Deep Learning

Using Python

Ahmad Ali AlZubi

Deep Learning Using Python

India | UAE | Nigeria | Uzbekistan | Montenegro | Iraq | Egypt | Thailand | Uganda | Philippines | Indonesia
www.parabpublications.com

Deep Learning Using Python

Authored By:

Ahmad Ali AlZubi
Professor, Computer Science Department, King Saud University,
Riyadh, Saudi Arabia

Copyright 2024 by Ahmad Ali AlZubi

First Impression: July 2024

Deep Learning Using Python

ISBN: 978-81-19585-32-8

Rs. 1000/- ($80)

No part of the book may be printed, copied, stored, retrieved, duplicated and reproduced in any form without the written permission of the editor/publisher.

DISCLAIMER

Information contained in this book has been published by Parab Publications and has been obtained by the author from sources believed to be reliable and correct to the best of their knowledge. The author is solely responsible for the contents of the articles compiled in this book. Responsibility of authenticity of the work or the concepts/views presented by the author through this book shall lie with the author and the publisher has no role or claim or any responsibility in this regard. Errors, if any, are purely unintentional and readers are requested to communicate such error to the author to avoid discrepancies in future.

Published by:
Parab Publications

Dedicated to

My Parents

Ali AlZubi and Ghazalah AlZubi

Preface

Deep Learning Using Python involves harnessing Python's robust libraries like TensorFlow, PyTorch, and Keras to develop and deploy sophisticated neural network models. Python's versatility and extensive ecosystem make it a preferred choice for implementing deep learning algorithms, enabling developers to tackle complex tasks such as image and speech recognition, natural language processing, and more. This book explores fundamental concepts, practical techniques, and advanced applications of deep learning within the Python environment.

The journey begins with an introduction to deep learning principles, covering neural network architecture, activation functions, loss functions, and optimization algorithms. Readers learn to preprocess data, including techniques for normalization, feature scaling, and handling missing values, essential for preparing datasets for training neural networks.

Next, the book delves into building various types of neural networks using Python libraries. From feedforward neural networks for basic tasks to convolutional neural networks (CNNs) for image data and recurrent neural networks (RNNs) for sequential data processing, each chapter provides hands-on examples and code snippets to facilitate understanding and implementation.

Advanced topics include transfer learning, where pre-trained models are finetuned for specific tasks, and generative adversarial networks (GANs) for creating synthetic data. Throughout the book, emphasis is placed on practical application, with discussions on model evaluation, hyperparameter tuning, and troubleshooting common issues encountered in deep learning projects.

Moreover, the integration of deep learning with Python's data visualization libraries like Matplotlib and Seaborn enables visualizing model performance metrics and data distributions, enhancing interpretability and decision-making in model development.

Real-world applications such as autonomous driving, medical diagnosis, and financial forecasting are explored to demonstrate the impact of deep learning in solving complex problems. Case studies and projects provide opportunities for readers to apply their knowledge, reinforcing concepts and enhancing practical skills in deep learning with Python.

Deep Learning Using Python equips readers with the essential tools and knowledge to embark on their journey into deep learning. Whether for academic study, professional development, or personal interest, this book serves as a comprehensive guide to mastering deep learning techniques and leveraging Python's capabilities to build intelligent systems for diverse applications.

This book explores the practical aspects of deep learning in Python, equipping readers with the skills to tackle complex AI challenges with confidence and clarity.

Acknowledgement

I would like to thank and heartfelt gratitude to King Saud University, Riyadh, Saudi Arabia for their supports.

I extend my heartfelt gratitude to the individuals who have contributed to the fruition of this endeavour, "**Deep Learning Using Python**" First and foremost, I express my deepest appreciation to the specialists in Machine Learning whose expertise and insights have shaped the content of this book, enabling a comprehensive understanding of the Deep Learning.

I extend my thanks to my colleagues for their invaluable support and encouragement throughout this journey.

I would like to thank my family and loved ones for their constant support, comprehension, and inspiration during the many hours that I have invested in the writing, research, and editing of this text.

Lastly, I express gratitude to the readers for their interest and trust in this work, with the hope that it serves as a meaningful resource in the field of Deep Learning.

Ahmad Ali AlZubi

Contents

Preface (*v* - *vi*)
Acknowledgement (*vii*)

1. An Introduction to Deep Learning **1**
- Principles of Deep Learning; Overview of Deep Lerning; Linear/ Logistic Regression; Deep Learning Algorithms over Traditional Machine Learning Algorithms?; Best Image Processing Tools Used in Machine Learning

2. Neural Network in Deep Learning **33**
- McCulloch Pitts Neuron — Deep Learning Building Block; Geometric Interpretation Of M-P Neuron; Biological Neurons: An Overly Simplified Illustration; Other important concepts of neural networks; Sigmoid Neuron — Building Block of Deep Neural Networks; Basic Introduction to Convolutional Neural Network in Deep Learning

3. Matlab and Python in Machine Learning **71**
- Matlab vs Python Performance; Deep Learning with MATLAB; MATLAB vs Python: Why and How to Make the Switch; Python Vs Matlab for Machine Learning; MATLAB vs. Python: The key differences; Generic Programming Tasks

4. Gradient Descent in Machine Learning **124**
- Plotting the Gradient Descent Algorithm; Gradient Descent or Steepest Descent; Gradient Descent; Gradient Descent algorithm and its variants; Gradient Descent: An Introduction to 1 of Machine Learning's Most Popular Algorithms; Applications of Machine learning; Image Processing With Deep Learning- A Quick Start Guide

5. **Natural Language Processing** 152
- The perceptron; Methods: Rules, statistics, neural networks

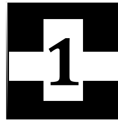
6. **Artificial Deep Neural Networks** 177
- Artificial neural network; Representation Power of functions; Crash Course on Multi-Layer Perceptron Neural Networks; Automatic speech recognition ; Deep Learning Applications to Know; Applications of Deep Learning You Need to Know

7. **Python Implementation of Neuron Model** 225
- McCulloch-Pitts Model of Neuron; Activation Functions in Neural Networks; Components of Neural Networks

8. **Sigmoid Neurons Function in Deep Learning** 246
- Sigmoid Neurons: An Introduction; A Gentle Introduction To Sigmoid Function; Multi-Layer Neural Networks: An Intuitive Approach; Activation Functions; Activation Functions and their Derivatives; Types of Activation Functions; Activation functions in Neural Networks; Intro to optimization in deep learning: Gradient Descent; Deep Learning Applications Used Across Industries

Bibliography 289

Index 291



An Introduction to Deep Learning

Deep learning is a subset of machine learning that focuses on algorithms inspired by the structure and function of the human brain's neural networks. It has revolutionized artificial intelligence (AI) by enabling machines to learn from large amounts of data and perform tasks that were previously thought to be exclusive to human cognition.

This introduction provides an overview of deep learning, its principles, applications, and key concepts.

PRINCIPLES OF DEEP LEARNING

At its core, deep learning utilizes artificial neural networks (ANNs) to process data and extract meaningful patterns. These networks are composed of layers of interconnected nodes, or neurons, that perform computations. Each layer processes the input data and passes it on to the next layer, with each subsequent layer learning more abstract features from the data.

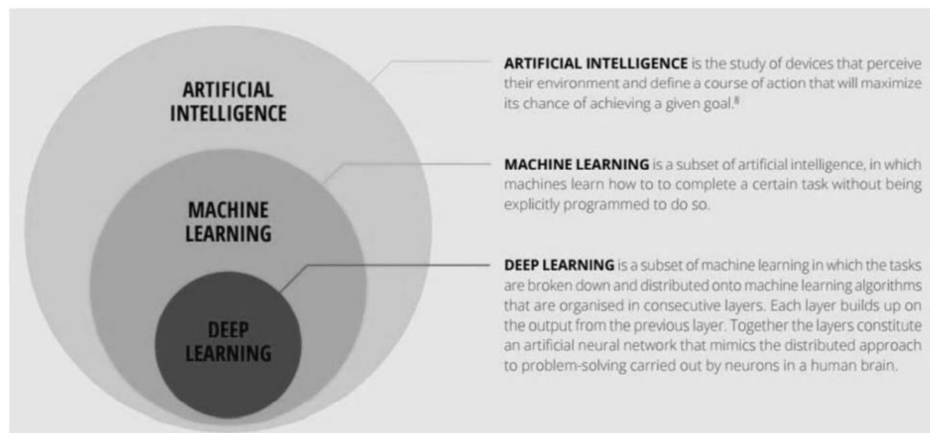
NEURAL NETWORK LAYERS

1. Input Layer: Receives raw data as input, such as images, text, or numerical data.
2. Hidden Layers: Intermediate layers between the input and output layers. Each hidden layer performs complex transformations and feature extraction.

3. Output Layer: Produces the final output based on the computations performed by the hidden layers. The number of nodes in the output layer depends on the type of problem (e.g., classification, regression).

FUTURE DIRECTIONS AND CHALLENGES

As deep learning continues to evolve, researchers are exploring new architectures (e.g., attention mechanisms, graph neural networks) and improving model interpretability and efficiency. Challenges include data privacy concerns, biases in training data, and the need for computational resources for training large models.



Deep learning is a sub-field of machine learning dealing with algorithms inspired by the structure and function of the brain called artificial neural networks. In other words, It mirrors the functioning of our brains. Deep learning algorithms are similar to how nervous system structured where each neuron connected each other and passing information.

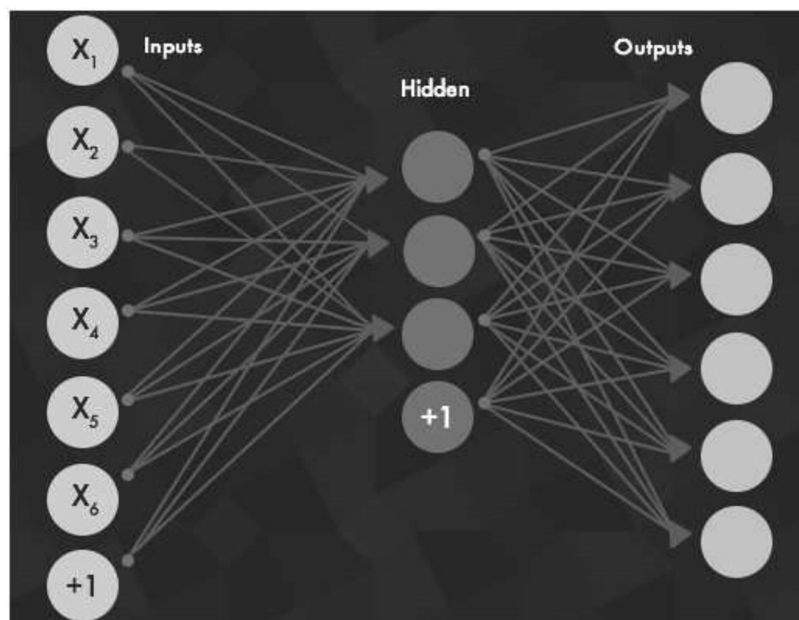
APPLICATIONS OF DEEP LEARNING

Deep learning has found applications across various domains, transforming industries and enhancing capabilities in:

- Computer Vision: Deep learning models like Convolutional Neural Networks (CNNs) are used for tasks such as image classification, object detection, and facial recognition. Applications include autonomous vehicles, medical imaging, and surveillance systems.
- Natural Language Processing (NLP): Deep learning techniques such as Recurrent Neural Networks (RNNs) and Transformer models are applied to tasks like language translation, sentiment analysis, and chatbots.

Companies use NLP for customer service automation, content generation, and information retrieval.

- Speech Recognition: Deep learning algorithms power speech recognition systems in virtual assistants (e.g., Siri, Alexa) and dictation software. They convert spoken language into text, enabling hands-free operation and accessibility features.
- Healthcare: Deep learning models analyze medical images (e.g., X-rays, MRIs) to assist in disease diagnosis and treatment planning. They also predict patient outcomes based on electronic health records (EHR) data, improving personalized medicine.
- Finance: Deep learning algorithms analyze financial data for fraud detection, risk assessment, and trading strategies. They process vast amounts of data in real-time, providing insights for investment decisions and market forecasting.

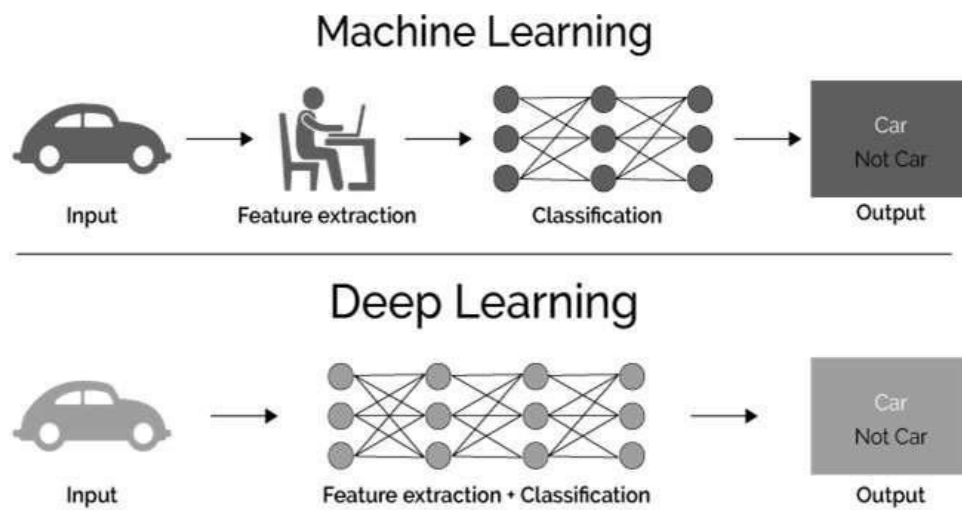


Key Concepts in Deep Learning

1. Backpropagation: A training algorithm that adjusts the weights of neural network connections based on the error between predicted and actual outputs. It enables networks to learn from data and improve performance over time.
2. Activation Functions: Functions applied to neuron outputs to introduce

non-linearity and enable the network to learn complex patterns. Common activation functions include ReLU (Rectified Linear Unit) and Sigmoid.

3. Loss Functions: Measure the difference between predicted and actual outputs during training. They guide the optimization process by quantifying the model's performance.
4. Optimization Algorithms: Techniques such as Gradient Descent and its variants adjust network parameters to minimize the loss function, optimizing model performance.



One of differences between machine learning and deep learning model is on the feature extraction area. Feature extraction is done by human in machine learning whereas deep learning model figure out by itself.

OVERVIEW OF DEEP LERNING

Most modern deep learning models are based on artificial neural networks, specifically convolutional neural networks (CNN)s, although they can also include propositional formulas or latent variables organized layer-wise in deep generative models such as the nodes in deep belief networks and deep Boltzmann machines.

In deep learning, each level learns to transform its input data into a slightly more abstract and composite representation. In an image recognition application, the raw input may be a matrix of pixels; the first representational layer may abstract the pixels and encode edges; the second layer may compose and encode arrangements of edges; the third layer may encode a nose and eyes; and the fourth layer may recognize that the image contains a face. Importantly, a deep learning

process can learn which features to optimally place in which level *on its own*. This does not eliminate the need for hand-tuning; for example, varying numbers of layers and layer sizes can provide different degrees of abstraction.

The word "deep" in "deep learning" refers to the number of layers through which the data is transformed. More precisely, deep learning systems have a substantial *credit assignment path* (CAP) depth. The CAP is the chain of transformations from input to output. CAPs describe potentially causal connections between input and output. For a feedforward neural network, the depth of the CAPs is that of the network and is the number of hidden layers plus one (as the output layer is also parameterized). For recurrent neural networks, in which a signal may propagate through a layer more than once, the CAP depth is potentially unlimited. No universally agreed-upon threshold of depth divides shallow learning from deep learning, but most researchers agree that deep learning involves CAP depth higher than 2. CAP of depth 2 has been shown to be a universal approximator in the sense that it can emulate any function. Beyond that, more layers do not add to the function approximator ability of the network. Deep models (CAP > 2) are able to extract better features than shallow models and hence, extra layers help in learning the features effectively.

Deep learning architectures can be constructed with a greedy layer-by-layer method. Deep learning helps to disentangle these abstractions and pick out which features improve performance.

For supervised learning tasks, deep learning methods eliminate feature engineering, by translating the data into compact intermediate representations akin to principal components, and derive layered structures that remove redundancy in representation.

Deep learning algorithms can be applied to unsupervised learning tasks. This is an important benefit because unlabeled data are more abundant than the labeled data. Examples of deep structures that can be trained in an unsupervised manner are deep belief networks.

INTERPRETATIONS

Deep neural networks are generally interpreted in terms of the universal approximation theorem or probabilistic inference.

The classic universal approximation theorem concerns the capacity of feedforward neural networks with a single hidden layer of finite size to approximate continuous functions. In 1989, the first proof was published by George Cybenko for sigmoid activation functions and was generalised to feed-forward multi-layer architectures in 1991 by Kurt Hornik. Recent work also showed that universal

approximation also holds for non-bounded activation functions such as the rectified linear unit.

The universal approximation theorem for deep neural networks concerns the capacity of networks with bounded width but the depth is allowed to grow. Lu et al. proved that if the width of a deep neural network with ReLU activation is strictly larger than the input dimension, then the network can approximate any Lebesgue integrable function; If the width is smaller or equal to the input dimension, then a deep neural network is not a universal approximator.

The probabilistic interpretation derives from the field of machine learning. It features inference, as well as the optimization concepts of training and testing, related to fitting and generalization, respectively. More specifically, the probabilistic interpretation considers the activation nonlinearity as a cumulative distribution function. The probabilistic interpretation led to the introduction of dropout as regularizer in neural networks. The probabilistic interpretation was introduced by researchers including Hopfield, Widrow and Narendra and popularized in surveys such as the one by Bishop.

HISTORY

Some sources point out that Frank Rosenblatt developed and explored all of the basic ingredients of the deep learning systems of today. He described it in his book "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms", published by Cornell Aeronautical Laboratory, Inc., Cornell University in 1962.

The first general, working learning algorithm for supervised, deep, feedforward, multilayer perceptrons was published by Alexey Ivakhnenko and Lapa in 1967. A 1971 paper described a deep network with eight layers trained by the group method of data handling. Other deep learning working architectures, specifically those built for computer vision, began with the Neocognitron introduced by Kunihiko Fukushima in 1980.

The term *Deep Learning* was introduced to the machine learning community by Rina Dechter in 1986, and to artificial neural networks by Igor Aizenberg and colleagues in 2000, in the context of Boolean threshold neurons.

In 1989, Yann LeCun et al. applied the standard backpropagation algorithm, which had been around as the reverse mode of automatic differentiation since 1970, to a deep neural network with the purpose of recognizing handwritten ZIP codes on mail. While the algorithm worked, training required 3 days.

Independently in 1988, Wei Zhang et al. applied the backpropagation algorithm to a convolutional neural network (a simplified Neocognitron by keeping only the

convolutional interconnections between the image feature layers and the last fully connected layer) for alphabets recognition and also proposed an implementation of the CNN with an optical computing system. Subsequently, Wei Zhang, et al. modified the model by removing the last fully connected layer and applied it for medical image object segmentation in 1991 and breast cancer detection in mammograms in 1994.

In 1994, André de Carvalho, together with Mike Fairhurst and David Bisset, published experimental results of a multi-layer boolean neural network, also known as a weightless neural network, composed of a 3-layers self-organising feature extraction neural network module (SOFT) followed by a multi-layer classification neural network module (GSN), which were independently trained. Each layer in the feature extraction module extracted features with growing complexity regarding the previous layer.

In 1995, Brendan Frey demonstrated that it was possible to train (over two days) a network containing six fully connected layers and several hundred hidden units using the wake-sleep algorithm, co-developed with Peter Dayan and Hinton. Many factors contribute to the slow speed, including the vanishing gradient problem analyzed in 1991 by Sepp Hochreiter.

Since 1997, Sven Behnke extended the feed-forward hierarchical convolutional approach in the Neural Abstraction Pyramid by lateral and backward connections in order to flexibly incorporate context into decisions and iteratively resolve local ambiguities.

Simpler models that use task-specific handcrafted features such as Gabor filters and support vector machines (SVMs) were a popular choice in the 1990s and 2000s, because of artificial neural network's (ANN) computational cost and a lack of understanding of how the brain wires its biological networks.

Both shallow and deep learning (e.g., recurrent nets) of ANNs have been explored for many years. These methods never outperformed non-uniform internal-handcrafting Gaussian mixture model/Hidden Markov model (GMM-HMM) technology based on generative models of speech trained discriminatively. Key difficulties have been analyzed, including gradient diminishing and weak temporal correlation structure in neural predictive models. Additional difficulties were the lack of training data and limited computing power.

Most speech recognition researchers moved away from neural nets to pursue generative modeling. An exception was at SRI International in the late 1990s. Funded by the US government's NSA and DARPA, SRI studied deep neural networks in speech and speaker recognition. The speaker recognition team led by

Larry Heck reported significant success with deep neural networks in speech processing in the 1998 National Institute of Standards and Technology Speaker Recognition evaluation. The SRI deep neural network was then deployed in the Nuance Verifier, representing the first major industrial application of deep learning.

The principle of elevating "raw" features over hand-crafted optimization was first explored successfully in the architecture of deep autoencoder on the "raw" spectrogram or linear filter-bank features in the late 1990s, showing its superiority over the Mel-Cepstral features that contain stages of fixed transformation from spectrograms. The raw features of speech, waveforms, later produced excellent larger-scale results.

Many aspects of speech recognition were taken over by a deep learning method called long short-term memory (LSTM), a recurrent neural network published by Hochreiter and Schmidhuber in 1997. LSTM RNNs avoid the vanishing gradient problem and can learn "Very Deep Learning" tasks that require memories of events that happened thousands of discrete time steps before, which is important for speech. In 2003, LSTM started to become competitive with traditional speech recognizers on certain tasks. Later it was combined with connectionist temporal classification (CTC) in stacks of LSTM RNNs. In 2015, Google's speech recognition reportedly experienced a dramatic performance jump of 49% through CTC-trained LSTM, which they made available through Google Voice Search.

In 2006, publications by Geoff Hinton, Ruslan Salakhutdinov, Osindero and Teh showed how a many-layered feedforward neural network could be effectively pre-trained one layer at a time, treating each layer in turn as an unsupervised restricted Boltzmann machine, then fine-tuning it using supervised backpropagation. The papers referred to *learning* for *deep belief nets.*

Deep learning is part of state-of-the-art systems in various disciplines, particularly computer vision and automatic speech recognition (ASR). Results on commonly used evaluation sets such as TIMIT (ASR) and MNIST (image classification), as well as a range of large-vocabulary speech recognition tasks have steadily improved. Convolutional neural networks (CNNs) were superseded for ASR by CTC for LSTM. but are more successful in computer vision.

The impact of deep learning in industry began in the early 2000s, when CNNs already processed an estimated 10% to 20% of all the checks written in the US, according to Yann LeCun. Industrial applications of deep learning to large-scale speech recognition started around 2010.

The 2009 NIPS Workshop on Deep Learning for Speech Recognition was motivated by the limitations of deep generative models of speech, and the possibility

that given more capable hardware and large-scale data sets that deep neural nets (DNN) might become practical. It was believed that pre-training DNNs using generative models of deep belief nets (DBN) would overcome the main difficulties of neural nets. However, it was discovered that replacing pre-training with large amounts of training data for straightforward backpropagation when using DNNs with large, context-dependent output layers produced error rates dramatically lower than then-state-of-the-art Gaussian mixture model (GMM)/Hidden Markov Model (HMM) and also than more-advanced generative model-based systems. The nature of the recognition errors produced by the two types of systems was characteristically different, offering technical insights into how to integrate deep learning into the existing highly efficient, run-time speech decoding system deployed by all major speech recognition systems. Analysis around 2009–2010, contrasting the GMM (and other generative speech models) vs. DNN models, stimulated early industrial investment in deep learning for speech recognition, eventually leading to pervasive and dominant use in that industry. That analysis was done with comparable performance (less than 1.5% in error rate) between discriminative DNNs and generative models.

In 2010, researchers extended deep learning from TIMIT to large vocabulary speech recognition, by adopting large output layers of the DNN based on context-dependent HMM states constructed by decision trees.

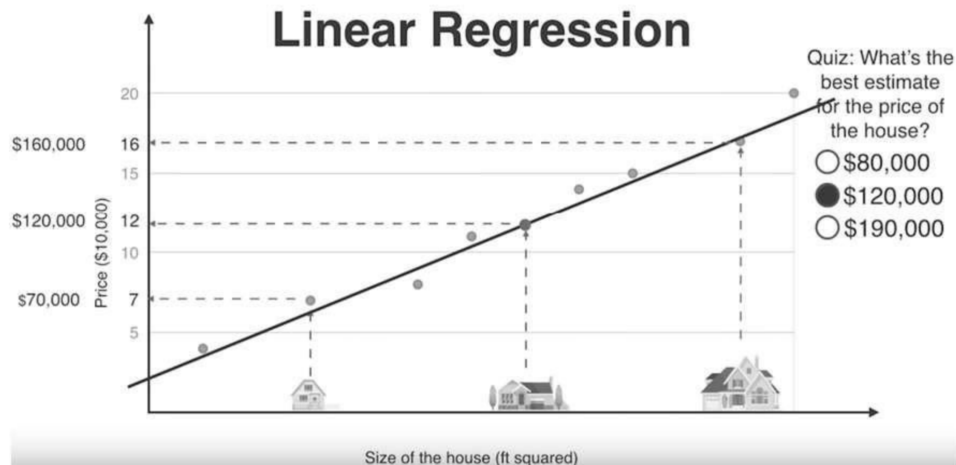
Advances in hardware have driven renewed interest in deep learning. In 2009, Nvidia was involved in what was called the "big bang" of deep learning, "as deep-learning neural networks were trained with Nvidia graphics processing units (GPUs)." That year, Andrew Ng determined that GPUs could increase the speed of deep-learning systems by about 100 times. In particular, GPUs are well-suited for the matrix/vector computations involved in machine learning. GPUs speed up training algorithms by orders of magnitude, reducing running times from weeks to days. Further, specialized hardware and algorithm optimizations can be used for efficient processing of deep learning models.

LINEAR/LOGISTIC REGRESSION

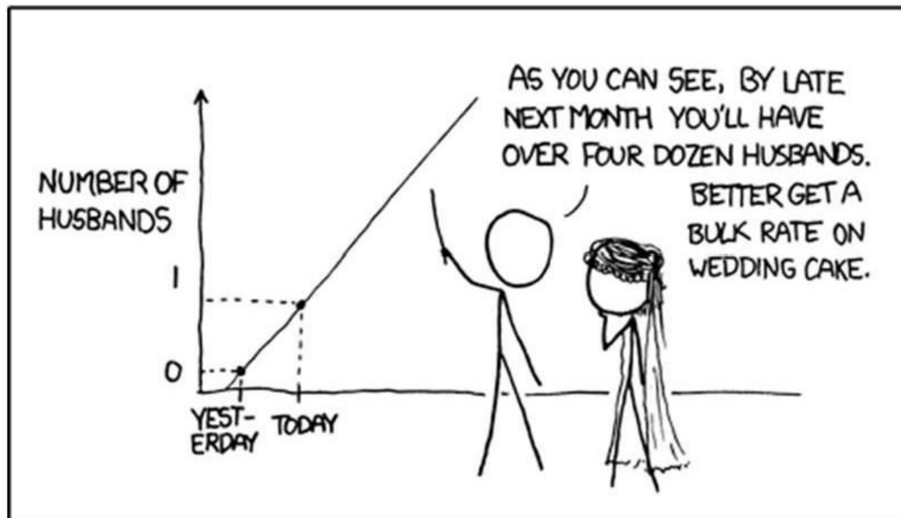
We cannot start deep learning without explaining linear and logistics regression which is the basis of deep learning.

Linear regression

It is a statistical method that allows us to summarise and study relationships between two continuous (quantitative) variables.



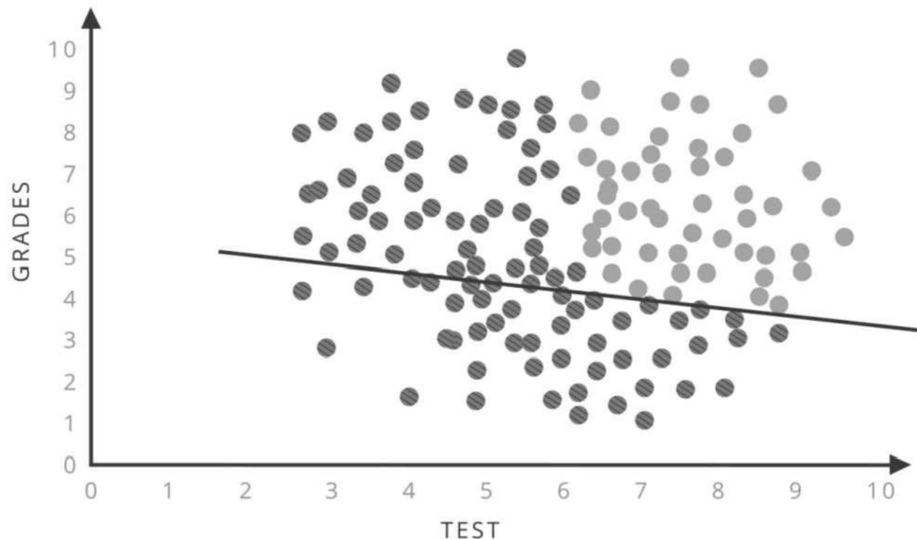
In this example, we have historical data based on the size of the house. We plot them into the graph as seen as dot points. Linear regression is the technique where finding a straight line between these points with less error(this will be explained later). Once we have a line with less error, we can predict the house price based on the size of the house.



Here is another example how linear regression predict in a joke manner.

Logistic regression

It is a statistical method for analysing a dataset in which there are one or more independent variables that determine an outcome. The outcome is measured in which there are only two possible outcomes: True or False.



In this example, we have historical dataset of student which have passed and not passed based on the grades and test scores. If we need to know a student will pass or not based on the grade and test score, logistic regression can be used.

In logistic regression, similar to linear regression, it will find best possible straight line that separate the two classification(passed and not passed).

ACTIVATION FUNCTION

Activation functions are functions that decide, given the inputs into the node, what should be the node's output? Because it's the activation function that decides the actual output, we often refer to the outputs of a layer as its "activations".

One of the simplest activation functions is the **Heaviside step function**. This function returns a **0** if the linear combination is less than 0. It returns a **1** if the linear combination is positive or equal to zero.

$$f(h) = \begin{cases} 0 & \text{if } h < 0 \\ 1 & \text{if } h \geq 0 \end{cases}$$

The output unit returns the result of f(h), where h is the input to the output unit:

WEIGHTS

When input data comes into a neuron, it gets multiplied by a weight value

that is assigned to this particular input. For example, the neuron above university example have two inputs, tests for test scores and grades, so it has two associated weights that can be adjusted individually.

Use of weights

These weights start out as random values, and as the neural network learns more about what kind of input data leads to a student being accepted into a university, the network adjusts the weights based on any errors in categorization that the previous weights resulted in. This is called **training** the neural network.

Remember we can associate weight as m(slope) in the orginal linear equation.

$y = mx+b$

BIAS

Weights and biases are the learnable parameters of the deep learning models.

Bias represented as b in the above linear equation.

DEEP LEARNING ALGORITHMS OVER TRADITIONAL MACHINE LEARNING ALGORITHMS?

The above graph depicts a performance comparison between the traditional machine learning algorithms and deep learning techniques on the amount of data. It is evident from the chart that the performance of deep learning algorithms increases with an increase in the amount of data. In contrast, for the traditional algorithms in machine learning, the performance rises to an extent but remains constant (flat line).

As per International Data Corporation (IDC), worldwide data will grow 61% to 175 zettabytes by 2025! The increasing usage of Social Media platforms, Mobile applications, etc. generates a humongous amount of data.

With the increase in the data and most of the data being unstructured (images, videos, audio, etc.) type, deep learning algorithms play a vital role in the data revolution era.

WHAT IS DEEP LEARNING?

Imagine when you were a kid, and you were asked to learn English Alphabets with the books having colorful pictures showing an Apple image for letter A, a Ball image for letter B, and so on. Why do you think they had those images when the objective was only to learn the alphabet?

We human beings have a knack for learning concepts visually, and we tend

to remember them by using a reference of these visuals.That is why you are likely to forget the answers you have written in a certification exam after a couple of days if you don't revise them again. Similarly, if you have binge-watched a sitcom on Netflix, you are likely to remember the dialogues and scenes for a long time if you watch it repeatedly.

INTRODUCTION TO DEEP LEARNING ALGORITHMS

Before we move on to the list of deep learning models in machine learning, let's understand the structure and working of deep learning algorithms with the famous MNIST dataset. The human brain is a network of billions of neurons that help represent a tremendous amount of knowledge. Deep Learning also uses the same analogy of a brain neuron for processing the information and recognizing them. Let's understand this with an example.

The above image is taken from the very famous MNIST dataset that gives a glimpse of the visual representation of digits. The MNIST dataset is widely used in many image processing techniques. Now, let's observe the image of how each number is written in different ways. We as human beings can easily understand these digits even if they are slightly tweaked or written differently because we have written them and seen them millions of times. But how will you make a computer recognize these digits if you are building an image processing system? That is where Deep learning comes into the picture!

What is a Neural Network in Deep Learning?

One can visually represent the fundamental structure of a Neural network cas in the above image, with mainly three components –

1. Input Layer
2. Hidden Layers
3. Output layer

The above image shows only one Hidden layer, and we can call it an Artificial Neural Network or a neural network. On the other hand, deep learning has several hidden layers, and that is where it gets its name "Deep". These hidden layers are interconnected and are used to make our model learn to give the final output.

Each node with information is passed in the form of inputs, and the node multiplies the inputs with random weight values and adds a bias before calculation. A nonlinear or activation function is then applied to determine which particular node will determine the output.

The activation functions used in artificial neural networks work like logic gates. So, if we require an output to be 1 for an OR gate. We will need to pass

the input values as 0,1 or 1,0. Different deep learning models use different activation functions and sometimes a combination of activation functions. We have similarities in neural networks and deep learning structures. But one cannot use neural networks for unstructured data like images, videos, sensor data, etc.

We need multiple hidden layers (sometimes even thousands) for these types of data, so we use deep neural networks.

How do Deep Learning Algorithms Work?

For the MNIST example discussed above, we can consider the digits as the input that are sent in a 28x28 pixel grid format to hidden layers for digit recognition. The hidden layers classify the digit (whether it is 0,1,2,...9) based on the shape. For example – If we consider a digit 8, it looks like having two knots interconnected to each other. The image data converted into pixel binaries(0,1) is sent as an input to the input layer.

Each connection in the layers has a weight associated with it, which determines the input value's importance. The initial weights are set randomly.

We can have negative weights also associated with these connections if the importance needs to be reduced.

The weights are updated after every iteration using the backpropagation algorithm.

In some cases, there might not be a prominent input image of the digit, and that is when several iterations have to be performed to train the deep learning model by increasing the number of hidden layers. Finally, the final output is generated based on the weights and number of iterations.

Now that we have a basic understanding of input and output layers in deep learning, let's understand some of the primary deep learning algorithms, how they work, and their use cases.

TOP DEEP LEARNING ALGORITHMS LIST

Multilayer Perceptrons (MLPs)

MLP is the most basic deep learning algorithm and also one of the oldest deep learning techniques. If you are a beginner in deep learning and have just started exploring it, we recommend you get started with MLP. MLPs can be referred to as a form of Feedforward neural networks.

How does MLP deep learning algorithm work?

- The working of MLP is the same as what we discussed above in our MNIST data example. The first layer takes the inputs, and the last produce

the output based on the hidden layers. Each node is connected to every node on the next layer, so the information is constantly fed forward between the multiple layers, which is why it is referred to as a feed-forward network.

- MLP uses a prevalent supervised learning technique called backpropagation for training.
- Each hidden layer is fed with some weights (randomly assigned values). The combination of the weights and input is supplied to an activation function which is passed further to the next layer to determine the output. If we don't arrive at the expected output, we calculate the loss (error) and we back-track to update the weights. It is an iterative process until the predicted output is obtained (trial and error). It is critical in training the deep learning model, as the correct weights will determine your final output.
- MLP's popularly use sigmoid functions, Rectified Linear unit (ReLU), and tanh as activation functions.

APPLICATIONS OF MLP

It is used by Social media sites (Instagram, Facebook) for compressing image data. That significantly helps to load the images even if the network strength is not too strong.

Other applications include Used in image and speech recognition, data compression, and also for classification problems.

Pros of MLP

1. They do not make any assumptions regarding the Probability density functions (PDF), unlike the other models that are based on Probability.
2. Ability to provide the decision function directly by training the perceptron.

Cons of MLP

1. Due to the hard-limit transfer function, the perceptrons can only give outputs in the form of 0 and 1.
2. While updating the weights in layers, the MLP network may be stuck in a local minimum which can hamper accuracy.

RADIAL BASIS FUNCTION NETWORKS (RBFNS)

As the name suggests, it is based on the Radial basis function (RBF) activation function. The model training process requires slightly less time using RBFN than MLP.

How do RBFN deep learning algorithms work?

A straightforward type of RBFN is a three-layer feedforward neural network with an input layer, a hidden layer consisting of several RBF nonlinear activation units, and a linear output layer that acts as a summation unit to give the final output.

RBFN uses trial and error to determine the structure of the network. That is done in two steps -

- In the first stage, the centers of the hidden layer using an unsupervised learning algorithm (k-means clustering) are determined.
- In the next step, the weights with linear regression are determined. Mean Squared Error (MSE) is used to determine the error and the weights are tweaked accordingly to minimize MSE.

Applications of RBFN

RBFNs are used to analyze stock market prices and also forecast sales prices in Retail industries because of their ability to work on time-series-based data. Other applications include Speech recognition, time-series analysis, image recognition, adaptive equalization, medical diagnosis, etc.

Pros of RBFN

1) The training process is faster when compared to MLP, as there is no backpropagation involved.
2) It is easy to interpret the roles of the hidden layer nodes compared to MLP.

Cons of RBFN

Although we have seen that the training is faster in the RBF network, classification takes time as compared to Multilayer Perceptrons.

The reason is that every node in the hidden layer must compute the RBF function for the input sample vector during classification.

It is easy to interpret the roles of the hidden layer nodes compared to MLP.

We want our model to recognize the objects irrespective of what surface they are upon and their position.

In CNN, the processing of data involves breaking the images into many numbers of overlapping tiles instead of feeding entire images into our network. And then, we use a technique called a sliding window over the whole original image and save the results as a separate tiny picture tile. The Sliding window is a kind of brute force solution where we scan all around for a given image to detect the object for all possible sections, each section at a time, until we get the expected object.

How do CNN deep learning algorithms work?

There are three basic building blocks of a CNN

1. Convolutional layers
2. Pooling layers
3. Full-Connected Layers

Convolutional Layer – It is the most significant building block of convolutional neural networks. A set of filters(kernels) are used in the layer's parameters that can be visualized as neurons of the layers. They have weighted inputs and based on the input size (a fixed square) that is also referred to as a receptive field; they provide the output. These filters, when applied to the input image, generate feature maps. It is the output of one filter that is applied to the previous layer. Moving one image pixel at a time, a given filter is drawn across the entire previous layer. For each position, a particular neuron is activated, and the output is collected in a feature map.

To avoid losing the arrangement of the original image tiles, we save the result obtained after processing each tile into a grid with the same tiles' arrangement as the original image.

In the convolution layer, the output is a grid array huge in size. To reduce the size of this array, we use an algorithm max pooling for down-sampling. The basic idea is to keep only the most significant input tile from the array.

Full-connected Network – The array is only a bunch of numerical values, so we can input them into a neural network that is fully connected (all neurons are connected). CNN most commonly uses ReLU as the activation function.

APPLICATIONS OF CNN

Facebook, Instagram, social media sites, etc., use CNNs for face detection and recognition. So when trying to tag your friend in your post, you are using CNN!

Other applications include Video analysis, Image recognition, Natural language processing, Forecasting, etc.

Pros of CNN

CNN results are more accurate, particularly for image/object recognition use cases, compared to other algorithms.

Cons of CNN

High computation power is required for training CNNs. So, they are not cost-effective.

RECURRENT NEURAL NETWORKS (RNNS)

Have you noticed when you start typing something, Google automatically completes the sentence for you! Now, if you are thinking about how it works, the secret is RNN. Recurrent Neural Networks have directed cycles among the interconnected nodes. They use their memory to process the next sequence of inputs to implement the auto-complete feature kind of functionalities. RNNs can take a series of inputs with no limit on their size, making them unique.

The Basic Structure of RNN

The above figure shows the different steps for each time state for an RNN. RNNs do not only account for the weights of hidden units to determine the output but also use the information learned from their prior inputs. RNNs are a special type of deep neural network that can remember those characters because of their internal memory. An output is produced, which is copied and provided back to the deep neural network like a loop. That is why the input could produce a different output based on previous inputs in the connected layers.

Let us understand this by an example -

Example - Imagine if you have built a feed-forward network that takes words as input and processes the word character by character. You pass the word ProjectPro, and by the time you reach the character "o", it would have already forgotten the last characters "P," "r", and "o".

Applications of RNN

Google, Search Engines, and Web Browsers extensively use RNN to auto-complete words and sentences. Other applications are Text Detection and Recognition, Analyzing video frames, etc.

Pros of RNN

The ability of RNN models to remember information throughout the training period plays a pivotal role in time series prediction.

Cons of RNN

1. The computation is time-taking because of its recurrent nature.
2. Hard to process long sequences in the training data set, mainly when we use ReLU or tanh as activation functions.

LONG SHORT-TERM MEMORY NETWORKS (LSTMS)

LSTMs are a special kind of RNN and are highly capable of learning long-term dependencies. Let's try to understand long-term dependencies by an example.

Suppose you have built a model to predict the next word based on the previous ones. Assume you are trying to predict the last word in the sentence, "the sun rises in the east," we don't need any further context, and obviously the following term will be east. In these types of cases, where there is not much gap between the relevant information and the place where it's needed, RNNs can learn and predict the output easily. But if we have a sentence like, "I am born in India. I speak fluent Hindi". This kind of prediction requires some context from the previous sentence about where a person was born, and RNNs might not be able to learn and connect the information in such cases.

How do LSTM deep learning algorithms work?

The cell state and hidden state are transferred to the next cell. As the name suggests, memory blocks remember things, and the changes to these memory blocks are done through mechanisms referred to as gates.

The key to LSTMs is the cell state (the horizontal line at the top, which runs through in the diagram). The key to LSTMs is the cell state (the horizontal line at the top which runs through in the diagram).

Step 1: - LSTM decides what information should be kept intact and what should be thrown away from the cell state. The sigmoid layer is responsible for making this decision.

Step 2: - LSTM decides what new information one should keep and replaces the irrelevant one identified in step 1 - the tanh and the sigmoid play an important role in identifying relevant information.

Step 3: - The output is determined with the help of the cell state that will now be a filtered version because of the applied sigmoid and tanh functions.

Applications

Anomaly detection in network traffic data or IDSs (intrusion detection systems), Time-series forecasting, Auto-completion, text and video analysis, and Caption generation.

Pros of LSTM

LSTMs when compared to conventional RNNs, are very handy in modeling the chronological sequences and long-range dependencies.

Cons of LSTM

1. High computation and resources are required to train the LSTM model, and it is also a very time-consuming process.
2. They are prone to overfitting.

RESTRICTED BOLTZMANN MACHINES (RBMS)

RBM is one of the oldest algorithm in deep learning invented in 1986 by Geoffrey Hinton. I bet you would have noticed how YouTube recommends videos similar to what you have watched recently. Also, if you have watched a web series or movie on Netflix, you will start getting a lot of recommendations related to them. They use a technique known as collaborative filtering that uses RBMs.

How do RBM deep learning algorithms work?

RBM is one of the simplest deep learning algorithms and has a basic structure with just two layers-

1. (Visible) Input layer
2. Hidden layer

The input x is multiplied by the respective weight(w) at each hidden node. A single input x can have 8 weights altogether (2 input nodes x 4 hidden nodes). The hidden nodes receive the inputs multiplied by their respective weights and a bias value. The result is passed through an activation function to the output layer for reconstruction. RBMs compare this reconstruction with the original input to determine the quality of the result. If the quality of the result is not good, the weights are updated, and a new reconstruction is built.

Applications

Netflix, Prime Video, and Streaming apps provide recommendations to users based on their watching patterns using the RBM algorithm. Feature extraction in pattern recognition, Recommendation Engines, Classification problems, Topic modeling, etc.

Pros of RBM

1. RBMs can be pre-trained in a completely unsupervised way as the learning algorithm can efficiently use extensive unlabelled data.
2. They don't require high computation and can encode any distribution.

Cons of RBM

1. Calculation of energy gradient function while training is challenging.
2. Adjusting weights using the CD-k algorithm is not as easy as backpropagation.

SELF ORGANIZING MAPS (SOMS)

Imagine you are working with a dataset with hundreds of features, and you want to visualize your data to understand the correlation between each feature.

It is not possible to imagine it using scatter or pair plots. Here comes SOMs. It reduces the data's dimension (less relevant features are removed) and helps us visualize the distribution of feature values.

How does the SOM deep learning algorithm work?

SOMs group similar data items together by creating a 1D or 2D map. Similar to the other algorithms, weights are initialized randomly for each node. At each step, one sample vector x is randomly taken from the input data set and the distances between x and all the other vectors are computed.

A *Best-Matching Unit* (BMU) closest to x is selected after voting among all the other vectors. Once BMU is identified, the weight vectors are updated, and the BMU and its topological neighbors are moved closer to the input vector x in the input space. This process is repeated until we get the expected output.

For our example, the program would first select a color from an array of samples, such as red, and then search the weights for those red locations. The weights surrounding those locations are red, and then the next color, blue is chosen, and the process continues.

Applications

Image analysis, fault diagnosis, process monitoring and control, etc. SOMs are used for 3D modeling human heads from stereo images because of their ability to generate powerful visualizations, and they are extensively valuable for the healthcare sector for creating 3D charts.

Pros of SOMs

1. We can easily interpret and understand the data using SOM.
2. Using dimensionality reduction further makes it much simpler to check for any similarities within our data.

Cons of SOMs

1. SOM requires neuron weights to be necessary and sufficient to cluster the input data.
2. If, while training SOM, we provide less or extensively more data, we may not get the informative or very accurate output.

GENERATIVE ADVERSARIAL NETWORKS (GANS)

It is an unsupervised learning algorithm capable of automatically discovering and learning the patterns in the data. GANs then generate new examples that resemble the original dataset.

How do GAN deep learning algorithms work?

GANs consist of two neural networks.

- **Generator Network** - First is a generator neural network that generates new examples.
- **Discriminator Network** – It is responsible for evaluating the generated examples and whether they belong to the actual training dataset.

Let us understand this by an example. Consider a currency note checking machine. The machine is responsible for checking if the notes are fake or real. The Generator network will try to create counterfeit notes and send them to the Discriminator. The Discriminator will take in both the real (input training data) and the fake notes and return a value between 0 to 1. This value is a probability where 1 represents completely genuine notes and 0 represents fake notes.

Both Generator and Discriminator will try to outperform each other and get trained at the same time.

Applications

GANs are widely used in the gaming industry for 3D object generations. They are also used for editing images, generating cartoon characters, etc.

They are also used for illustrations for novels, articles, etc.

Pros of GANs

1. GANs can learn any data's internal representation (messy and complex distributions). They can be trained effectively using unlabeled data so they can quickly produce realistic and high-quality results.
2. They can recognize objects as well as can calculate the distance between them.

Cons of GANs

1. As they generate new data from original data, there is no such evaluation metric to judge the accuracy of output.
2. High computation and time required for model training.

AUTOENCODERS DEEP LEARNING ALGORITHM

Autoencoders are unsupervised algorithms very similar to Principal Component Analysis (PCA) in machine learning. They are used to convert multi-dimensional data into low-dimensional data. And if we want the original data, we can regenerate it back.

A simple example is -Suppose your friend has asked you to share a software you have saved on your computer. The folder size of that software is close to 1 GB. If you directly upload this whole folder to your Google drive, it will take a lot of time. But if you compress it, then the data size will reduce, and you can upload it easily. Your friend can directly download this folder, extract the data, and get the original folder.

In the above example, the original folder is the input, the compressed folder is encoded data, and when your friend extracts the compressed folder, it is decoding.

How do autoencoders work?

There are 3 main components in Autoencoders –

1. **Encoder** – The encoder compresses the input into a latent space representation which can be reconstructed later to get the original input.
2. **Code** – This is the compressed part (latent space representation) that is obtained after encoding.
3. **Decoder** – The decoder aims to reconstruct the code to its original form. The reconstruction output obtained may not be as accurate as the original and might have some loss.

The code layer present between the encoder and decoder is also referred to as Bottleneck. It is used to decide which aspects of input data are relevant and what can be neglected. The bottleneck is a very significant layer in our network. Without it, the network could easily learn to memorize the input values by passing them along through the network.

Applications

Colouring of images, image compression, denoising, etc.

They are used in the healthcare industry for medical imaging (technique and process of imaging the human body's interior for performing clinical analysis) Eg - breast cancer detection.

Pros of Autoencoders

Using multiple encoder and decoder layers reduces the computational cost of representing some functions to a certain extent.

Cons of Autoencoders

1. It is not as efficient as GANs when reconstructing images as for complex images,it usually does not work well.
2. We might lose essential data from our original input after encoding.

DEEP BELIEF NETWORKS

A deep belief network (DBN) is built by appending several Restricted Boltzmann Machines (RBM) layers. Each RBM layer can communicate with both the previous and subsequent layers.

DBNs are pre-trained by using the Greedy algorithm. It uses a layer-by-layer approach to learn all the generative weights and the top-down approaches. The weights associated determine how all variables in one layer rely on the other variables in the above layer.

How do deep belief networks work?

- DBNs are pre-trained by using the Greedy algorithm. They use a layer-by-layer approach to learn all the generative weights and the top-down approaches. The weights associated determine how all variables in one layer rely on the other variables in the above layer.
- Several steps of Gibbs sampling (for obtaining a sequence of observations approximated from a specified multivariate probability distribution when direct sampling is difficult) are run on the top two hidden layers of the network. The idea is to draw a sample from the RBM defined by the top two hidden layers.
- In the next step, we use a single pass of ancestral sampling through the rest of the model to draw a sample from the visible units.
- A single, bottom-up pass can conclude learning the values of the latent variables in every layer.
- The Greedy pre-training starts with an observed data vector in the lowest layer. Then it uses the generative weights in the opposite direction with the help of fine-tuning.

Applications

- Variational Autoencoder(VAE), a type of autoencoder is used to generate anime characters in the entertainment/gaming industry
- To recognize, cluster, and create images, video sequences, and motion-capture data.

Pros of DBNs

1. They can work with even a tiny labeled dataset.
2. DBNs provide robustness in classification. (view angle, size, position, color, etc.)

Cons of DBNS

Hardware requirements are high to process inputs.

BEST IMAGE PROCESSING TOOLS USED IN MACHINE LEARNING

Image processing is a very useful technology and the demand from the industry seems to be growing every year. Historically, image processing that uses machine learning appeared in the 1960s as an attempt to simulate the human vision system and automate the image analysis process. As the technology developed and improved, solutions for specific tasks began to appear.

The rapid acceleration of computer vision in 2010, thanks to deep learning and the emergence of open source projects and large image databases only increased the need for image processing tools.

Currently, many useful libraries and projects have been created that can help you solve image processing problems with machine learning or simply improve the processing pipelines in the computer vision projects where you use ML.

Frameworks and libraries

In theory, you could build your image processing application from scratch, just you and your computer. But in reality, it's way better to stand on the shoulders of giants and use what other people have built and extend or adjust it where needed.

This is where libraries and frameworks come in and in image processing, where creating efficient implementations is often a difficult task this is even more true.

So, let me give you my list of libraries and frameworks that you can use in your image processing projects:

OpenCV

Open-source library of computer vision and image processing algorithms.

Designed and well optimized for real-time computer vision applications.

Designed to develop open infrastructure.

Functionality:

- Basic data structures
- Image processing algorithms
- Basic algorithms for computer vision

- Input and output of images and videos
- Human face detection
- Search for stereo matches (FullHD)
- Optical flow
- Continuous integration system
- CUDA-optimized architecture
- Android version
- Java API
- Built-in performance testing system
- Cross-platform

TensorFlow

Open-source software library for machine learning.

Created to solve problems of constructing and training a neural network with the aim of automatically finding and classifying images, reaching the quality of human perception.

Functionality:

- Work on multiple parallel processors
- Calculation through multidimensional data arrays – tensors
- Optimization for tensor processors
- Immediate model iteration
- Simple debugging
- Own logging system
- Interactive log visualizer

PyTorch

Open-source machine learning platform.

Designed to speed up the development cycle from research prototyping to industrial development.

Functionality:

- Easy transition to production
- Distributed learning and performance optimization
- Rich ecosystem of tools and libraries
- Good support for major cloud platforms

- Optimization and automatic differentiation modules

Caffe

A deep learning framework focused on solving the problem of image classification and segmentation.

Functionality:

- Computation using blobs – multidimensional data arrays used in parallel computing
- Model definition and configuration optimization, no hard coding
- Easy switching between CPU and GPU
- High speed of work

EmguCV

Cross platform .Net addon for OpenCV for image processing.

Functionality:

- Working with .NET compatible languages – C #, VB, VC ++, IronPython, etc.
- Compatible with Visual Studio, Xamarin Studio and Unity
- Can run on Windows, Linux, Mac OS, iOS, and Android

VXL

A collection of open-source C ++ libraries.

Functionality:

- Load, save, and modify images in many common file formats, including very large images
- Geometry for points, curves and other elementary objects in 1, 2 or 3 dimensions
- Camera geometry
- Restoring structure from movement
- Designing a graphical user interface
- Topology
- 3D images

GDAL

Library for reading and writing raster and vector geospatial data formats.

Functionality:

- Getting information about raster data
- Convert to various formats
- Data re-projection
- Creation of mosaics from rasters
- Creation of shapefiles with raster tile index

MIScnn

Framework for 2D/3D Medical Image Segmentation.

Functionality:

- Creation of segmentation pipelines
- Preliminary processing
- Input Output
- Data increase
- Patch analysis
- Automatic assessment
- Cross validation

Tracking

JavaScript library for computer vision.

Functionality:

- Color tracking
- Face recognition
- Using modern HTML5 specifications
- Lightweight kernel (~ 7 KB)

WebGazer

Library for eye tracking.

Uses a webcam to determine the location of visitors' gaze on the page in real-time (where the person is looking).

Functionality:

- Self-calibration of the model, which observes the interaction of Internet visitors with a web page, and trains the display between eye functions and position on the screen

- Real time look prediction in most modern browsers
- Easy integration with just a few lines of JavaScript
- Ability to predict multiple views
- Work in the browser on the client side, without transferring data to the server

Marvin

A framework for working with video and images.

Functionality:

- Capture video frames
- Frame processing for video filtering
- Multi-threaded image processing
- Support for plugin integration via GUI
- Feature extraction from image components
- Generation of fractals
- Object tracking
- Motion Detection

Kornia

Library for computer vision in PyTorch.

Functionality:

- Image conversion
- Epipolar geometry
- Depth estimation
- Low-level image processing (such as filtering and edge detection directly on tensors)
- Color correction
- Feature recognition
- Image filtering
- Border recognition

Datasets

You cannot build machine learning models without the data. This is especially important in image processing applications where adding more labeled data to your

training dataset usually gets you bigger improvements than state-of-the-art network architectures or training methods.

With that in mind, let me give you a list of image datasets that you can use in your projects:

Diversity in Faces

A dataset designed to reduce the bias of algorithms.

A million labeled images of faces of people of different nationalities, ages and genders, as well as other indicators – head size, face contrast, nose length, forehead height, face proportions, etc. and their relationships to each other.

FaceForencis

Dataset for recognizing fake photos and videos.

A set of images (over half a million) created using the Face2Face, FaceSwap and DeepFakes methods.

1000 videos with faces made using each of the falsification methods.

YouTube-8M Segments

Dataset of Youtube videos, with marked up content in dynamics.

Approximately 237 thousand layouts and 1000 categories.

SketchTransfer

Dataset for training neural networks to generalize

The data consists of real-world tagged images and unlabeled sketches.

DroneVehicle

Dataset for counting objects in drone images.

15,532 RGB drone shots, there is an infrared shot for each image.

Object marking is available for both RGB and infrared images.

The dataset contains directional object boundaries and object classes.

In total, 441,642 objects were marked in the dataset for 31,064 images.

Waymo Open Dataset

Dataset for training autopilot vehicles.

Includes videos of driving with marked objects.

3,000 driving videos totaling 16.7 hours, 600,000 frames, about 25 million 3D object boundaries and 22 million 2D object boundaries.

To eliminate the problem of uniformity of videos, the recordings were made under various conditions. Video options include weather, pedestrians, lighting, cyclists, and construction sites. Diversity in the data increases the generalization ability of the models that are trained on it.

ImageNet-A

A dataset of images that the neural network cannot classify correctly. Based on the test results, the models predicted objects from the dataset with an accuracy of 3%. Contains 7.5 thousand images, the peculiarity of which is that they contain natural optical illusions.

Designed to study the stability of neural networks to ambiguous images of objects, which will help to increase the generalizing ability of models.

READY-MADE SOLUTIONS

Ready-made solutions are open-source repositories and software tools that are built to solve particular, often specialized tasks. By using those solutions you can “outsource” your model building or image processing pipeline to a tool that does it with one(ish) click or one command execution. With that in mind let me give you my list.

MobileNet

A set of computer vision algorithms optimized for mobile devices.

Functionality:

- Facial analysis
- Determination of location by environment
- Recognition directly on the smartphone
- Low latency and low power consumption

Fritz

A machine learning platform for iOS and Android developers.

Functionality:

- Runs directly on mobile devices, no data transfer
- Porting models to other frameworks and updating models in applications without having to release a new version

Computer Vision Annotation Tool

Functionality:

- Shapes for marking – rectangles, polygons, polylines, points

- No need for installation
- Ability to work together
- Automation of the marking process
- Support for various annotation scripts

3D-BoNet

Segmentation of objects in 3D images.

Solving the instance segmentation problem is 10 times computationally better than other existing approaches.

End-to-end neural network that accepts a 3D image as input, and gives out the boundary of recognized objects at the output.

Reasoning-RCNN

Object recognition from thousands of categories.

Detection of hard-to-see objects in the image.

An architecture that allows you to work on top of any existing detector.

STEAL

Detection of object boundaries on noisy data.

Increase the precision of marked object boundaries.

An additional layer to any semantic editor and loss function.

VQ-VAE-2

Generation of realistic versatile images.

Some fix for the disadvantages of using GAN for image generation.

Communication system of encoder and decoder on two levels.

CorrFlow

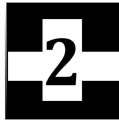
Automatic marking of videos. Distribution of markup from one image to the entire video. Based on a self-supervised model.

FUNIT

Replacing objects with others.

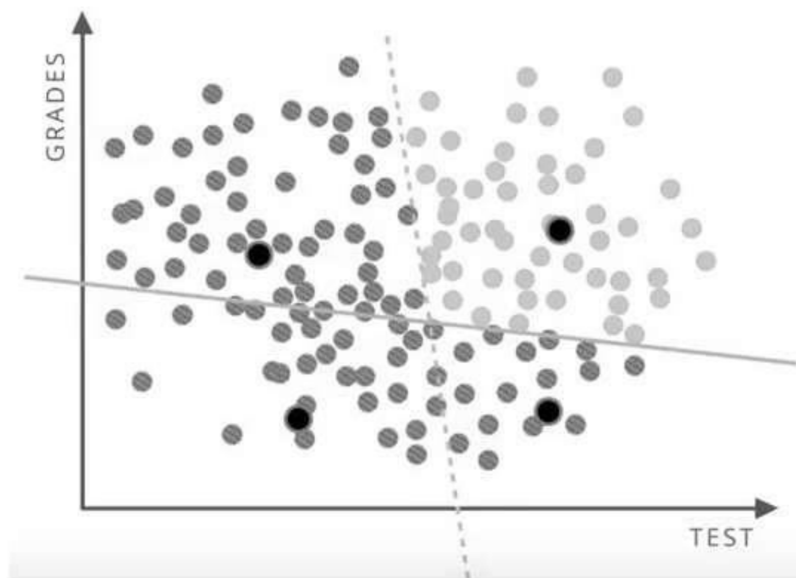
Converting object images from one class to another with a minimum amount of training data.

Based on GAN architecture.

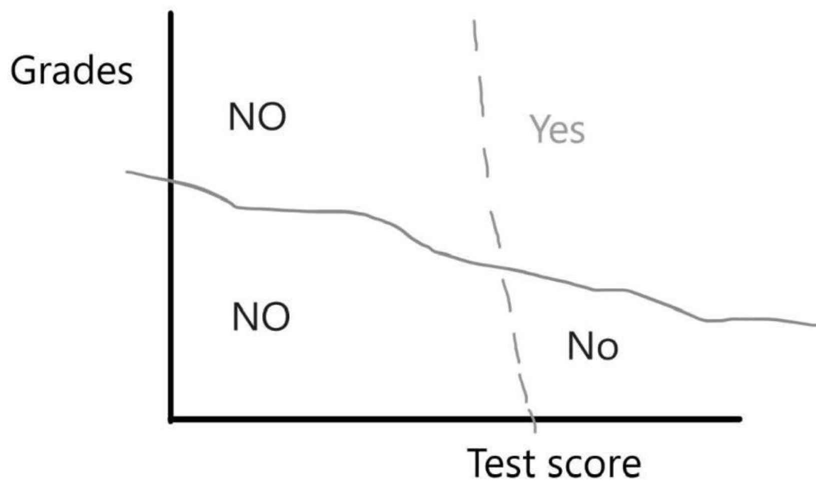


Neural Network in Deep Learning

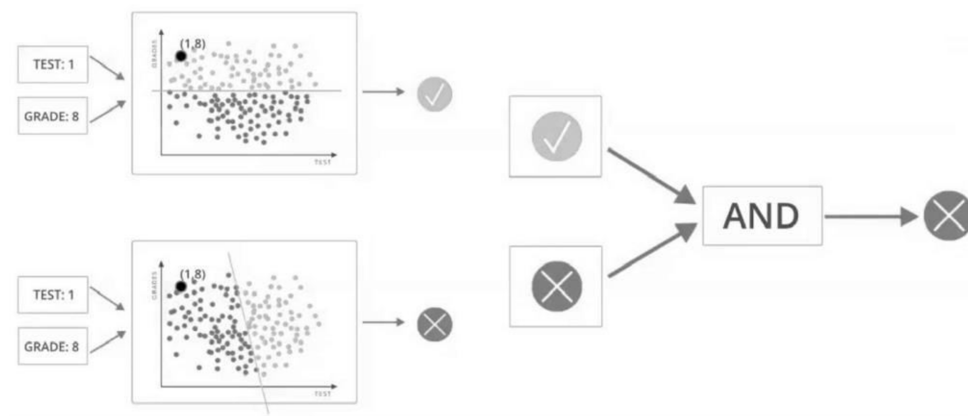
As explained above, deep learning is a sub-field of machine learning dealing with algorithms inspired by the structure and function of the brain called artificial neural networks. We will explain here how we can construct a simple neural network from the example. In the above example, Logistic regression is the technique to be used to separate data using single line. But most of the time we cannot classify the dataset using a single line with high accuracy.



How about if we separate, data points with two lines.



In this case, we say anything below blue line will be "No(not passed)" and above it will be "Yes(passed)". Similarly, we say anything on the left side will be "No(not passed)" and on the right side "Yes(passed)".

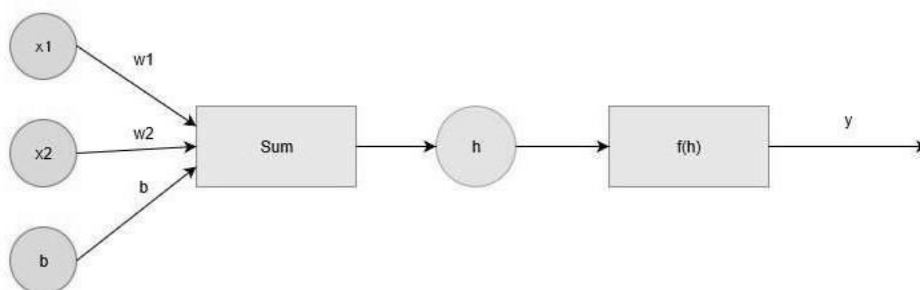


As we have neurons in nervous system, we can define each line as one neuron and connected to next layer neurons along with neurons in the same layer. In this case we have two neurons that represents the two lines. The above picture is an example of simple neural network where two neurons accept that input data and compute yes or no based on their condition and pass it to the second layer neuron to concatenate the result from previous layer. For this specific example test score

1 and grade 8 input, the output will be "Not passed" which is accurate, but in logistic regression out we may get as "passed".

To summarise this, using multiple neurons in different layers, essentially we can increase the accuracy of the model. This is the basis of neural network.

The diagram below shows a simple network. The linear combination of the weights, inputs, and bias form the input h, which passes through the activation function f(h), giving the final output, labeled y.



The good fact about this architecture, and what makes neural networks possible, is that the activation function, f(h) can be any function, not just the step function shown earlier.

For example, if you let f(h)=h, the output will be the same as the input. Now the output of the network is

$$y = \sum_i w_i x_i + b$$

This equation should be familiar to you, it's the same as the linear regression model!

Other activation functions you'll see are the logistic (often called the sigmoid), tanh, and softmax functions.

sigmoid(x)=1/(1+e"x)

The sigmoid function is bounded between 0 and 1, and as an output can be interpreted as a probability for success. It turns out, again, using a sigmoid as the activation function results in the same formulation as logistic regression.

We can finally say output of the simple neural network based on sigmoid as below:

$$y = f(h) = \text{sigmoid}(\sum_i w_i x_i + b)$$

MCCULLOCH PITTS NEURON — DEEP LEARNING BUILDING BLOCK

The fundamental block of deep learning is artificial neuron i.e. it takes a weighted aggregate of inputs, applies a function and gives an output. The very first step towards the artificial neuron was taken by Warren McCulloch and Walter Pitts in 1943 inspired by neurobiology, created a model known as McCulloch-Pitts Neuron.

MOTIVATION — BIOLOGICAL NEURON

The inspiration for the creation of an artificial neuron comes from the biological neuron.

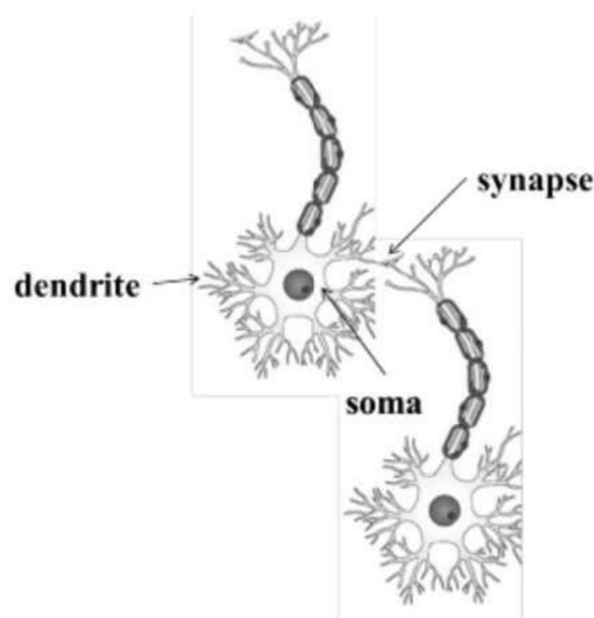


Fig : Biological Neuron — Padhai Deep Learning

In a simplistic view, neurons receive signals and produce a response. The general structure of a neuron. *Dendrites* are the transmission channels to bring inputs from another neuron or another organ. *Synapse* — Governs the strength of the interaction between the neurons, consider it like weights we use in neural networks. *Soma* — The processing unit of the neuron.

At the higher level, neuron takes a signal input through the dendrites, process it in the soma and passes the output through the axon.

McCulloch-Pitts Neuron Model

MP Neuron Model introduced by Warren McCulloch and Walter Pitts in 1943. MP neuron model is also known as linear threshold gate model.

Mathematical Model

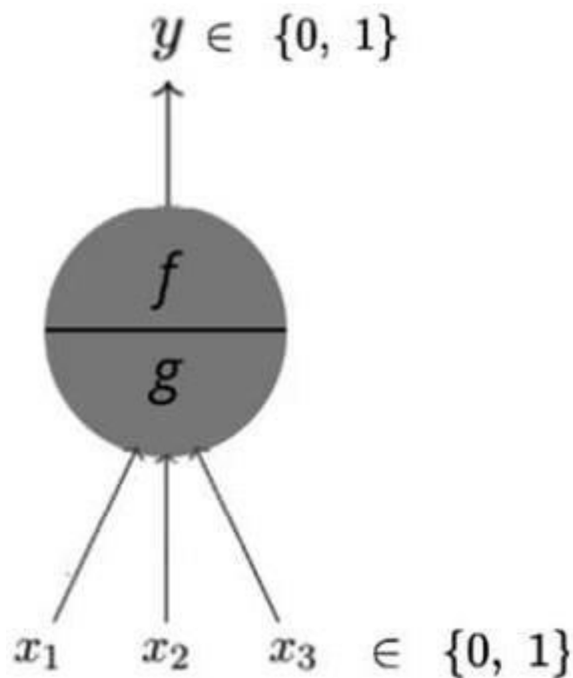


Fig : Simple representation of MP Neuron Model

The function (soma) is actually split into two parts: **g** — The aggregates the inputs to a single numeric value and the function **f** produces the output of this neuron by taking the output of the **g** as the input i,e.. a single value as its argument.

The function **f** will output the value 1 if the aggregation performed by the function **g** is greater than some threshold else it will return 0.

The inputs x1, x2, ….. xn for the MP Neuron can only take boolean values and the inputs can be inhibitory or excitatory. Inhibitory inputs can have maximum effect on the decision-making process of the model. In some cases, inhibitory inputs can influence the final outcome of the model.

$y = 0$ if any x_i is inhibitory, else

$$g(x_1, x_2, ...x_n) = g(x) = \sum_{i=1}^{n} x_i$$

$$y = f(g(x)) = 1 \text{ if } g(x) \geq b$$

$$= 0 \text{ if } g(x) < b$$

Fig: Mathematical representation

For example, I can predict my own decision of whether I would like to watch a movie in a nearby IMAX theater tomorrow or not using an MP Neuron Model.

All the inputs to the model are boolean i.e., [0,1] we can see that output from the model will also be boolean. (0 — Not going to movie, 1 — going to the movie)

Inputs for the above problem could be

- x1 — IsRainingTomorrow (Whether it's going to rain tomorrow or not)
- x2 — IsScifiMovie (I like science fiction movies)
- x3 — IsSickTomorrow (Whether I am going to be sick tomorrow or not depends on any symptoms, eg: fever)
- x4 — IsDirectedByNolan (Movie directed by Christopher Nolan or not.) etc….

In this scenario, if x3 — IsSickTomorrow is equal to 1, then the output will always be 0. If I am not feeling well on the day of the movie then no matter whoever is the actor or director of the movie, I wouldn't be going for a movie.

Loss Function

Let's take an example of buying a phone based on some features of the features in the binary format. { y — 0: Not buying a phone and y — 1: buying a phone}

For each particular phone (observation) with a certain threshold value **b**, using the MP Neuron Model, we can predict the outcome using a condition that the summation of the inputs is greater than b then the predicted value will be 1 or

else it will be 0. The loss for the particular observation will be squared difference between the Yactual and Ypredicted.

Launch (within 6 months)	0	1	1	0	0	1	0	1	1
Weight (<160g)	1	0	1	0	0	0	1	0	0
Screen size (<5.9 in)	1	0	1	0	1	0	1	0	1
dual sim	1	1	0	0	0	1	0	1	0
Internal memory (>= 64 GB, 4GB RAM)	1	1	1	1	1	1	1	1	1
NFC	0	1	1	0	1	0	1	1	1
Radio	1	0	0	1	1	1	0	0	0
Battery(>3500mAh)	0	0	0	1	0	1	0	1	0
Price > 20k	0	1	1	0	0	0	1	1	1
Like (y)	1	0	1	0	1	1	0	1	0

Fig: Buying a phone

$$\hat{y} = \sum_{i=1}^{n} x_i \geq b$$

Fig: MP Neuron Model for Buying a phone

Similarly, for all the observations, calculate the summation of the squared difference between the Yactual and Ypredicted to get the total loss of the model for a particular threshold value **b**.

$$loss = \sum_{i} (y_i - \hat{y}_i)^2$$

Fig: Loss of the Model

Learning Algorithm

The purpose of the learning algorithm is to find out the best value for the

parameter **b** so that the loss of the model will be minimum. In the ideal scenario, the loss of the model for the best value of **b** would be zero.

For n features in the data, the summation we are computing can take only values between 0 and n because all of our inputs are binary (0 or 1). 0 — indicates all the features are off and 1 — indicates all the features are on. Therefore the different values the threshold **b** can take will also vary from 0 to n. As we have only one parameter with a range of values 0 to n, we can use the brute force approach to find the best value of b.

- Initialize the b with a random integer [0,n]
- For each observation
- Find the predicted outcome, by using the formula

Calculate the summation of inputs and check whether its greater than or equal to b. If its greater than or equal to b, then the predicted outcome will be 1 or else it will be 0.

- After finding the predicting outcome compute the loss for each observation.
- Finally, compute the total loss of the model by summing up all the individual losses.
- Similarly, we can iterate over all the possible values of b and find the total loss of the model. Then we can choose the value of b, such that the loss is minimum.

MODEL EVALUATION

After finding the best threshold value **b** from the learning algorithm, we can evaluate the model on the test data by comparing the predicted outcome and the actual outcome.

	Training data										Test data			
Launch (within 6 months)	0	1	1	0	0	1	0	1	1	0	1	0	0	1
Weight (<160g)	1	0	1	0	0	0	1	0	0	1	0	1	1	1
Screen size (<5.9 in)	1	0	1	0	1	0	1	0	1	0	0	1	1	1
dual sim	1	1	0	0	0	1	0	1	0	0	0	1	0	0
Internal memory (>= 64 GB, 4GB RAM)	1	1	1	1	1	1	1	1	1	0	1	0	0	0
NFC	0	1	1	0	1	0	1	1	1	0	0	0	1	0
Radio	1	0	0	1	1	1	0	0	0	0	1	1	1	0
Battery(>3500mAh)	0	0	0	1	0	1	0	1	0	0	1	1	1	0
Price > 20k	0	1	1	0	0	0	1	1	1	0	0	0	1	0
Like? (y)	1	1	1	0	0	1	1	1	0	0	0	1	0	0
predicted	1	1	0	1	1	1	1	0	0	0	0	1	1	0

Fig: Predictions on the test data for b = 5

For evaluation, we will calculate the accuracy score of the model.

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Fig: Accuracy metric

For the above-shown test data, the accuracy of the MP neuron model = 75%.

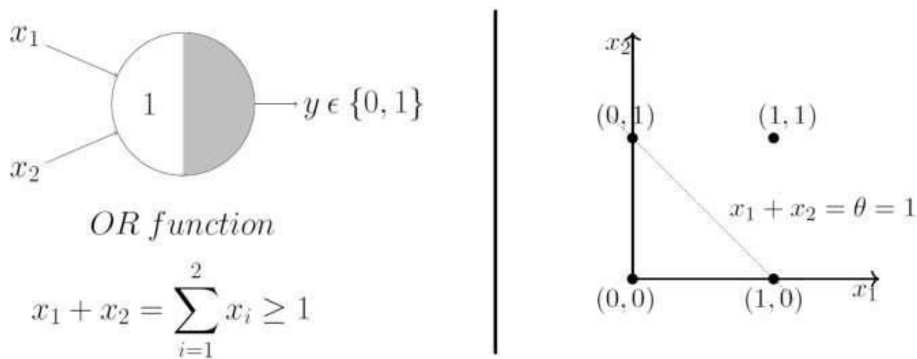
GEOMETRIC INTERPRETATION OF M-P NEURON

This is the best part of the post according to me. Lets start with the OR function.

OR Function

We already discussed that the OR function's thresholding parameter ***theta*** is 1, for obvious reasons. The inputs are obviously boolean, so only 4 combinations are possible — (0,0), (0,1), (1,0) and (1,1). Now plotting them on a 2D graph and making use of the OR function's aggregation equation

i.e., ***x_1 + x_2* e” 1** using which we can draw the decision boundary as shown in the graph below. Mind you again, this is not a real number graph.

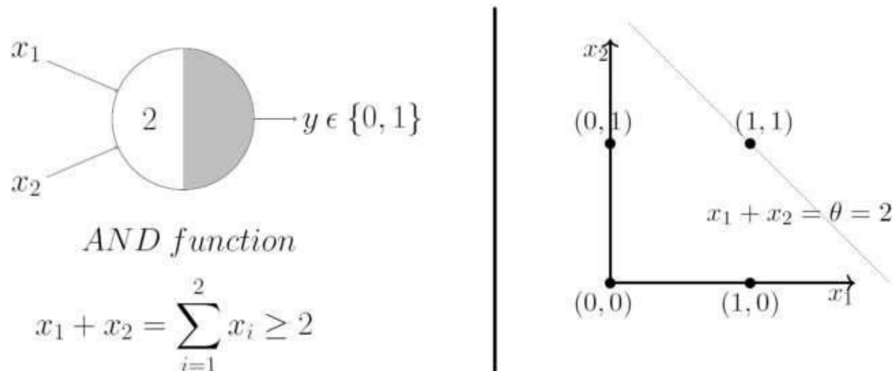


We just used the aggregation equation i.e., ***x_1 + x_2* =1** to graphically show that all those inputs whose output when passed through the OR function M-P neuron lie ON or ABOVE that line and all the input points that lie BELOW that line are going to output 0.

Voila!! The M-P neuron just learnt a linear decision boundary! The M-P neuron is splitting the input sets into two classes — positive and negative. Positive ones (which output 1) are those that lie ON or ABOVE the decision boundary and negative ones (which output 0) are those that lie BELOW the decision

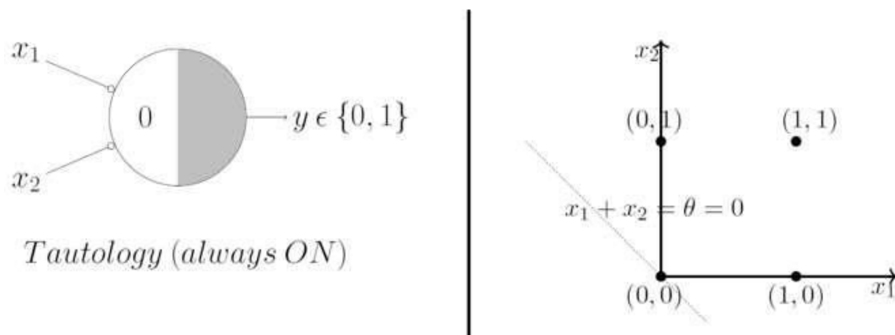
boundary. Lets convince ourselves that the M-P unit is doing the same for all the boolean functions by looking at more examples (if it is not already clear from the math).

AND Function



In this case, the decision boundary equation is ***x_1 + x_2* =2**. Here, all the input points that lie ON or ABOVE, just (1,1), output 1 when passed through the AND function M-P neuron. It fits! The decision boundary works!

Tautology



Too easy, right?

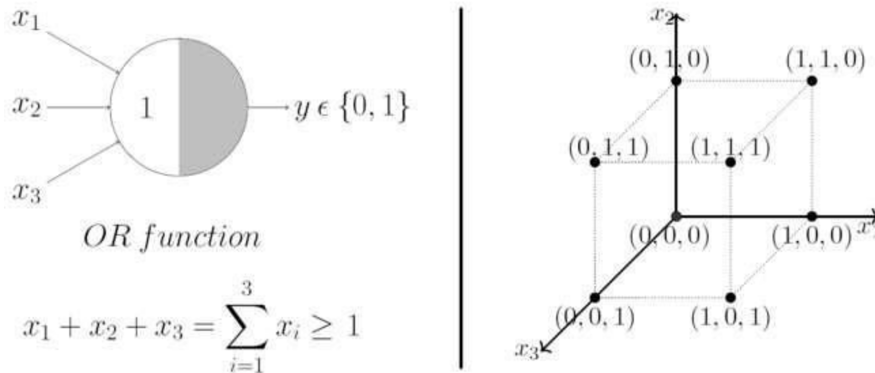
I think you get it by now but what if we have more than 2 inputs?

OR Function With 3 Inputs

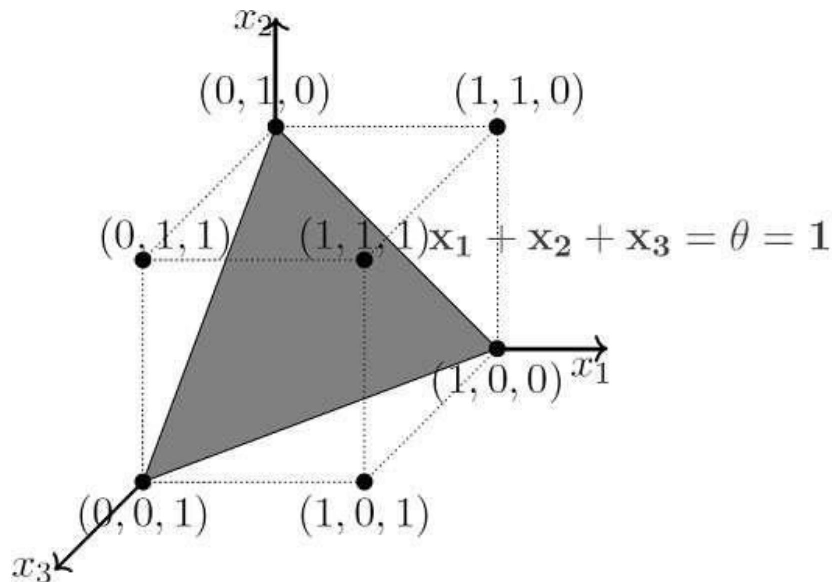
Lets just generalize this by looking at a 3 input OR function M-P unit. In this case, the possible inputs are 8 points — (0,0,0), (0,0,1), (0,1,0), (1,0,0), (1,0,1),… you got the point(s). We can map these on a 3D graph and this time we draw a decision boundary in 3 dimensions.

"Is it a bird? Is it a plane?"

Yes, it is a PLANE!



The plane that satisfies the decision boundary equation ***x_1 + x_2 + x_3 = 1*** is shown below:



Take your time and convince yourself by looking at the above plot that all the points that lie ON or ABOVE that plane (positive half space) will result in output 1 when passed through the OR function M-P unit and all the points that lie BELOW that plane (negative half space) will result in output 0.

Just by hand coding a thresholding parameter, M-P neuron is able to conveniently represent the boolean functions which are linearly separable.

Linear separability (for boolean functions): There exists a line (plane) such that all inputs which produce a 1 lie on one side of the line (plane) and all inputs which produce a 0 lie on other side of the line (plane).

Limitations Of M-P Neuron

- What about non-boolean (say, real) inputs?
- Do we always need to hand code the threshold?
- Are all inputs equal? What if we want to assign more importance to some inputs?
- What about functions which are not linearly separable? Say XOR function.

I hope it is now clear why we are not using the M-P neuron today. Overcoming the limitations of the M-P neuron, Frank Rosenblatt, an American psychologist, proposed the classical perception model, the mighty *artificial neuron*, in 1958. It is more generalized computational model than the McCulloch-Pitts neuron where weights and thresholds can be learnt over time.

More on *perceptron* and how it learns the weights and thresholds etc. in my future posts.

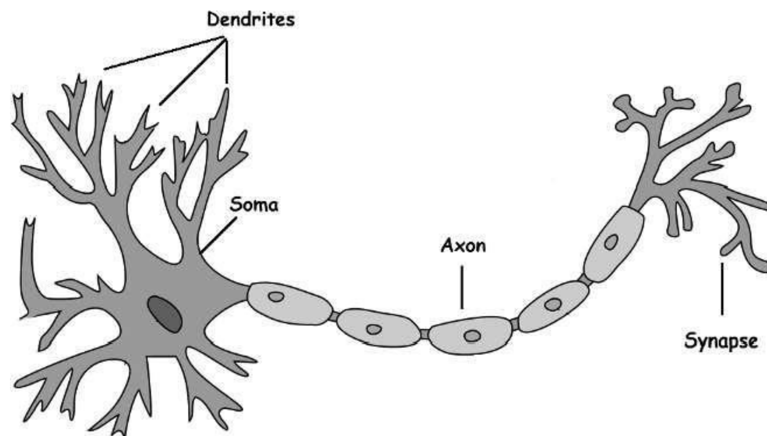
BIOLOGICAL NEURONS: AN OVERLY SIMPLIFIED ILLUSTRATION

Dendrite: Receives signals from other neurons

Soma: Processes the information

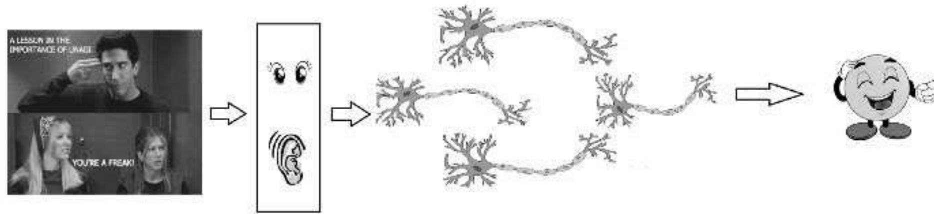
Axon: Transmits the output of this neuron

Synapse: Point of connection to other neurons



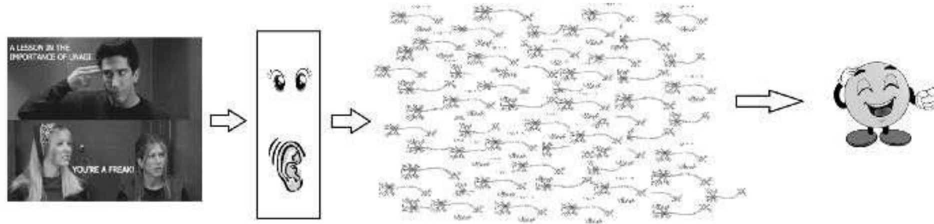
Basically, a neuron takes an input signal (dendrite), processes it like the CPU (soma), passes the output through a cable like structure to other connected neurons (axon to synapse to other neuron's dendrite). Now, this might be biologically inaccurate as there is a lot more going on out there but on a higher level, this is what is going on with a neuron in our brain — takes an input, processes it, throws out an output.

Our sense organs interact with the outer world and send the visual and sound information to the neurons. Let's say you are watching Friends. Now the information your brain receives is taken in by the "laugh or not" set of neurons that will help you make a decision on whether to laugh or not. Each neuron gets fired/activated only when its respective criteria (more on this later) is met like shown below.



Not real.

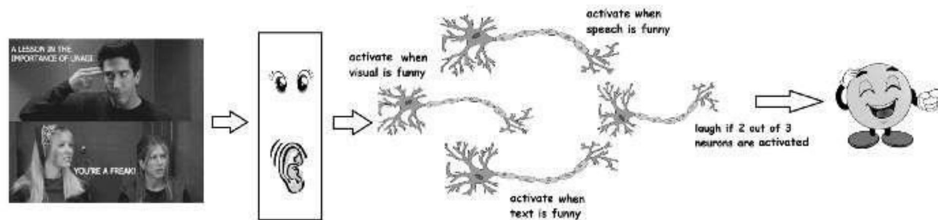
Of course, this is not entirely true. In reality, it is not just a couple of neurons which would do the decision making. There is a massively parallel interconnected network of 10^{11} neurons (100 billion) in our brain and their connections are not as simple as I showed you above. It might look something like this:



Still not real but closer.

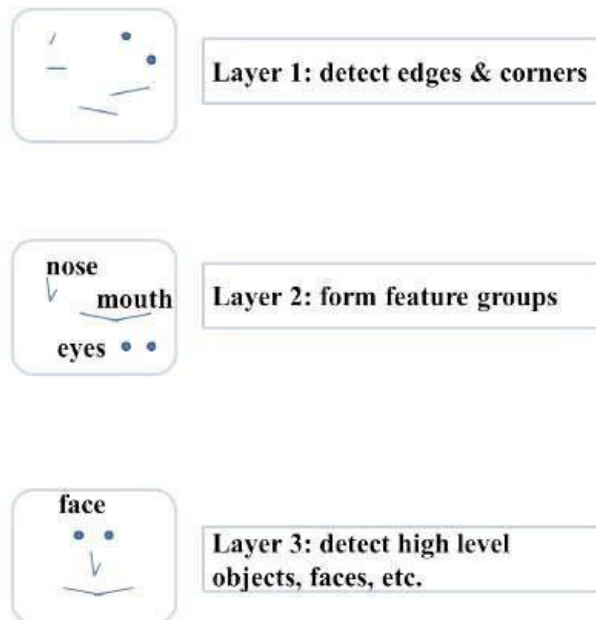
Now the sense organs pass the information to the first/lowest layer of neurons to process it. And the output of the processes is passed on to the next layers in a hierarchical manner, some of the neurons will fire and some won't and this process goes on until it results in a final response — in this case, laughter.

This massively parallel network also ensures that there is a division of work. Each neuron only fires when its intended criteria is met i.e., a neuron may perform a certain role to a certain stimulus, as shown below.



Division of work

It is believed that neurons are arranged in a hierarchical fashion (however, many credible alternatives with experimental support are proposed by the scientists) and each layer has its own role and responsibility. To detect a face, the brain could be relying on the entire network and not on a single layer.

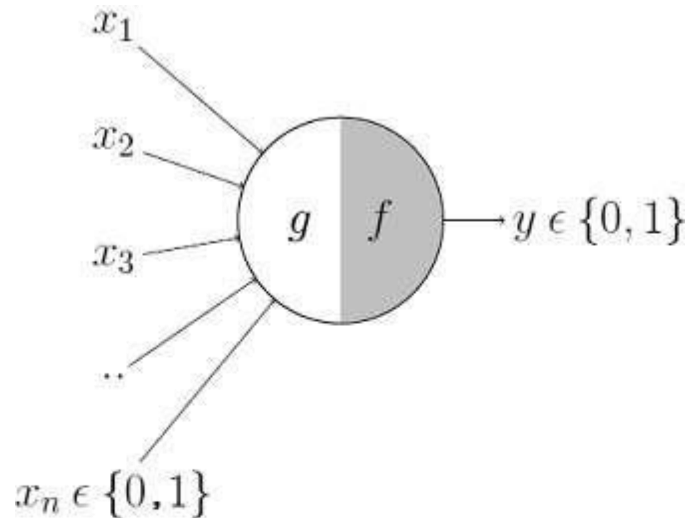


Sample illustration of hierarchical processing. **Credits:** Mitesh M. Khapra's lecture slides. Now that we have established how a biological neuron works, lets look at what McCulloch and Pitts had to offer.

Note: My understanding of how the brain works is very very very limited. The above illustrations are overly simplified.

McCulloch-Pitts Neuron

The first computational model of a neuron was proposed by Warren MuCulloch (neuroscientist) and Walter Pitts (logician) in 1943.



This is where it all began..

It may be divided into 2 parts. The first part, ***g*** takes an input (ahem dendrite ahem), performs an aggregation and based on the aggregated value the second part, ***f*** makes a decision.

Lets suppose that I want to predict my own decision, whether to watch a random football game or not on TV. The inputs are all boolean i.e., {0,1} and my output variable is also boolean {0: Will watch it, 1: Won't watch it}.

- So, ***x_1*** could be *isPremierLeagueOn* (I like Premier League more)
- ***x_2*** could be *isItAFriendlyGame* (I tend to care less about the friendlies)
- ***x_3*** could be *isNotHome* (Can't watch it when I'm running errands. Can I?)
- ***x_4*** could be *isManUnitedPlaying* (I am a big Man United fan. GGMU!) and so on.

These inputs can either be *excitatory* or *inhibitory*. Inhibitory inputs are those that have maximum effect on the decision making irrespective of other inputs i.e., if ***x_3*** is 1 (not home) then my output will always be 0 i.e., the neuron will never fire, so ***x_3*** is an inhibitory input.

Excitatory inputs are NOT the ones that will make the neuron fire on their own but they might fire it when combined together. Formally, this is what is going on:

$$g(x_1, x_2, x_3, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

$$y = f(g(\mathbf{x})) = 1 \quad if \quad g(\mathbf{x}) \geq \theta$$
$$= 0 \quad if \quad g(\mathbf{x}) < \theta$$

We can see that ***g*(x)** is just doing a sum of the inputs — a simple aggregation.

And ***theta*** here is called thresholding parameter.

For example, if I always watch the game when the sum turns out to be 2 or more, the ***theta*** is 2 here.

This is called the Thresholding Logic.

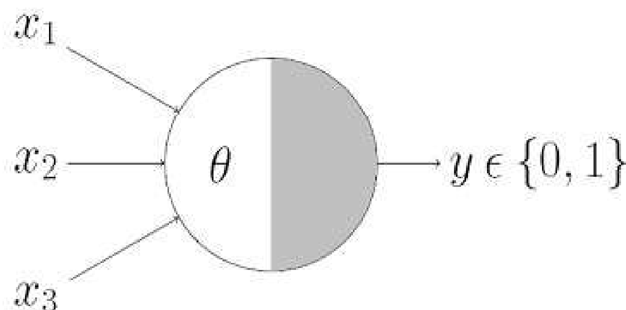
Boolean Functions Using M-P Neuron

So far we have seen how the M-P neuron works. Now lets look at how this very neuron can be used to represent a few boolean functions.

Mind you that our inputs are all boolean and the output is also boolean so essentially, the neuron is just trying to learn a boolean function.

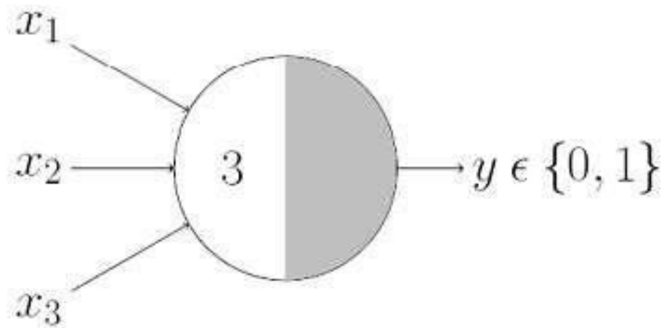
A lot of boolean decision problems can be cast into this, based on appropriate input variables— like whether to continue reading this post, whether to watch Friends after reading this post etc. can be represented by the M-P neuron.

M-P Neuron: A Concise Representation



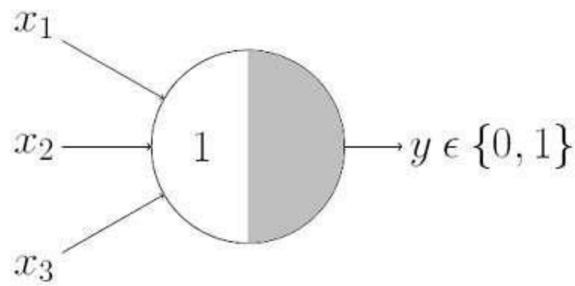
This representation just denotes that, for the boolean inputs ***x_1***, ***x_2*** and ***x_3*** if the ***g*(x)** i.e., **sum e" theta**, the neuron will fire otherwise, it won't.

AND Function



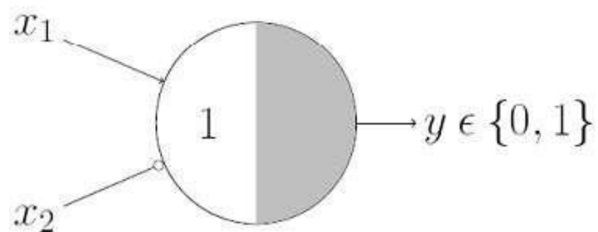
An AND function neuron would only fire when ALL the inputs are ON i.e., ***g*(x)** e” 3 here.

OR Function



I believe this is self explanatory as we know that an OR function neuron would fire if ANY of the inputs is ON i.e., ***g*(x)** · 1 here.

A Function With An Inhibitory Input



$$x_1\ AND\ !x_2{}^*$$

Now this might look like a tricky one but it's really not. Here, we have an inhibitory input i.e., ***x_2*** so whenever ***x_2*** is 1, the output will be 0.

Keeping that in mind, we know that ***x_1 AND !x_2*** would output 1 only when ***x_1*** is 1 and ***x_2*** is 0 so it is obvious that the threshold parameter should be 1.

Lets verify that, the ***g*(x)** i.e., ***x_1 + x_2*** would be e" 1 in only 3 cases:

Case 1: when ***x_1*** is 1 and ***x_2*** is 0

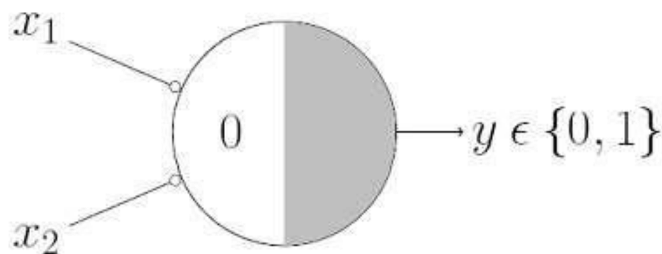
Case 2: when ***x_1*** is 1 and ***x_2*** is 1

Case 3: when ***x_1*** is 0 and ***x_2*** is 1

But in both Case 2 and Case 3, we know that the output will be 0 because ***x_2*** is 1 in both of them, thanks to the inhibition.

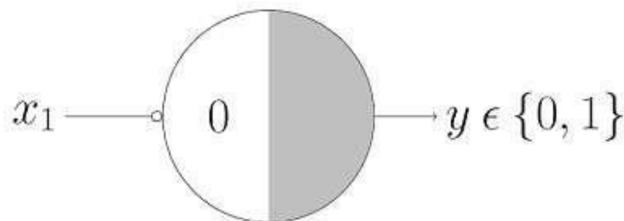
And we also know that ***x_1 AND !x_2*** would output 1 for Case 1 (above) so our thresholding parameter holds good for the given function.

NOR Function



For a NOR neuron to fire, we want ALL the inputs to be 0 so the thresholding parameter should also be 0 and we take them all as inhibitory input.

NOT Function



For a NOT neuron, 1 outputs 0 and 0 outputs 1. So we take the input as an inhibitory input and set the thresholding parameter to 0. It works!

Can any boolean function be represented using the M-P neuron? Before you answer that, lets understand what M-P neuron is doing geometrically.

OTHER IMPORTANT CONCEPTS OF NEURAL NETWORKS

Training

Weights start out as random values, and as the neural network learns more about what kind of input data leads to a student being accepted into a university(above example), the network adjusts the weights based on any errors in categorization that the previous weights resulted in.

This is called **training** the neural network. Once we have the trained network, we can use it for predicting the output for the similar input.

Error

This very important concept to define how well a network performing during the training.

In the training phase of the network, it make use of error value to adjust the weights so that it can get reduced error at each step.

The goal of the training phase to minimize the error

Mean Squared Error is one of the popular error function. it is a modified version **Sum Squared Error.**

$$SSE = \sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2$$

$$MSE = \frac{1}{n} \times SSE$$

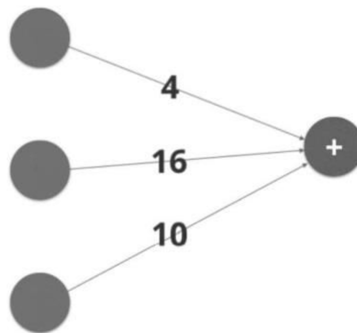
Or we can write MSE as:

$$E = \frac{1}{2m} \sum_\mu (y^\mu - \hat{y}^\mu)^2$$

Forward Propagation

By propagating values from the first layer (the input layer) through all the mathematical functions represented by each node, the network outputs a value. This process is called a **forward pass**.

Input layer Output layer



Code for implementing the forward propagation using numpy:

```
import numpy as np
def sigmoid(x):
   """
   Calculate  sigmoid
   """
   return  1/(1+np.exp(-x))
# Network size
N_input = 4
N_hidden = 3
N_output = 2
np.random.seed(42)
# Make some fake data
X = np.random.randn(4)
weights_input_to_hidden = np.random.normal(0, scale=0.1,
size=(N_input, N_hidden))
weights_hidden_to_output = np.random.normal(0, scale=0.1,
size=(N_hidden, N_output))

# TODO: Make a forward pass through the network
hidden_layer_in = np.dot(X, weights_input_to_hidden)
hidden_layer_out = sigmoid(hidden_layer_in)
print('Hidden-layer Output:')
print(hidden_layer_out)
output_layer_in = np.dot(hidden_layer_out,
weights_hidden_to_output)
output_layer_out = sigmoid(output_layer_in)
```

```
print('Output-layer Output:')
print(output_layer_out)
```

Gradient Descent

Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost).Gradient descent is best used when the parameters cannot be calculated analytically (e.g. using linear algebra) and must be searched for by an optimization algorithm.Gradient descent is used to find the minimum error by minimizing a "cost" function.

In the university example(explained it in the neural network section), the correct lines to divide the dataset is already defined. How does we find the correct line? As we know, weights are adjusted during the training process. Adjusting the weight will enable each neuron to correctly divide the dataset with given dataset.

To figure out how we're going to find these weights, start by thinking about the goal. We want the network to make predictions as close as possible to the real values. To measure this, we need a metric of how wrong the predictions are, the **error**. A common metric is the sum of the squared errors (SSE):

$$E = \frac{1}{2} \sum_{\mu} \sum_{j} \left[y_j^{\mu} - \hat{y}_j^{\mu} \right]^2$$

where *y^* is the prediction and *y* is the true value, and you take the sum over all output units *j* and another sum over all data points *ì*.

The SSE is a good choice for a few reasons. The square ensures the error is always positive and larger errors are penalized more than smaller errors. Also, it makes the math nice, always a plus.

Remember that the output of a neural network, the prediction, depends on the weights

$$\hat{y}_j^{\mu} = f\left(\sum_i w_{ij} x_i^{\mu} \right)$$

and accordingly the error depends on the weights

$$E = \frac{1}{2} \sum_{\mu} \sum_{j} \left[y_j^{\mu} - f\left(\sum_i w_{ij} x_i^{\mu} \right) \right]^2$$

We want the network's prediction error to be as small as possible and the weights are the knobs we can use to make that happen. Our goal is to find weights wij that minimize the squared error E. To do this with a neural network, typically we use gradient descent. With gradient descent, we take multiple small steps towards our goal. In this case, we want to change the weights in steps that reduce the error. Continuing the analogy, the error is our mountain and we want to get to the bottom. Since the fastest way down a mountain is in the steepest direction, the steps taken should be in the direction that minimizes the error the most.

Back Propagation

In neural networks, you forward propagate to get the output and compare it with the real value to get the error. Now, to minimise the error, you propagate backwards by finding the derivative of error with respect to each weight and then subtracting this value from the weight value. This is called back propagation.

Before, we saw how to update weights with gradient descent. The back propagation algorithm is just an extension of that, using the chain rule to find the error with the respect to the weights connecting the input layer to the hidden layer (for a two layer network).

Here is the back propagation algorithm from Udacity:

- Set the weight steps for each layer to zero
  - The input to hidden weights $\Delta w_{ij} = 0$
  - The hidden to output weights $\Delta W_j = 0$
- For each record in the training data:
  - Make a forward pass through the network, calculating the output $\hat{y}$
  - Calculate the error gradient in the output unit, $\delta^o = (y - \hat{y})f'(z)$ where $z = \sum_j W_j a_j$, the input to the output unit.
  - Propagate the errors to the hidden layer $\delta_j^h = \delta^o W_j f'(h_j)$
  - Update the weight steps,:
    - $\Delta W_j = \Delta W_j + \delta^o a_j$
    - $\Delta w_{ij} = \Delta w_{ij} + \delta_j^h a_i$
- Update the weights, where η is the learning rate and m is the number of records:
  - $W_j = W_j + \eta \Delta W_j / m$
  - $w_{ij} = w_{ij} + \eta \Delta w_{ij} / m$
- Repeat for e epochs.

Code for implementing the propagation in numpy:

```
import numpy as np
from data_prep import features, targets, features_test,
targets_test
```

```
np.random.seed(21)
def sigmoid(x):
   """
   Calculate  sigmoid
   """
   return 1 / (1 + np.exp(-x))

# Hyperparameters
n_hidden = 2  # number of hidden units
epochs = 900
learnrate = 0.005
n_records, n_features = features.shape
last_loss = None
# Initialize weights
weights_input_hidden = np.random.normal(scale=1 / n_features
** .5,
                                         size=(n_features,
n_hidden))
weights_hidden_output = np.random.normal(scale=1 / n_features
** .5,
                                         size=n_hidden)
for e in range(epochs):
  del_w_input_hidden = np.zeros(weights_input_hidden.shape)
                        del_w_hidden_output        =
np.zeros(weights_hidden_output.shape)
   for x, y in zip(features.values, targets):
       ## Forward pass ##
       # TODO: Calculate the output
       hidden_input  =  np.dot(x,  weights_input_hidden)
       hidden_output = sigmoid(hidden_input)
       output  =  sigmoid(np.dot(hidden_output,
                                 weights_hidden_output))
       ## Backward pass ##
       # TODO: Calculate the network's prediction error
       error = y - output
       # TODO: Calculate error term for the output unit
       output_error_term = error * output * (1 - output)
       ## propagate errors to hidden layer
      # TODO: Calculate the hidden layer's contribution to
the error
```

```
            hidden_error = np.dot(output_error_term,
weights_hidden_output)
         # TODO: Calculate the error term for the hidden
layer
        hidden_error_term = hidden_error * hidden_output *
(1 - hidden_output)
        # TODO: Update the change in weights
            del_w_hidden_output += output_error_term *
hidden_output
        del_w_input_hidden += hidden_error_term * x[:, None]
    # TODO: Update weights
    weights_input_hidden += learnrate * del_w_input_hidden
/ n_records
   weights_hidden_output += learnrate * del_w_hidden_output
/ n_records
   # Printing out the mean square error on the training set
   if e % (epochs / 10) == 0:
             hidden_output = sigmoid(np.dot(x,
weights_input_hidden))
        out = sigmoid(np.dot(hidden_output,
                              weights_hidden_output))
        loss = np.mean((out - targets) ** 2)
        if last_loss and last_loss < loss:
             print("Train loss: ", loss, " WARNING - Loss
Increasing")
        else:
            print("Train loss: ", loss)
        last_loss = loss
# Calculate accuracy on test data
hidden = sigmoid(np.dot(features_test, weights_input_hidden))
out = sigmoid(np.dot(hidden, weights_hidden_output))
predictions = out > 0.5
accuracy = np.mean(predictions == targets_test)
print("Prediction  accuracy: {:.3f}".format(accuracy))
```

Regularisation

Regularisation is the technique used to solve the over-fitting problem. Over-fitting happens when model is biased to one type of datset. There are different types of regularisation techniques, I think the mostly used regularisation is dropout.

Dropout *is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data. It is a very efficient way of performing model averaging with neural networks.[1] The term "dropout" refers to dropping out units (both hidden and visible) in a neural network.*

During the training, randomly selected neurons are not considered. We can set the number of neurons for the dropout. Their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. I think best practise is to remove 20 % of neurons.

SIGMOID NEURON — BUILDING BLOCK OF DEEP NEURAL NETWORKS

The building block of the deep neural networks is called the sigmoid neuron. Sigmoid neurons are similar to perceptrons, but they are slightly modified such that the output from the sigmoid neuron is much smoother than the step functional output from perceptron.

In this post, we will talk about the motivation behind the creation of sigmoid neuron and working of the sigmoid neuron model.

This is the 1st part in the two-part series discussing the working of sigmoid neuron and it's learning algorithm:

1 | Sigmoid Neuron — Building Block of Deep Neural Networks

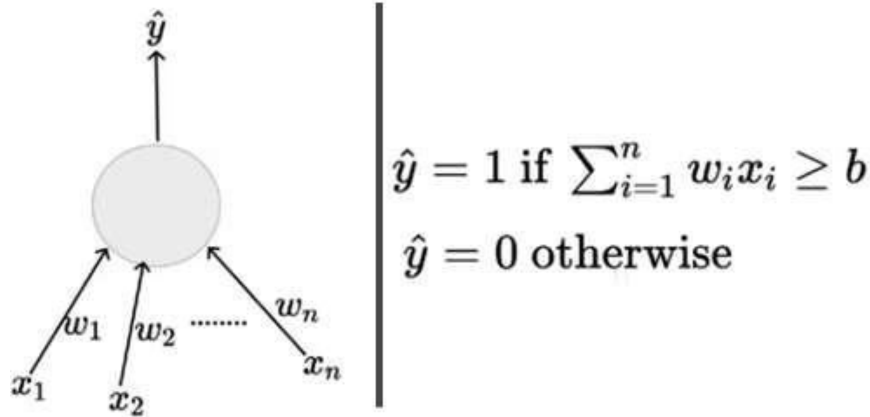
2 | Sigmoid Neuron Learning Algorithm Explained With Math

Why Sigmoid Neuron

Before we go into the working of a sigmoid neuron, let's talk about the perceptron model and its limitations in brief.

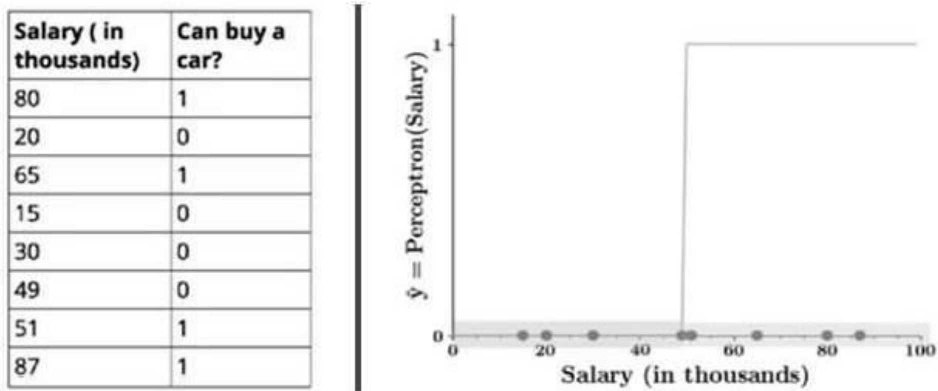
Perceptron model takes several real-valued inputs and gives a single binary output. In the perceptron model, every input xi has weight wi associated with it. The weights indicate the importance of the input in the decision-making process. The model output is decided by a threshold **W** if the weighted sum of the inputs is greater than threshold **W** output will be 1 else output will be 0. In other words, the model will fire if the weighted sum is greater than the threshold.

From the mathematical representation, we might say that the thresholding logic used by the perceptron is very harsh. Let's see the harsh thresholding logic with an example.



Perceptron (Left) & Mathematical Representation (Right)

Consider the decision making process of a person, whether he/she would like to purchase a car or not based on only one input X1—Salary and by setting the threshold **b**(**W**) = -10 and the weight **W** = 0.2. The output from the perceptron model will look like in the figure shown below.



Salary (in thousands)	Can buy a car?
80	1
20	0
65	1
15	0
30	0
49	0
51	1
87	1

Data (Left) & Graphical Representation of Output(Right)

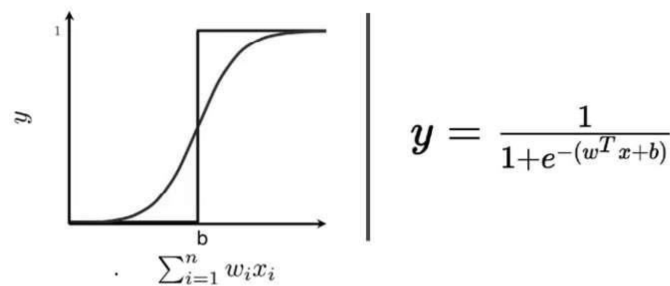
Red points indicates that a person would not buy a car and green points indicate that person would like to buy a car. Isn't it a bit odd that a person with 50.1K will buy a car but someone with a 49.9K will not buy a car? The small change in the input to a perceptron can sometimes cause the output to completely flip, say from 0 to 1. This behavior is not a characteristic of the specific problem we choose or the specific weight and the threshold we choose. It is a characteristic of the perceptron neuron itself which behaves like a step function. We can overcome

this problem by introducing a new type of artificial neuron called a *sigmoid* neuron.

SIGMOID NEURON

Can we have a smoother (not so harsh) function?

Introducing sigmoid neurons where the output function is much smoother than the step function. In the sigmoid neuron, a small change in the input only causes a small change in the output as opposed to the stepped output. There are many functions with the characteristic of an " **S**" shaped curve known as sigmoid functions. The most commonly used function is the logistic function.



Sigmoid Neuron Representation (logistic function)

We no longer see a sharp transition at the threshold **b**. The output from the sigmoid neuron is not 0 or 1. Instead, it is a real value between 0–1 which can be interpreted as a probability.

REGRESSION AND CLASSIFICATION

The inputs to the sigmoid neuron can be real numbers unlike the boolean inputs in MP Neuron and the output will also be a real number between 0–1. In the sigmoid neuron, we are trying to regress the relationship between **X** and **Y** in terms of probability. Even though the output is between 0–1, we can still use the sigmoid function for binary classification tasks by choosing some threshold.

Learning Algorithm

An algorithm for learning the parameters **w** and **b** of the sigmoid neuron model by using the gradient descent algorithm.

Find w and b such that:

$$\underset{w,b}{\text{minimize}}\, \mathscr{L}(w, b) = \sum_{i=1}^{N} (y_i - f(x_i))^2$$

Minimize the Squared Error Loss

The objective of the learning algorithm is to determine the best possible values for the parameters, such that the overall loss (squared error loss) of the model is minimized as much as possible. Here goes the learning algorithm:

Initialise w, b

Iterate over data:

$compute\ \hat{y}$

$compute\ \mathscr{L}(w, b)$

$$w_{t+1} = w_t - \eta \Delta w_t$$

$$b_{t+1} = b_t - \eta \Delta b_t$$

till satisfied

Sigmoid Learning Algorithm

We initialize **w** and **b** randomly. We then iterate over all the observations in the data, for each observation find the corresponding predicted outcome using the sigmoid function and compute the squared error loss. Based on the loss value, we will update the weights such that the overall loss of the model at the new parameters will be **less than the current loss** of the model.

$$\mathscr{L}(w, b) > \mathscr{L}(w + \eta \Delta w, b + \eta \Delta b)$$

Loss Optimization

We will keep doing the update operation until we are satisfied. Till satisfied could mean any of the following:

- The overall loss of the model becomes zero.

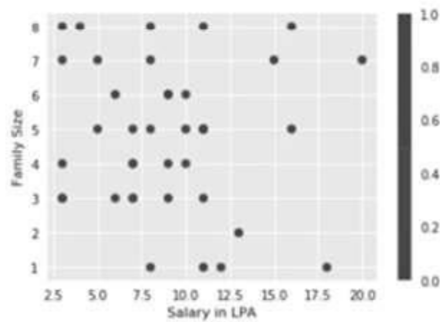
- The overall loss of the model becomes a very small value closer to zero.
- Iterating for a fixed number of passes based on computational capacity.

Can It Handle Non-Linear Data?

One of the limitations of the perceptron model is that the learning algorithm works only if the data is linearly separable. That means that the positive points will lie on one side of the boundary and negative points lie another side of the boundary. Can sigmoid neuron handle non-linearly separable data?.

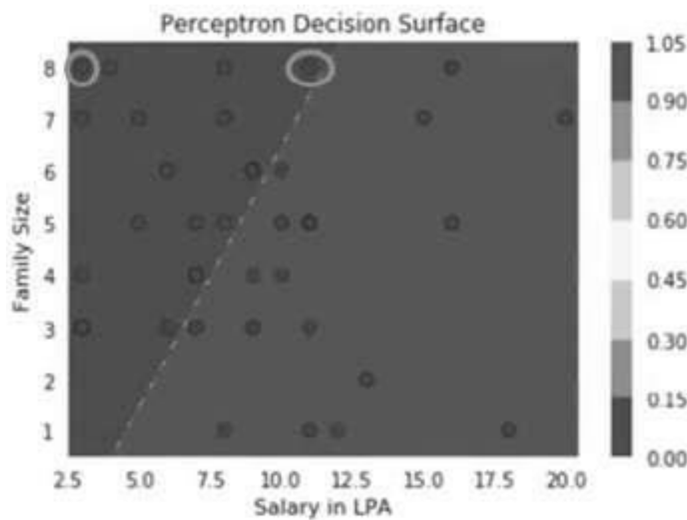
Let’s take an example of whether a person is going to buy a car or not based on two inputs, X — Salary in Lakhs Per Annum (LPA) and X — Size of the family. I am assuming that there is a relationship between **X** and **Y**, it is approximated using the sigmoid function.

	Salary in LPA	Family Size	Buys Car?
0	11	8	1
1	20	7	1
2	4	8	0
3	8	7	0
4	11	5	1



Input Data(Left) & Scatter Plot of Data(Right)

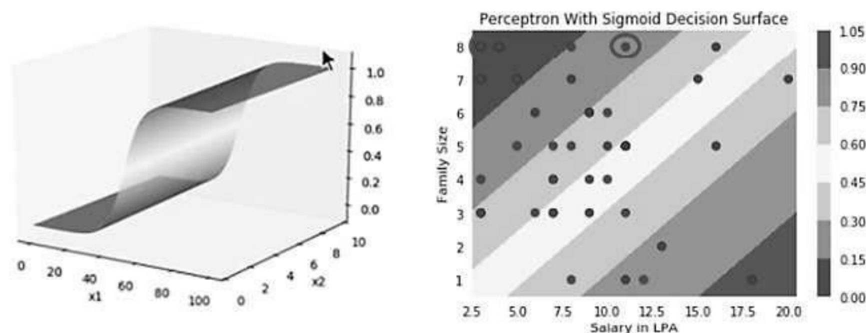
The red points indicate that the output is 0 and green points indicate that it is 1.



As we can see from the figure, there is no line or a linear boundary that can effectively separate red and green points. If we train a perceptron on this data, the learning algorithm will **never converge** because the data is not linearly separable. Instead of going for convergence, I will run the model for a certain number of iterations so that the errors will be minimized as much as possible.

Perceptron Decision boundary for fixed iterations

From the perceptron decision boundary, we can see that the perceptron doesn't distinguish between the points that lie close to the boundary and the points lie far inside because of the harsh thresholding logic. But in the real world scenario, we would expect a person who is sitting on the fence of the boundary can go either way, unlike the person who is way inside from the decision boundary. Let's see how sigmoid neuron will handle this non-linearly separable data. Once I fit our two-dimensional data using the sigmoid neuron, I will be able to generate the 3D contour plot shown below to represent the decision boundary for all the observations.



Sigmoid Neuron Decision Boundary (Left) & Top View of Decision Boundary (Right)

For comparison, let's take the same two observations and see what will be predicted outcome from the sigmoid neuron for these observations. As you can see the predicted value for the observation present in the far left of the plot is zero (present in the dark red region) and the predicted value of another observation is around 0.35 i.e. there is a 35% chance that the person might buy a car. Unlike the rigid output from the perceptron, now we a smooth and continuous output between 0–1 which can be interpreted as a probability.

STILL DOES NOT COMPLETELY SOLVE OUR PROBLEM FOR NON-LINEAR DATA.

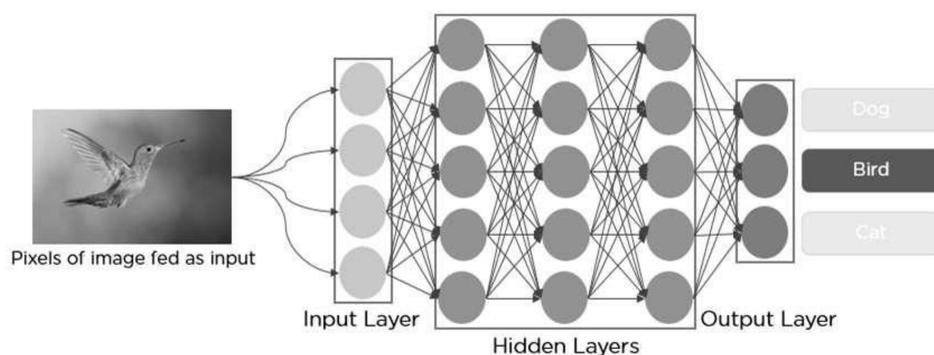
Although we have introduced the non-linear sigmoid neuron function, it is still not able to effectively separate red points from green points. The important point

is that from a rigid decision boundary in perceptron, we have taken our first step in the direction of creating a decision boundary that works well for non-linearly separable data. Hence the sigmoid neuron is the building block of deep neural network eventually we have to use a network of neurons to helps us out to create a "*perfect*" decision boundary.

BASIC INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORK IN DEEP LEARNING

Hidden layers have ushered in a new era, with the old techniques being non-efficient, particularly when it comes to problems like Pattern Recognition, Object Detection, Image Segmentation, and other image processing-based problems. CNN is one of the most deployed deep

learning neural networks.



BACKGROUND OF CNNS

Around the 1980s, CNNs were developed and deployed for the first time. A CNN could only detect handwritten digits at the time. CNN was primarily used in various areas to read zip and pin codes etc.

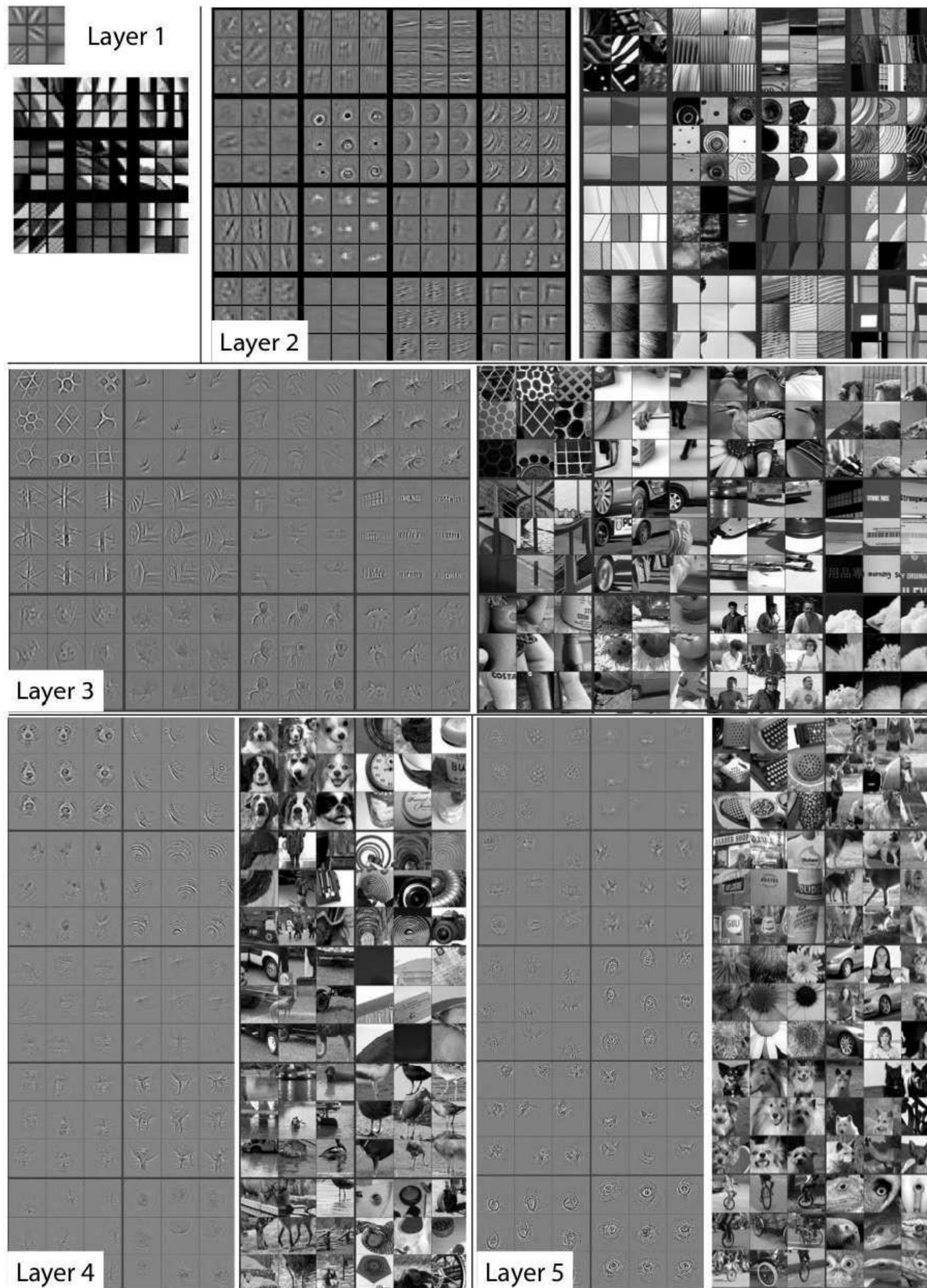
The most common aspect of any A.I. model is that it requires a massive amount of data to train. This was one of the biggest problems that CNN faced at the time, and due to this, they were only used in the postal industry. Yann LeCun was the first to introduce convolutional neural networks.

Kunihiko Fukushima, a renowned Japanese scientist, who even invented recognition, which was a very simple Neural Network used for image identification, had developed on the work done earlier by LeCun

What is CNN?

In the field of deep learning, convolutional neural network (CNN) is among

the class of deep neural networks, which was being mostly deployed in the field of analyzing/image recognition.



Convolutional Neural uses a very special kind of method which is being

known as Convolution. The mathematical definition of convolution is a mathematical operation being applied on the two functions that give output in a form of a third function that shows how the shape of one function is being influenced, modified by the other function.

The Convolutional neural networks(CNN) consists of various layers of artificial neurons. Artificial neurons, similar to that neuron cells that are being used by the human brain for passing various sensory input signals an d o th er respon ses, are mathematical functions that are being used for calculating the sum of various inputs and giving output in the form of an activation value.

The behaviour of each CNN neuron is being defined by the value of its weights. When being fed with the values (of the pixel), the artificial neurons of a CNN recognizes various visual features and specifications.

When we give an input image into a CNN, each of its inner layers generates various activation maps. Activation maps point out the relevant features of the given input image. Each of the CNN neurons generally takes input in the form of a group/patch of the pixel, multiplies their values(colours) by the value of its weights, adds them up, and input them through the respective activation function.

The first (or maybe the bottom) layer of the CNN usually recognizes the various features of the input image such as edges horizontally, vertically, and diagonally.

The output of the first layer is being fed as an input of the next layer, which in turn will extract other complex features of the input image like corners and combinations of edges.

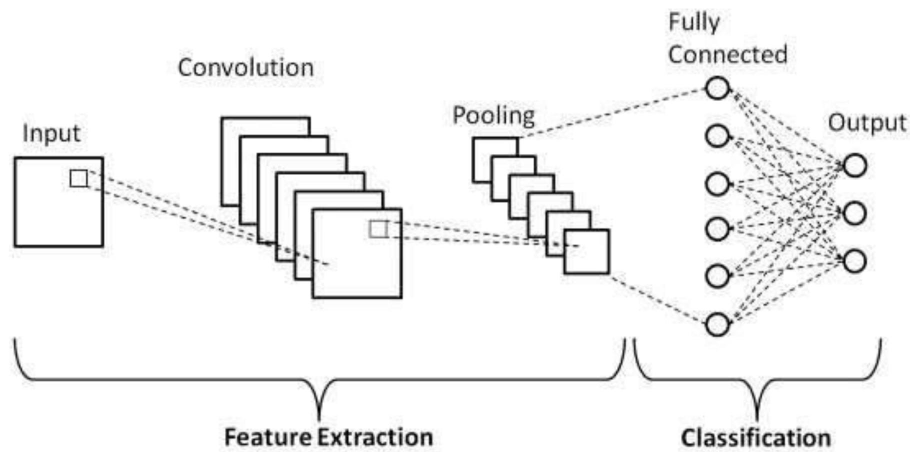
The deeper one moves into the convolutional neural network, the more the layers start detecting various higher-level features such as objects, faces, etc

CNN's Basic Architecture

A CNN architecture consists of two key components:

- A convolution tool that separates and identifies the distinct features of an image for analysis in a process known as Feature Extraction
- A fully connected layer that takes the output of the convolution process and predicts the image's class based on the features retrieved earlier.

The CNN is made up of three types of layers: convolutional layers, pooling layers, and fully-connected (FC) layers.



Convolution Layers

This is the very first layer in the CNN that is responsible for the extraction of the different features from the input images. The convolution mathematical operation is done between the input image and a filter of a specific size MxM in this layer.

THE FULLY CONNECTED

The Fully Connected (FC) layer comprises the weights and biases together with the neurons and is used to connect the neurons between two separate layers. The last several layers of a CNN Architecture are usually positioned before the output layer.

Pooling layer

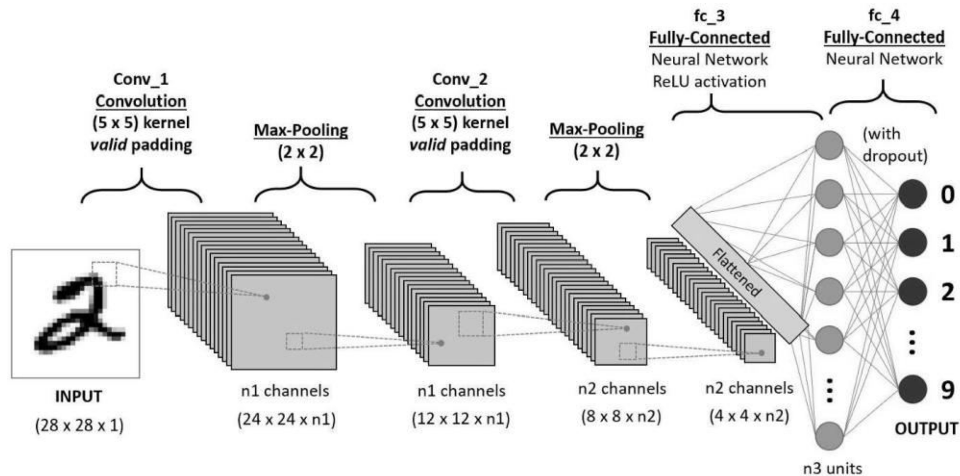
The Pooling layer is responsible for the reduction of the size(spatial) of the Convolved Feature. This decrease in the computing power is being required to process the data by a significant reduction in the dimensions.

There are two types of pooling

1 average pooling

2 max pooling.

A Pooling Layer is usually applied after a Convolutional Layer. This layer's major goal is to lower the size of the convolved feature map to reduce computational expenses. This is accomplished by reducing the connections between layers and operating independently on each feature map. There are numerous sorts of Pooling operations, depending on the mechanism utilised.



The largest element is obtained from the feature map in Max Pooling. The average of the elements in a predefined sized Image segment is calculated using Average Pooling. Sum Pooling calculates the total sum of the components in the predefined section. The Pooling Layer is typically used to connect the Convolutional Layer and the FC Layer.

Dropout

To avoid overfitting (when a model performs well on training data but not on new data), a dropout layer is utilised, in which a few neurons are removed from the neural network during the training phase, resulting in a smaller model.

Activation Functions

They're utilised to learn and approximate any form of network variable-to-variable association that's both continuous and complex.

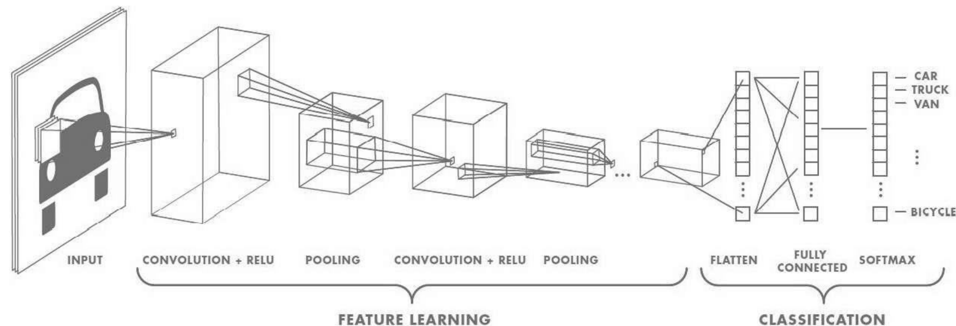
It gives the network non-linearity. The ReLU, Softmax, and tanH are some of the most often utilised activation functions.

TRAINING THE CONVOLUTIONAL NEURAL NETWORK

The process of adjusting the value of the weights is defined as the "training" of the neural network.

Firstly, the CNN initiates with the random weights. During the training of CNN, the neural network is being fed with a large dataset of images being labelled with their corresponding class labels (cat, dog, horse, etc.). The CNN network processes each image with its values being assigned randomly and then make comparisons with the class label of the input image.

If the output does not match the class label(which mostly happen initially at the beginning of the training process and therefore makes a respective small adjustment to the weights of its CNN neurons so that output correctly matches the class label image.



The corrections to the value of weights are being made through a technique which is known as backpropagation. Backpropagation optimizes the tuning process and makes it easier for adjustments for better accuracy every run of the training of the image dataset is being called an "epoch."

The CNN goes through several series of epochs during the process of training, adjusting its weights as per the required small amounts.

After each epoch step, the neural network becomes a bit more accurate at classifying and correctly predicting the class of the training images. As the CNN improves, the adjustments being made to the weights become smaller and smaller accordingly.

After training the CNN, we use a test dataset to verify its accuracy. The test dataset is a set of labelled images that were not being included in the training process. Each image is being fed to CNN, and the output is compared to the actual class label of the test image. Essentially, the test dataset evaluates the prediction performance of the CNN

If a CNN accuracy is good on its training data but is bad on the test data, it is said as "overfitting." This happens due to less size of the dataset (training)

Limitations

They (CNN) use massive computing power and resources for the recognition of various visual patterns/trends that is very much impossible to achieve by the human eye.

One usually needs a very long time to train a convolutional neural network, especially with a large size of image datasets.

One generally requires very specialized hardware (like a GPU) to perform the training of the dataset

PYTHON CODE IMPLEMENTATION FOR IMPLEMENTING CNN FOR CLASSIFICATION

Importing Relevant Libraries

```
import NumPy as np
%matplotlib inline
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import TensorFlow as tf
tf.compat.v1.set_random_seed(2019)
```

Loading MNIST Dataset

```
(X_train,Y_train),(X_test,Y_test) =
keras.datasets.mnist.load_data()
```

Scaling The Data

```
X_train = X_train / 255
X_test = X_test / 255

#flatenning
X_train_flattened = X_train.reshape(len(X_train), 28*28)
X_test_flattened = X_test.reshape(len(X_test), 28*28)
```

Designing The Neural Network

```
model = keras.Sequential([
     keras.layers.Dense(10, input_shape=(784,),
activation='sigmoid')
])
model.compile(optimizer='adam',
             loss='sparse_categorical_crossentropy',
             metrics=['accuracy'])
model.fit(X_train_flattened, Y_train, epochs=5)
```

Output:

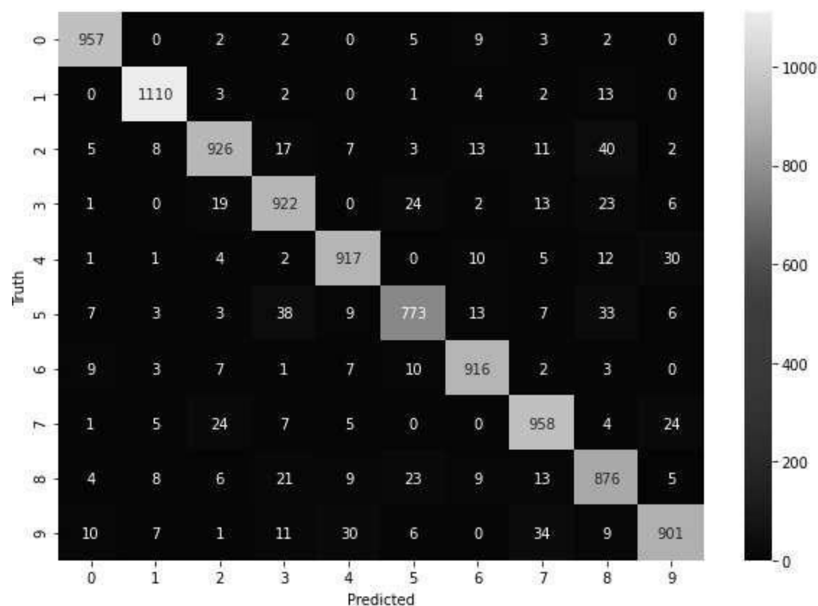
```
Epoch 1/5
1875/1875 [==============================] - 8s 4ms/step -
loss: 0.7187 - accuracy: 0.8141
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step -
loss: 0.3122 - accuracy: 0.9128
```

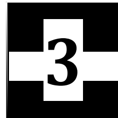
```
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step -
loss: 0.2908 - accuracy: 0.9187
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step -
loss: 0.2783 - accuracy: 0.9229
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step -
loss: 0.2643 - accuracy: 0.9262
```

Confusion Matrix for visualization of predictions

```
Y_predict = model.predict(X_test_flattened)
Y_predict_labels = [np.argmax(i) for i in Y_predict]
cm =
tf.math.confusion_matrix(labels=Y_test,predictions=Y_predict_labels)
%matplotlib inline
plt.figure(figsize = (10,7))
sn.heatmap(cm, annot=True, fmt='d')
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

Output





Matlab and Python in Machine Learning

MATLAB VS PYTHON PERFORMANCE

The essential version of MATLAB was written in FORTRAN77. After 1983, Clever Moler and his partners modified the whole application in C Programming. Be that as it may, Matlab is a confused mixture of multiple programming languages like C, C++, and Java.

That's the only reason Matlab is used for technical computing which makes this a high-performance language.

Python is well known for its straightforward syntax. You can improve Python Programming performance according to your need like avoid unwanted loops, get updated with the latest versions, the syntax is easy so try to make your code small and light that leads to high speed in execution.

MATLAB VS PYTHON FOR DEEP LEARNING

Python is viewed as in any case in the rundown of all AI development languages because of the simple syntax.

In Matlab, if you have good command in code, you can apply profound learning strategies to your work whether you're structuring algorithms, getting ready and marking information, or creating code and sending to inserted frameworks. But you need to buy the deep learning toolbox in Matlab.

Matlab or Python for Machine Learning

Matlab is most uncommonly seen as a business numerical handling condition, yet moreover as a programming language. It likewise has a standard library. Be that as it may, it utilizes joint cross-section variable based math and a broad framework for data taking care of and plotting. It is like a manner containing tool compartments for the students. In any case, these will cost the customer extra.

Python is a kind of programming language. The most widely recognized usage to this programming language is that in C (otherwise called C Python). In addition to the fact that Python is a programming language, yet it comprises a substantial standard library. This library is organized to concentrate on general programming and contain modules for OS explicit, stringing, systems administration, and databases.

Matlab vs python for Scientific Computing

Matlab has been there for a long time for scientific computing, and Python has its computing packages like SciPy, NumPy have not been outdated. So Matlab becomes a gift for the communities of Data analysis, Visualization, and Scientific. Matlab is a Math oriented language with different kinds of toolboxes that have several purposes, like Visual processing and Driving a Robot, etc. But you have to pay for the toolkits, those are professionally developed and tested by experts. In Python, you have to rely on community-authored packages for scientific and engineering usages.

Everyone has their way of learning and capabilities, so programming has the same rule every language has its pros and cons. So do an experiment with both words for a few days then decide which suits you best.

MATLAB VS PYTHON FOR ENGINEERING

MATLAB is the simplest and most beneficial computing environment for specialists and engineers. It incorporates the MATLAB language, the top primary programming language committed to numerical and scientific computing.

Python is use in mechanical engineering. The most popular application of python is to perform numerical analysis.

DEEP LEARNING WITH MATLAB

The term ‘artificial intelligence’ or ‘AI’ as it is now commonly known was first coined by the renowned computer scientist John McCarthy in 1955 – 56. There are various definitions of what AI connotes. The European Commission (EC) defines AI as follows:

"Artificial intelligence (AI) refers to systems that display intelligent behaviour by analyzing their environment and taking actions – with some degree of autonomy – to achieve specific goals. AI -based systems can be purely software-based, acting in the virtual world (e.g. voice assistants, image analysis software, search engines, speech and face recognition systems) or AI can be embedded in hardware devices (e.g. advanced robots, autonomous cars, drones or Internet of Things applications)."

And if it is good enough for EC, it should be good enough for us. But to keep it simple, AI involves using computers to do things that traditionally require human intelligence. AI involves creating algorithms to classify, analyze, and make predictions using this data.

Machine learning (ML) is a subset of AI and is essentially a modelling technique that figures out a model from the data given to it. The data can be anything from audio, images to documents. The primary objective of ML is to create a model using the training data when mathematical equations and laws are not adequate to arrive at the model.

Deep Learning, the main topic of this chapter, is a subset of ML, in which a model learns to perform classification tasks directly from images, text, or sound. Many people confuse between ML and Deep Learning. The two main (there are others too) differences between ML and Deep Learning is that unlike ML, Deep Learning needs a large training dataset. And while ML takes shorter training times, Deep Learning tends to take longer training times. In order to understand Deep Learning and how MATLAB can help, it is necessary to understand what the 'deep' in Deep Learning stands for. And in order to do that, we need to first understand what a neural network means.

The concept of neural network is based on biological neurons, which are the elements in the brain that establish communication with the nerves. A neural network architecture simulates this communication in the computational environment by programs composed of nodes and values that work together to process data. A typical system of neural network architecture will attempt to solve a problem by asking a series of 'yes' and 'no' questions about the subject.

By discarding certain elements and accepting others, an acceptable answer is eventually found. The 'deep' in Deep Learning isn't a reference to any kind of deeper understanding achieved by the approach; rather, it stands for this idea of successive layers of representations. When Deep Learning was first introduced, it consisted only of one input and one output layers, with may be one layer between the two. Anything more than 2 layers qualify as Deep Learning. Today's Deep Learning networks can have literally 100s of layers, thanks to increase in processing power. How many layers contribute to a model of the data is called the depth of

the model. A deep neural network thus combines multiple nonlinear processing layers, using simple elements operating in parallel and inspired by biological nervous systems. It consists of an input layer, several hidden layers, and an output layer. The layers are interconnected via nodes, or neurons, with each hidden layer using the output of the previous layer as its input. It is generally perceived that the greater the number of layers, better the accuracy of the final model.

Deep Learning is especially well-suited to identification applications such as face recognition, text translation, voice recognition, and advanced driver assistance systems, including, lane classification and traffic sign recognition since it is deemed as having better accuracy than ML. Advanced tools and techniques have dramatically improved Deep Learning algorithms—to the point where they can outperform humans at classifying images, win against the world's best GO player, or enable a voice-controlled assistant used by the likes of Amazon and google Home.

USING MATLAB® FOR DEEP LEARNING

MATLAB® is a programming platform designed specifically for engineers and scientists to analyze and design systems and products in a fast and efficient manner. The heart of MATLAB is the MATLAB language, a matrix-based language allowing the most natural expression of computational mathematics. MATLAB is considered one of the best programming languages at being able to handle the matrices of Deep Learning in a simple and intuitive manner.

Advantages of MATLAB for Deep Learning

MATLAB has interactive Deep Learning apps for labeling that includes signal data, audio data, images, and video. Labelling is one of the most tedious tasks in Deep Learning, and MATLAB is the ideal app that automates it.

MATLAB can help with generating synthetic data when you don't have enough data of the right scenarios. This is a huge benefit as Deep Learning depends a lot on huge amount of datasets.

- In the case of automated driving, you can author scenarios and simulate the output of different sensors using a 3D simulation environment.
- In radar and communications, this includes generating data for waveform-modulation-identification and target classification applications.
- MATLAB has a variety of ways to interact and transfer data between Deep Learning frameworks.

- MATLAB supports ONNX to import and export models between other frameworks. A model designed in PyTorch, for example, can be brought into MATLAB, and models trained in MATLAB can be exported using the ONNX framework.
- MATLAB also supports Python interoperability: You can call Python from MATLAB and MATLAB from Python.
- The MATLAB Deep Learning Toolbox™ provides a framework for designing and implementing deep neural networks with algorithms, pre-trained models, and apps.
- MATLAB also provides specialized toolboxes and functionality for Reinforcement Learning, Automated Driving, Natural Language Processing, Medical Image Processing and Computer Vision

MATLAB VS PYTHON: WHY AND HOW TO MAKE THE SWITCH

MATLAB VS PYTHON: COMPARING FEATURES AND PHILOSOPHY

Python is a high-level, general-purpose programming language designed for ease of use by human beings accomplishing all sorts of tasks. Python was created by Guido van Rossum and first released in the early 1990s. Python is a mature language developed by hundreds of collaborators around the world.

Python is used by developers working on small, personal projects all the way up to some of the largest internet companies in the world. Not only does Python run Reddit and Dropbox, but the original Google algorithm was written in Python. Also, the Python-based Django Framework runs Instagram and many other websites. On the science and engineering side, the data to create the 2019 photo of a black hole was processed in Python, and major companies like Netflix use Python in their data analytics work.

There is also an important philosophical difference in the MATLAB vs Python comparison. **MATLAB** is **proprietary, closed-source** software. For most people, a license to use MATLAB is quite expensive, which means that if you have code in MATLAB, then only people who can afford a license will be able to run it. Plus, users are charged for each additional toolbox they want to install to extend the basic functionality of MATLAB. Aside from the cost, the MATLAB language is developed exclusively by Mathworks. If Mathworks were ever to go out of business, then MATLAB would no longer be able to be developed and might eventually stop functioning.

On the other hand, **Python** is **free and open-source** software. Not only can you download Python at no cost, but you can also download, look at, and modify the source code as well. This is a big advantage for Python because it means that anyone can pick up the development of the language if the current developers were unable to continue for some reason.

If you're a researcher or scientist, then using open-source software has some pretty big benefits. Paul Romer, the 2018 Nobel Laureate in Economics, is a recent convert to Python. By his estimation, switching to open-source software in general, and Python in particular, brought greater integrity and accountability to his research. This was because all of the code could be shared and run by any interested reader.

Moreover, since Python is available at no cost, a much broader audience can use the code you develop. As you'll see a little later on in the article, Python has an awesome community that can help you get started with the language and advance your knowledge. There are tens of thousands of tutorials, articles, and books all about Python software development. Here are a few to get you started:

- Introduction to Python 3
- Basic Data Types in Python
- Python 3 Basics Learning Path

Plus, with so many developers in the community, there are hundreds of thousands of free packages to accomplish many of the tasks that you'll want to do with Python.

Like **MATLAB**, **Python** is an **interpreted** language. This means that Python code can be ported between all of the major operating system platforms and CPU architectures out there, with only small changes required for different platforms. There are distributions of Python for desktop and laptop CPUs and microcontrollers like Adafruit. Python can also talk to other microcontrollers like Arduino with a simple programming interface that is almost identical no matter the host operating system.

For all of these reasons, and many more, Python is an excellent choice to replace MATLAB as your programming language of choice. Now that you're convinced to try out Python, read on to find out how to get it on your computer and how to switch from MATLAB!

Note: GNU Octave is a free and open-source clone of MATLAB. In this sense, GNU Octave has the same philosophical advantages that Python has around code reproducibility and access to the software.

Octave's syntax is mostly compatible with MATLAB syntax, so it provides a short learning curve for MATLAB developers who want to use open-source software. However, Octave can't match Python's community or the number of different kinds of applications that Python can serve, so we definitely recommend you switch whole hog over to Python.

Besides, this website is called *Real Python*, not *Real Octave* =Ø

SETTING UP YOUR ENVIRONMENT FOR PYTHON

Getting Python via Anaconda

Python can be downloaded from a number of different sources, called **distributions**. For instance, the Python that you can download from the official Python website is one distribution. Another very popular Python distribution, particularly for math, science, engineering, and data science applications, is the Anaconda distribution.

There are two main reasons that Anaconda is so popular:

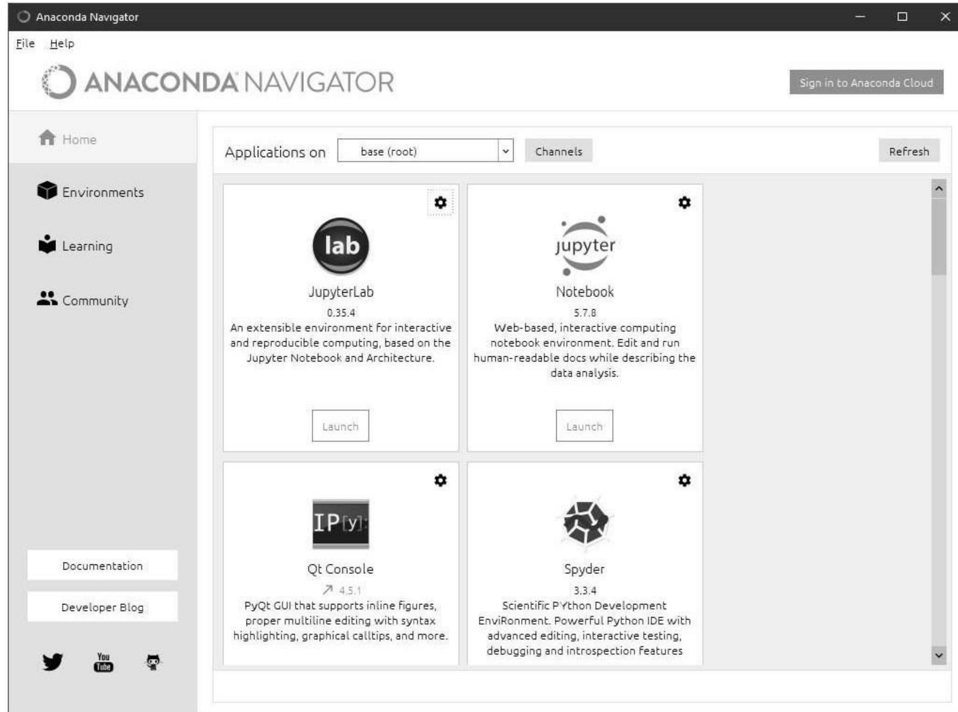
1. Anaconda distributes pre-built packages for Windows, macOS, and Linux, which means that the installation process is really easy and the same for all three major platforms.
2. Anaconda includes all of the most popular packages for engineering and data science type workloads in one single installer.

For the purposes of creating an environment that is very similar to MATLAB, you should download and install Anaconda. As of this writing, there are two major versions of Python available: Python 2 and Python 3.

You should definitely install the version of Anaconda for Python 3, since Python 2 will not be supported past January 1, 2020. Python 3.7 is the most recent version at the time of this writing, but Python 3.8 should be out a few months after this chapter is published. Either 3.7 or 3.8 will work the same for you, so choose the most recent version you can.

Once you have downloaded the Anaconda installer, you can follow the default set up procedures depending on your platform. You should install Anaconda in a directory that does not require administrator permission to modify, which is the default setting in the installer.

With Anaconda installed, there are a few specific programs you should know about. The easiest way to launch applications is to use the Anaconda Navigator. On Windows, you can find this in the Start Menu and on macOS you can find it in Launchpad. Here's a screenshot of the Anaconda Navigator on Windows:



In the screenshot, you can see several installed applications, including **JupyterLab**, **Jupyter Notebook**, and **Spyder**.

On Windows, there is one other application that you should know about. This is called **Anaconda Prompt**, and it is a command prompt set up specifically to work with conda on Windows.

If you want to type conda commands in a terminal, rather than using the Navigator GUI, then you should use Anaconda Prompt on Windows.

On macOS, you can use any terminal application such as the default Terminal.app or iTerm2 to access conda from the command line.

On Linux, you can use the terminal emulator of your choice and which specific emulator is installed will depend on your Linux distribution.

Terminology Note: You may be a little bit confused about conda versus Anaconda. The distinction is subtle but important. **Anaconda** is a distribution of Python that includes many of the necessary packages for scientific work of all kinds. **conda** is a cross-platform package management software that is included with the Anaconda distribution of Python. conda is the software that you use to build, install, and remove packages within the Anaconda distribution.

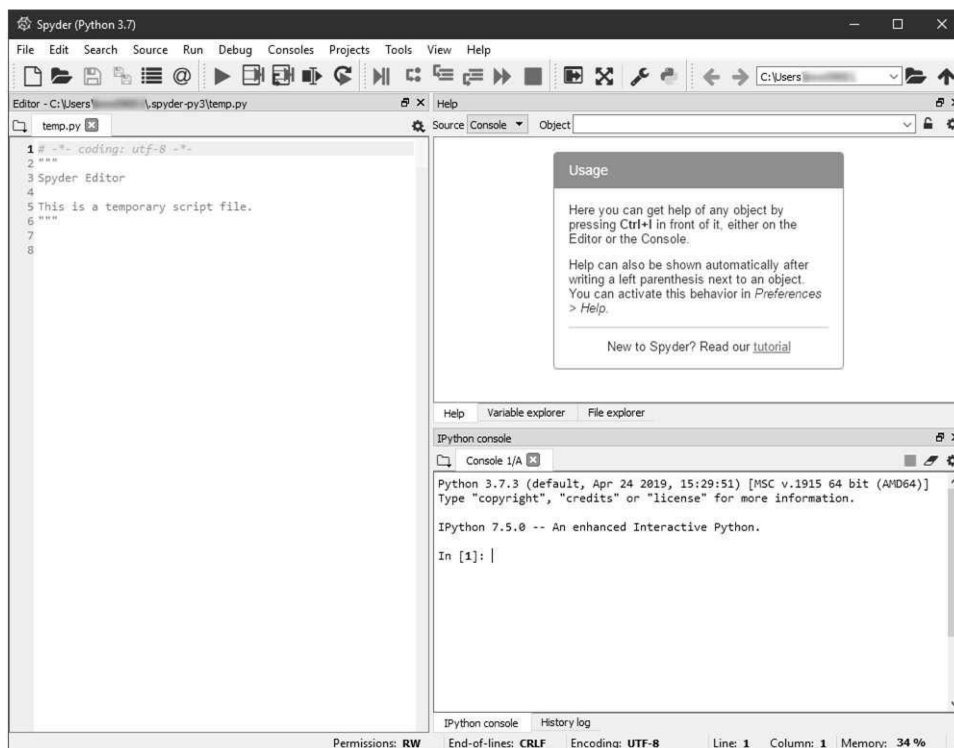
Python also includes another way to install packages, called pip. If you're using Anaconda, you should always prefer to install packages using conda whenever

possible.

Sometimes, though, a package is only available with pip, and for those cases, you can read What Is Pip? A Guide for New Pythonistas.

Changing the Default Window Layout in Spyder

The default window in Spyder looks like the image below. This is for version 3.3.4 of Spyder running on Windows 10. It should look quite similar on macOS or Linux:

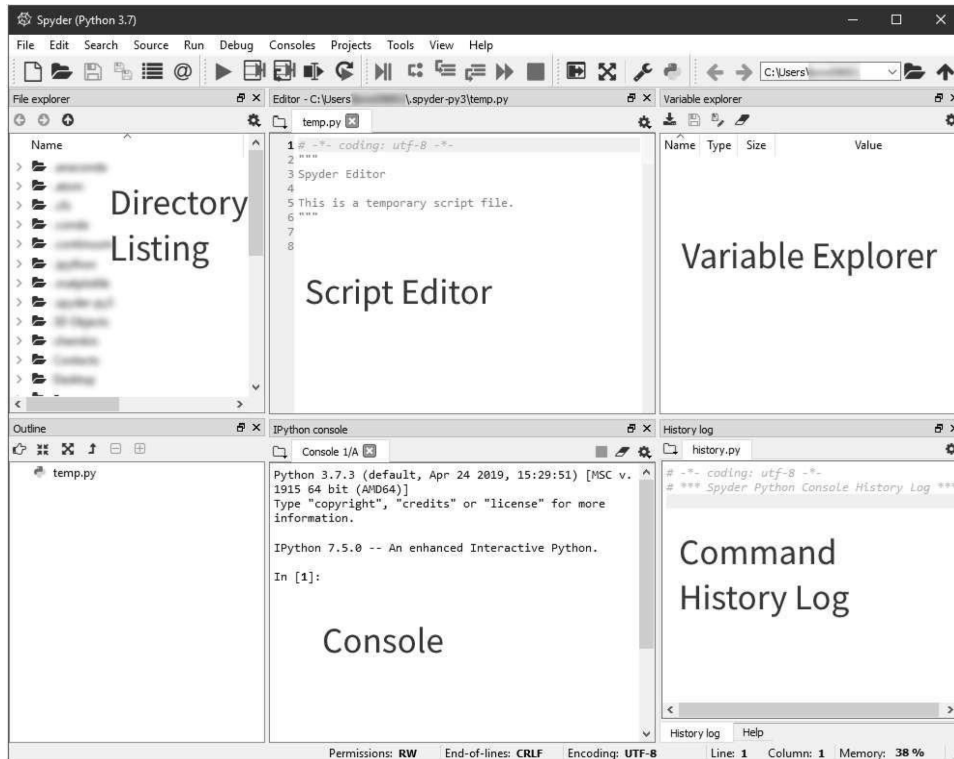


Before you take a tour of the user interface, you can make the interface look a little more like MATLAB. In the *View ! Window layouts* menu choose *MATLAB layout.* That will change the window automatically so it has the same areas that you're used to from MATLAB, annotated on the figure below:

In the top left of the window is the *File Explorer* or **directory listing**. In this pane, you can find files that you want to edit or create new files and folders to work with.

In the top center is a file editor. In this editor, you can work on Python scripts that you want to save to re-run later on. By default, the editor opens a file called temp.py located in Spyder's configuration directory. This file is meant as

a temporary place to try things out before you save them in a file somewhere else on your computer.



In the bottom center is the console. Like in MATLAB, the console is where you can run commands to see what they do or when you want to debug some code. Variables created in the console are not saved if you close Spyder and open it up again. The console is technically running IPython by default.

Any commands that you type in the console will be logged into the history file in the bottom right pane of the window. Furthermore, any variables that you create in the console will be shown in the variable explorer in the top right pane.

Notice that you can adjust the size of any pane by putting your mouse over the divider between panes, clicking, and dragging the edge to the size that you want. You can close any of the panes by clicking the *x* in the top of the pane.

You can also break any pane out of the main window by clicking the button that looks like two windows in the top of the pane, right next to the *x* that closes the pane.

When a pane is broken out of the main window, you can drag it around and rearrange it however you want. If you want to put the pane back in the main

window, drag it with the mouse so a transparent blue or gray background appears and the neighboring panes resize, then let go and the pane will snap into place.

Once you have the panes arranged exactly how you want, you can ask Spyder to save the layout. Go to the *View* menu and find the *Window layouts* flyout again.

Then click *Save current layout* and give it a name. This lets you reset to your preferred layout at any time if something gets changed by accident. You can also reset to one of the default configurations from this menu.

Getting an Integrated Development Environment

One of the big advantages of MATLAB is that it includes a development environment with the software. This is the window that you're most likely used to working in. There is a console in the center where you can type commands, a variable explorer on the right, and a directory listing on the left.

Unlike MATLAB, Python itself does not have a default development environment.

It is up to each user to find one that fits their needs. Fortunately, Anaconda comes with two different integrated development environments (IDEs) that are similar to the MATLAB IDE to make your switch seamless. These are called Spyder and JupyterLab. In the next two sections, you'll see a detailed introduction to Spyder and a brief overview of JupyterLab.

Spyder

Spyder is an IDE for Python that is developed specifically for scientific Python work. One of the really nice things about Spyder is that it has a mode specifically designed for people like you who are converting from MATLAB to Python. You'll see that a little later on.

Running Statements in the Console in Spyder

In this chapter, you're going to be writing some simple Python commands, but don't worry if you don't quite understand what they mean yet.

You'll learn more about Python syntax a little later on in this chapter. What you want to do right now is get a sense for how Spyder's interface is similar to and different from the MATLAB interface.

You'll be working a lot with the Spyder console in this chapter, so you should learn about how it works.

In the console, you'll see a line that starts with In [1]:, for input line 1. Spyder (really, the IPython console) numbers all of the input lines that you type. Since this is the first input you're typing, the line number is 1. In the rest of this chapter,

you'll see references to "input line X," where X is the number in the square brackets.

One of the first things I like to do with folks who are new to Python is show them the *Zen of Python*. This short poem gives you a sense of what Python is all about and how to approach working with Python.

To see the *Zen of Python*, type import this on input line 1 and then run the code by pressing Enter. You'll see an output like below:

```
In [1]: import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one— and preferably only one —obvious way to
do it.
Although that way may not be obvious at first unless you're
Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good
idea.
Namespaces are one honking great idea — let's do more of
those!
```

This code has import this on input line 1. The output from running import this is to print the *Zen of Python* onto the console. We'll return to several of the stanzas in this poem later on in the chapter.

In many of the code blocks in this chapter, you'll see three greater-than signs (>>>) in the top right of the code block. If you click that, it will remove the input prompt and any output lines, so you can copy and paste the code right into your console.

Many Pythonistas maintain a healthy sense of humor. This is displayed in many places throughout the language, including the *Zen of Python*. For another one, in the Spyder console, type the following code, followed by Enter to run it:

In [2]: import antigravity

That statement will open your web browser to the webcomic called *XKCD*, specifically comic #353, where the author has discovered that Python has given him the ability to fly!

You've now successfully run your first two Python statements! Congratulations =<

If you look at the History Log, you should see the first two commands you typed in the console (import this and import antigravity). Let's define some variables and do some basic arithmetic now. In the console, type the following statements, pressing Enter after each one:

```
In [3]: var_1 = 10
In [4]: var_2 = 20
In [5]: var_3 = var_1 + var_2
In [6]: var_3
Out[6]: 30
```

In this code, you defined 3 variables: var_1, var_2, and var_3. You assigned var_1 the value 10, var_2 the value 20, and var_3 the sum of var_1 and var_2. Then you showed the value of the var_3 variable by writing it as the only thing on the input line. The output from that statement is shown on the next Out line, and the number on the Out line matches the associated In line.

There are two main things for you to notice in these commands:

1. If a statement does not include an assignment (with an =), it is printed onto an Out line. In MATLAB, you would need to include a semicolon to suppress the output even from assignment statements, but that is not necessary in Python.
2. On input lines 3, 4, and 5, the *Variable explorer* in the top right pane updated.

After you run these three commands, your *Variable explorer* should look like the image below:

In this image, you can see a table with four columns:

1. ***Name*** shows the name that you gave to var_1, var_2, and var_3.
2. ***Type*** shows the Python type of the variable, in this case, all int for integer numbers.

3. ***Size*** shows the size of the data stored variable, which is more useful for lists and other data structures.
4. ***Value*** shows the current value of the variable.

Variable explorer

Name	Type	Size	Value
var_1	int	1	10
var_2	int	1	20
var_3	int	1	30

Running Code in Files in Spyder

The last stop in our brief tour of the Spyder interface is the File editor pane. In this pane, you can create and edit Python scripts and run them using the console. By default, Spyder creates a temporary file called temp.py which is intended for you to temporarily store commands as you're working before you move or save them in another file. Let's write some code into the temp.py file and see how to run it. The file starts with the following code, which you can just leave in place:

```
1# -*- coding: utf-8 -*-
2"""
3Spyder Editor
4
5This is a temporary script file.
6"""
```

In this code, you can see two Python syntax structures:

- Line 1 has a comment. In Python, the comment character is the hash or pound sign (#). MATLAB uses the percent symbol (%) as the comment character. Anything following the hash on the line is a comment and is usually ignored by the Python interpreter.
- Starting on line 2 is a string that provides some context for the contents of the file. This is often referred to as a **documentation string** or docstring for short.

Now you can start adding code to this file. Starting on line 8 in temp.py, enter

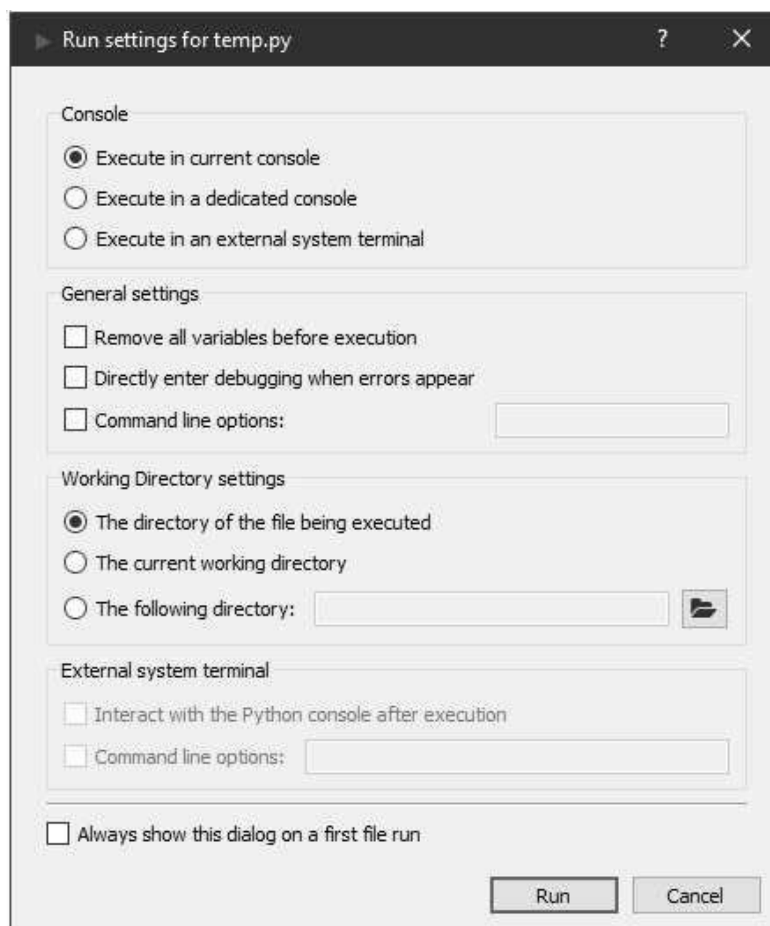
the following code that is similar to what you already typed in the console:

```
8var_4 = 10
9var_5 = 20
10var_6 = var_4 + var_5
```

Then, there are three ways to run the code:

1. You can use the F5 keyboard shortcut to run the file just like in MATLAB.
2. You can click the green right-facing triangle in the menu bar just above the *Editor* and *File explorer* panes.
3. You can use the *Run ! Run* menu option.

The first time you run a file, Spyder will open a dialog window asking you to confirm the options you want to use. For this test, the default options are fine and you can click *Run* at the bottom of the dialog box:



This will automatically execute the following code in the console:

```
In  [7]:  runfile('C:/Users/Eleanor/.spyder-py3/temp.py',
   ...:          wdir='C:/Users/Eleanor/.spyder-py3')
```

This code will run the file that you were working on. Notice that running the file added three variables into the Variable explorer: var_4, var_5, and var_6. These are the three variables that you defined in the file. You will also see runfile() added to the History log.

In Spyder, you can also create code cells that can be run individually. To create a code cell, add a line that starts with # %% into the file open in the editor:

```
11# %% This is a code cell
12var_7 = 42
13var_8 = var_7 * 2
14
15# %% This is a second code cell
16print("This code will be executed in this cell")
```

In this code, you have created your first code cell on line 11 with the # %% code. What follows is a line comment and is ignored by Python. On line 12, you are assigning var_7 to have the value 42 and then line 13 assigns var_8 to be var_7 times two. Line 15 starts another code cell that can be executed separately from the first one.

To execute the code cells, click the *Run Current Cell* or *Run Current Cell and Go to the Next One* buttons next to the generic *Run* button in the toolbar. You can also use the keyboard shortcuts Ctrl+Enter to run the current cell and leave it selected, or Shift+Enter to run the current cell and select the next cell.

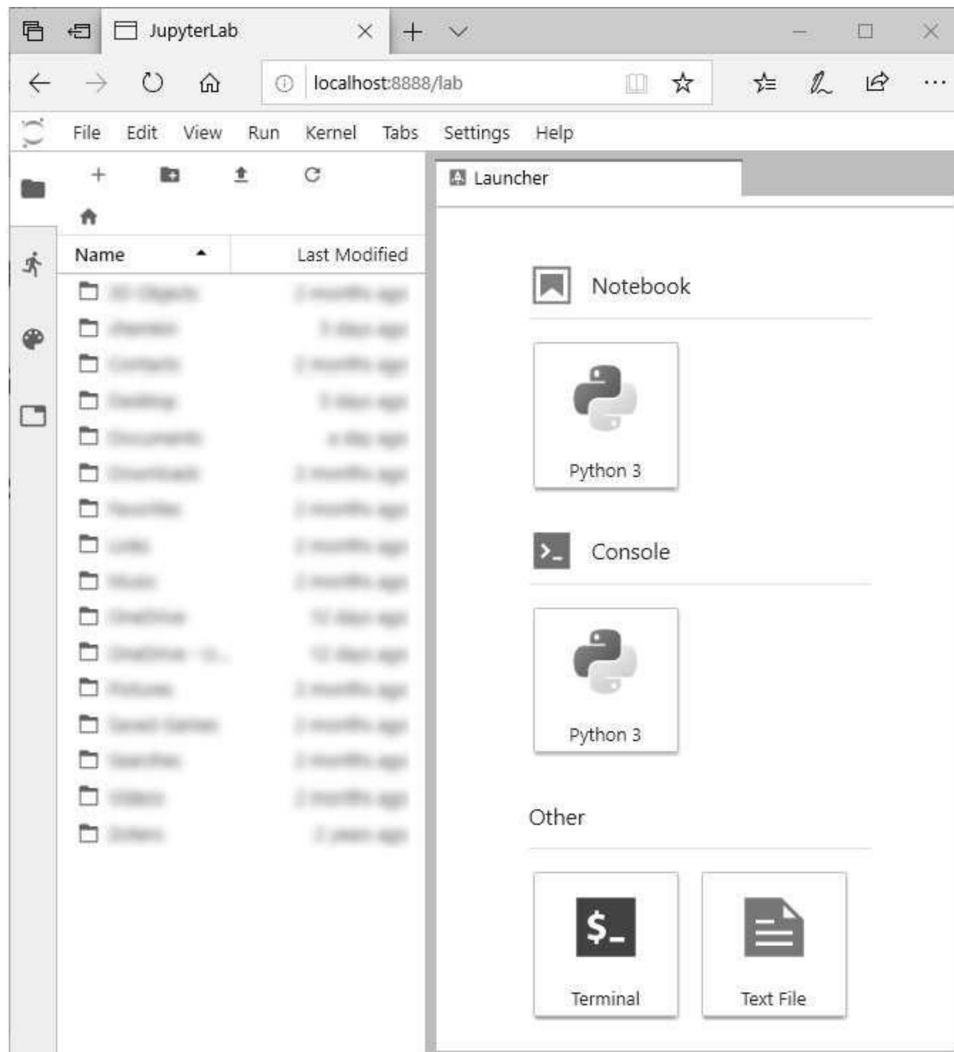
Spyder also offers easy-to-use debugging features, just like in MATLAB. You can double-click any of the line numbers in the *Editor* to set a breakpoint in your code. You can run the code in debug mode using the blue right-facing triangle with two vertical lines from the toolbar, or the Ctrl+F5 keyboard shortcut. This will pause execution at any breakpoints you specify and open the ipdb debugger in the console which is an IPython-enhanced way to run the Python debugger pdb. You can read more in Python Debugging With pdb.

JupyterLab

JupyterLab is an IDE developed by Project Jupyter. You may have heard of Jupyter Notebooks, particularly if you're a data scientist. Well, JupyterLab is the next iteration of the Jupyter Notebook. Although at the time of this writing JupyterLab is still in beta, Project Jupyter expects that JupyterLab will eventually replace the current Notebook server interface. However, JupyterLab

is fully compatible with existing Notebooks so the transition should be fairly seamless.

The main JupyterLab window is shown in the picture below:



There are two main sections of the interface:

1. On the left is a *File explorer* that lets you open files from your computer.
2. On the right side of the window is how you can open create new Notebook files, work in an IPython console or system terminal, or create a new text file.

If you're interested in learning more about JupyterLab, you can read a lot more about the next evolution of the Notebook in the blog post announcing the beta release or in the JupyterLab documentation. You can also learn about the Notebook interface in Jupyter Notebook: An Introduction and the Using Jupyter Notebooks course. One neat thing about the Jupyter Notebook-style document is that the code cells you created in Spyder are very similar to the code cells in a Jupyter Notebook.

Summarizing Your Experience in Spyder

Now you have the basic tools to use Spyder as a replacement for the MATLAB integrated development environment. You know how to run code in the console or type code into a file and run the file. You also know where to look to see your directories and files, the variables that you've defined, and the history of the commands you typed.

Once you're ready to start organizing your code into modules and packages, you can check out the following resources:

- Python Modules and Packages – An Introduction
- How to Publish an Open-Source Python Package to PyPI
- How to Publish Your Own Python Package to PyPI

Spyder is a really big piece of software, and you've only just scratched the surface. You can learn a lot more about Spyder by reading the official documentation, the troubleshooting and FAQ guide, and the Spyder wiki.

LEARNING ABOUT PYTHON'S MATHEMATICAL LIBRARIES

Now you've got Python on your computer and you've got an IDE where you feel at home. So how do you learn about how to actually accomplish a task in Python? With MATLAB, you can use a search engine to find the topic you're looking for just by including MATLAB in your query. With Python, you'll usually get better search results if you can be a bit more specific in your query than just including Python.

You'll take the next step to really feeling comfortable with Python by learning about how Python functionality is divided into several libraries. You'll also learn what each library does so you can get top-notch results with your searches!

Python is sometimes called a **batteries-included** language. This means that most of the important functions you need are already included when you install Python. For instance, Python has built-in math and statistics libraries that include the basic operations.

Sometimes, though, you want to do something that isn't included in the language. One of the big advantages of Python is that someone else has probably done whatever you need to do and published the code to accomplish that task. There are *several hundred-thousand* publicly available and free packages that you can easily install to perform various tasks. These range from processing PDF files to building and hosting an interactive website to working with highly optimized mathematical and scientific functions.

Working with arrays or matrices, optimization, or plotting requires additional libraries to be installed. Fortunately, if you install Python with the Anaconda installer these libraries come preinstalled and you don't need to worry. Even if you're not using Anaconda, they are usually pretty easy to install for most operating systems.

The set of important libraries you'll need to switch over from MATLAB are typically called the **SciPy stack**. At the base of the stack are libraries that provide fundamental array and matrix operations (**NumPy**), integration, optimization, signal processing, and linear algebra functions (**SciPy**), and plotting (**Matplotlib**). Other libraries that build on these to provide more advanced functionality include **Pandas**, **scikit-learn**, **SymPy**, and more.

NumPy (Numerical Python)

NumPy is probably the most fundamental package for scientific computing in Python. It provides a highly efficient interface to create and interact with multi-dimensional arrays. Nearly every other package in the SciPy stack uses or integrates with NumPy in some way.

NumPy arrays are the equivalent to the basic array data structure in MATLAB. With NumPy arrays, you can do things like inner and outer products, transposition, and element-wise operations. NumPy also contains a number of useful methods for reading text and binary data files, fitting polynomial functions, many mathematical functions (sine, cosine, square root, and so on), and generating random numbers.

The performance-sensitive parts of NumPy are all written in the C language, so they are very fast. NumPy can also take advantage of optimized linear algebra libraries such as Intel's MKL or OpenBLAS to further increase performance.

SciPy (Scientific Python)

The SciPy package (as distinct from the SciPy stack) is a library that provides a huge number of useful functions for scientific applications. If you need to do work that requires optimization, linear algebra or sparse linear algebra, discrete

Fourier transforms, signal processing, physical constants, image processing, or numerical integration, then SciPy is the library for you! Since SciPy implements so many different features, it's almost like having access to a bunch of the MATLAB toolboxes in one package.

SciPy relies heavily on NumPy arrays to do its work. Like NumPy, many of the algorithms in SciPy are implemented in C or Fortran, so they are also very fast. Also like NumPy, SciPy can take advantage of optimized linear algebra libraries to further improve performance.

Matplotlib (MATLAB-like Plotting Library)

Matplotlib is a library to produce high-quality and interactive two-dimensional plots. Matplotlib is designed to provide a plotting interface that is similar to the plot() function in MATLAB, so people switching from MATLAB should find it somewhat familiar. Although the core functions in Matplotlib are for 2-D data plots, there are extensions available that allow plotting in three dimensions with the mplot3d package, plotting geographic data with cartopy, and many more listed in the Matplotlib documentation.

Other Important Python Libraries

With NumPy, SciPy, and Matplotlib, you can switch a lot of your MATLAB code to Python. But there are a few more libraries that might be helpful to know about.

- **Pandas** provides a DataFrame, an array with the ability to name rows and columns for easy access.
- **SymPy** provides symbolic mathematics and a computer algebra system.
- **scikit-learn** provides many functions related to machine learning tasks.
- **scikit-image** provides functions related to image processing, compatible with the similar library in SciPy.
- **Tensorflow** provides a common platform for many machine learning tasks.
- **Keras** provides a library to generate neural networks.
- **multiprocessing** provides a way to perform multi-process based parallelism. It's built into Python.
- **Pint** provides a unit library to conduct automatic conversion between physical unit systems.

- **PyTables** provides a reader and writer for HDF5 format files.
- **PyMC3** provides Bayesian statistical modeling and probabilistic machine learning functionality.

SYNTAX DIFFERENCES BETWEEN MATLAB® AND PYTHON

You'll learn how to convert your MATLAB code into Python code. You'll learn about the main syntax differences between MATLAB and Python, see an overview of basic array operations and how they differ between MATLAB and Python, and find out about some ways to attempt automatic conversion of your code.

The biggest technical difference between MATLAB and Python is that in MATLAB, everything is treated as an array, while in Python everything is a more general object. For instance, in MATLAB, strings are arrays of characters or arrays of strings, while in Python, strings have their own type of object called str. This has profound consequences for how you approach coding in each language, as you'll see below.

With that out of the way, let's get started! To help you, the sections below are organized into groups based on how likely you are to run into that syntax.

You Will Probably See This Syntax

The examples in this chapter represent code that you are very likely to see in the wild. These examples also demonstrate some of the more basic Python language features. You should make sure that you have a good grasp of these examples before moving on.

Comments Start With # in Python

In MATLAB, a comment is anything that follows a percent sign (%) on a line. In Python, comments are anything that follow the hash or pound sign (#). You already saw a Python comment. In general, the Python interpreter ignores the content of comments, just like the MATLAB interpreter, so you can write whatever content you want in the comment. One exception to this rule in Python is the example you saw earlier in the section about Spyder:

```
# -*- coding: utf-8 -*-
```

When the Python interpreter reads this line, it will set the encoding that it uses to read the rest of the file. This comment must appear in one of the first two lines of the file to be valid.

Another difference between MATLAB and Python is in how inline documentation is written. In MATLAB, documentation is written at the start of a function in a comment, like the code sample below:

```
function [total] = addition(num_1,num_2)
% ADDITION  Adds two numbers together
%    TOTAL = ADDITION(NUM_1,NUM_2) adds NUM_1 and NUM_2
together
%
%   See also SUM and PLUS
```

However, Python does not use comments in this way. Instead, Python has an idea called **documentation strings** or **docstrings** for short. In Python, you would document the MATLAB function shown above like this:

```
def addition(num_1, num_2):
   """Adds two numbers together.
   Example
   ----
   >>> total = addition(10, 20)
   >>> total
   30
   """
```

Notice in this code that the docstring is between two sets of three quote characters (“””). This allows the docstring to run onto multiple lines with the whitespace and newlines preserved. The triple quote characters are a special case of **string literals**. Don’t worry too much about the syntax of defining a function yet.

Whitespace at the Beginning of a Line Is Significant in Python

When you write code in MATLAB, blocks like if statements for and while loops, and function definitions are finished with the end keyword. It is generally considered a good practice in MATLAB to indent the code within the blocks so that the code is visually grouped together, but it is not syntactically necessary. For example, the following two blocks of code are functionally equivalent in MATLAB:

```
1num = 10;
2
3if num == 10
4disp("num is equal to 10")
5else
6disp("num is not equal to 10")
7end
8
9disp("I am now outside the if block")
```

In this code, you are first creating num to store the value 10 and then checking whether the value of num is equal to 10. If it is, you are displaying the phrase num is equal to 10 on the console from line 2. Otherwise, the else clause will kick in and display num is not equal to 10. Of course, if you run this code, you will see the num is equal to 10 output and then I am now outside the if block.

Now you should modify your code so it looks like the sample below:

```
1num = 10;
2
3if num == 10
4    disp("num is equal to 10")
5else
6    disp("num is not equal to 10")
7end
8
9disp("I am now outside the if block")
```

In this code, you have only changed lines 3 and 5 by adding some spaces or indentation in the front of the line. The code will perform identically to the previous example code, but with the indentation, it is much easier to tell what code goes in the if part of the statement and what code is in the else part of the statement.

In Python, indentation at the start of a line is used to delimit the beginning and end of class and function definitions, if statements, and for and while loops. There is no end keyword in Python. This means that indentation is very important in Python!

In addition, in Python the definition line of an if/else/elif statement, a for or while loop, a function, or a class is ended by a colon. In MATLAB, the colon is not used to end the line.

Consider this code example:

```
1num = 10
2
3if num == 10:
4    print("num is equal to 10")
5else:
6    print("num is not equal to 10")
7
8print("I am now outside the if block")
```

On the first line, you are defining num and setting its value to 10. On line 2, writing if num == 10: tests the value of num compared to 10. Notice the colon at

the end of the line. Next, line 3 *must* be indented in Python's syntax. On that line, you are using print() to display some output to the console, in a similar way to disp() in MATLAB. You'll read more about print() versus disp().

On line 4, you are starting the else block. Notice that the e in the else keyword is vertically aligned with the i in the if keyword, and the line is ended by a colon. Because the else is dedented relative to print() on line 3, and because it is aligned with the if keyword, Python knows that the code within the if part of the block has finished and the else part is starting. Line 5 is indented by one level, so it forms the block of code to be executed when the else statement is satisfied.

Lastly, on line 6 you are printing a statement from outside the if/else block. This statement will be printed regardless of the value of num. Notice that the p in print() is vertically aligned with the i in if and the e in else. This is how Python knows that the code in the if/else block has ended. If you run the code above, Python will display num is equal to 10 followed by I am now outside the if block.

Now you should modify the code above to remove the indentation and see what happens. If you try to type the code without indentation into the Spyder/ IPython console, you will get an IndentationError:

```
In [1]: num = 10
In [2]: if num == 10:
   ...: print("num is equal to 10")
 File "<ipython-input-2-f453ffd2bc4f>", line 2
   print("num is equal to 10")
       ^
IndentationError: expected an indented block
```

In this code, you first set the value of num to 10 and then tried to write the if statement without indentation. In fact, the IPython console is smart and automatically indents the line after the if statement for you, so you'll have to delete the indentation to produce this error.

When you're indenting your code, the official Python style guide called PEP 8 recommends using 4 space characters to represent one indentation level. Most text editors that are set up to work with Python files will automatically insert 4 spaces if you press the Tab key on your keyboard. You can choose to use the tab character for your code if you want, but you shouldn't mix tabs and spaces or you'll probably end up with a TabError if the indentation becomes mismatched.

Conditional Statements Use elif in Python

In MATLAB, you can construct conditional statements with if, elseif, and else.

These kinds of statements allow you to control the flow of your program in response to different conditions.

You should try this idea out with the code below, and then compare the example of MATLAB vs Python for conditional statements:

```
1num = 10;
2if num == 10
3    disp("num is equal to 10")
4elseif num == 20
5    disp("num is equal to 20")
6else
7    disp("num is neither 10 nor 20")
8end
```

In this code block, you are defining num to be equal to 10. Then you are checking if the value of num is 10, and if it is, using disp() to print output to the console. If num is 20, you are printing a different statement, and if num is neither 10 nor 20, you are printing the third statement.

In Python, the elseif keyword is replaced with elif:

```
1num = 10
2if num == 10:
3    print("num is equal to 10")
4elif num == 20:
5    print("num is equal to 20")
6else:
7    print("num is neither 10 nor 20")
```

This code block is functionally equivalent to the previous MATLAB code block. There are 2 main differences. On line 4, elseif is replaced with elif, and there is no end statement to end the block. Instead, the if block ends when the next dedented line of code is found after the else. You can read more in the Python documentation for if statements.

Calling Functions and Indexing Sequences Use Different Brackets in Python

In MATLAB, when you want to call a function or when you want to index an array, you use round brackets (()), sometimes also called parentheses. Square brackets ([]) are used to create arrays.

You can test out the differences in MATLAB vs Python with the example code below:

```
>> arr = [10, 20, 30];
```

```
>> arr(1)
ans =
   10
>> sum(arr)
ans =
   60
```

In this code, you first create an array using the square brackets on the right side of the equal sign. Then, you retrieve the value of the first element by arr(1), using the round brackets as the indexing operator. On the third input line, you are calling sum() and using the round brackets to indicate the parameters that should be passed into sum(), in this case just arr. MATLAB computes the sum of the elements in arr and returns that result.

Python uses separate syntax for calling functions and indexing sequences. In Python, using round brackets means that a function should be executed and using square brackets will index a sequence:

```
In [1]: arr = [10, 20, 30]
In [2]: arr[0]
Out[2]: 10
In [3]: sum(arr)
Out[3]: 60
```

In this code, you are defining a Python list on input line 1. Python lists have some important distinctions from arrays in MATLAB and arrays from the NumPy package.

On the input line 2, you are displaying the value of the first element of the list with the indexing operation using square brackets. On input line 3, you are calling sum() using round brackets and passing in the list stored in arr. This results in the sum of the list elements being displayed on the last line. Notice that Python uses square brackets for indexing the list and round brackets for calling functions.

The First Index in a Sequence Is 0 in Python

In MATLAB, you can get the first value from an array by using 1 as the index. This style follows the natural numbering convention and starts how you would count the number of items in the sequence. You can try out the differences of MATLAB vs Python with this example:

```
>> arr = [10, 20, 30];
>> arr(1)
ans =
   10
>> arr(0)
```

Array indices must be positive integers or logical values.

In this code, you are creating an array with three numbers: 10, 20, and 30. Then you are displaying the value of the first element with the index 1, which is 10. Trying to access the zeroth element results in an error in MATLAB, as shown on the last two lines.

In Python, the index of the first element in a sequence is 0, not 1:

```
In [1]: arr = [10, 20, 30]
In [2]: arr[0]
Out[2]: 10
In [3]: arr[1]
Out[3]: 20
In [4]: a_string = "a string"
In [5]: a_string[0]
Out[5]:  'a'
In [6]: a_string[1]
Out[6]: ' '
```

In this code, you are defining arr as a Python list with three elements on input line 1. On input line 2, you are displaying the value of the first element of the list, which has the index 0. Then you are displaying the second element of the list, which has the index 1.

On input lines 4, 5, and 6, you are defining a_string with the contents "a string" and then getting the first and second elements of the string. Notice that the second element (character) of the string is a space. This demonstrates a general Python feature, that many variable types operate as sequences and can be indexed, including lists, tuples, strings, and arrays.

The Last Element of a Sequence Has Index -1 in Python

In MATLAB, you can get the last value from an array by using end as the index. This is really useful when you don't know how long an array is, so you don't know what number to access the last value.

Try out the differences in MATLAB vs Python with this example:

```
>> arr = [10, 20, 30];
>> arr(end)
ans =
   30
```

In this code, you are creating an array with three numbers, 10, 20, and 30. Then you are displaying the value of the last element with the index end, which is 30.

In Python, the last value in a sequence can be retrieved by using the index -1:

```
In [1]: arr = [10, 20, 30]
In [2]: arr[-1]
Out[2]: 30
```

In this code, you are defining a Python list with three elements on input line 1. On input line 2, you are displaying the value of the last element of the list, which has the index -1 and the value 30.

In fact, by using negative numbers as the index values you can work your way backwards through the sequence:

```
In [3]: arr[-2]
Out[3]: 20
In [4]: arr[-3]
Out[4]: 10
```

In this code, you are retrieving the second-to-last and third-to-last elements from the list, which have values of 20 and 10, respectively.

Exponentiation Is Done With ** in Python

In MATLAB, when you want to raise a number to a power you use the caret operator (^). The caret operator is a **binary operator** that takes two numbers. Other binary operators include addition (+), subtraction (-), multiplication (*), and division (/), among others. The number on the left of the caret is the base and the number on the right is the exponent.

Try out the differences of MATLAB vs Python with this example:

```
>> 10^2
ans =
   100
```

In this code, you are raising 10 to the power of 2 using the caret resulting an answer of 100.

In Python, you use two asterisks (**) when you want to raise a number to a power:

```
In [1]: 10 ** 2
Out[1]: 100
```

In this code, you are raising 10 to the power of 2 using two asterisks resulting an answer of 100.

Notice that there is no effect of including spaces on either side of the asterisks. In Python, the typical style is to have spaces on both sides of a binary operator.

The Length of a Sequence Is Found With len() in Python

In MATLAB, you can get the length of an array with length(). This function takes an array as the argument and returns back the size of the largest dimension in the array. You can see the basics of this function with this example:

```
>> length([10, 20, 30])
ans =
   3
>> length("a string")
ans =
   1
```

In this code, on the first input line you are finding the length of an array with 3 elements. As expected, length() returns an answer of 3. On the second input line, you are finding the length of the **string array** that contains one element. Notice that MATLAB implicitly creates a string array, even though you did not use the square brackets to indicate it is an array.

In Python, you can get the length of a sequence with len():

```
In [1]: len([10, 20, 30])
Out[1]: 3
In [2]: len("a string")
Out[2]: 8
```

In this code, on the input line 1 you are finding the length of a list with 3 elements. As expected, len() returns a length of 3. On input line 2, you are finding the length of a string as the input. In Python, strings are sequences and len() counts the number of characters in the string. In this case, a string has 8 characters.

Console Output Is Shown With print() in Python

In MATLAB, you can use disp(), fprintf(), and sprintf() to print the value of variables and other output to the console. In Python, print() serves a similar function as disp(). Unlike disp(), print() can send its output to a file similar to fprintf().

Python's print() will display any number of arguments passed to it, separating them by a space in the output. This is different from disp() in MATLAB, which only takes one argument, although that argument can be an array with multiple values. The following example shows how Python's print() can take any number of arguments, and each argument is separated by a space in the output:

In [1]: val_1 = 10

In [2]: val_2 = 20

In [3]: str_1 = “any number of arguments”

In [4]: print(val_1, val_2, str_1)

10 20 any number of arguments

In this code, the input lines 1, 2, and 3 define val_1, val_2, and str_1, where val_1 and val_1 are integers, and str_1 is a string of text. On input line 4, you are printing the three variables using print(). The output below this line the value of the three variables are shown in the console output, separated by spaces.

You can control the separator used in the output between arguments to print() by using the sep keyword argument:

In [5]: print(val_1, val_2, str_1, sep=”; “)

10; 20; any number of arguments

In this code, you are printing the same three variables but setting the separator to be a semicolon followed by a space. This separator is printed between the first and second and the second and third arguments, but not after the third argument. To control the character printed after the last value, you can use the end keyword argument to print():

In [6]: print(val_1, val_2, str_1, sep=”; “, end=”;”)

10; 20; any number of arguments;

In this code, you have added the end keyword argument to print(), setting it to print a semicolon after the last value. This is shown in the output on line below the input.

Like disp() from MATLAB, print() cannot directly control the output format of variables and relies on you to do the formatting. If you want more control over the format of the output, you should use f-strings or str.format(). In these strings, you can use very similar formatting style codes as fprintf() in MATLAB to format numbers:

In [7]: print(f”The value of val_1 = {val_1:8.3f}”)

The value of val_1 =   10.000

In [8]: # The following line will only work in Python 3.8

In [9]: print(f”The value of {val_1=} and {val_2=}”)

The value of val_1=10, and val_2=20

In this code, input line 7 includes an f-string, indicated by the f to start the string. This means that Python will substitute the value of any variables it encounters between {}, or curly braces, within the string. You can see that in the output, Python has replaced {val_1:8.3f} with a floating point number with 8 columns in the output and 3 digits of precision.

Input line 9 demonstrates a new feature in Python 3.8. If a variable name is immediately followed by an equals sign inside curly braces, the name of the variable and the value will be printed automatically.

You can take a deep dive into Python's print() by checking out The Ultimate Guide to Python Print.

You Will Probably See These, but You Can Learn Them When You Need To

You'll find examples of code that you'll probably see in the wild, but you can wait a little while to understand them if you want. These examples use some intermediate features in Python but are still in the core of how Python works.

Function Definitions Start With def and return Values in Python

In MATLAB, you can define a function by placing the function keyword at the start of a line. This is followed by the name of any output variables, an equals (=) sign, then the name of the function and any input arguments in parentheses. Within the the function you have to assign to any variables you specified in the definition line as outputs. A simple example MATLAB function is shown below:

```
1function [total] = addition(num_1,num_2)
2total = num_1 + num_2;
3end
```

In this code, you see the function definition on line 1. There is only one output variable, called total, for this function. The name of the function is addition and it takes two arguments, which will be assigned the names num_1 and num_2 in the function body. Line 2 is the implementation of the function. The value of total is set equal to the sum of num_1 and num_2. The last line of the function is the end keyword that tells the MATLAB interpreter the definition of the function has finished. To use this function in MATLAB, you should save it in a file called addition.m, matching the name of the function. Alternatively, it can be placed in file with other commands provided that the function definition is the last thing in the file and the file is *not* named addition.m. Then, you can run the function by typing the following code in the MATLAB console:

```
>> var_1 = 20;
>> var_2 = 10;
>> sum_of_vars = addition(var_1,var_2)
sum_of_vars =
  30
```

In this code, you have defined two variables called var_1 and var_2 that hold the values 20 and 10, respectively. Then you created a third variable called sum_of_vars that stores the output from addition(). Check out the *Variable explorer*, and you'll see that sum_of_vars has the value 30, as expected. Notice that the name sum_of_vars did not have to be the same name as the output variable used in the function definition, which was total.

MATLAB does not require a function to provide an output value. In this case, you would remove the output variable and the equals sign from the function definition. Modify your addition.m file so that the code looks like this:

```
1function addition(num_1,num_2)
2total = num_1 + num_2;
3end
```

The only change in this code from the earlier code is that you deleted the [total] = from line 1, the other lines are exactly the same. Now if you try to assign the result of calling this function to a variable, MATLAB will generate an error in the console:

```
>> var_1 = 20;
>> var_2 = 10;
>> sum_of_vars = addition(var_1,var_2);
Error using addition
Too many output arguments.
```

In this code, you defined the same two variables var_1 and var_2 as before and called addition() in the same way as before. However, since addition() no longer specifies an output variable, MATLAB generates an error message that there are too many output arguments. Clicking on the word addition will open the definition of the function for you to edit or view the source code to fix the problem. In Python, the def keyword starts a function definition. The def keyword must be followed by the name of the function and any arguments to the function inside parentheses, similar to MATLAB. The line with def must be ended with a colon (:).

Starting on the next line, the code that should be executed as part of the function must be indented one level. In Python, the function definition ends when a line of code starts at the same indentation level as the def keyword on the first line.

If your function returns some output back to the caller, Python does not require that you specify a name for an output variable. Instead, you use the return statement to send an output value from the function.

An equivalent function in Python to your first addition() example with an output variable is shown below:

```
1def addition(num_1, num_2):
2     total = num_1 + num_2
3     return total
```

In this code, you see the def keyword followed by the function name and the two arguments num_1 and num_2 on line 1. On line 2 you can see the creation of a new variable total to store the sum of num_1 and num_2, and on line 3 the value of total is returned to the point where this function was called. Notice that lines 2 and 3 are indented by 4 spaces because they make up the body of the function.

The variable that stores the sum of num_1 and num_2 can have any name, it doesn't have to be called total. In fact, you don't need to create a variable there at all. You can simplify your previous function definition by eliminating total and simply returning the value of num_1 + num_2:

```
1def addition(num_1, num_2):
2    return num_1 + num_1
```

Line 1 in this code is the same as it was before, you have only changed line 2 and deleted line 3. Line 2 now computes the value of num_1 + num_2 and returns that value back to the caller of the function. Line 2 is indented by 4 spaces because it makes up the body of the function.

To use this function in Python, you do not need to save it in a file with a special name. You can place the function definition in any Python file, at any point in the file. There is no restriction that the function definition has to be last. In fact, you can even define functions right from the console, which is not possible in MATLAB.

Open Spyder and in the Console pane type:

```
>>>
In [1]: def addition(num_1, num_2):
```

On this line of code you are creating the function definition. In the Spyder/IPython console, once you start a function definition and press Enter, the start of the line becomes three dots and the cursor is automatically indented. Now you can type the remainder of the function definition. You'll have to press Enter twice to complete the definition:

```
>>>
In [1]: def addition(num_1, num_2):
```

```
...:     return num_1 + num_2
...:
```

In this code, you have the definition of the function on the first line and the body of the function on the second line. The console automatically adds the ...: at the start of the lines to indicate these are **continuation lines** that apply to the function definition.

Once you've completed the definition, you can execute the function from the console as well. You should type this code:

```
In [2]: var_1 = 20
In [3]: var_2 = 10
In [4]: sum_of_vars = addition(var_1, var_2)
In [5]: sum_of_vars
Out[5]: 30
```

In this code, you first create two variables var_1 and var_2 that store the values you want to add together. Then, on input line 4, you assign sum_of_vars to the result that is returned from addition(). On input line 5, you are outputting the value of sum_of_vars to the console screen. This displays 30, the sum of 10 and 20.

In Python, if you do not explicitly put a return statement, your function will implicitly return the special value None. You should change your Python definition of addition() to see how this works. In the Spyder/IPython console, type the following:

```
In [6]: def addition(num_1, num_2):
   ...:     total = num_1 + num_2
   ...:
```

In this code, you have the same def line on input line 6. You have changed the first continuation line to assign the result of the addition to total instead of returning. Now you should see what happens when we execute this modified function:

```
In [7]: sum_of_vars = addition(var_1, var_2)
In [8]: sum_of_vars
In [9]:
```

In this code, on input line 7 you are assigning sum_of_vars to be the returned value from addition(). Then, on input line 8, you are showing the value of sum_of_vars on the console screen, just like before. This time though, there is

no output! By default, Python prints nothing when it outputs a variable whose value is None. You can double check the value of the sum_of_vars variable by looking at the *Variable explorer*. In the *Type* column, it should list NoneType, telling you that sum_of_vars is the special None value.

Functions Accept Positional and Keyword Arguments in Python

In MATLAB, functions have input arguments specified on the first line, in the function definition. When you call a function in MATLAB, you can pass from zero up to the number of arguments that are specified. In the body of the function, you can check the number of input arguments the caller actually passed to execute different code. This is useful when you want different arguments to have different meaning, like in the example below:

```
1function [result] = addOrSubtract(num_1,num_2,subtract)
2% ADDORSUBTRACT Add or subtract two value
3% RESULT = addOrSubtract(NUM_1,NUM_2) adds NUM_1 and NUM_2
together
4%
5% RESULT = addOrSubtract(NUM_1,NUM_2,true) subtracts NUM_2
from NUM_1
6
7    switch nargin
8        case 2
9            result = num_1 + num_2;
10        case 3
11            result = num_1 - num_2;
12        otherwise
13            result = 0;
14    end
15end
```

In this code, you are defining a function with three possible input arguments. On line 7, you are starting a switch/case block that determines how many input arguments were passed to the function by using the special variable nargin. This variable stores the actual number of arguments the caller passed into the function.

In your code above, you are defining three cases:

1. If the number of input arguments is 2, you are adding num_1 and num_2 together.

2. If the number of input arguments is 3, you are subtracting num_2 from num_1.
3. If fewer than 2 arguments are passed, the output will be 0.

If more than 3 arguments are passed, MATLAB will raise an error.

Now you should experiment with this function. Save the code above into a file called addOrSubtract.m and then on the MATLAB console, try the version with two input arguments:

```
>> addOrSubtract(10,20)
ans =
   30
```

In this code, you are calling addOrSubtract() with two arguments, so the arguments are added together, resulting in an answer of 30. Next, try calling addOrSubtract() with three arguments:

```
>>>
>> addOrSubtract(10,20,true)
ans =
  -10
```

In this code, you used three input arguments, and found that the second argument was subtracted from the first, resulting in an answer of -10. Third, try calling addOrSubtract() with one argument:

```
>> addOrSubtract(10)
ans =
    0
```

In this code, you used one input argument and found the answer was 0, because MATLAB only found one argument to the function and used the otherwise case. Finally, try calling addOrSubtract() with four arguments:

```
>> addOrSubtract(10,20,true,30)
Error using addOrSubtract
Too many input arguments.
```

In this code, you find that MATLAB raises an error because there were more input arguments passed than were defined in the function line.

There are four key takeaways from this example with MATLAB:

1. There is only one kind of argument in a function definition.
2. The meaning of an argument in the code is determined by its position in the function definition.

3. The maximum number of arguments that can be passed to a function is determined by the number of arguments specified in the function definition.
4. Any number of arguments up to the maximum can be passed by the caller.

In Python, there are two kinds of arguments you can specify when defining a function. These are **required** and **optional** arguments. The key difference between these is that required arguments must be passed when a function is called, while optional are given a default value in the function definition.

You can see the differences between these two styles in the next example:

```
1def add_or_subtract(num_1, num_2, subtract=False):
2   “””Add or subtract two numbers, depending on the value of subtract.”””
3    if subtract:
4         return num_1 - num_2
5    else:
6         return num_1 + num_2
```

In this code, you are defining a function called add_or_subtract() that has three arguments: num_1, num_2, and subtract. In the function definition, you can see the two types of arguments. The first two arguments, num_1 and num_2, are required arguments.

The third argument, subtract, has a default value assigned to it by specifying a value after an equals sign in the function definition. This means that when the function is called, passing a value for subtract is optional. If no value is passed, the default as defined in the function definition line will be used. In this case, the default value is False.

In the body of the function, you are testing the value of subtract with the if statement to determine whether addition or subtraction should be performed. If subtract is True, num_2 will be subtracted from num_1. Otherwise, if subtract is False, then num_1 will be added to num_2. In either case, the result of the arithmetic operation will be returned to the caller.

In addition to the two types of arguments you can use when **defining** a function, there are two kinds of arguments you can specify when **calling** a function. These are called **positional** and **keyword** arguments. You can see the difference between these in the following example. First, try passing only two arguments to the function:

```
In [1]: add_or_subtract(10, 20)
```

```
Out[1]: 30
```

In this code, you passed only two arguments to add_or_subtract(), 10 and 20. In this case, you passed these values as positional arguments, and the meaning of the arguments is defined by their position in the function call.

Since only the two required arguments were passed, subtract will take on the default value, which is False. Therefore, 10 and 20 will be added together, which you can see on the output line. Next, try passing a value for subtract:

```
In [2]: add_or_subtract(10, 20, False)
Out[2]: 30
In [3]: add_or_subtract(10, 20, True)
Out[3]: -10
```

In this code, you passed three arguments to add_or_subtract(), with two different values for the subtract argument. First, you passed False on input line 2. The result was the addition of 10 and 20. Then, you passed True on input line 3, resulting in the difference between 10 and 20, or -10.

In these examples, you saw that it is possible in Python to define default values for arguments to a function. This means when you call the function, any arguments with default values are optional and do not have to be passed. If no value is passed for any default arguments, the default value will be used. However, you *must* pass a value for every argument without a default value. Otherwise, Python will raise an error:

```
In [4]: add_or_subtract(10)
Traceback (most recent call last):
 File "<ipython-input-4-f9d1f2ae4494>", line 1, in <module>
   add_or_subtract(10)
TypeError: add_or_subtract() missing 1 required positional argument: 'num_2'
```

In this code, you have only passed one of the two required arguments to add_or_subtract(), so Python raises a TypeError. The error message tells you that you did not pass a value for num_2, because it does not have a default value.

In these last three example, you have used **positional** arguments, so which parameter is assigned to the variables in the function depends on the order they are passed. There is another method to pass arguments to functions in Python, called **keyword** arguments. To use keyword arguments, you specify the name of the argument in the function call:

```
In [5]: add_or_subtract(num_1=10, num_2=20, subtract=True)
```

Out[5]: -10

In this code, you have used keyword arguments for all three arguments to add_or_subtract(). Keyword arguments are specified by stating the argument name, then an equals sign, then the value that argument should have. One of the big advantages of keyword arguments is that they make your code more explicit. (As the *Zen of Python* says, explicit is better than implicit.)

However, they make the code somewhat longer, so it's up to your judgement when to use keyword arguments or not.

Another benefit of keyword arguments is that they can be specified in any order:

In [6]: add_or_subtract(subtract=True, num_2=20, num_1=10)

Out[6]: -10

In this code, you have specified the three arguments for add_or_subtract() as keyword arguments, but the order is different from in the function definition. Nonetheless, Python connects the right variables together because they are specified as keywords instead of positional arguments.

You can also mix positional and keyword arguments together in the same function call. If positional and keyword arguments are mixed together, the positional arguments *must* be specified first, before any keyword arguments:

In [7]: add_or_subtract(10, 20, subtract=True)

Out[7]: -10

In this code, you have specified the values for num_1 and num_2 using positional arguments, and the value for subtract using a keyword argument. This is probably the most common case of using keyword arguments, because it provides a good balance between being explicit and being concise.

Finally, there is one last benefit of using keyword arguments and default values. Spyder, and other IDEs, provide introspection of function definitions. This will tell you the names of all of the defined function arguments, which ones have default arguments, and the value of the default arguments. This can save you time and make your code easier and faster to read.

There Are No switch/case Blocks in Python

In MATLAB, you can use switch/case blocks to execute code by checking the value of a variable for equality with some constants. This type of syntax is quite useful when you know you want to handle a few discrete cases. Try out a switch/case block with this example:

```
num = 10;
switch num
   case 10
      disp("num is 10")
   case 20
      disp("num is 20")
   otherwise
      disp("num is neither 10 nor 20")
end
```

In this code, you start by defining num and setting it equal to 10 and on the following lines you test the value of num. This code will result in the output num is 10 being displayed on the console, since num is equal to 10.

This syntax is an interesting comparison of MATLAB vs Python because Python does not have a similar syntax. Instead, you should use an if/elif/else block:

```
num = 10
if num == 10:
   print("num is 10")
elif num == 20:
   print("num is 20")
else:
   print("num is neither 10 nor 20")
```

In this code, you start by defining num and setting it equal to 10. On the next several lines you are writing an if/elif/else block to check the different values that you are interested in.

Namespaces Are One Honking Great Idea in Python

In MATLAB, all functions are found in a single scope. MATLAB has a defined search order for finding functions within the current scope. If you define your own function for something that MATLAB already includes, you may get unexpected behavior.

As you saw in the *Zen of Python*, namespaces are one honking great idea. **Namespaces** are a way to provide different scopes for names of functions, classes, and variables. This means you have to tell Python which library has the function you want to use. This is a good thing, especially in cases where multiple libraries provide the same function.

For instance, the built-in math library provides a square root function, as does the more advanced NumPy library. Without namespaces, it would be more difficult to tell Python which square root function you wanted to use.

To tell Python where a function is located, you first have to import the library, which creates the namespace for that library's code. Then, when you want to use a function from the library, you tell Python which namespace to look in:

```
In [1]: import math
In [2]: math.sqrt(4)
Out[2]: 2.0
```

In this code, on input line 1 you imported the math library that is built-in to Python. Then, input line 2 computes the square root of 4 using the square root function from within the math library. The math.sqrt() line should be read as "from within math, find sqrt()."

The import keyword searches for the named library and binds the namespace to the same name as the library by default. You can read more about how Python searches for libraries in Python Modules and Packages – An Introduction.

You can also tell Python what name it should use for a library. For instance, it is very common to see numpy shortened to np with the following code:

```
In [3]: import numpy as np
In [4]: np.sqrt(4)
Out[4]: 2.0
```

In this code, input line 3 imports NumPy and tells Python to put the library into the np namespace. Then, whenever you want to use a function from NumPy, you use the np abbreviation to find that function. On input line 4, you are computing the square root of 4 again, but this time, using np.sqrt(). np.sqrt() should be read as "from within NumPy, find sqrt()."

There are two main caveats to using namespaces where you should be careful:

1. You should not name a variable with the same name as one of the functions built into Python. You can find a complete list of these functions in the Python documentation. The most common variable names that are also built-in functions and should not be used are dir, id, input, list, max, min, sum, str, type, and vars.
2. You should not name a Python file (one with the extension .py) with the same name as a library that you have installed. In other words, you should not create a Python file called math.py. This is because Python searches the current working directory first when it tries to

import a library. If you have a file called math.py, that file will be found before the built-in math library and you will probably see an AttributeError.

PYTHON VS MATLAB FOR MACHINE LEARNING

Machine learning is a branch of artificial intelligence that enables the systems to learn and improve their performance on their own by acclimatizing themselves to the experience received from inputs by the user over time.

This process of improving with experience happens without any explicit programs being executed for the same. The concept of machine learning is the focus on the progress of computer systems that can access a given set of data and then use it to learn and enhance itself automatically.

Of late two programming languages have seen much difference of opinion concerning their role in machine learning. Many individuals try to search MatLab vs Python for machine learning.

FACTORS OF PREFERENCES FOR UTILIZING PYTHON FOR MACHINE LEARNING

Free and open-source

Even though few of them are, actually, free and open-source, it's as yet one of the highlights of Python that makes it stand apart as a programming language. You can download Python for nothing, which implies that Python engineers can download its source code, cause adjustments to it to and even convey it. Python accompanies a broad assortment of libraries that help you to do your undertakings.

Magnificent Collection of Inbuilt Libraries

Python offers an immense number of in-assembled libraries that the Python advancement organizations can use for information control, information mining, and AI, for example,

- NumPy — it is used for the logical count.
- Scikit-learn — it is used for information mining and investigation which enhances Python's AI ease of use. Panda — this library offers engineers with superior structures and information examination apparatuses that assist them with lessening the venture usage time.
- SciPy — this library is used for cutting edge calculation.
- Pybrain — the developer utilizes this library for AI.

Moderate Learning Curve

Numerous individuals guarantee that Python is extremely easy to comprehend, and given the usefulness and versatility it offers, Python as a programming language is anything but difficult to learn and utilize. It centers around code clarity and is an adaptable and very much organized language. How hard Python is, relies upon you. For example, if a beginner is given acceptable examination material and a not too bad instructor, Python can without much of a stretch be comprehended. Indeed, even good Python engineers can instruct Python to a novice.

Universally useful programming language

What it implies is that Python can be utilized to construct pretty much anything. It is amazingly valuable for backend Web Development, Artificial Intelligence, Scientific Computing, and Data Analysis. Python is be utilized for web advancement, framework activities, Server and Administrative apparatuses, logical demonstrating, and can likewise be utilized by a few engineers to construct profitability devices, work area applications, and games.

Simple to coordinate

Python is being utilized as a coordination language in numerous spots, to stick the current parts together. Python is anything but difficult to coordinate with other lower-level dialects, for example, C, C++, or Java. Additionally, it is anything but difficult to consolidate a Python based-stack with information researcher's work, which permits it to bring productivity into creation.

Simple to make models

As we realize that Python is easy to learn and can create sites rapidly. Python requires less coding, which implies that you can make models and test your ideas rapidly and effectively in Python when contrasted with a few other programming dialects. Creating models spares engineers' time and diminishes your organization's general use also.

POINTS OF INTEREST IN USING MATLAB FOR MACHINE LEARNING

Extension For Preprocessing

Matlab gives scope for preprocessing datasets effectively with space explicit applications for sound, video, and picture information. Clients can picture, check, and repair issues before preparing the Deep Network Designer application to manufacture complex system models or adjust prepared systems for move learning.

Multi-Program Deployment

Matlab can utilize profound learning models wherever including CUDA, C code, endeavor frameworks, or the cloud. It gives an extraordinary presentation where a client can deliver code that supports upgraded libraries like Intel(MKL-DNN), NVIDIA (TensorRT, cuDNN), and (ARM Compute Library) to manufacture deplorable examples with elite surmising action.

Profound Learning Toolbox

Profound Learning Toolbox actualizes a system for making and performing profound neural systems with calculations, prepared models, and applications. A client can apply convolution neural systems and long momentary memory (LSTM) systems to give grouping and relapse on the picture, time-arrangement, and content information. Applications and plots bolster clients to picture actuation, alter organize structures, and screen arrangement progress

Interoperability

MATLAB underpins interoperability with other open-source profound learning systems, for example, ONNX. Clients can pick MATLAB for finding abilities and prebuilt purposes and applications which are not accessible in other programming dialects.

The final verdict

Well, we have discussed here the advantages of each language separately. You now have to weigh their advantages to see which one would you want to work with for your machine learning endeavors. Every language has got something different to offer. It depends on your expertise too to decide the answer for MatLab vs Python for machine learning.

MATLAB VS. PYTHON: THE KEY DIFFERENCES

NATURE

MATLAB is closed-source software and a proprietary commercial product. Thus, you need to purchase it to be able to use it. For every additional MATLAB toolbox you wish to install and run, you need to incur extra charges. The cost aspect aside, it is essential to note that since MATLAB is specially designed for MathWorks, its user base is quite limited. Also, if MathWorks were to ever go out of business, MATLAB would lose its industrial importance.

Unlike MATLAB, Python is an open-source programming language, meaning

it is entirely free. You can download and install Python and make alterations to the source code to best suit your needs. Due to this reason, Python enjoys a bigger fan following and user base. Naturally, the Python community is pretty extensive, with hundreds and thousands of developers contributing actively to enrich the language continually. As we stated earlier, Python offers numerous free packages, making it an appealing choice for developers worldwide.

Syntax

The most notable technical difference between MATLAB and Python lies in their syntax. While MATLAB treats everything as an array, Python treats everything as a general object.

For instance, in MATLAB, strings can either be arrays of strings or arrays of characters, but in Python, strings are denoted by a unique object called "str." Another example highlighting the difference between MATLAB and Python's syntax is that in MATLAB, a comment is anything that starts after the percent sign (%). In contrast, comments in Python typically follow the hash symbol (#).

IDE

MATLAB boasts of having an integrating development environment. It is a neat interface with a console located in the center where you can type commands, while a variable explorer lies on the right, you'll find a directory listing on the left.

On the other hand, Python does not include a default development environment. Users need to choose an IDE that fits their requirement specifications. Anaconda, a popular Python package, encompasses two different IDEs – Spyder and JupyterLab – that function as efficiently as the MATLAB IDE.

Tools

Programming languages are usually accompanied by a suite of specialized tools to support a wide range of user requirements, from modeling scientific data to building ML models. Integrated tools make the development process easier, quicker, and more seamless.

Although MATLAB does not have a host of libraries, its standard library includes integrated toolkits to cover complex scientific and computational challenges. The best thing about MATLAB toolkits is that experts develop them, rigorously tested, and well-documented for scientific and engineering operations. The toolkits are designed to collaborate efficiently and also integrate seamlessly with parallel computing environments and GPUs. Moreover, since they are updated together, you get fully-compatible versions of the tools.

As for Python, all of its libraries contain many useful modules for different programming needs and frameworks. Some of the best Python libraries include NumPy, SciPy, PyTorch, OpenCV Python, Keras, TensorFlow, Matplotlib, Theano, Requests, and NLTK. Being an open-source programming language, Python offers the flexibility and freedom to developers to design Python-based software tools (like GUI toolkits) for extending the capabilities of the language.

WHICH IS BETTER FOR DEEP LEARNING MATLAB OR PYTHON?

Deep Learning techniques has changed the field of computer vision significantly during the last decade, providing state-of-the-art solutions such as, object detection and image classification and opened the door for challenges and new problems, like image-to-image translation and visual question answering (VQA).

The success and popularization of Deep Learning in the field of computer vision and related areas are fostered, in great part, by the availability of rich tools, apps and frameworks in the Python and MATLAB ecosystems.

MATLAB is a robust computing environment for mathematical or technical computing operations involving the arrays, matrices, and linear algebra while, Python is a high-level launguage, general-purpose programming language designed for ease of use by human beings accomplishing all sorts of tasks.

MATLAB has scientific computing for a long while Python has evolved as an efficient programming language with the emergence of artificial intelligence, deep learning, and machine learning. Though which both are used to execute various data analysis and rendering tasks, there are some elementary differences.

MATLAB VS PYTHON

MATLAB was designed by Cleve Moler Matlab is also known as matrix laboratory as a multi-paradigm programming language developed by MathWorks. It helpful for matrix manipulation, Implementation of algorithms and interfacing the programs written in other programming languages. MATLAB Primarily used for numerical computing.

Whereas, Python was created by Guido van Rossum in 1991 and it is a high-level general-purpose Programming language. Python supports multiple paradigms such as Procedural, Functional programming and Object-Oriented Programming.

Python is the most widely used language in the modern machine learning research industry and academia. It is the number in which one language for natural language processing (NLP), computer vision (CV), and reinforcement learning and other available packages such as NLTK, OpenCV, OpenAI Gym, etc.

MATLAB VS PYTHON: THE KEY DIFFERENCES

Nature

MATLAB is a closed-source software and proprietary commercial product. Thus, you need to purchase the software to be able to use it. For every additional MATLAB toolbox you wish to install and run the software, you need to incur extra charges.

Python is an open-source programming language, meaning that it is entirely free. You can download and install Python from internet and make alterations to the source code for best suit your needs.

Syntax

The most notable and technical difference between MATLAB and Python lies in their syntax. While MATLAB treats everything as an array While, Python treats everything as a general object. For instance MATLAB, strings can either be arrays of strings or arrays of characters, but in Python, the strings are denoted by a unique object called "str."

IDE

MATLAB having an integrating development environment. It have a neat interface with a console located at the center where you can type commands, while a variable explorer lies on the right, you'll find a directory listing on the left side. On the other hand, Python does not have a default development environment. Users need to choose an IDE, that fits their requirement and specifications. Anaconda, one of the popular Python package, encompasses two different IDEs – Spyder and JupyterLab – their function as efficiently as of the MATLAB IDE.

Tools

MATLAB does not have a host library, it's a standard library includes integrated toolkits to cover the complex scientific and computational challenges. The best thing about MATLAB toolkits is that the experts develops rigorously an wil be tested and well-documented for scientific and engineering operations. The toolkits are designed to collaborate integrate seamlessly with parallel computing environments and GPUs. In Python, all of its libraries contain many useful modules in different programming and frameworks. Some of the best Python libraries include SciPy, PyTorch, NumPy, OpenCV Python, Keras, TensorFlow, Matplotlib, Theano, Requests, and NLTK. Being an open-source programming language, Python offers a flexibile freedom to developers to design based software tools (like GUI toolkits) for extending the capabilities of the language.

Graphics

MATLAB's capabile for signal processing and modeling in a graphical interface while, Python lacks a graphical interface that can perform these advanced functions. Overall, both MATLAB and Python have excellent tools. While one is designed for specific tasks (MATLAB) and another can perform a wide variety of generic operations.

WHICH IS FASTER MATLAB VS PYTHON?

The python results the fundamentally the same which indicates the statsmodels OLS functions are exceptionally advance. Matlab shows a huge speed and exhibits how local direct variables are based in the math code is favored for speed. Overall, Matlab is around multiple times faster than python.

Is Python better than Matlab?

Obviously, Matlab has its on points of interest as well: It has a strong measured functions. Simulink is an item in which there is nothing worth mentioning elective yet. It may be simpler for beginners and the bundle incorporates all you need, while in Python you have to introduce additional bundles and an IDE.

Can Python Replace Matlab?

Each of them are marginally more effective than the other, yet when all is said in done Python can be a replacement for MATLAB. Most of the applications in MATLAB toolbox can be found in Python libraries or viably duplicated. Python is more versatile than MATLAB as a general language and it's indications of better execution.

GENERIC PROGRAMMING TASKS

Generic programming tasks are problems that are not specific to any application. For example, reading and saving data to a file, preprocessing CSV or text file, writing scripts or functions for basic problems like counting the number of occurrences of an event, plotting data, performing basic statistical tasks such as computing the mean, median, standard deviation, etc.

MACHINE LEARNING

This is the area where Python and R have a clear advantage over Matlab. They both have access to numerous libraries and packages for both classical (random forest, regression, SVM, etc.) and modern (deep learning and neural networks such as CNN, RNN, etc.) machine learning models. However, Python is the most widely

used language for modern machine learning research in industry and academia. It is the number one language for natural language processing (NLP), computer vision (CV), and reinforcement learning, thanks to many available packages such as NLTK, OpenCV, OpenAI Gym, etc.

Python is also the number one language for most research or work involving neural networks and deep learning, thanks to many available libraries and platforms such as Tensorflow, Pytorch, Keras, etc.

Probabilistic Graphical Modeling (PGM)

Probabilistic graphical models are a class of models for inference and learning on graphs. They are divided into undirected graphical models or sometimes referred to as Markov random field and directed graphical models or Bayesian network.

Python, R, and Matlab all have support for PGM. However, Python and R are outperforming Matlab in this area. Matlab, thanks to the BNT (Bayesian Network Toolbox) by Kevin Murphy, has support for the static and dynamic Bayesian network. The Matlab standard library (hmmtrain) supports the discrete hidden Markov model (HMM), a well-known class of dynamic Bayesian networks. Matlab also supports the conditional random field (CRF) thanks to crfChain (by Mark Schmidt and Kevin Swersky) and UGM by Mark Schmidt.

Python has excellent support for PGM thanks to hmmlearn (Full support for discrete and continuous HMM), pomegranate, bnlearn (a wrapper around the bnlearn in R), pypmc, bayespy, pgmpy, etc. It also has better support for CRF through sklearn-crfsuite.

R has excellent support for PGM. It has numerous stunning packages and libraries such as bnlearn, bnstruct, depmixS4, etc. The support for CRF is done through the CRF and crfsuite packages.

Causal Inference

R by far is the most widely used language in causal inference research (along with SAS and STATA; however, R is free while the other two are not). It has numerous libraries such as bnlearn, bnstruct for causal discovery (structure learning) to learn the DAG (directed acyclic graph) from data. It has libraries and functions for various techniques such as outcome regression, IPTW, g-estimation, etc.

Python also, thanks to the dowhy package by Microsoft research, is capable of combining the Pearl causal network framework with the Rubin potential outcome model and provides an easy interface for causal inference modeling.

Time-Series Analysis

R is also the strongest and by far the most widely used language for time series analysis and forecasting. Numerous books have been written about time series forecasting using R. There are many libraries to implement algorithms such as ARIMA, Holt-Winters, exponential smoothing. For example, the forecast package by Rob Hyndman is the most used package for time series forecasting. Python, thanks to neural networks, especially the LSTM, receives lots of attention in time series forecasting [1]. Furthermore, the Prophet package by Facebook written in both R and Python provides excellent and automated support for time series analysis and forecasting.

Signal Processing and Digital Communication

This is the area where Matlab is the strongest and is used often in research and industry. Matlab communications toolbox provides all functionalities needed to implement a complete communication system. It has functionalities to implement all well-known modulation schemes, channel and source coding, equalizer, and necessary decoding and detection algorithms in the receiver. The DSP system toolbox provides all functionalities to design IIR (Infinite Impulse Response), FIR (Finite Impulse Response), and adaptive filters. It has complete support for FFT (Fast Fourier Transform), IFFT, wavelet, etc.

Python, although is not as capable as Matlab in this area but has support for digital communication algorithms through CommPy and Komm packages.

Control and Dynamical System

Matlab is still the most widely used language for implementing the control and dynamical system algorithms thanks to the control system toolbox. It has extensive supports for all well-known methods such as PID controller, state-space design, root locus, transfer function, pole-zero diagrams, Kalman Filter, and many more. However, the main strength of Matlab is coming from its excellent and versatile graphical editor Simulink. Simulink lets you simulate the real-world system using drag and drop blocks (It is similar to the LabView). The Simulink output can then be imported to Matlab for further analysis. Python has support for control and dynamical system through the control and dynamical systems library.

Optimization and Numerical Analysis

All three programming languages have excellent support for optimization problems such as linear programming (LP), convex optimization, nonlinear optimization with and without constraint.

The support for optimization and numerical analysis in Matlab is done through the optimization toolbox. This supports linear programming (LP), mixed-integer linear programming (MILP), quadratic programming (QP), second-order cone programming (SOCP), nonlinear programming (NLP), constrained linear least squares, nonlinear least squares, nonlinear equations, etc. CVX is another strong package in Matlab written by Stephen Boys and his Ph.D. student for convex optimization. Python supports optimization through various packages such as CVXOPT, pyOpt (Nonlinear optimization), PuLP(Linear Programming), and CVXPY (python version of CVX for convex optimization problems). R supports convex optimization through CVXR (Similar to CVX and CVXPY), optimx (quasi-Newton and conjugate gradient method), and ROI (linear, quadratic, and conic optimization problems).

Web Development

This is an area where Python outperforms R and Matlab by a large margin. Actually, neither R nor Matlab are used for any web development design.

Python, thanks to Django and Flask, is a compelling language for backend development. Many existing websites, such as Google, Pinterest, and Instagram, use Python as part of their backend development.

Django is a full-stack platform that gives you everything you need right off the box (Battery-included). It also has support for almost all well-known databases. On the other hand, Flask is a lightweight platform that is mainly used to design less complex websites.

Pros and Cons of Each Language

This chapter will discuss the cons and pros of each programming language and summarize.

Matlab

Advantage:

- Many wonderful libraries and the number one choice in signal processing, communication system, and control theory.
- Simulink: One of the best toolboxes in MATLAB is used extensively in control and dynamical system applications.
- Lots of available and robust packages for optimization, control, and numerical analysis.
- Nice toolbox for graphical work (Lets you plot beautiful looking graphs) and inherent support for matrix and vector manipulation.
- Easy to learn and has a user-friendly interface.

Disadvantage:

- Proprietary and not free or open-source, which makes it very hard for collaboration.
- Lack of good packages and libraries for machine learning, AI, time series analysis, and causal inference.
- Limited in terms of functionality: cannot be used for web development and app design.
- Not object-oriented language.
- Smaller user community compared to Python.

PYTHON

Advantage:

- Many wonderful libraries in machine learning, AI, web development, and optimization.
- Number one language for deep learning and machine learning in general.
- Open-source and free.
- A large community of users across GitHub, Stackoverflow, and ...
- It can be used for other applications besides engineering, unlike MATLAB. For example, GUI (Graphical User Interface) development using Tkinter and PyQt.

- Object-oriented language.
- Easy to learn and user-friendly syntax.

Disadvantage:

- Lack of good packages for signal processing and communication (still behind for engineering applications).
- Steeper learning curve than MATLAB since it is an object-oriented programming(OOP) language and is harder to master.
- Requires more time and expertise to setup and install the working environment.

R

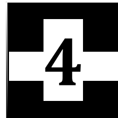
Advantage:

- So many wonderful libraries in statistics and machine learning.
- Open-source and free.
- Number one language for time series analysis, causal inference, and PGM.
- A large community of researchers, especially in academia.
- Ability to create web applications, for example, through the Shiney app.

Disadvantage:

- Slower compared to Python and Matlab.
- More limited scope in terms of applications compared to Python. (Cannot be used for game development or cannot be as a backend for web developments)
- Not object-oriented language.
- Lack of good packages for signal processing and communication (still behind for engineering applications).
- Smaller user communities compared to Python.
- Harder and not user-friendly compared to Python and Matlab.

To summarize, Python is the most popular language for machine learning, AI, and web development while it provides excellent support for PGM and optimization. On the other hand, Matlab is a clear winner for engineering applications while it has lots of good libraries for numerical analysis and optimization. The biggest disadvantage of Matlab is that it is not free or open-source. R is a clear winner for time series analysis, causal inference, and PGM. It also has excellent support for machine learning and data science applications.



Gradient Descent in Machine Learning

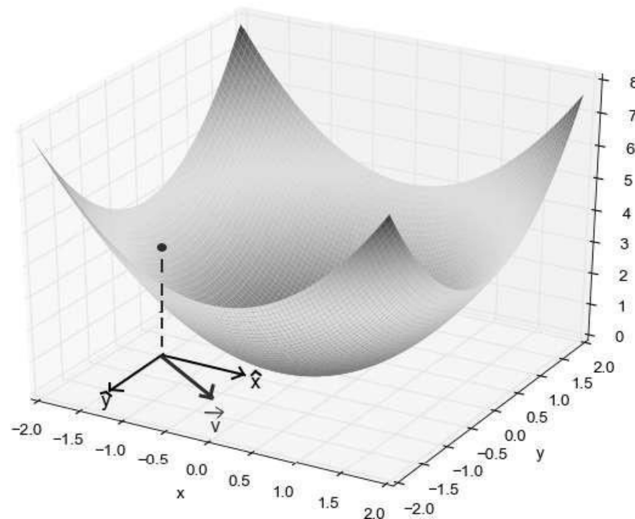
Gradient Descent is known as one of the most commonly used optimization algorithms to train machine learning models by means of minimizing errors between actual and expected results. Further, gradient descent is also used to train Neural Networks.

In mathematical terminology, Optimization algorithm refers to the task of minimizing/maximizing an objective function f(x) parameterized by x. Similarly, in machine learning, optimization is the task of minimizing the cost function parameterized by the model's parameters. The main objective of gradient descent is to minimize the convex function using iteration of parameter updates. Once these machine learning models are optimized, these models can be used as powerful tools for Artificial Intelligence and various computer science applications.

In this chapter on Gradient Descent in Machine Learning, we will learn in detail about gradient descent, the role of cost functions specifically as a barometer within Machine Learning, types of gradient descents, learning rates, etc.

PLOTTING THE GRADIENT DESCENT ALGORITHM

When we have a single parameter (theta), we can plot the dependent variable cost on the y-axis and theta on the x-axis. If there are two parameters, we can go with a 3-D plot, with cost on one axis and the two parameters (thetas) along the other two axes.



Cost along z-axis and parameters(thetas) along x-axis and y-axis

It can also be visualized by using **Contours.** This shows a 3-D plot in two dimensions with parameters along both axes and the response as a contour. The value of the response increases away from the center and has the same value along with the rings. The response is directly proportional to the distance of a point from the center (along a direction).

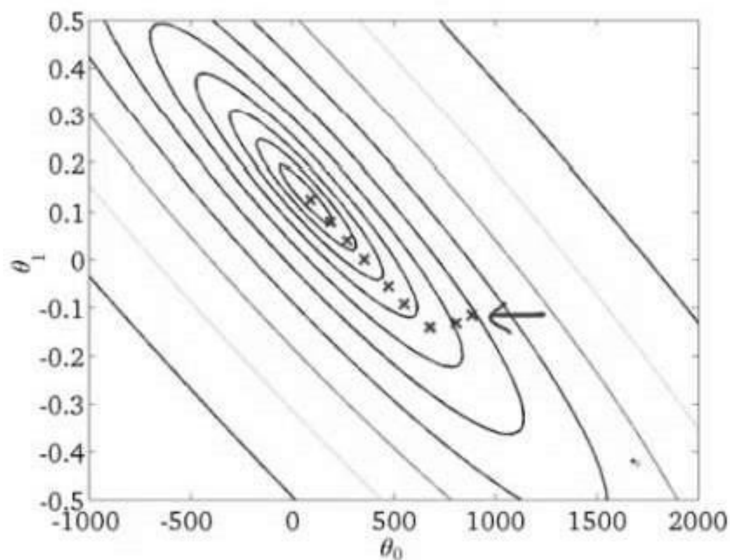


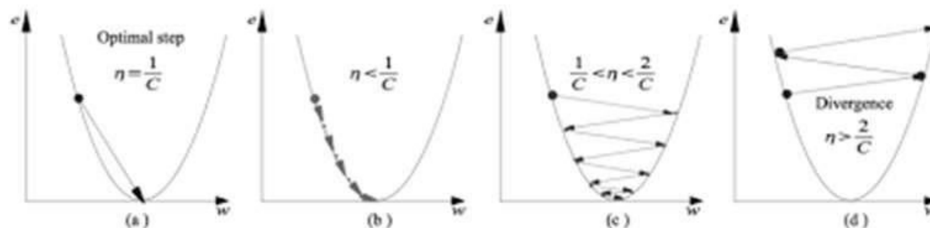
Fig. Gradient descent using Contour Plot.

ALPHA – THE LEARNING RATE

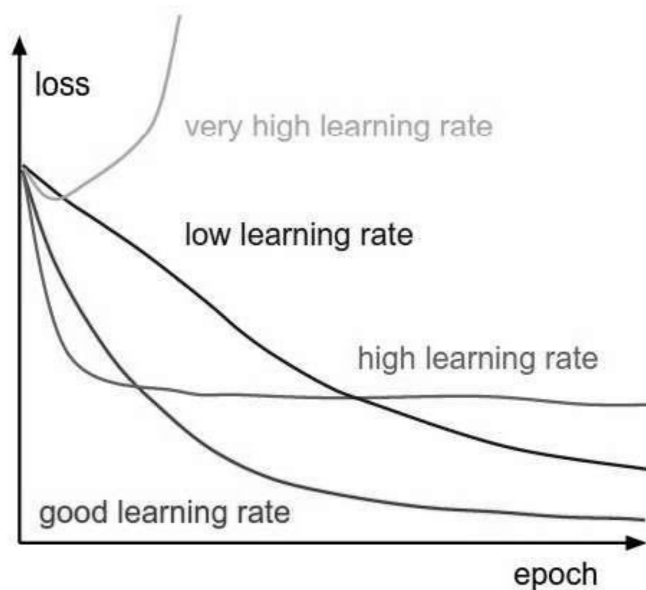
We have the direction we want to move in, now we must decide the size of the step we must take.

***It must be chosen carefully to end up with local minima.**

- If the learning rate is too high, we might **OVERSHOOT** the minima and keep bouncing, without reaching the minima
- If the learning rate is too small, the training might turn out to be too long



1. a) Learning rate is optimal, model converges to the minimum
2. b) Learning rate is too small, it takes more time but converges to the minimum
3. c) Learning rate is higher than the optimal value, it overshoots but converges (1/C < ç <2/C)
4. d) Learning rate is very large, it overshoots and diverges, moves away from the minima, performance decreases on learning



Local Minima

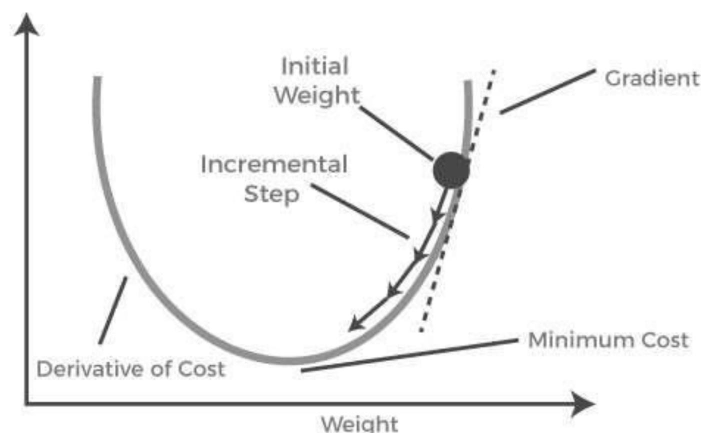
The cost function may consist of many minimum points. The gradient may settle on any one of the minima, which depends on the initial point (i.e initial parameters(theta)) and the learning rate. Therefore, the optimization may converge to different points with different starting points and learning rate.

GRADIENT DESCENT OR STEEPEST DESCENT

Gradient descent was initially discovered by **”Augustin-Louis Cauchy”** in mid of 18th century. ***Gradient Descent is defined as one of the most commonly used iterative optimization algorithms of machine learning to train the machine learning and deep learning models. It helps in finding the local minimum of a function.***

The best way to define the local minimum or local maximum of a function using gradient descent is as follows:

- o If we move towards a negative gradient or away from the gradient of the function at the current point, it will give the **local minimum** of that function.
- o Whenever we move towards a positive gradient or towards the gradient of the function at the current point, we will get the **local maximum** of that function.



This entire procedure is known as Gradient Ascent, which is also known as steepest descent. ***The main objective of using a gradient descent algorithm is to minimize the cost function using iteration.*** To achieve this goal, it performs two steps iteratively:

- o Calculates the first-order derivative of the function to compute the gradient or slope of that function.

- o Move away from the direction of the gradient, which means slope increased from the current point by alpha times, where Alpha is defined as Learning Rate. It is a tuning parameter in the optimization process which helps to decide the length of the steps.

What is Cost-function?

The cost function is defined as the measurement of difference or error between actual values and expected values at the current position and present in the form of a single real number. It helps to increase and improve machine learning efficiency by providing feedback to this model so that it can minimize error and find the local or global minimum.

Further, it continuously iterates along the direction of the negative gradient until the cost function approaches zero. At this steepest descent point, the model will stop learning further. Although cost function and loss function are considered synonymous, also there is a minor difference between them.

The slight difference between the loss function and the cost function is about the error within the training of machine learning models, as loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.

How does Gradient Descent work?

Before starting the working principle of gradient descent, we should know some basic concepts to find out the slope of a line from linear regression. The equation for simple linear regression is given as:

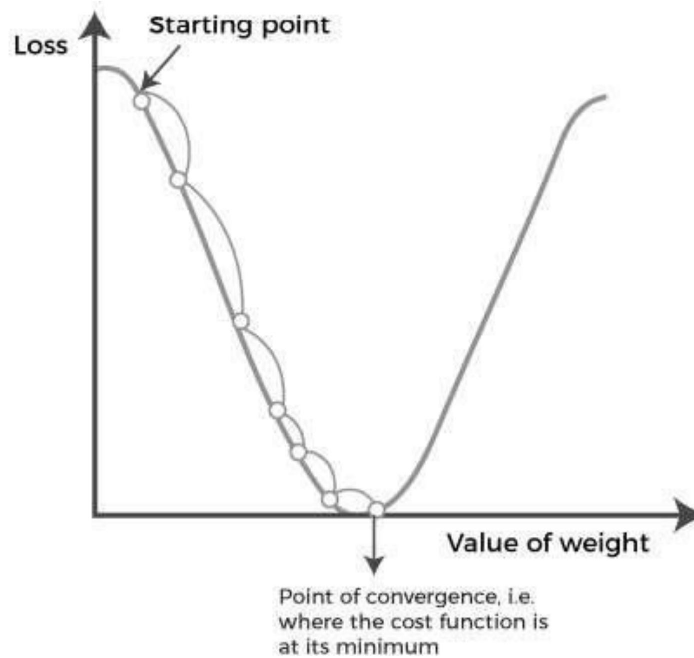
1. Y=mX+c

Where ‘m’ represents the slope of the line, and ‘c’ represents the intercepts on the y-axis.

The starting point is used to evaluate the performance as it is considered just as an arbitrary point. At this starting point, we will derive the first derivative or slope and then use a tangent line to calculate the steepness of this slope. Further, this slope will inform the updates to the parameters (weights and bias).

The slope becomes steeper at the starting point or arbitrary point, but whenever new parameters are generated, then steepness gradually reduces, and at the lowest point, it approaches the lowest point, which is called **a point of convergence.**

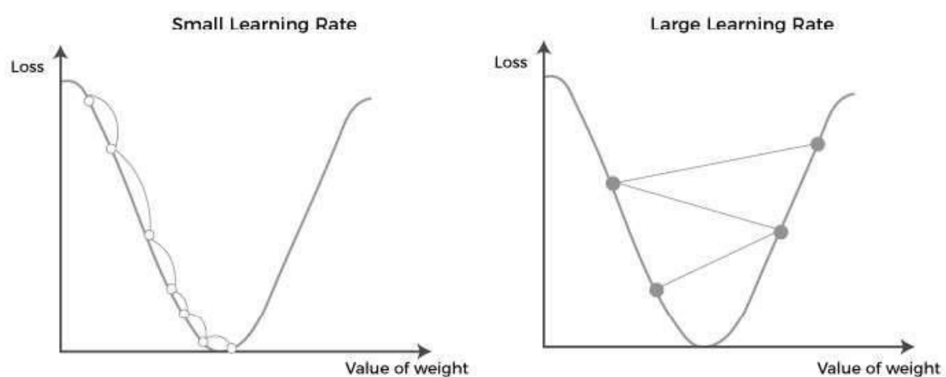
The main objective of gradient descent is to minimize the cost function or the error between expected and actual. To minimize the cost function, two data points are required:



Direction & Learning Rate

These two factors are used to determine the partial derivative calculation of future iteration and allow it to the point of convergence or local minimum or global minimum.

LEARNING RATE



It is defined as the step size taken to reach the minimum or lowest point. This is typically a small value that is evaluated and updated based on the behavior of

the cost function. If the learning rate is high, it results in larger steps but also leads to risks of overshooting the minimum. At the same time, a low learning rate shows the small step sizes, which compromises overall efficiency but gives the advantage of more precision.

TYPES OF GRADIENT DESCENT

Based on the error in various training models, the Gradient Descent learning algorithm can be divided into **Batch gradient descent, stochastic gradient descent, and mini-batch gradient descent.** Let's understand these different types of gradient descent:

Batch Gradient Descent:

Batch gradient descent (BGD) is used to find the error for each point in the training set and update the model after evaluating all training examples. This procedure is known as the training epoch. In simple words, it is a greedy approach where we have to sum over all examples for each update.

Advantages of Batch gradient descent:

- o It produces less noise in comparison to other gradient descent.
- o It produces stable gradient descent convergence.
- o It is Computationally efficient as all resources are used for all training samples.

Stochastic gradient descent

Stochastic gradient descent (SGD) is a type of gradient descent that runs one training example per iteration. Or in other words, it processes a training epoch for each example within a dataset and updates each training example's parameters one at a time. As it requires only one training example at a time, hence it is easier to store in allocated memory. However, it shows some computational efficiency losses in comparison to batch gradient systems as it shows frequent updates that require more detail and speed. Further, due to frequent updates, it is also treated as a noisy gradient. However, sometimes it can be helpful in finding the global minimum and also escaping the local minimum.

Advantages of Stochastic gradient descent:

In Stochastic gradient descent (SGD), learning happens on every example, and it consists of a few advantages over other gradient descent.

- o It is easier to allocate in desired memory.
- o It is relatively fast to compute than batch gradient descent.
- o It is more efficient for large datasets.

MiniBatch Gradient Descent:

Mini Batch gradient descent is the combination of both batch gradient descent and stochastic gradient descent. It divides the training datasets into small batch sizes then performs the updates on those batches separately. Splitting training datasets into smaller batches make a balance to maintain the computational efficiency of batch gradient descent and speed of stochastic gradient descent. Hence, we can achieve a special type of gradient descent with higher computational efficiency and less noisy gradient descent.

Advantages of Mini Batch gradient descent:

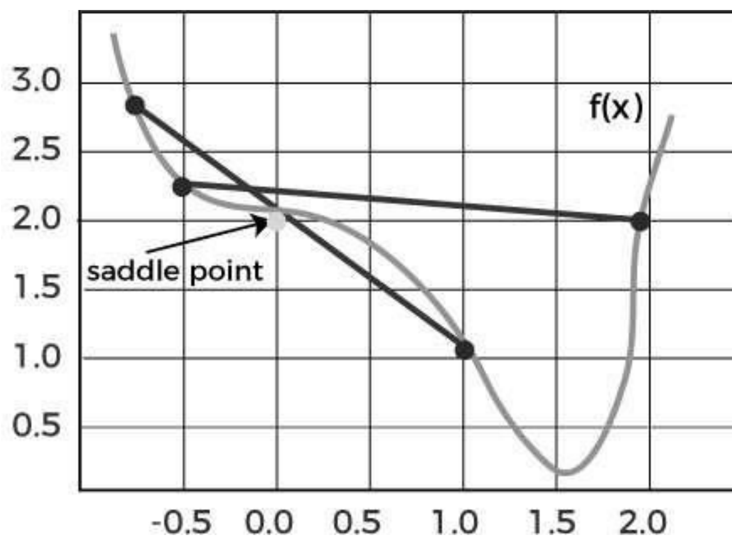
- o It is easier to fit in allocated memory.
- o It is computationally efficient.
- o It produces stable gradient descent convergence.

CHALLENGES WITH THE GRADIENT DESCENT

Although we know Gradient Descent is one of the most popular methods for optimization problems, it still also has some challenges. There are a few challenges as follows:

Local Minima and Saddle Point:

For convex problems, gradient descent can find the global minimum easily, while for non-convex problems, it is sometimes difficult to find the global minimum, where the machine learning models achieve the best results.



Whenever the slope of the cost function is at zero or just close to zero, this model stops learning further. Apart from the global minimum, there occur some scenarios that can show this slop, which is saddle point and local minimum. Local minima generate the shape similar to the global minimum, where the slope of the cost function increases on both sides of the current points.

In contrast, with saddle points, the negative gradient only occurs on one side of the point, which reaches a local maximum on one side and a local minimum on the other side. The name of a saddle point is taken by that of a horse's saddle.

The name of local minima is because the value of the loss function is minimum at that point in a local region. In contrast, the name of the global minima is given so because the value of the loss function is minimum there, globally across the entire domain the loss function.

Vanishing and Exploding Gradient

In a deep neural network, if the model is trained with gradient descent and backpropagation, there can occur two more issues other than local minima and saddle point.

Vanishing Gradients:

Vanishing Gradient occurs when the gradient is smaller than expected. During backpropagation, this gradient becomes smaller that causing the decrease in the learning rate of earlier layers than the later layer of the network. Once this happens, the weight parameters update until they become insignificant.

Exploding Gradient:

Exploding gradient is just opposite to the vanishing gradient as it occurs when the Gradient is too large and creates a stable model. Further, in this scenario, model weight increases, and they will be represented as NaN. This problem can be solved using the dimensionality reduction technique, which helps to minimize complexity within the model.

GRADIENT DESCENT

Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost).

Gradient descent is best used when the parameters cannot be calculated analytically (e.g. using linear algebra) and must be searched for by an optimization algorithm.

Intuition for Gradient Descent

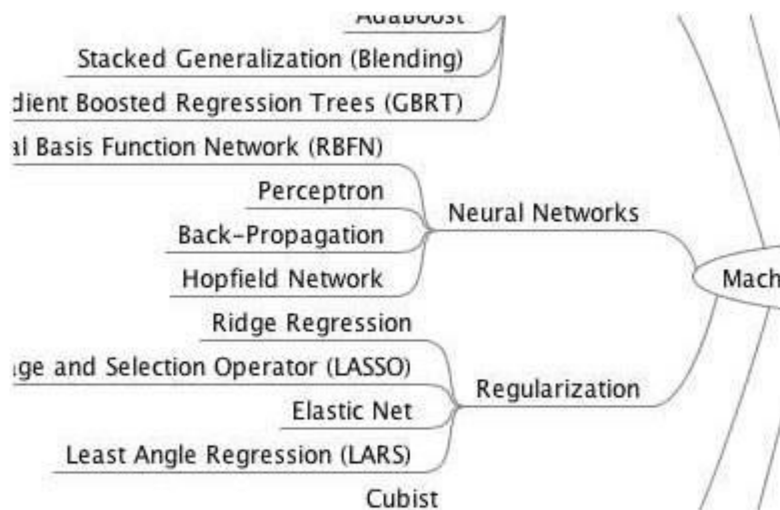
Think of a large bowl like what you would eat cereal out of or store fruit in. This bowl is a plot of the cost function (f).

A random position on the surface of the bowl is the cost of the current values of the coefficients (cost).

The bottom of the bowl is the cost of the best set of coefficients, the minimum of the function. The goal is to continue to try different values for the coefficients, evaluate their cost and select new coefficients that have a slightly better (lower) cost.

Repeating this process enough times will lead to the bottom of the bowl and you will know the values of the coefficients that result in the minimum cost.

GET YOUR FREE ALGORITHMS MIND MAP



Sample of the handy machine learning algorithms mind map.

I've created a handy mind map of 60+ algorithms organized by type.

GRADIENT DESCENT PROCEDURE

The procedure starts off with initial values for the coefficient or coefficients for the function. These could be 0.0 or a small random value.

coefficient = 0.0

The cost of the coefficients is evaluated by plugging them into the function and calculating the cost.

cost = f(coefficient)

or

cost = evaluate(f(coefficient))

The derivative of the cost is calculated. The derivative is a concept from calculus and refers to the slope of the function at a given point. We need to know the slope so that we know the direction (sign) to move the coefficient values in order to get a lower cost on the next iteration.

delta = derivative(cost)

Now that we know from the derivative which direction is downhill, we can now update the coefficient values. A learning rate parameter (alpha) must be specified that controls how much the coefficients can change on each update.

coefficient = coefficient – (alpha * delta)

This process is repeated until the cost of the coefficients (cost) is 0.0 or close enough to zero to be good enough.

You can see how simple gradient descent is. It does require you to know the gradient of your cost function or the function you are optimizing, but besides that, it's very straightforward.

BATCH GRADIENT DESCENT FOR MACHINE LEARNING

The goal of all supervised machine learning algorithms is to best estimate a target function (f) that maps input data (X) onto output variables (Y). This describes all classification and regression problems.

Some machine learning algorithms have coefficients that characterize the algorithms estimate for the target function (f). Different algorithms have different representations and different coefficients, but many of them require a process of optimization to find the set of coefficients that result in the best estimate of the target function.

Common examples of algorithms with coefficients that can be optimized using gradient descent are Linear Regression and Logistic Regression.

The evaluation of how close a fit a machine learning model estimates the target function can be calculated a number of different ways, often specific to the machine learning algorithm. The cost function involves evaluating the coefficients in the machine learning model by calculating a prediction for the model for each training instance in the dataset and comparing the predictions to the actual output values and calculating a sum or average error (such as the Sum of Squared Residuals or SSR in the case of linear regression).

From the cost function a derivative can be calculated for each coefficient so that it can be updated using exactly the update equation described above.

The cost is calculated for a machine learning algorithm over the entire training dataset for each iteration of the gradient descent algorithm. One iteration of the algorithm is called one batch and this form of gradient descent is referred to as batch gradient descent.

Batch gradient descent is the most common form of gradient descent described in machine learning.

STOCHASTIC GRADIENT DESCENT FOR MACHINE LEARNING

Gradient descent can be slow to run on very large datasets.

Because one iteration of the gradient descent algorithm requires a prediction for each instance in the training dataset, it can take a long time when you have many millions of instances.

In situations when you have large amounts of data, you can use a variation of gradient descent called stochastic gradient descent.

In this variation, the gradient descent procedure described above is run but the update to the coefficients is performed for each training instance, rather than at the end of the batch of instances.

The first step of the procedure requires that the order of the training dataset is randomized. This is to mix up the order that updates are made to the coefficients. Because the coefficients are updated after every training instance, the updates will be noisy jumping all over the place, and so will the corresponding cost function. By mixing up the order for the updates to the coefficients, it harnesses this random walk and avoids it getting distracted or stuck.

The update procedure for the coefficients is the same as that above, except the cost is not summed over all training patterns, but instead calculated for one training pattern.

The learning can be much faster with stochastic gradient descent for very large training datasets and often you only need a small number of passes through the dataset to reach a good or good enough set of coefficients, e.g. 1-to-10 passes through the dataset.

Tips for Gradient Descent

This chapter lists some tips and tricks for getting the most out of the gradient descent algorithm for machine learning.

- **Plot Cost versus Time**: Collect and plot the cost values calculated by the algorithm each iteration. The expectation for a well performing gradient

descent run is a decrease in cost each iteration. If it does not decrease, try reducing your learning rate.

- **Learning Rate**: The learning rate value is a small real value such as 0.1, 0.001 or 0.0001. Try different values for your problem and see which works best.
- **Rescale Inputs**: The algorithm will reach the minimum cost faster if the shape of the cost function is not skewed and distorted. You can achieved this by rescaling all of the input variables (X) to the same range, such as [0, 1] or [-1, 1].
- **Few Passes**: Stochastic gradient descent often does not need more than 1-to-10 passes through the training dataset to converge on good or good enough coefficients.
- **Plot Mean Cost**: The updates for each training dataset instance can result in a noisy plot of cost over time when using stochastic gradient descent. Taking the average over 10, 100, or 1000 updates can give you a better idea of the learning trend for the algorithm.

GRADIENT DESCENT ALGORITHM AND ITS VARIANTS

Gradient Descent is an optimization algorithm used for minimizing the cost function in various machine learning algorithms. It is basically used for updating the parameters of the learning model.

Types of gradient Descent:

1. **Batch Gradient Descent:** This is a type of gradient descent which processes all the training examples for each iteration of gradient descent. But if the number of training examples is large, then batch gradient descent is computationally very expensive. Hence if the number of training examples is large, then batch gradient descent is not preferred. Instead, we prefer to use stochastic gradient descent or mini-batch gradient descent.
2. **Stochastic Gradient Descent:** This is a type of gradient descent which processes 1 training example per iteration. Hence, the parameters are being updated even after one iteration in which only a single example has been processed. Hence this is quite faster than batch gradient descent. But again, when the number of training examples is large, even then it processes only one example which can be additional overhead for the system as the number of iterations will be quite large.

3. **Mini Batch gradient descent:** This is a type of gradient descent which works faster than both batch gradient descent and stochastic gradient descent. Here *b* examples where $b<m$ are processed per iteration. So even if the number of training examples is large, it is processed in batches of b training examples in one go. Thus, it works for larger training examples and that too with lesser number of iterations.

Variables used:

Let m be the number of training examples.

Let n be the number of features.

Note: if b == m, then mini batch gradient descent will behave similarly to batch gradient descent.

Algorithm for batch gradient descent :

Let h (x) be the hypothesis for linear regression. Then, the cost function is given by:

Let represents the sum of all training examples from i=1 to m.

J_{train}(▪) = (1/2m) ▪(h ($x^{(i)}$) - $y^{(i)})^2$

Repeat {

▪j = ▪j - q (learning rate/m) * ▪(h ($x^{(i)}$) - $y^{(i)})x^{(i)}_j$

For every j =0 …n

}

Where $x_j^{(i)}$ Represents the jth feature of the ith training example. So if *m* is very large(e.g. 5 million training samples), then it takes hours or even days to converge to the global minimum.

That's why for large datasets, it is not recommended to use batch gradient descent as it slows down the learning.

Algorithm for stochastic gradient descent:

1) Randomly shuffle the data set so that the parameters can be trained evenly for each type of data.

2) As mentioned above, it takes into consideration one example per iteration.

Hence,

Let ($x^{(i)}$,$y^{(i)}$) be the training example

Cost(▪, ($x^{(i)}$,$y^{(i)}$)) = (1/2) Ó(h▪($x^{(i)}$) - $y^{(i)})^2 J_{train}$(▪)

= (1/m) Ó Cost(▪, ($x^{(i)}$,$y^{(i)}$))

Repeat {

For i=1 to m{

▪$_j$ = ▪$_j$ - q (learning rate) * ▪(h ($x^{(i)}$) - $y^{(i)}$)$x_j^{(i)}$For every j =0 …n

}

}

ALGORITHM FOR MINI BATCH GRADIENT DESCENT

Say b be the no of examples in one batch, where b < m.

Assume b = 10, m = 100;

Note: However we can adjust the batch size. It is generally kept as power of 2. The reason behind it is because some hardware such as GPUs achieve better run time with common batch sizes such as power of 2.

Repeat {

For i=1,11, 21,…..,91

Let Ó be the summation from i to i+9 represented by k.

▪$_j$ = ▪$_j$ - p øß¢ (learning rate/size of (b)) * ▪(h ($x^{(k)}$) - $y^{(k)}$)$x_j^{(k)}$ For every j =0 …n

}

Convergence trends in different variants of Gradient Descents

In case of Batch Gradient Descent, the algorithm follows a straight path towards the minimum. If the cost function is convex, then it converges to a global minimum and if the cost function is not convex, then it converges to a local minimum. Here the learning rate is typically held constant. In case of stochastic gradient Descent and mini-batch gradient descent, the algorithm does not converge but keeps on fluctuating around the global minimum. Therefore in order to make it converge, we have to slowly change the learning rate. However the convergence of Stochastic gradient descent is much noisier as in one iteration, it processes only one training example.

GRADIENT DESCENT: AN INTRODUCTION TO 1 OF MACHINE LEARNING'S MOST POPULAR ALGORITHMS

Gradient descent is by far the most popular optimization strategy used in machine learning and deep learning at the moment. It is used when training data models, can be combined with every algorithm and is easy to understand and implement. Everyone working with machine learning should understand its concept.

We'll walk through how gradient descent works, what types of it are used today, and its advantages and tradeoffs.

INTRODUCTION TO GRADIENT DESCENT

Gradient descent is an optimization algorithm that's used when training a machine learning model. It's based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum.

Gradient Descent is an optimization algorithm for finding a local minimum of a differentiable function. Gradient descent is simply used in machine learning to find the values of a function's parameters (coefficients) that minimize a cost function as far as possible.

You start by defining the initial parameter's values and from there gradient descent uses calculus to iteratively adjust the values so they minimize the given cost-function. To understand this concept fully, it's important to know about gradients.

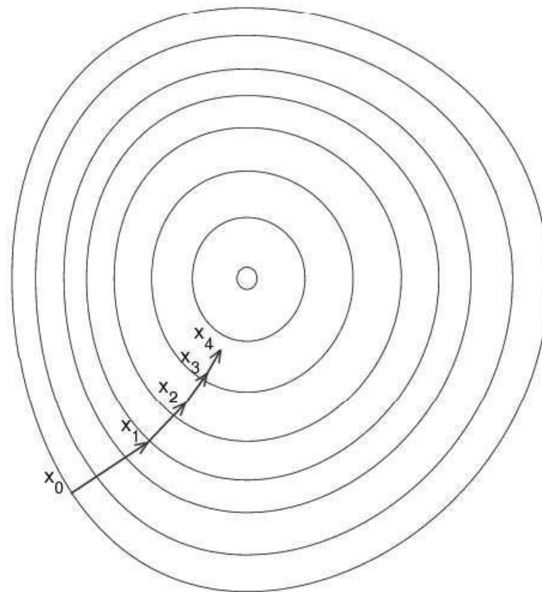
What is a Gradient?

"A gradient measures how much the output of a function changes if you change the inputs a little bit." —Lex Fridman (MIT)

A gradient simply measures the change in all weights with regard to the change in error. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. In mathematical terms, a gradient is a partial derivative with respect to its inputs.

In machine learning, a gradient is a derivative of a function that has more than one input variable. Known as the slope of a function in mathematical terms, the gradient simply measures the change in all weights with regard to the change in error.

Imagine a blindfolded man who wants to climb to the top of a hill with the fewest steps along the way as possible. He might start climbing the hill by taking really big steps in the steepest direction, which he can do as long as he is not close to the top. As he comes closer to the top, however, his steps will get smaller and smaller to avoid overshooting it. This process can be described mathematically using the gradient.



Imagine the image below illustrates our hill from a top-down view and the red arrows are the steps of our climber. Think of a gradient in this context as a vector that contains the direction of the steepest step the blindfolded man can take and also how long that step should be.

Note that the gradient ranging from X0 to X1 is much longer than the one reaching from X3 to X4.

This is because the steepness/slope of the hill, which determines the length of the vector, is less. This perfectly represents the example of the hill because the hill is getting less steep the higher it's climbed. Therefore a reduced gradient goes along with a reduced slope and a reduced step size for the hill climber.

How Gradient Descent works

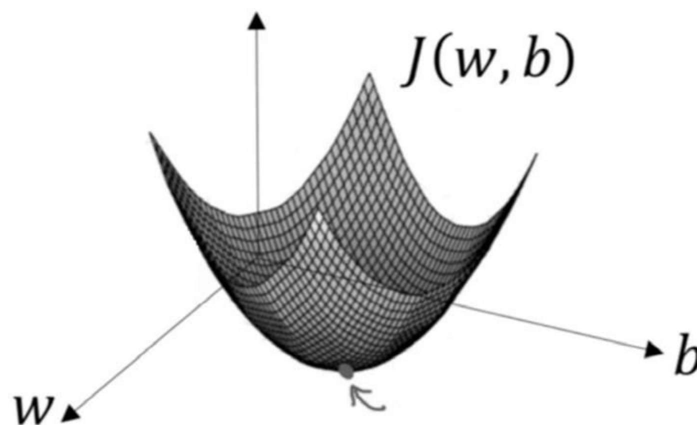
Instead of climbing up a hill, think of gradient descent as hiking down to the bottom of a valley. This is a better analogy because it is a minimization algorithm that minimizes a given function.

The equation below describes what gradient descent does: *b* is the next position of our climber, while *a* represents his current position. The minus sign refers to the minimization part of gradient descent. The gamma in the middle is a waiting factor and the gradient term (Äf(a)) is simply the direction of the steepest descent.

$$\mathbf{b} = \mathbf{a} - \gamma \ \nabla \ \mathbf{f}(\mathbf{a})$$

So this formula basically tells us the next position we need to go, which is the direction of the steepest descent. Let's look at another example to really drive the concept home.

Imagine you have a machine learning problem and want to train your algorithm with gradient descent to minimize your cost-function *J*(*w*, *b*) and reach its local minimum by tweaking its parameters (*w* and *b*). The image below shows the horizontal axes representing the parameters (*w* and *b*), while the cost function *J*(*w*, *b*) is represented on the vertical axes. Gradient descent is a convex function.



We know we want to find the values of *w* and *b* that correspond to the minimum of the cost function (marked with the red arrow). To start finding the right values we initialize *w* and *b* with some random numbers. Gradient descent

then starts at that point (somewhere around the top of our illustration), and it takes one step after another in the steepest downside direction (i.e., from the top to the bottom of the illustration) until it reaches the point where the cost function is as small as possible.

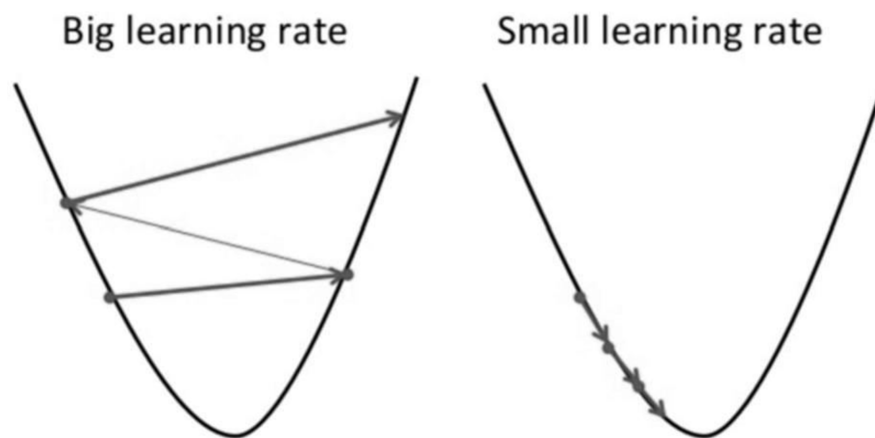
IMPORTANCE OF THE LEARNING RATE

How big the steps the gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.

For gradient descent to reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high.

This is important because if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent. If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while.

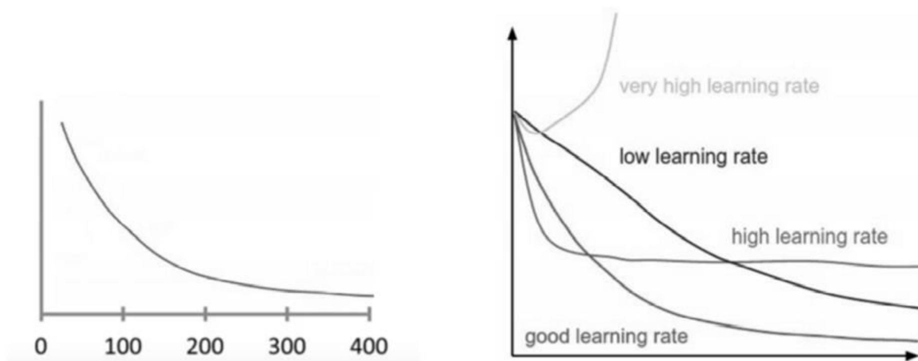
So, the learning rate should never be too high or too low for this reason. You can check if your learning rate is doing well by plotting it on a graph.



How to make sure it works properly

A good way to make sure gradient descent runs properly is by plotting the cost function as the optimization runs. Put the number of iterations on the x-axis and the value of the cost-function on the y-axis. This helps you see the value of your cost function after each iteration of gradient descent, and provides a way to easily spot how appropriate your learning rate is. You can just try different values for it and plot them all together. The left image below shows such a plot, while

the image on the right illustrates the difference between good and bad learning rates.



If gradient descent is working properly, the cost function should decrease after every iteration.

When gradient descent can't decrease the cost-function anymore and remains more or less on the same level, it has converged. The number of iterations gradient descent needs to converge can sometimes vary a lot. It can take 50 iterations, 60,000 or maybe even 3 million, making the number of iterations to convergence hard to estimate in advance.

There are some algorithms that can automatically tell you if gradient descent has converged, but you must define a threshold for the convergence beforehand, which is also pretty hard to estimate. For this reason, simple plots are the preferred convergence test.

Another advantage of monitoring gradient descent via plots is it allows us to easily spot if it doesn't work properly, for example if the cost function is increasing. Most of the time the reason for an increasing cost-function when using gradient descent is a learning rate that's too high.

If the plot shows the learning curve just going up and down, without really reaching a lower point, try decreasing the learning rate. Also, when starting out with gradient descent on a given problem, simply try 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, etc., as the learning rates and look at which one performs the best.

This introductory video to gradient descent helps to explain one of machine learning's most useful algorithms.

TYPES OF GRADIENT DESCENT

There are three popular types of gradient descent that mainly differ in the amount of data they use:

Batch Gradient Descent

Batch gradient descent, also called vanilla gradient descent, calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated. This whole process is like a cycle and it's called a training epoch.

Some advantages of batch gradient descent are its computational efficiency, it produces a stable error gradient and a stable convergence. Some disadvantages are that the stable error gradient can sometimes result in a state of convergence that isn't the best the model can achieve. It also requires the entire training dataset be in memory and available to the algorithm.

Stochastic Gradient Descent

By contrast, stochastic gradient descent (SGD) does this for each training example within the dataset, meaning it updates the parameters for each training example one by one. Depending on the problem, this can make SGD faster than batch gradient descent. One advantage is the frequent updates allow us to have a pretty detailed rate of improvement.

The frequent updates, however, are more computationally expensive than the batch gradient descent approach. Additionally, the frequency of those updates can result in noisy gradients, which may cause the error rate to jump around instead of slowly decreasing.

Mini-batch Gradient Descent

Mini-batch gradient descent is the go-to method since it's a combination of the concepts of SGD and batch gradient descent. It simply splits the training dataset into small batches and performs an update for each of those batches. This creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.

Common mini-batch sizes range between 50 and 256, but like any other machine learning technique, there is no clear rule because it varies for different applications. This is the go-to algorithm when training a neural network and it is the most common type of gradient descent within deep learning.

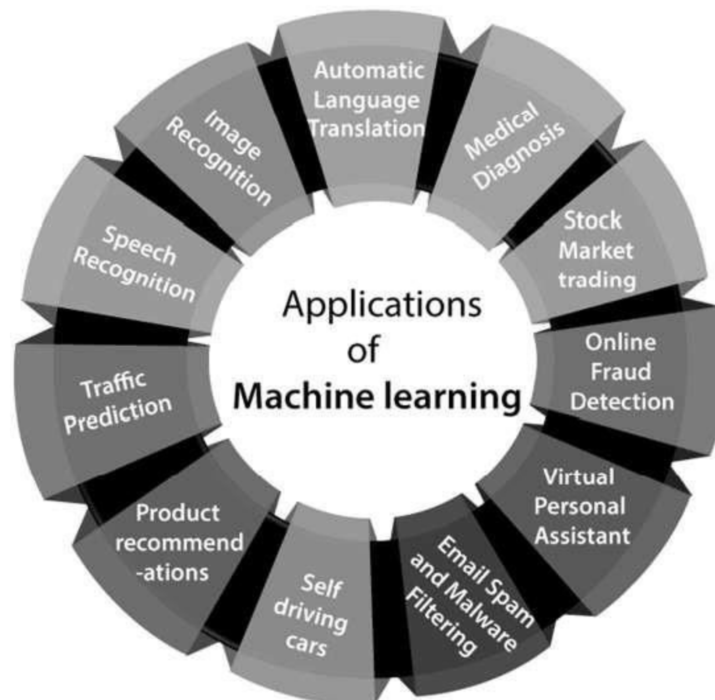
APPLICATIONS OF MACHINE LEARNING

Machine learning is a buzzword for today's technology, and it is growing very rapidly day by day. We are using machine learning in our daily life even without knowing it such as Google Maps, Google assistant, Alexa, etc. Below are some most trending real-world applications of Machine Learning:

IMAGE RECOGNITION

Image recognition is one of the most common applications of machine learning. It is used to identify objects, persons, places, digital images, etc. The popular use case of image recognition and face detection is, **Automatic friend tagging suggestion**:

Facebook provides us a feature of auto friend tagging suggestion. Whenever we upload a photo with our Facebook friends, then we automatically get a tagging suggestion with name, and the technology behind this is machine learning's **face detection** and **recognition algorithm**.



Speech Recognition

While using Google, we get an option of "**Search by voice**," it comes under speech recognition, and it's a popular application of machine learning.

Speech recognition is a process of converting voice instructions into text, and it is also known as "**Speech to text**", or "**Computer speech recognition**." At present, machine learning algorithms are widely used by various applications of speech recognition. **Google assistant**, **Siri**, **Cortana**, and **Alexa** are using speech recognition technology to follow the voice instructions.

Traffic prediction:

If we want to visit a new place, we take help of Google Maps, which shows us the correct path with the shortest route and predicts the traffic conditions.

It predicts the traffic conditions such as whether traffic is cleared, slow-moving, or heavily congested with the help of two ways:

- **Real Time location** of the vehicle form Google Map app and sensors
- **Average time has taken** on past days at the same time.

Everyone who is using Google Map is helping this app to make it better. It takes information from the user and sends back to its database to improve the performance.

Product recommendations:

Machine learning is widely used by various e-commerce and entertainment companies such as **Amazon**, **Netflix**, etc., for product recommendation to the user. Whenever we search for some product on Amazon, then we started getting an advertisement for the same product while internet surfing on the same browser and this is because of machine learning.

Google understands the user interest using various machine learning algorithms and suggests the product as per customer interest.

As similar, when we use Netflix, we find some recommendations for entertainment series, movies, etc., and this is also done with the help of machine learning.

Self-driving cars:

One of the most exciting applications of machine learning is self-driving cars. Machine learning plays a significant role in self-driving cars. Tesla, the most popular car manufacturing company is working on self-driving car. It is using unsupervised learning method to train the car models to detect people and objects while driving.

Email Spam and Malware Filtering:

Whenever we receive a new email, it is filtered automatically as important, normal, and spam. We always receive an important mail in our inbox with the important symbol and spam emails in our spam box, and the technology behind this is Machine learning. Below are some spam filters used by Gmail:

- Content Filter
- Header filter

- o General blacklists filter
- o Rules-based filters
- o Permission filters

Some machine learning algorithms such as **Multi-Layer Perceptron**, **Decision tree**, and **Naïve Bayes classifier** are used for email spam filtering and malware detection.

Virtual Personal Assistant:

We have various virtual personal assistants such as **Google assistant**, **Alexa**, **Cortana**, **Siri**. As the name suggests, they help us in finding the information using our voice instruction. These assistants can help us in various ways just by our voice instructions such as Play music, call someone, Open an email, Scheduling an appointment, etc.

These virtual assistants use machine learning algorithms as an important part.

These assistant record our voice instructions, send it over the server on a cloud, and decode it using ML algorithms and act accordingly.

Online Fraud Detection:

Machine learning is making our online transaction safe and secure by detecting fraud transaction. Whenever we perform some online transaction, there may be various ways that a fraudulent transaction can take place such as **fake accounts**, **fake ids**, and **steal money** in the middle of a transaction. So to detect this, **Feed Forward Neural network** helps us by checking whether it is a genuine transaction or a fraud transaction.

For each genuine transaction, the output is converted into some hash values, and these values become the input for the next round. For each genuine transaction, there is a specific pattern which gets change for the fraud transaction hence, it detects it and makes our online transactions more secure.

Stock Market trading:

Machine learning is widely used in stock market trading. In the stock market, there is always a risk of up and downs in shares, so for this machine learning's **long short term memory neural network** is used for the prediction of stock market trends.

Medical Diagnosis:

In medical science, machine learning is used for diseases diagnoses. With this, medical technology is growing very fast and able to build 3D models that can

predict the exact position of lesions in the brain. It helps in finding brain tumors and other brain-related diseases easily.

Automatic Language Translation:

Nowadays, if we visit a new place and we are not aware of the language then it is not a problem at all, as for this also machine learning helps us by converting the text into our known languages. Google's GNMT (Google Neural Machine Translation) provide this feature, which is a Neural Machine Learning that translates the text into our familiar language, and it called as automatic translation.

IMAGE PROCESSING WITH DEEP LEARNING- A QUICK START GUIDE

Imagine how much more valuable your data would be to your business if your document-intake solution could extract data from images as seamlessly as it does from the text.

Thanks to deep learning, intelligent document processing (IDP) is able to combine various AI technologies to not only automatically classify photos, but also describe the various elements in pictures and write short sentences describing each segment with proper English grammar.

IDP leverages a deep learning network known as CNN (Convolutional Neural Networks) to learn patterns that naturally occur in photos. IDP is then able to adapt as new data is processed, using Imagenet, one of the biggest databases of labeled images, which has been instrumental in advancing computer vision.

One of the ways this type of technology is implemented with impact is in the document-heavy insurance industry. Claims processing starts with a small army of humans manually entering data from forms.

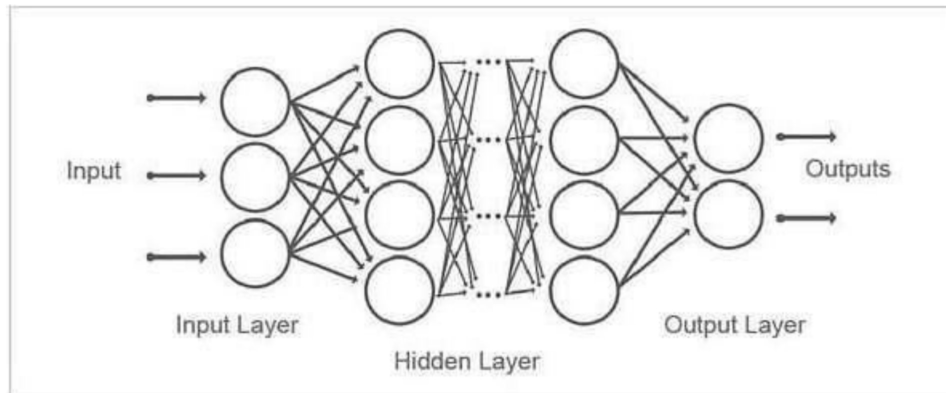
In a typical use case, the claim includes a set of documents such as: claim forms, police reports, accident scene and vehicle damage pictures, vehicle operator driver's license, insurance copy, bills, invoices, and receipts.

Documents like these aren't standard, and the business systems that automate most of the claims processing can't function without data from the forms.

To turn those documents into data, the Convolutional Neural Networks are trained using GPU-accelerated deep learning frameworks such as Caffe2, Chainer, Microsoft Cognitive Toolkit, MXNet, PaddlePaddle, Pytorch, TensorFlow, and inference optimizers such as TensorRT.

Neural networks were first used in 2009 for speech recognition, and were

only implemented by Google in 2012. Deep learning, also called neural networks, is a subset of machine learning that uses a model of computing that's very much inspired by the structure of the brain.



"Deep learning is already working in Google search and in image search; it allows you to image-search a term like 'hug.' It's used to getting you Smart Replies to your Gmail. It's in speech and vision. It will soon be used in machine translation, I believe." said Geoffrey Hinton, considered the Godfather of neural networks.

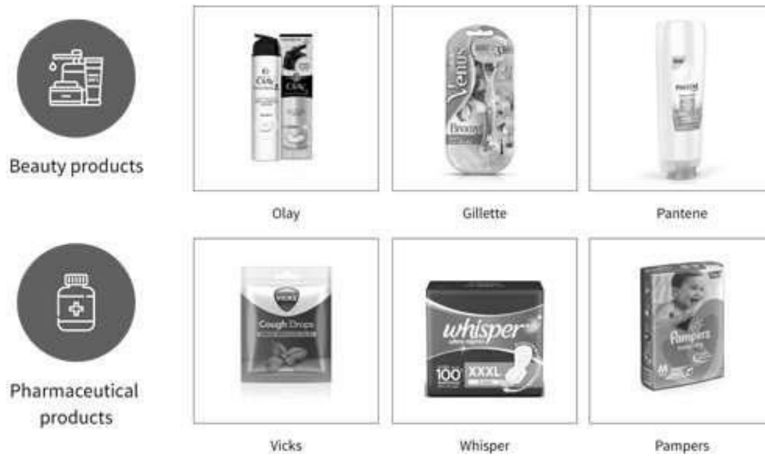
Deep Learning models, with their multi-level structures, as shown above, are very helpful in extracting complicated information from input images. Convolutional neural networks are also able to drastically reduce computation time by taking advantage of GPU for computation, which many networks fail to utilize.

Let's take a deeper dive into IDP's image data preparation using deep learning. Preparing images for further analysis is needed to offer better local and global feature detection, which is how IDP enables straight-through processing and drives ROI for your business. Below are the steps:

IMAGE CLASSIFICATION

For increased accuracy, image classification using CNN is most effective. First and foremost, your IDP solution will need a set of images. In this case, images of beauty and pharmacy products are used as the initial training data set. The most common image data input parameters are the number of images, image dimensions, number of channels, and number of levels per pixel.

With classification, you are able to categorize images (in this case, as beauty and pharmacy). Each category again has different classes of objects as shown in the picture below:



Data Labeling

It's better to manually label the input data so that the deep learning algorithm can eventually learn to make the predictions on its own. Some off the shelf manual data labeling tools are given here.

The objective at this point will be mainly to identify the actual object or text in a particular image, demarcating whether the word or object is oriented improperly, and identifying whether the script (if present) is in English or other languages.

To automate the tagging and annotation of images, NLP pipelines can be applied. ReLU (rectified linear unit) is then used for the non-linear activation functions, as they perform better and decrease training time.

To increase the training dataset, we can also try data augmentation by emulating the existing images and transforming them. We could transform the available images by making them smaller, blowing them up, cropping elements etc.

Using RCNN

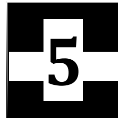
With the usage of Region-based Convolutional Neural Network (aka RCNN), locations of objects in an image can be detected with ease. Within just 3 years the RCNN has moved from Fast RCNN, Faster RCNN to Mask RCNN, making tremendous progress towards human-level cognition of images. Below is an example of the final output of the image recognition model where it was trained by deep learning CNN to identify categories and products in images.

If you are new to deep learning methods and don't want to train your own model, you could have a look on Google Cloud Vision. It works pretty well for general cases.

Category Detection

Product Detection

If you are looking for a specific IDP solution or customization, our ML experts will ensure your time and resources are well spent in partnering with us.



Natural Language Processing

Natural language processing (**NLP**) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data. The goal is a computer capable of "understanding" the contents of documents, including the contextual nuances of the language within them. The technology can then accurately extract information and insights contained in the documents as well as categorize and organize the documents themselves.

Challenges in natural language processing frequently involve speech recognition, natural-language understanding, and natural-language generation.

HISTORY

The proposed test includes a task that involves the automated interpretation and generation of natural language.

Symbolic NLP (1950s – early 1990s)

The premise of symbolic NLP is well-summarized by John Searle's Chinese room experiment: Given a collection of rules (e.g., a Chinese phrasebook, with questions and matching answers), the computer emulates natural language understanding (or other NLP tasks) by applying those rules to the data it confronts.

- **1950s**: The Georgetown experiment in 1954 involved fully automatic translation of more than sixty Russian sentences into English. The

authors claimed that within three or five years, machine translation would be a solved problem. However, real progress was much slower, and after the ALPAC report in 1966, which found that ten-year-long research had failed to fulfill the expectations, funding for machine translation was dramatically reduced. Little further research in machine translation was conducted until the late 1980s when the first statistical machine translation systems were developed.

- **1960s**: Some notably successful natural language processing systems developed in the 1960s were SHRDLU, a natural language system working in restricted "blocks worlds" with restricted vocabularies, and ELIZA, a simulation of a Rogerian psychotherapist, written by Joseph Weizenbaum between 1964 and 1966. Using almost no information about human thought or emotion, ELIZA sometimes provided a startlingly human-like interaction. When the "patient" exceeded the very small knowledge base, ELIZA might provide a generic response, for example, responding to "My head hurts" with "Why do you say your head hurts?".
- **1970s**: During the 1970s, many programmers began to write "conceptual ontologies", which structured real-world information into computer-understandable data. Examples are MARGIE (Schank, 1975), SAM (Cullingford, 1978), PAM (Wilensky, 1978), TaleSpin (Meehan, 1976), QUALM (Lehnert, 1977), Politics (Carbonell, 1979), and Plot Units (Lehnert 1981). During this time, the first chatterbots were written (e.g., PARRY).
- **1980s**: The 1980s and early 1990s mark the heyday of symbolic methods in NLP. Focus areas of the time included research on rule-based parsing (e.g., the development of HPSG as a computational operationalization of generative grammar), morphology (e.g., two-level morphology), semantics (e.g., Lesk algorithm), reference (e.g., within Centering Theory) and other areas of natural language understanding (e.g., in the Rhetorical Structure Theory). Other lines of research were continued, e.g., the development of chatterbots with Racter and Jabberwacky. An important development (that eventually led to the statistical turn in the 1990s) was the rising importance of quantitative evaluation in this period.

Statistical NLP (1990s–2010s)

Up to the 1980s, most natural language processing systems were based on complex sets of hand-written rules. Starting in the late 1980s, however, there was

a revolution in natural language processing with the introduction of machine learning algorithms for language processing. This was due to both the steady increase in computational power and the gradual lessening of the dominance of Chomskyan theories of linguistics (e.g. transformational grammar), whose theoretical underpinnings discouraged the sort of corpus linguistics that underlies the machine-learning approach to language processing.

- **1990s**: Many of the notable early successes on statistical methods in NLP occurred in the field of machine translation, due especially to work at IBM Research. These systems were able to take advantage of existing multilingual textual corpora that had been produced by the Parliament of Canada and the European Union as a result of laws calling for the translation of all governmental proceedings into all official languages of the corresponding systems of government. However, most other systems depended on corpora specifically developed for the tasks implemented by these systems, which was (and often continues to be) a major limitation in the success of these systems. As a result, a great deal of research has gone into methods of more effectively learning from limited amounts of data.
- **2000s**: With the growth of the web, increasing amounts of raw (unannotated) language data has become available since the mid-1990s. Research has thus increasingly focused on unsupervised and semi-supervised learning algorithms. Such algorithms can learn from data that has not been hand-annotated with the desired answers or using a combination of annotated and non-annotated data. Generally, this task is much more difficult than supervised learning, and typically produces less accurate results for a given amount of input data. However, there is an enormous amount of non-annotated data available (including, among other things, the entire content of the World Wide Web), which can often make up for the inferior results if the algorithm used has a low enough time complexity to be practical.

Neural NLP (present)

In the 2010s, representation learning and deep neural network-style machine learning methods became widespread in natural language processing. That popularity was due partly to a flurry of results showing that such techniques can achieve state-of-the-art results in many natural language tasks, e.g., in language modeling and parsing. This is increasingly important in medicine and healthcare, where NLP

helps analyze notes and text in electronic health records that would otherwise be inaccessible for study when seeking to improve care.

THE PERCEPTRON

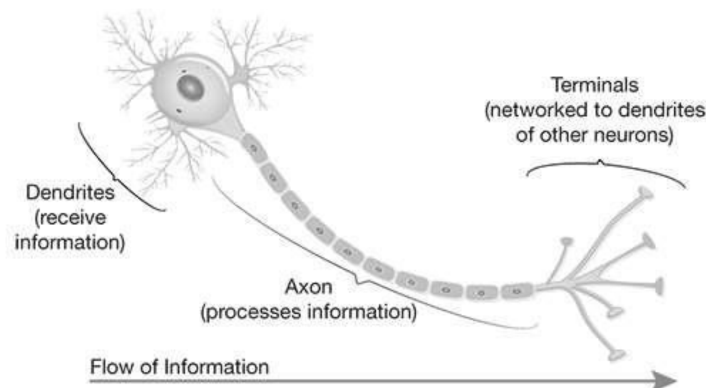


Figure . ***A human neuron collects inputs from other neurons using dendrites and sums all the inputs. If the total is greater than a threshold value, it produces an output.***

The average human brain has approximately 100 billion neurons. A human neuron uses dendrites to collect inputs from other neurons, adds all the inputs, and if the resulting sum is greater than a threshold, it fires and produces an output. The fired output is then sent to other connected neurons.

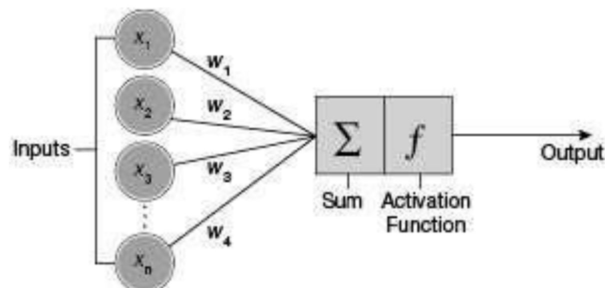


Figure . ***A perceptron is a mathematical model of a neuron. It receives weighted inputs, which are added together and passed to an activation function. The activation function decides whether it should produce an output.***

A perceptron is a mathematical model of a biological neuron. Just like a real neuron, it receives inputs and computes an output. Each input has an associated weight. All the inputs are individually multiplied by their weights, added together, and passed into an activation function that determines whether the neuron should fire and produce an output.

There are many different types of activation functions with different properties, but one of the simplest is the step function. A step function outputs a 1 if the input is higher than a certain threshold, otherwise it outputs a 0. For example, if a perceptron has two inputs (x_1 and x_2):

$x_1 = 0.9$

$x_2 = 0.7$

which have weightings (w_1 and w_2) of:

$w_1 = 0.2$

$w_2 = 0.9$

and the activation function threshold is equal to 0.75, then weighing the inputs and adding them together yields:

$x_1w_1 + x_2w_2 = (0.9\times0.7) + (0.2\times0.9) = 0.81$

Because the total input is higher than the threshold (0.75), the neuron will fire. Since we chose a simple step function, the output would be 1.

So how does all this lead to intelligence? It starts with the ability to learn something simple through training.

Training a perceptron

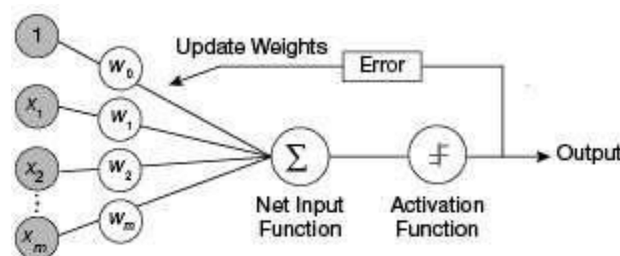


Figure . ***To train a perceptron, the weights are adjusted to minimize the output error. Output error is defined as the difference between the desired output and the actual output.***

Training a perceptron involves feeding it multiple training samples and calculating the output for each of them. After each sample, the weights are adjusted to minimize the output error, usually defined as the difference between the desired (target) and the actual outputs.

By following this simple training algorithm to update weights, a perceptron can learn to perform binary linear classification. For example, it can learn to separate dogs from cats given size and domestication data, if the data are linearly classifiable.

The perceptron's ability to learn classification is significant because classification underlies many acts of intelligence. A common example of classification is detecting spam emails.

Given a training dataset of spam-like emails labeled as "spam" and regular emails labeled as "not-spam," an algorithm that can learn characteristics of spam emails would be very useful.

Similarly, such algorithms could learn to classify tumors as cancerous or benign, learn your music preferences and classify songs as "likely-to-like" and "unlikely-to-like," or learn to distinguish normally behaving valves from abnormally behaving valves.

Perceptrons are powerful classifiers. However, individually they can only learn linearly classifiable patterns and are unable to handle nonlinear or more complicated patterns.

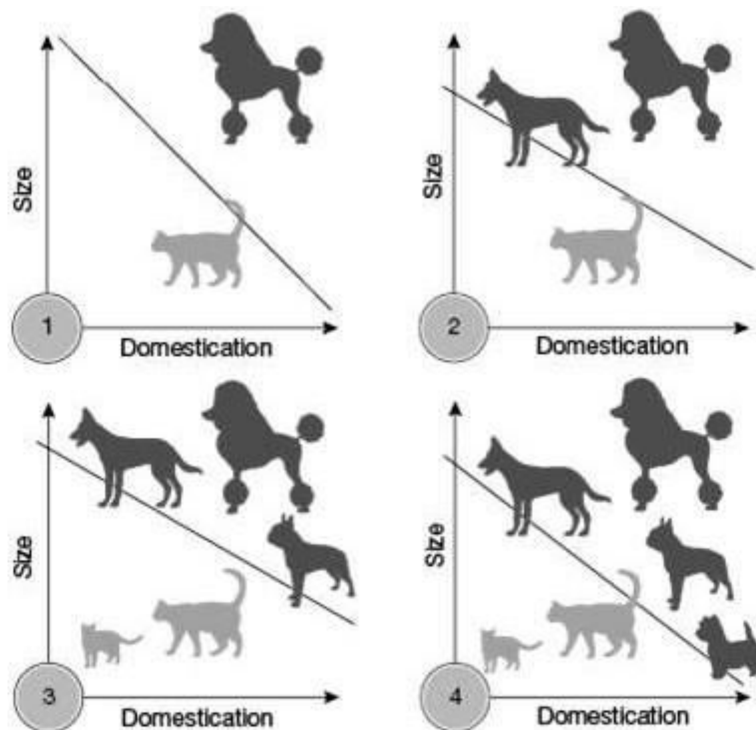


Figure . ***A perceptron can learn to separate dogs and cats given size and domestication data. As more training examples are added, the perceptron updates its linear boundary.***

Multilayer perceptrons

A single neuron is capable of learning simple patterns, but when many neurons

are connected together, their abilities increase dramatically. Each of the 100 billion neurons in the human brain has, on average, 7,000 connections to other neurons.

It has been estimated that the brain of a three-year-old child has about one quadrillion connections between neurons. And, theoretically, there are more possible neural connections in the brain than there are atoms in the universe.

A multilayer perceptron (MLP) is an artificial neural network with multiple layers of neurons between input and output. MLPs are also called feedforward neural networks. Feedforward means that data flow in one direction from the input to the output layer.

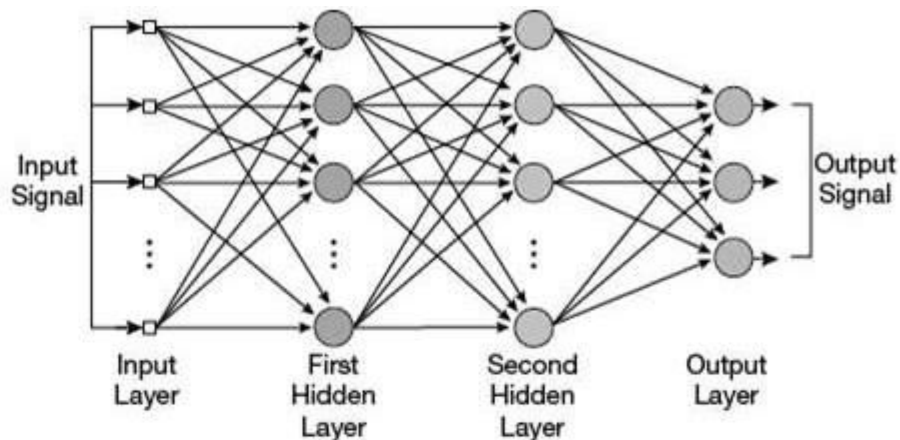


Figure . ***A multilayer perceptron has multiple layers of neurons between the input and output. Each neuron's output is connected to every neuron in the next layer.***

Typically, every neuron's output is connected to every neuron in the next layer. Layers that come between the input and output layers are referred to as hidden layers.

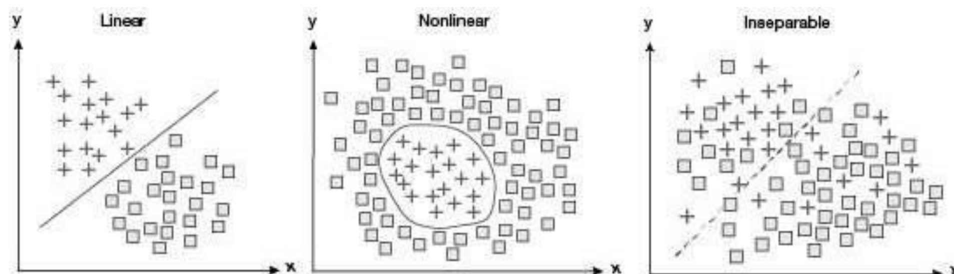


Figure . ***Although single perceptrons can learn to classify linear patterns, they are unable to handle nonlinear or other more complicated datasets. Multilayer perceptrons are more capable of handling nonlinear patterns, and can even classify inseparable data.***

MLPs are widely used for pattern classification, recognition, prediction, and approximation, and can learn complicated patterns that are not separable using linear or other easily articulated curves. The capacity of an MLP network to learn complicated patterns increases with the number of neurons and layers.

MLPs have been successful at a wide range of AI tasks, from speech recognition to predicting thermal conductivity of aqueous electrolyte solutions and controlling a continuous stirred-tank reactor. For example, an MLP for recognizing printed digits (*e.g.,* the account and routing number printed on a check) would be comprised of a grid of inputs to read individual pixels of digits (say, a 9×12 bitmap), followed by one or more hidden layers, and finally 10 output neurons to indicate which number was recognized in the input (0–9).

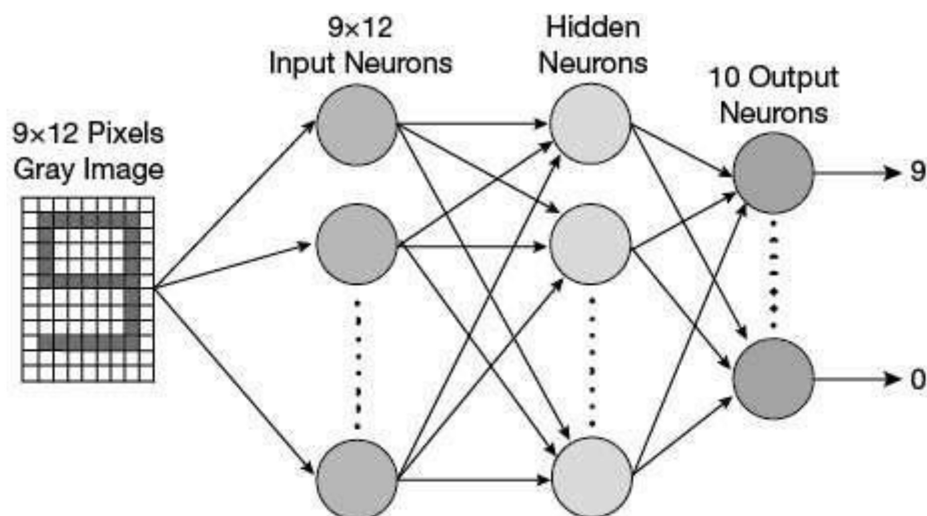


Figure . ***A multilayer perceptron for recognizing digits printed on a check would have a grid of inputs to read individual pixels of digits, followed by one or more layers of hidden neurons, and 10 output neurons to indicate which number was recognized.***

Such an MLP for recognizing digits would typically be trained by showing it images of digits and telling it whether it recognized them correctly or not. Initially, the MLP's output would be random, but as it is trained, it will adjust weights between the neurons and start classifying inputs correctly.

A typical real-life MLP for recognizing handwritten digits consists of 784 perceptrons that accept inputs from a 28×28 pixel bitmap representing a handwritten digit, 15 neurons in the hidden layer, and 10 output neurons. Typically, such an MLP is trained using a pool of 50,000 labeled images of handwritten digits. It can learn to recognize previously unseen handwritten digits with 95% accuracy

after a few minutes of training on a well-configured computer.In a similar fashion, others have used data from *Perry's Chemical Engineers' Handbook* to train an MLP to predict viscosity of a compound. In another study, scientists were able to detect faults in a heat exchanger by training an MLP to recognize deviations in temperature and flowrate as symptoms of tube plugging and partial fouling in the heat exchanger internals.

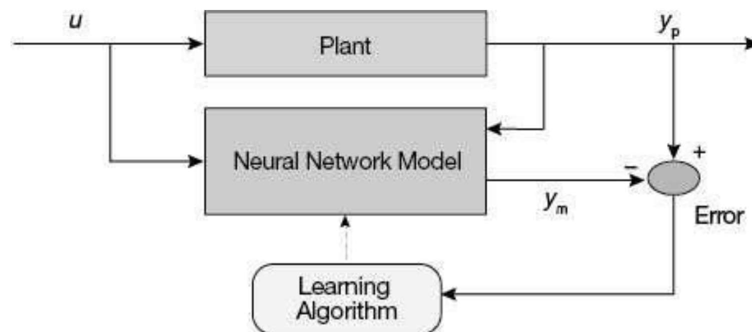


Figure . ***An MLP can learn the dynamics of the plant by evaluating the error between the actual plant output and the neural network output.***

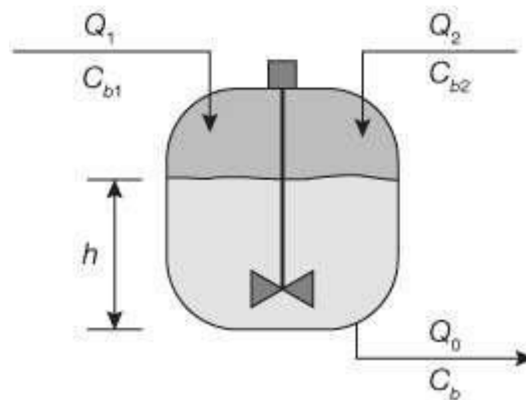


Figure . ***A continuous stirred-tank reactor can be trained to maintain appropriate product concentration and flow by using past data about inflow, concentration, liquid level, and outflow.***

As another example, MLPs have been used for predictive control of chemical reactors. The typical setup trains a neural network to learn the forward dynamics of the plant. The prediction error between the plant output and the neural network output is used for training the neural network. The neural network learns from previous inputs and outputs to predict future values of the plant output. For example, a controller for a catalytic continuous stirred-tank reactor can be trained

to maintain appropriate product concentration and flow by using past data about inflow Q_1 and Q_2 at concentrations C_{b1} and C_{b2}, respectively, liquid level h, and outflow Q_0 at concentration C_b.

In general, given a statistically relevant dataset, an artificial neural network can learn from it.

TRAINING A MULTILAYER PERCEPTRON

Training a single perceptron is easy — all weights are adjusted repeatedly until the output matches the expected value for all training data. For a single perceptron, weights can be adjusted using the formulas:

$$\Delta w_i = \eta(t - o)x_i$$
$$w_i + \Delta w_i \rightarrow w_i$$

where w_i is the weight, ▪w_i is the weight adjustment, t is the target output, o is the actual output, and ▪ is the learning rate—usually a small value used to moderate the rate of change of weights.

However, this approach of tweaking each weight independently does not work for an MLP because each neuron's output is an input for all neurons in the next layer.

Tweaking the weight on one connection impacts not only the neuron it propagates to directly, but also all of the neurons in the following layers as well, and thus affects all the outputs. Therefore, you cannot obtain the best set of weights by optimizing one weight at a time. Instead, the entire space of possible weight combinations must be searched simultaneously. The primary method for doing this relies on a technique called gradient descent.

Imagine you are at the top of a hill and you need to get to the bottom of the hill in the quickest way possible. One approach could be to look in every direction to see which way has the steepest grade, and then step in that direction.

If you repeat this process, you will gradually go farther and farther downhill. That is how gradient descent works: If you can define a function over all weights that reflects the difference between the desired output and calculated output, then the function will be lowest (*i.e.,* the bottom of the hill) when the MLP's output matches the desired output. Moving toward this lowest value will become a matter of calculating the gradient (or derivative of the function) and taking a small step in the direction of the gradient.

Backpropagation, short for "backward propagation of errors," is the most commonly used algorithm for training MLPs using gradient descent. The backward

part of the name stems from the fact that calculation of the gradient proceeds backward through the network. The gradient of the final layer of weights is calculated first and the gradient of the first layer of weights is calculated last.

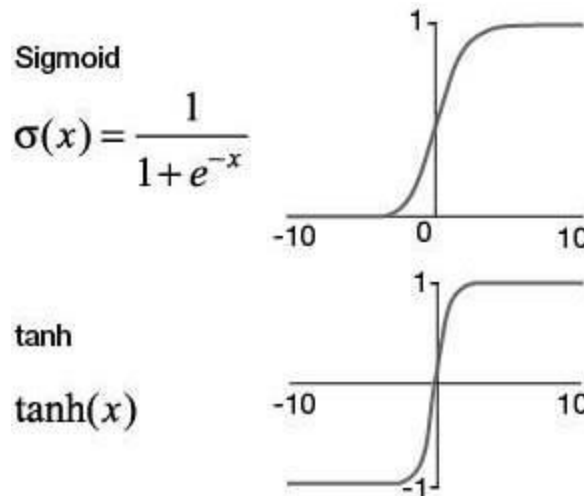


Figure . ***Sigmoid and tanh functions are nonlinear activation functions. The output of the sigmoid function is a value between 0 and 1. The output of the sigmoid function can be used to represent a probability, often the probability that the input belongs to a category (*****e.g.,** ***cat or dog).***

Before looking at how backpropagation works, recall that a perceptron calculates a weighted sum of its input and then decides whether it should fire. The decision about whether or not to fire is made by the activation function. In the perceptron example, we used a step function that outputted a 1 if the input was higher than a certain threshold, otherwise it out-putted a 0. In practice, ANNs use nonlinear activation functions like the sigmoid or tanh functions, at least in part because a simple step function does not lend itself to calculating gradients — its derivative is 0.

The sigmoid function maps its input to the range 0 to 1. You might recall that probabilities, too, are represented by values between 0 and 1. Hence, the output of the sigmoid function can be used to represent a probability — often the probability that the input belongs to a category (*e.g.,* cat or dog). For this reason, it is one of the most widely used activation functions for artificial neural networks.

Example: Training an MLP with backpropagation

Consider a simple MLP with three layers: two neurons in the input layer (X_{i1}, X_{i2}) connected to three neurons (X_{h1}, X_{h2}, X_{h3}) in the hidden layer via weights W_1–

W_6, which are connected to a single output neuron (X_o) via weights W_7–W_9. Assume that we are using the sigmoid activation function, initial weights are randomly assigned, and input values [1, 1] will lead to an output of 0.77.

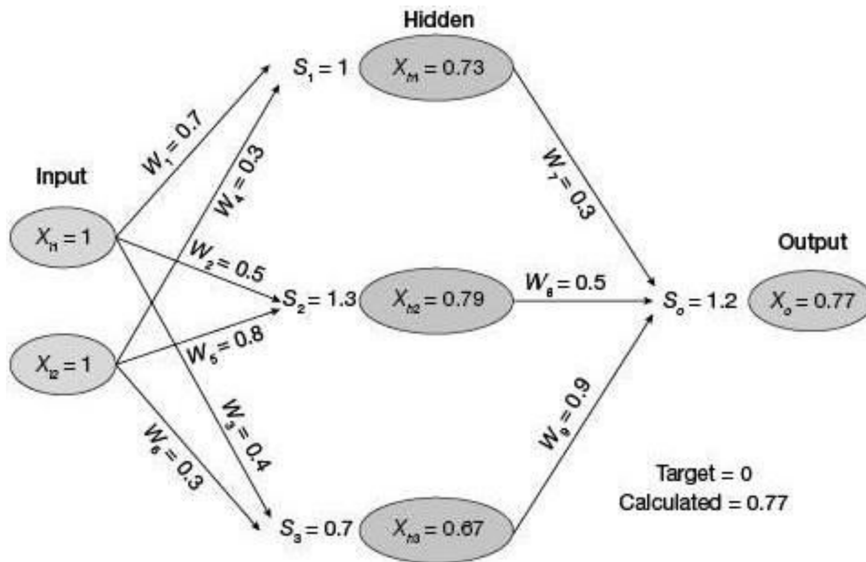


Figure . ***An example MLP with three layers accepts an input of [1, 1] and computes an output of 0.77.***

Let's assume that the desired output for inputs [1, 1] is 0. The backpropagation algorithm can be used to adjust weights. First, calculate the error at the last neuron's (X_o) output:

$$\text{Error} = \text{Target value} - \text{Calculated value}$$
$$\text{Error} = 0 - 0.77 = -0.77$$

Recall that the output (0.77) was obtained by applying the sigmoid activation function to the weighted sum of the previous layer's outputs (1.2):

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$\cdot(1.2) = 1/(1 + e^{-1.2}) = 0.77$

The derivative of the sigmoid function represents the gradient or rate of change:

$$\frac{d\sigma(x)}{d(x)} = \sigma(x)\big(1-\sigma(x)\big)$$

Hence, the gradient or rate of change of the sigmoid function at $x = 1.2$ is: $(0.77) \times (1 - 0.77) = 0.177$

If we multiply the error in output (–0.77) by this rate of change (0.177) we get –0.13. This can be proposed as a small change in input that could move the system toward the proverbial "bottom of the hill."

Recall that the sum of the weighted inputs of the output neuron (1.2) is the product of the output of the three neurons in the previous layer and the weights between them and the output neuron:

$$S_o = X_{h1} \times W_7 + X_{h2} \times W_8 + X_{h3} \times W_9$$
$$1.2 = 0.73 \times 0.3 + 0.79 \times 0.5 + 0.67 \times 0.9$$

To change this sum (S_o) by –0.13, we can adjust each incoming weight (W_7, W_8, W_9) proportional to the corresponding output of the previous (hidden layer) neuron (X_{h1}, X_{h2}, X_{h3}). So, the weights between the hidden neurons and the output neuron become:

$W_{7new} = W_{7old} + (-0.13/X_{h1}) = 0.3 + (-0.13/0.73) = 0.11$

$W_{8new} = W_{8old} + (-0.13/X_{h2}) = 0.5 + (-0.13/0.79) = 0.33$

$W_{9new} = W_{9old} + (-0.13/X_{h3}) = 0.9 + (-0.13/0.67) = 0.7$

After adjusting the weights between the hidden layer neurons and the output neuron, we repeat the process and similarly adjust the weights between the input and hidden layer neurons.

This is done by first calculating the gradient at the input coming into each neuron in the hidden layer. For example, the gradient at X_{h3} is: $0.67 \times (1-0.67) = 0.22$.

The proposed change in the sum of weighted inputs of X_{h3} (*i.e.,* S_3) can be calculated by multiplying the gradient (0.22) by the proposed change in the sum of weighted inputs of the following neuron (–0.13), and dividing by the weight from this neuron to the following neuron (W_9).

Note that we are propagating errors backward, so it was the error in the following neuron (X_o) that we proportionally propagated backward to this neuron's inputs.

The proposed change in the sum of weighted inputs of X_{h3} (*i.e.,* S_3) is:

Change in S_3 = Gradient at X_{h3} × Proposed change in S_o/W_9

Change in $S_3 = 0.22 \times (-0.13)/0.9 = -0.03$

Note that we use the original value of W_9 (0.9) rather than the recently

calculated new value (0.7) to propagate the error backward. This is because although we are working one step at a time, we are trying to search the entire space of possible weight combinations and change them in the right direction (toward the bottom of the hill). In each iteration, we propagate the output error through original weights, leading to new weights for the iteration. This global backward propagation of the output neuron's error is the key concept that lets all weights change toward ideal values.

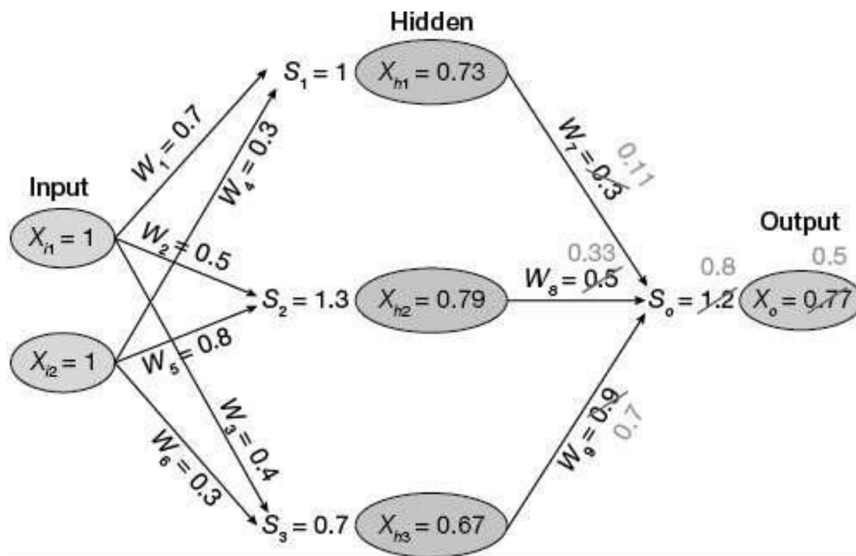


Figure . ***A backpropagation algorithm is used to adjust the weightings between the hidden layer neurons and the output neurons, so that the output is closer to the target value (0).***

Once you know the proposed change in the weighted sum of inputs of each neuron (S_1, S_2, S_3), you can change the weights leading to the neuron (W_1 through W_6) proportional to the output from the previous neuron. Thus, W_6 changes from 0.3 to 0.27.

Upon repeating this process for all weights, the new output in this example becomes 0.68, which is a little closer to the ideal value (0) than what we started with (0.77). By performing just one such iteration of forward and back propagation, the network is already learning!

A small neural network like the one in this example will typically learn to produce correct outputs after a few hundred such iterations of weight adjustments. On the other hand, training AlphaGo's neural network, which has tens of thousands of neurons arranged in more than a dozen layers, takes more serious computing power, which is becoming increasingly available.

Looking forward

Even with all the amazing progress in AI, such as self-driving cars, the technology is still very narrow in its accomplishments and far from autonomous. Today, 99% of machine learning requires human work and large amounts of data that need to be normalized and labeled (*i.e.*, this is a dog; this is a cat). And, people need to supply and fine-tune the appropriate algorithms. All of this relies on manual labor.

Other challenges that plague neural networks include:

- *Bias*. Machine learning is looking for patterns in data. If you start with bad data, you will end up with bad models.
- *Over-fitting*. In general, a model is typically trained by maximizing its performance on a particular training dataset. The model thus memorizes the training examples, but may not learn to generalize to new situations and datasets.
- *Hyper-parameter optimization*. The value of a hyper-parameter is defined prior to the commencement of the learning process (*e.g.,* number of layers, number of neurons per layer, type of activation function, initial value of weights, value of the learning rate, etc.). Changing the value of such parameters by a small amount can invoke large changes in the performance of the network.
- *Black-box problems.* Neural networks are essentially black boxes, and researchers have a hard time understanding how they deduce particular conclusions. Their operation is largely invisible to humans, rendering them unsuitable for domains in which verifying the process is important.

Thus far, we have looked at neural networks that learn from data. This approach is called supervised learning. The training of a neural network under supervised learning, an input is presented to the network and it produces an output that is compared with the desired/target output. An error is generated if there is a difference between the actual output and the target output and the weights are adjusted based on this error until the actual output matches the desired output. Supervised learning relies on manual human labor for collecting, preparing, and labeling a large amount of training data.

Unsupervised learning does not depend on target outputs for learning. Instead, inputs of a similar type are combined to form clusters. When a new input pattern is applied, the neural network gives an output indicating the class to which the input pattern belongs.

Reinforcement learning involves learning by trial and error, solely from rewards or punishments. Such neural networks construct and learn their own knowledge directly from raw inputs, such as vision, without any hand-engineered features or domain heuristics. AlphaGo Zero, the successor to AlphaGo, is based on reinforcement learning. Unlike AlphaGo, which was initially trained on thousands of human games to learn how to play Go, AlphaGo Zero learned to play simply by playing games against itself. Although it began with completely random play, it eventually surpassed human level of play and defeated the previous version of AlphaGo by 100 games to 0.

METHODS: RULES, STATISTICS, NEURAL NETWORKS

In the early days, many language-processing systems were designed by symbolic methods, i.e., the hand-coding of a set of rules, coupled with a dictionary lookup: such as by writing grammars or devising heuristic rules for stemming.

More recent systems based on machine-learning algorithms have many advantages over hand-produced rules:

- The learning procedures used during machine learning automatically focus on the most common cases, whereas when writing rules by hand it is often not at all obvious where the effort should be directed.
- Automatic learning procedures can make use of statistical inference algorithms to produce models that are robust to unfamiliar input (e.g. containing words or structures that have not been seen before) and to erroneous input (e.g. with misspelled words or words accidentally omitted). Generally, handling such input gracefully with handwritten rules, or, more generally, creating systems of handwritten rules that make soft decisions, is extremely difficult, error-prone and time-consuming.
- Systems based on automatically learning the rules can be made more accurate simply by supplying more input data. However, systems based on handwritten rules can only be made more accurate by increasing the complexity of the rules, which is a much more difficult task. In particular, there is a limit to the complexity of systems based on handwritten rules, beyond which the systems become more and more unmanageable. However, creating more data to input to machine-learning systems simply requires a corresponding increase in the number of man-hours worked, generally without significant increases in the complexity of the annotation process.

Despite the popularity of machine learning in NLP research, symbolic methods are still (2020) commonly used:

- when the amount of training data is insufficient to successfully apply machine learning methods, e.g., for the machine translation of low-resource languages such as provided by the Apertium system,
- for preprocessing in NLP pipelines, e.g., tokenization, or
- for postprocessing and transforming the output of NLP pipelines, e.g., for knowledge extraction from syntactic parses.

Statistical methods

Since the so-called "statistical revolution" in the late 1980s and mid-1990s, much natural language processing research has relied heavily on machine learning. The machine-learning paradigm calls instead for using statistical inference to automatically learn such rules through the analysis of large *corpora* (the plural form of *corpus*, is a set of documents, possibly with human or computer annotations) of typical real-world examples.

Many different classes of machine-learning algorithms have been applied to natural-language-processing tasks. These algorithms take as input a large set of "features" that are generated from the input data. Increasingly, however, research has focused on statistical models, which make soft, probabilistic decisions based on attaching real-valued weights to each input feature (complex-valued embeddings, and neural networks in general have also been proposed, for e.g. speech). Such models have the advantage that they can express the relative certainty of many different possible answers rather than only one, producing more reliable results when such a model is included as a component of a larger system.

Some of the earliest-used machine learning algorithms, such as decision trees, produced systems of hard if-then rules similar to existing hand-written rules. However, part-of-speech tagging introduced the use of hidden Markov models to natural language processing, and increasingly, research has focused on statistical models, which make soft, probabilistic decisions based on attaching real-valued weights to the features making up the input data. The cache language models upon which many speech recognition systems now rely are examples of such statistical models. Such models are generally more robust when given unfamiliar input, especially input that contains errors (as is very common for real-world data), and produce more reliable results when integrated into a larger system comprising multiple subtasks. Since the neural turn, statistical methods in NLP research have been largely replaced by neural networks. However, they continue to be relevant for contexts in which statistical interpretability and transparency is required.

Neural networks

A major drawback of statistical methods is that they require elaborate feature engineering. Since 2015, the field has thus largely abandoned statistical methods and shifted to neural networks for machine learning. Popular techniques include the use of word embeddings to capture semantic properties of words, and an increase in end-to-end learning of a higher-level task (e.g., question answering) instead of relying on a pipeline of separate intermediate tasks (e.g., part-of-speech tagging and dependency parsing).

In some areas, this shift has entailed substantial changes in how NLP systems are designed, such that deep neural network-based approaches may be viewed as a new paradigm distinct from statistical natural language processing. For instance, the term *neural machine translation* (NMT) emphasizes the fact that deep learning-based approaches to machine translation directly learn sequence-to-sequence transformations, obviating the need for intermediate steps such as word alignment and language modeling that was used in statistical machine translation (SMT).

COMMON NLP TASKS

The following is a list of some of the most commonly researched tasks in natural language processing. Some of these tasks have direct real-world applications, while others more commonly serve as subtasks that are used to aid in solving larger tasks.

Though natural language processing tasks are closely intertwined, they can be subdivided into categories for convenience. A coarse division is given below.

TEXT AND SPEECH PROCESSING

Optical character recognition (OCR)

Given an image representing printed text, determine the corresponding text.

Speech recognition

Given a sound clip of a person or people speaking, determine the textual representation of the speech. This is the opposite of text to speech and is one of the extremely difficult problems colloquially termed "AI-complete". In natural speech there are hardly any pauses between successive words, and thus speech segmentation is a necessary subtask of speech recognition. In most spoken languages, the sounds representing successive letters blend into each other in a process termed coarticulation, so the conversion of the analog signal to discrete characters can be a very difficult process. Also, given that words in the same language are spoken by people with different accents, the speech recognition software must be able

to recognize the wide variety of input as being identical to each other in terms of its textual equivalent.

Speech segmentation

Given a sound clip of a person or people speaking, separate it into words. A subtask of speech recognition and typically grouped with it.

Text-to-speech

Given a text, transform those units and produce a spoken representation. Text-to-speech can be used to aid the visually impaired.

Word segmentation (Tokenization)

Separate a chunk of continuous text into separate words. For a language like English, this is fairly trivial, since words are usually separated by spaces. However, some written languages like Chinese, Japanese and Thai do not mark word boundaries in such a fashion, and in those languages text segmentation is a significant task requiring knowledge of the vocabulary and morphology of words in the language. Sometimes this process is also used in cases like bag of words (BOW) creation in data mining.

MORPHOLOGICAL ANALYSIS

Lemmatization

The task of removing inflectional endings only and to return the base dictionary form of a word which is also known as a lemma. Lemmatization is another technique for reducing words to their normalized form. But in this case, the transformation actually uses a dictionary to map words to their actual form.

Morphological segmentation

Separate words into individual morphemes and identify the class of the morphemes. The difficulty of this task depends greatly on the complexity of the morphology (*i.e.*, the structure of words) of the language being considered. English has fairly simple morphology, especially inflectional morphology, and thus it is often possible to ignore this task entirely and simply model all possible forms of a word (e.g., “open, opens, opened, opening”) as separate words. In languages such as Turkish or Meitei, a highly agglutinated Indian language, however, such an approach is not possible, as each dictionary entry has thousands of possible word forms.

Part-of-speech tagging

Given a sentence, determine the part of speech (POS) for each word. Many

words, especially common ones, can serve as multiple parts of speech. For example, "book" can be a noun ("the book on the table") or verb ("to book a flight"); "set" can be a noun, verb or adjective; and "out" can be any of at least five different parts of speech.

Stemming

The process of reducing inflected (or sometimes derived) words to a base form (e.g., "close" will be the root for "closed", "closing", "close", "closer" etc.). Stemming yields similar results as lemmatization, but does so on grounds of rules, not a dictionary.

SYNTACTIC ANALYSIS

Grammar induction

Generate a formal grammar that describes a language's syntax.

Sentence breaking (also known as "sentence boundary disambiguation")

Given a chunk of text, find the sentence boundaries. Sentence boundaries are often marked by periods or other punctuation marks, but these same characters can serve other purposes (e.g., marking abbreviations).

Parsing

Determine the parse tree (grammatical analysis) of a given sentence. The grammar for natural languages is ambiguous and typical sentences have multiple possible analyses: perhaps surprisingly, for a typical sentence there may be thousands of potential parses (most of which will seem completely nonsensical to a human). There are two primary types of parsing: *dependency parsing* and *constituency parsing*. Dependency parsing focuses on the relationships between words in a sentence (marking things like primary objects and predicates), whereas constituency parsing focuses on building out the parse tree using a probabilistic context-free grammar (PCFG).

LEXICAL SEMANTICS (OF INDIVIDUAL WORDS IN CONTEXT)

Lexical semantics

What is the computational meaning of individual words in context?

Distributional semantics

How can we learn semantic representations from data?

Named entity recognition (NER)

Given a stream of text, determine which items in the text map to proper names, such as people or places, and what the type of each such name is (e.g. person, location, organization). Although capitalization can aid in recognizing named entities in languages such as English, this information cannot aid in determining the type of named entity, and in any case, is often inaccurate or insufficient. For example, the first letter of a sentence is also capitalized, and named entities often span several words, only some of which are capitalized. Furthermore, many other languages in non-Western scripts (e.g. Chinese or Arabic) do not have any capitalization at all, and even languages with capitalization may not consistently use it to distinguish names. For example, German capitalizes all nouns, regardless of whether they are names, and French and Spanish do not capitalize names that serve as adjectives.

Sentiment analysis

Extract subjective information usually from a set of documents, often using online reviews to determine "polarity" about specific objects. It is especially useful for identifying trends of public opinion in social media, for marketing.

Terminology extraction

The goal of terminology extraction is to automatically extract relevant terms from a given corpus.

Word-sense disambiguation (WSD)

Many words have more than one meaning; we have to select the meaning which makes the most sense in context. For this problem, we are typically given a list of words and associated word senses, e.g. from a dictionary or an online resource such as WordNet.

Entity linking

Many words—typically proper names—refer to named entities; here we have to select the entity (a famous individual, a location, a company, etc.) which is referred to in context.

RELATIONAL SEMANTICS (SEMANTICS OF INDIVIDUAL SENTENCES)

Relationship extraction

Given a chunk of text, identify the relationships among named entities (e.g. who is married to whom).

Semantic parsing

Given a piece of text (typically a sentence), produce a formal representation of its semantics, either as a graph (e.g., in AMR parsing) or in accordance with a logical formalism (e.g., in DRT parsing). This challenge typically includes aspects of several more elementary NLP tasks from semantics (e.g., semantic role labelling, word-sense disambiguation) and can be extended to include full-fledged discourse analysis.

Semantic role labelling

Given a single sentence, identify and disambiguate semantic predicates (e.g., verbal frames), then identify and classify the frame elements (semantic roles).

DISCOURSE (SEMANTICS BEYOND INDIVIDUAL SENTENCES)

Coreference resolution

Given a sentence or larger chunk of text, determine which words ("mentions") refer to the same objects ("entities"). Anaphora resolution is a specific example of this task, and is specifically concerned with matching up pronouns with the nouns or names to which they refer. The more general task of coreference resolution also includes identifying so-called "bridging relationships" involving referring expressions. For example, in a sentence such as "He entered John's house through the front door", "the front door" is a referring expression and the bridging relationship to be identified is the fact that the door being referred to is the front door of John's house (rather than of some other structure that might also be referred to).

Discourse analysis

This rubric includes several related tasks. One task is discourse parsing, i.e., identifying the discourse structure of a connected text, i.e. the nature of the discourse relationships between sentences (e.g. elaboration, explanation, contrast). Another possible task is recognizing and classifying the speech acts in a chunk of text (e.g. yes-no question, content question, statement, assertion, etc.).

Implicit semantic role labelling

Given a single sentence, identify and disambiguate semantic predicates (e.g., verbal frames) and their explicit semantic roles in the current sentence. Then, identify semantic roles that are not explicitly realized in the current sentence, classify them into arguments that are explicitly realized elsewhere in the text and those that are not specified, and resolve the former against the local text. A closely

related task is zero anaphora resolution, i.e., the extension of coreference resolution to pro-drop languages.

Recognizing textual entailment

Given two text fragments, determine if one being true entails the other, entails the other's negation, or allows the other to be either true or false.

Topic segmentation and recognition

Given a chunk of text, separate it into segments each of which is devoted to a topic, and identify the topic of the segment.

Argument mining

The goal of argument mining is the automatic extraction and identification of argumentative structures from natural language text with the aid of computer programs. Such argumentative structures include the premise, conclusions, the argument scheme and the relationship between the main and subsidiary argument, or the main and counter-argument within discourse.

HIGHER-LEVEL NLP APPLICATIONS

Automatic summarization (text summarization)

Produce a readable summary of a chunk of text. Often used to provide summaries of the text of a known type, such as research papers, articles in the financial section of a newspaper.

Book generation

Not an NLP task proper but an extension of natural language generation and other NLP tasks is the creation of full-fledged books. The first machine-generated book was created by a rule-based system in 1984 (Racter, *The policeman's beard is half-constructed*). The first published work by a neural network was published in 2018, *1 the Road*, marketed as a novel, contains sixty million words. Both these systems are basically elaborate but non-sensical (semantics-free) language models. The first machine-generated science book was published in 2019 (Beta Writer, *Lithium-Ion Batteries*, Springer, Cham). Unlike *Racter* and *1 the Road*, this is grounded on factual knowledge and based on text summarization.

Dialogue management

Computer systems intended to converse with a human.

Document AI

A Document AI platform sits on top of the NLP technology enabling users

with no prior experience of artificial intelligence, machine learning or NLP to quickly train a computer to extract the specific data they need from different document types. NLP-powered Document AI enables non-technical teams to quickly access information hidden in documents, for example, lawyers, business analysts and accountants.

Grammatical error correction

Grammatical error detection and correction involves a great band-width of problems on all levels of linguistic analysis (phonology/orthography, morphology, syntax, semantics, pragmatics). Grammatical error correction is impactful since it affects hundreds of millions of people that use or acquire English as a second language. It has thus been subject to a number of shared tasks since 2011. As far as orthography, morphology, syntax and certain aspects of semantics are concerned, and due to the development of powerful neural language models such as GPT-2, this can now (2019) be considered a largely solved problem and is being marketed in various commercial applications.

Machine translation

Automatically translate text from one human language to another. This is one of the most difficult problems, and is a member of a class of problems colloquially termed “AI-complete”, i.e. requiring all of the different types of knowledge that humans possess (grammar, semantics, facts about the real world, etc.) to solve properly.

Natural-language generation (NLG):

Convert information from computer databases or semantic intents into readable human language.

Natural-language understanding (NLU)

Convert chunks of text into more formal representations such as first-order logic structures that are easier for computer programs to manipulate. Natural language understanding involves the identification of the intended semantic from the multiple possible semantics which can be derived from a natural language expression which usually takes the form of organized notations of natural language concepts. Introduction and creation of language metamodel and ontology are efficient however empirical solutions. An explicit formalization of natural language semantics without confusions with implicit assumptions such as closed-world assumption (CWA) vs. open-world assumption, or subjective Yes/No vs. objective True/False is expected for the construction of a basis of semantics formalization.

Question answering

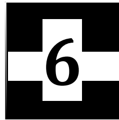
Given a human-language question, determine its answer. Typical questions have a specific right answer (such as “What is the capital of Canada?”), but sometimes open-ended questions are also considered (such as “What is the meaning of life?”).

Text-to-image generation

Given a description of an image, generate an image that matches the description.

Text-to-scene generation

Given a description of a scene, generate a 3D model of the scene.



Artificial Deep Neural Networks

Artificial neural networks (**ANNs**) or **connectionist systems** are computing systems inspired by the biological neural networks that constitute animal brains. Such systems learn (progressively improve their ability) to do tasks by considering examples, generally without task-specific programming. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the analytic results to identify cats in other images. They have found most use in applications difficult to express with a traditional computer algorithm using rule-based programming.

An ANN is based on a collection of connected units called artificial neurons, (analogous to biological neurons in a biological brain). Each connection (synapse) between neurons can transmit a signal to another neuron. The receiving (postsynaptic) neuron can process the signal(s) and then signal downstream neurons connected to it. Neurons may have state, generally represented by real numbers, typically between 0 and 1. Neurons and synapses may also have a weight that varies as learning proceeds, which can increase or decrease the strength of the signal that it sends downstream.

Typically, neurons are organized in layers. Different layers may perform different kinds of transformations on their inputs. Signals travel from the first (input), to the last (output) layer, possibly after traversing the layers multiple times.

The original goal of the neural network approach was to solve problems in the same way that a human brain would. Over time, attention focused on matching

specific mental abilities, leading to deviations from biology such as backpropagation, or passing information in the reverse direction and adjusting the network to reflect that information.

Neural networks have been used on a variety of tasks, including computer vision, speech recognition, machine translation, social network filtering, playing board and video games and medical diagnosis.

As of 2017, neural networks typically have a few thousand to a few million units and millions of connections. Despite this number being several order of magnitude less than the number of neurons on a human brain, these networks can perform many tasks at a level beyond that of humans (e.g., recognizing faces, or playing “Go”).

Deep neural networks

A deep neural network (DNN) is an artificial neural network (ANN) with multiple layers between the input and output layers. There are different types of neural networks but they always consist of the same components: neurons, synapses, weights, biases, and functions. These components as a whole function similarly to a human brain, and can be trained like any other ML algorithm.

For example, a DNN that is trained to recognize dog breeds will go over the given image and calculate the probability that the dog in the image is a certain breed.

The user can review the results and select which probabilities the network should display (above a certain threshold, etc.) and return the proposed label. Each mathematical manipulation as such is considered a layer, and complex DNN have many layers, hence the name “deep” networks.

DNNs can model complex non-linear relationships. DNN architectures generate compositional models where the object is expressed as a layered composition of primitives. The extra layers enable composition of features from lower layers, potentially modeling complex data with fewer units than a similarly performing shallow network. For instance, it was proved that sparse multivariate polynomials are exponentially easier to approximate with DNNs than with shallow networks.

Deep architectures include many variants of a few basic approaches. Each architecture has found success in specific domains. It is not always possible to compare the performance of multiple architectures, unless they have been evaluated on the same data sets.

DNNs are typically feedforward networks in which data flows from the input layer to the output layer without looping back. At first, the DNN creates a map of virtual neurons and assigns random numerical values, or “weights”, to connections

between them. The weights and inputs are multiplied and return an output between 0 and 1. If the network did not accurately recognize a particular pattern, an algorithm would adjust the weights. That way the algorithm can make certain parameters more influential, until it determines the correct mathematical manipulation to fully process the data.

Recurrent neural networks (RNNs), in which data can flow in any direction, are used for applications such as language modeling. Long short-term memory is particularly effective for this use. Convolutional deep neural networks (CNNs) are used in computer vision. CNNs also have been applied to acoustic modeling for automatic speech recognition (ASR).

Challenges

As with ANNs, many issues can arise with naively trained DNNs. Two common issues are overfitting and computation time.

DNNs are prone to overfitting because of the added layers of abstraction, which allow them to model rare dependencies in the training data. Regularization methods such as Ivakhnenko's unit pruning or weight decay or sparsity can be applied during training to combat overfitting. Alternatively dropout regularization randomly omits units from the hidden layers during training. This helps to exclude rare dependencies. Finally, data can be augmented via methods such as cropping and rotating such that smaller training sets can be increased in size to reduce the chances of overfitting.

DNNs must consider many training parameters, such as the size (number of layers and number of units per layer), the learning rate, and initial weights. Sweeping through the parameter space for optimal parameters may not be feasible due to the cost in time and computational resources.

Various tricks, such as batching (computing the gradient on several training examples at once rather than individual examples) speed up computation. Large processing capabilities of many-core architectures (such as GPUs or the Intel Xeon Phi) have produced significant speedups in training, because of the suitability of such processing architectures for the matrix and vector computations.

Alternatively, engineers may look for other types of neural networks with more straightforward and convergent training algorithms. CMAC (cerebellar model articulation controller) is one such kind of neural network. It doesn't require learning rates or randomized initial weights for CMAC. The training process can be guaranteed to converge in one step with a new batch of data, and the computational complexity of the training algorithm is linear with respect to the number of neurons involved.

HARDWARE

Since the 2010s, advances in both machine learning algorithms and computer hardware have led to more efficient methods for training deep neural networks that contain many layers of non-linear hidden units and a very large output layer. By 2019, graphic processing units (GPUs), often with AI-specific enhancements, had displaced CPUs as the dominant method of training large-scale commercial cloud AI. OpenAI estimated the hardware computation used in the largest deep learning projects from AlexNet (2012) to AlphaZero (2017), and found a 300,000-fold increase in the amount of computation required, with a doubling-time trendline of 3.4 months. Special electronic circuits called deep learning processors were designed to speed up deep learning algorithms. Deep learning processors include neural processing units (NPUs) in Huawei cellphones and cloud computing servers such as tensor processing units (TPU) in the Google Cloud Platform. Cerebras Systems has also built a dedicated system to handle large deep learning models, the CS-2, based on the largest processor in the industry, the second-generation Wafer Scale Engine (WSE-2).

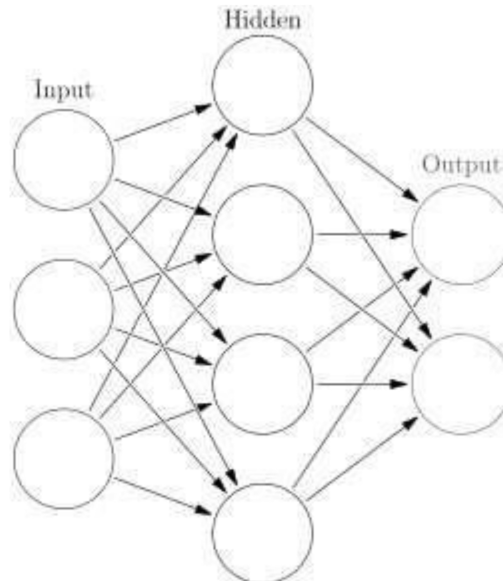
Atomically thin semiconductors are considered promising for energy-efficient deep learning hardware where the same basic device structure is used for both logic operations and data storage. In 2020, Marega et al. published experiments with a large-area active channel material for developing logic-in-memory devices and circuits based on floating-gate field-effect transistors (FGFETs).

In 2021, J. Feldmann et al. proposed an integrated photonic hardware accelerator for parallel convolutional processing. The authors identify two key advantages of integrated photonics over its electronic counterparts: (1) massively parallel data transfer through wavelength division multiplexing in conjunction with frequency combs, and (2) extremely high data modulation speeds. Their system can execute trillions of multiply-accumulate operations per second, indicating the potential of integrated photonics in data-heavy AI applications.

ARTIFICIAL NEURAL NETWORK

An artificial neural network is an interconnected group of nodes, inspired by a simplification of neurons in a brain. Here, each circular node represents an artificial neuron and an arrow represents a connection from the output of one artificial neuron to the input of another.

Artificial neural networks (**ANNs**), usually simply called **neural networks** (**NNs**) or **neural nets**, are computing systems inspired by the biological neural networks that constitute animal brains.



An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives signals then processes them and can signal neurons connected to it. The “signal” at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called *edges*. Neurons and edges typically have a *weight* that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold.

Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.

TRAINING

Neural networks learn (or are trained) by processing examples, each of which contains a known “input” and “result,” forming probability-weighted associations between the two, which are stored within the data structure of the net itself. The training of a neural network from a given example is usually conducted by determining the difference between the processed output of the network (often a prediction) and a target output. This difference is the error. The network then

adjusts its weighted associations according to a learning rule and using this error value. Successive adjustments will cause the neural network to produce output which is increasingly similar to the target output. After a sufficient number of these adjustments the training can be terminated based upon certain criteria. This is known as supervised learning.

Such systems "learn" to perform tasks by considering examples, generally without being programmed with task-specific rules. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images. They do this without any prior knowledge of cats, for example, that they have fur, tails, whiskers, and cat-like faces. Instead, they automatically generate identifying characteristics from the examples that they process.

History

Warren McCulloch and Walter Pitts (1943) opened the subject by creating a computational model for neural networks. In the late 1940s, D. O. Hebb created a learning hypothesis based on the mechanism of neural plasticity that became known as Hebbian learning. Farley and Wesley A. Clark (1954) first used computational machines, then called "calculators", to simulate a Hebbian network. In 1958, psychologist Frank Rosenblatt invented the perceptron, the first artificial neural network, funded by the United States Office of Naval Research. The first functional networks with many layers were published by Ivakhnenko and Lapa in 1965, as the Group Method of Data Handling. The basics of continuous backpropagation were derived in the context of control theory by Kelley in 1960 and by Bryson in 1961, using principles of dynamic programming. Thereafter research stagnated following Minsky and Papert (1969), who discovered that basic perceptrons were incapable of processing the exclusive-or circuit and that computers lacked sufficient power to process useful neural networks.

In 1970, Seppo Linnainmaa published the general method for automatic differentiation (AD) of discrete connected networks of nested differentiable functions. In 1973, Dreyfus used backpropagation to adapt parameters of controllers in proportion to error gradients. Werbos's (1975) backpropagation algorithm enabled practical training of multi-layer networks. In 1982, he applied Linnainmaa's AD method to neural networks in the way that became widely used.

The development of metal–oxide–semiconductor (MOS) very-large-scale integration (VLSI), in the form of complementary MOS (CMOS) technology, enabled increasing MOS transistor counts in digital electronics. This provided

more processing power for the development of practical artificial neural networks in the 1980s.

In 1986 Rumelhart, Hinton and Williams showed that backpropagation learned interesting internal representations of words as feature vectors when trained to predict the next word in a sequence.

From 1988 onward, the use of neural networks transformed the field of protein structure prediction, in particular when the first cascading networks were trained on *profiles* (matrices) produced by multiple sequence alignments.

In 1992, max-pooling was introduced to help with least-shift invariance and tolerance to deformation to aid 3D object recognition. Schmidhuber adopted a multi-level hierarchy of networks (1992) pre-trained one level at a time by unsupervised learning and fine-tuned by backpropagation.

Neural networks' early successes included predicting the stock market and in 1995 a (mostly) self-driving car.

Geoffrey Hinton et al. (2006) proposed learning a high-level representation using successive layers of binary or real-valued latent variables with a restricted Boltzmann machine to model each layer. In 2012, Ng and Dean created a network that learned to recognize higher-level concepts, such as cats, only from watching unlabeled images. Unsupervised pre-training and increased computing power from GPUs and distributed computing allowed the use of larger networks, particularly in image and visual recognition problems, which became known as "deep learning".

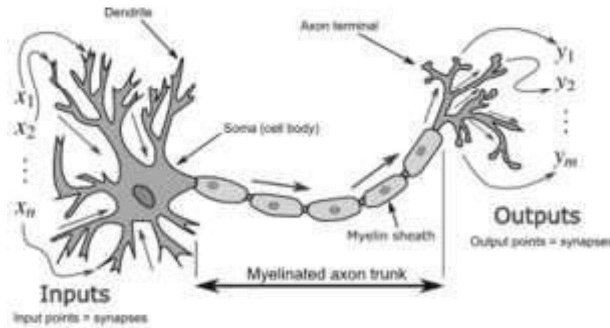
Ciresan and colleagues (2010) showed that despite the vanishing gradient problem, GPUs make backpropagation feasible for many-layered feedforward neural networks. Between 2009 and 2012, ANNs began winning prizes in image recognition contests, approaching human level performance on various tasks, initially in pattern recognition and handwriting recognition. For example, the bi-directional and multi-dimensional long short-term memory (LSTM) of Graves et al. won three competitions in connected handwriting recognition in 2009 without any prior knowledge about the three languages to be learned.

Ciresan and colleagues built the first pattern recognizers to achieve human-competitive/superhuman performance on benchmarks such as traffic sign recognition (IJCNN 2012).

MODELS

ANNs began as an attempt to exploit the architecture of the human brain to perform tasks that conventional algorithms had little success with. They soon reoriented towards improving empirical results, mostly abandoning attempts to remain true to their biological precursors. Neurons are connected to each other

in various patterns, to allow the output of some neurons to become the input of others. The network forms a directed, weighted graph.



Neuron and myelinated axon, with signal flow from inputs at dendrites to outputs at axon terminals

An artificial neural network consists of a collection of simulated neurons. Each neuron is a node which is connected to other nodes via links that correspond to biological axon-synapse-dendrite connections. Each link has a weight, which determines the strength of one node's influence on another.

Artificial neurons

ANNs are composed of artificial neurons which are conceptually derived from biological neurons. Each artificial neuron has inputs and produces a single output which can be sent to multiple other neurons. The inputs can be the feature values of a sample of external data, such as images or documents, or they can be the outputs of other neurons. The outputs of the final *output neurons* of the neural net accomplish the task, such as recognizing an object in an image.

To find the output of the neuron we take the weighted sum of all the inputs, weighted by the *weights* of the *connections* from the inputs to the neuron. We add a *bias* term to this sum. This weighted sum is sometimes called the *activation*. This weighted sum is then passed through a (usually nonlinear) activation function to produce the output. The initial inputs are external data, such as images and documents. The ultimate outputs accomplish the task, such as recognizing an object in an image.

Organization

The neurons are typically organized into multiple layers, especially in deep learning. Neurons of one layer connect only to neurons of the immediately preceding and immediately following layers. The layer that receives external data is the *input layer*. The layer that produces the ultimate result is the *output layer*.

In between them are zero or more *hidden layers*. Single layer and unlayered networks are also used. Between two layers, multiple connection patterns are possible. They can be 'fully connected', with every neuron in one layer connecting to every neuron in the next layer. They can be *pooling*, where a group of neurons in one layer connect to a single neuron in the next layer, thereby reducing the number of neurons in that layer. Neurons with only such connections form a directed acyclic graph and are known as *feedforward networks*. Alternatively, networks that allow connections between neurons in the same or previous layers are known as *recurrent networks.*

Hyperparameter

A hyperparameter is a constant parameter whose value is set before the learning process begins. The values of parameters are derived via learning. Examples of hyperparameters include learning rate, the number of hidden layers and batch size. The values of some hyperparameters can be dependent on those of other hyperparameters. For example, the size of some layers can depend on the overall number of layers.

Learning

Learning is the adaptation of the network to better handle a task by considering sample observations. Learning involves adjusting the weights (and optional thresholds) of the network to improve the accuracy of the result. This is done by minimizing the observed errors. Learning is complete when examining additional observations does not usefully reduce the error rate. Even after learning, the error rate typically does not reach 0. If after learning, the error rate is too high, the network typically must be redesigned. Practically this is done by defining a cost function that is evaluated periodically during learning. As long as its output continues to decline, learning continues. The cost is frequently defined as a statistic whose value can only be approximated. The outputs are actually numbers, so when the error is low, the difference between the output (almost certainly a cat) and the correct answer (cat) is small. Learning attempts to reduce the total of the differences across the observations. Most learning models can be viewed as a straightforward application of optimization theory and statistical estimation.

Learning rate

The learning rate defines the size of the corrective steps that the model takes to adjust for errors in each observation. A high learning rate shortens the training time, but with lower ultimate accuracy, while a lower learning rate takes longer, but with the potential for greater accuracy. Optimizations such as Quickprop are

primarily aimed at speeding up error minimization, while other improvements mainly try to increase reliability. In order to avoid oscillation inside the network such as alternating connection weights, and to improve the rate of convergence, refinements use an adaptive learning rate that increases or decreases as appropriate. The concept of momentum allows the balance between the gradient and the previous change to be weighted such that the weight adjustment depends to some degree on the previous change. A momentum close to 0 emphasizes the gradient, while a value close to 1 emphasizes the last change.

Cost function

While it is possible to define a cost function ad hoc, frequently the choice is determined by the function's desirable properties (such as convexity) or because it arises from the model (e.g. in a probabilistic model the model's posterior probability can be used as an inverse cost).

Backpropagation

Backpropagation is a method used to adjust the connection weights to compensate for each error found during learning. The error amount is effectively divided among the connections. Technically, backprop calculates the gradient (the derivative) of the cost function associated with a given state with respect to the weights. The weight updates can be done via stochastic gradient descent or other methods, such as Extreme Learning Machines, "No-prop" networks, training without backtracking, "weightless" networks, and non-connectionist neural networks.

Learning paradigms

Machine learning is commonly separated into three main learning paradigms, supervised learning, unsupervised learning and reinforcement learning. Each corresponds to a particular learning task.

Supervised learning

Supervised learning uses a set of paired inputs and desired outputs. The learning task is to produce the desired output for each input. In this case the cost function is related to eliminating incorrect deductions. A commonly used cost is the mean-squared error, which tries to minimize the average squared error between the network's output and the desired output. Tasks suited for supervised learning are pattern recognition (also known as classification) and regression (also known as function approximation). Supervised learning is also applicable to sequential data (e.g., for hand writing, speech and gesture recognition). This can be thought

of as learning with a "teacher", in the form of a function that provides continuous feedback on the quality of solutions obtained thus far.

Self-learning

Self-learning in neural networks was introduced in 1982 along with a neural network capable of self-learning named Crossbar Adaptive Array (CAA). It is a system with only one input, situation s, and only one output, action (or behavior) a. It has neither external advice input nor external reinforcement input from the environment. The CAA computes, in a crossbar fashion, both decisions about actions and emotions (feelings) about encountered situations. The system is driven by the interaction between cognition and emotion. Given the memory matrix, W =||w(a,s)||, the crossbar self-learning algorithm in each iteration performs the following computation:

```
In situation s perform action a;
Receive consequence situation s';
Compute emotion of being in consequence situation v(s');
Update crossbar memory w'(a,s) = w(a,s) + v(s').
```

The backpropagated value (secondary reinforcement) is the emotion toward the consequence situation. The CAA exists in two environments, one is behavioral environment where it behaves, and the other is genetic environment, where from it initially and only once receives initial emotions about to be encountered situations in the behavioral environment. Having received the genome vector (species vector) from the genetic environment, the CAA will learn a goal-seeking behavior, in the behavioral environment that contains both desirable and undesirable situations.

Neuroevolution

Neuroevolution can create neural network topologies and weights using evolutionary computation. It is competitive with sophisticated gradient descent approaches. One advantage of neuroevolution is that it may be less prone to get caught in "dead ends".

Stochastic neural network

Stochastic neural networks originating from Sherrington–Kirkpatrick models are a type of artificial neural network built by introducing random variations into the network, either by giving the network's artificial neurons stochastic transfer functions, or by giving them stochastic weights. This makes them useful tools for optimization problems, since the random fluctuations help the network escape from local minima.

Other

In a Bayesian framework, a distribution over the set of allowed models is chosen to minimize the cost. Evolutionary methods, gene expression programming, simulated annealing, expectation-maximization, non-parametric methods and particle swarm optimization are other learning algorithms. Convergent recursion is a learning algorithm for cerebellar model articulation controller (CMAC) neural networks.

Modes

Two modes of learning are available: stochastic and batch. In stochastic learning, each input creates a weight adjustment. In batch learning weights are adjusted based on a batch of inputs, accumulating errors over the batch. Stochastic learning introduces "noise" into the process, using the local gradient calculated from one data point; this reduces the chance of the network getting stuck in local minima. However, batch learning typically yields a faster, more stable descent to a local minimum, since each update is performed in the direction of the batch's average error. A common compromise is to use "mini-batches", small batches with samples in each batch selected stochastically from the entire data set.

TYPES

ANNs have evolved into a broad family of techniques that have advanced the state of the art across multiple domains. The simplest types have one or more static components, including number of units, number of layers, unit weights and topology. Dynamic types allow one or more of these to evolve via learning. The latter are much more complicated, but can shorten learning periods and produce better results. Some types allow/require learning to be "supervised" by the operator, while others operate independently. Some types operate purely in hardware, while others are purely software and run on general purpose computers.

Some of the main breakthroughs include: convolutional neural networks that have proven particularly successful in processing visual and other two-dimensional data; long short-term memory avoid the vanishing gradient problem and can handle signals that have a mix of low and high frequency components aiding large-vocabulary speech recognition, text-to-speech synthesis, and photo-real talking heads; competitive networks such as generative adversarial networks in which multiple networks (of varying structure) compete with each other, on tasks such as winning a game or on deceiving the opponent about the authenticity of an input.

NETWORK DESIGN

Neural architecture search (NAS) uses machine learning to automate ANN

design. Various approaches to NAS have designed networks that compare well with hand-designed systems. The basic search algorithm is to propose a candidate model, evaluate it against a dataset and use the results as feedback to teach the NAS network. Available systems include AutoML and AutoKeras.

Design issues include deciding the number, type and connectedness of network layers, as well as the size of each and the connection type (full, pooling, ...).

Hyperparameters must also be defined as part of the design (they are not learned), governing matters such as how many neurons are in each layer, learning rate, step, stride, depth, receptive field and padding (for CNNs), etc.

Use

Using Artificial neural networks requires an understanding of their characteristics.

- Choice of model: This depends on the data representation and the application. Overly complex models are slow learning.
- Learning algorithm: Numerous trade-offs exist between learning algorithms. Almost any algorithm will work well with the correct hyperparameters for training on a particular data set. However, selecting and tuning an algorithm for training on unseen data requires significant experimentation.
- Robustness: If the model, cost function and learning algorithm are selected appropriately, the resulting ANN can become robust.

ANN capabilities fall within the following broad categories:

- Function approximation, or regression analysis, including time series prediction, fitness approximation and modeling.
- Classification, including pattern and sequence recognition, novelty detection and sequential decision making.
- Data processing, including filtering, clustering, blind source separation and compression.
- Robotics, including directing manipulators and prostheses.

APPLICATIONS

Because of their ability to reproduce and model nonlinear processes, artificial neural networks have found applications in many disciplines. Application areas include system identification and control (vehicle control, trajectory prediction, process control, natural resource management), quantum chemistry, general game playing, pattern recognition (radar systems, face identification, signal classification,

3D reconstruction, object recognition and more), sensor data analysis, sequence recognition (gesture, speech, handwritten and printed text recognition), medical diagnosis, finance (e.g. automated trading systems), data mining, visualization, machine translation, social network filtering and e-mail spam filtering. ANNs have been used to diagnose several types of cancers and to distinguish highly invasive cancer cell lines from less invasive lines using only cell shape information.

ANNs have been used to accelerate reliability analysis of infrastructures subject to natural disasters and to predict foundation settlements. ANNs have also been used for building black-box models in geoscience: hydrology, ocean modelling and coastal engineering, and geomorphology. ANNs have been employed in cybersecurity, with the objective to discriminate between legitimate activities and malicious ones. For example, machine learning has been used for classifying Android malware, for identifying domains belonging to threat actors and for detecting URLs posing a security risk. Research is underway on ANN systems designed for penetration testing, for detecting botnets, credit cards frauds and network intrusions.

ANNs have been proposed as a tool to solve partial differential equations in physics and simulate the properties of many-body open quantum systems. In brain research ANNs have studied short-term behavior of individual neurons, the dynamics of neural circuitry arise from interactions between individual neurons and how behavior can arise from abstract neural modules that represent complete subsystems. Studies considered long-and short-term plasticity of neural systems and their relation to learning and memory from the individual neuron to the system level.

THEORETICAL PROPERTIES

Computational power

The multilayer perceptron is a universal function approximator, as proven by the universal approximation theorem. However, the proof is not constructive regarding the number of neurons required, the network topology, the weights and the learning parameters.

A specific recurrent architecture with rational-valued weights (as opposed to full precision real number-valued weights) has the power of a universal Turing machine, using a finite number of neurons and standard linear connections. Further, the use of irrational values for weights results in a machine with super-Turing power.

Capacity

A model's "capacity" property corresponds to its ability to model any given

function. It is related to the amount of information that can be stored in the network and to the notion of complexity. Two notions of capacity are known by the community. The information capacity and the VC Dimension. The information capacity of a perceptron is intensively discussed in Sir David MacKay's book which summarizes work by Thomas Cover. The capacity of a network of standard neurons (not convolutional) can be derived by four rules that derive from understanding a neuron as an electrical element. The information capacity captures the functions modelable by the network given any data as input. The second notion, is the VC dimension. VC Dimension uses the principles of measure theory and finds the maximum capacity under the best possible circumstances. This is, given input data in a specific form. As noted in, the VC Dimension for arbitrary inputs is half the information capacity of a Perceptron. The VC Dimension for arbitrary points is sometimes referred to as Memory Capacity.

Convergence

Models may not consistently converge on a single solution, firstly because local minima may exist, depending on the cost function and the model. Secondly, the optimization method used might not guarantee to converge when it begins far from any local minimum. Thirdly, for sufficiently large data or parameters, some methods become impractical.

Another issue worthy to mention is that training may cross some Saddle point which may lead the convergence to the wrong direction.

The convergence behavior of certain types of ANN architectures are more understood than others. When the width of network approaches to infinity, the ANN is well described by its first order Taylor expansion throughout training, and so inherits the convergence behavior of affine models. Another example is when parameters are small, it is observed that ANNs often fits target functions from low to high frequencies. This behavior is referred to as the spectral bias, or frequency principle, of neural networks. This phenomenon is the opposite to the behavior of some well studied iterative numerical schemes such as Jacobi method. Deeper neural networks have been observed to be more biased towards low frequency functions.

Generalization and statistics

Applications whose goal is to create a system that generalizes well to unseen examples, face the possibility of over-training. This arises in convoluted or over-specified systems when the network capacity significantly exceeds the needed free parameters. Two approaches address over-training. The first is to use cross-validation and similar techniques to check for the presence of over-training and

to select hyperparameters to minimize the generalization error. The second is to use some form of *regularization*. This concept emerges in a probabilistic (Bayesian) framework, where regularization can be performed by selecting a larger prior probability over simpler models; but also in statistical learning theory, where the goal is to minimize over two quantities: the 'empirical risk' and the 'structural risk', which roughly corresponds to the error over the training set and the predicted error in unseen data due to overfitting.

Confidence analysis of a neural network

Supervised neural networks that use a mean squared error (MSE) cost function can use formal statistical methods to determine the confidence of the trained model. The MSE on a validation set can be used as an estimate for variance. This value can then be used to calculate the confidence interval of network output, assuming a normal distribution. A confidence analysis made this way is statistically valid as long as the output probability distribution stays the same and the network is not modified.

By assigning a softmax activation function, a generalization of the logistic function, on the output layer of the neural network (or a softmax component in a component-based network) for categorical target variables, the outputs can be

interpreted as posterior probabilities. This is useful in classification as it gives a certainty measure on classifications.

The softmax activation function is:

CRITICISM

Training

A common criticism of neural networks, particularly in robotics, is that they require too much training for real-world operation. Potential solutions include randomly shuffling training examples, by using a numerical optimization algorithm that does not take too large steps when changing the network connections following an example, grouping examples in so-called mini-batches and/or introducing a recursive least squares algorithm for CMAC.

Theory

A fundamental objection is that ANNs do not sufficiently reflect neuronal function. Backpropagation is a critical step, although no such mechanism exists in biological neural networks. How information is coded by real neurons is not known. Sensor neurons fire action potentials more frequently with sensor activation and muscle cells pull more strongly when their associated motor neurons receive action potentials more frequently. Other than the case of relaying information from a sensor neuron to a motor neuron, almost nothing of the principles of how information is handled by biological neural networks is known.

A central claim of ANNs is that they embody new and powerful general principles for processing information. These principles are ill-defined. It is often claimed that they are emergent from the network itself. This allows simple statistical association (the basic function of artificial neural networks) to be described as learning or recognition. In 1997, Alexander Dewdney commented that, as a result, artificial neural networks have a "something-for-nothing quality, one that imparts a peculiar aura of laziness and a distinct lack of curiosity about just how good these computing systems are. No human hand (or mind) intervenes; solutions are found as if by magic; and no one, it seems, has learned anything". One response to Dewdney is that neural networks handle many complex and diverse tasks, ranging from autonomously flying aircraft to detecting credit card fraud to mastering the game of Go.

Technology writer Roger Bridgman commented:

Neural networks, for instance, are in the dock not only because they have been hyped to high heaven, (what hasn't?) but also because you could create a successful net without understanding how it worked: the bunch of numbers that captures its

behaviour would in all probability be "an opaque, unreadable table...valueless as a scientific resource". In spite of his emphatic declaration that science is not technology, Dewdney seems here to pillory neural nets as bad science when most of those devising them are just trying to be good engineers. An unreadable table that a useful machine could read would still be well worth having.

Biological brains use both shallow and deep circuits as reported by brain anatomy, displaying a wide variety of invariance. Weng argued that the brain self-wires largely according to signal statistics and therefore, a serial cascade cannot catch all major statistical dependencies.

Hardware

Large and effective neural networks require considerable computing resources. While the brain has hardware tailored to the task of processing signals through a graph of neurons, simulating even a simplified neuron on von Neumann architecture may consume vast amounts of memory and storage. Furthermore, the designer often needs to transmit signals through many of these connections and their associated neurons – which require enormous CPU power and time.

Schmidhuber noted that the resurgence of neural networks in the twenty-first century is largely attributable to advances in hardware: from 1991 to 2015, computing power, especially as delivered by GPGPUs (on GPUs), has increased around a million-fold, making the standard backpropagation algorithm feasible for training networks that are several layers deeper than before. The use of accelerators such as FPGAs and GPUs can reduce training times from months to days.

Neuromorphic engineering or a physical neural network addresses the hardware difficulty directly, by constructing non-von-Neumann chips to directly implement neural networks in circuitry. Another type of chip optimized for neural network processing is called a Tensor Processing Unit, or TPU.

Practical counterexamples

Analyzing what has been learned by an ANN is much easier than analyzing what has been learned by a biological neural network. Furthermore, researchers involved in exploring learning algorithms for neural networks are gradually uncovering general principles that allow a learning machine to be successful. For example, local vs. non-local learning and shallow vs. deep architecture.

Hybrid approaches

Advocates of hybrid models (combining neural networks and symbolic approaches), claim that such a mixture can better capture the mechanisms of the human mind.

REPRESENTATION POWER OF FUNCTIONS

The representation power of functions refers to their ability to model or approximate various types of relationships within data. This concept is crucial in fields such as machine learning, where functions are used to map inputs to outputs. A function with high representation power can capture complex patterns and relationships in data.

Expressiveness: This refers to the range of functions or models that can be represented. For example, linear functions have limited expressiveness, as they can only model linear relationships. In contrast, neural networks with sufficient depth and non-linear activation functions have high expressiveness, capable of modeling intricate, non-linear relationships.

This chapter covers the content discussed in the Representation Power of Functions module of the Deep Learning course and all the images are taken from the same module. So far we have seen three models: MP Neuron, Perceptron and Sigmoid Neuron but none of them were able to deal with the non-linearly separable data.

The Representation power of functions which will help us understand why we need complex functions as our model.

We need to find continuous functions and the reason for that is very simple as we are going to use the Gradient Descent Algorithm where we move in a direction opposite to the gradient while updating the parameters as per the below update rule.

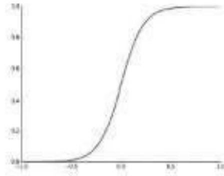
$$w_{t+1} = w_t - \eta \Delta w_t$$

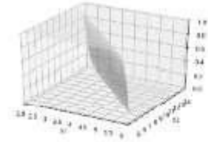
$$b_{t+1} = b_t - \eta \Delta b_t$$

$$\Delta w_t = \frac{\partial L}{\partial w}$$

$$\Delta b_t = \frac{\partial L}{\partial b}$$

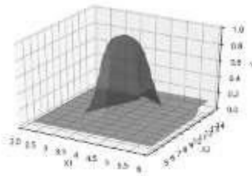
Now to compute the derivative with respect to the parameters, the function must be differentiable so that's why we want to have a continuous function. Example of continuous functions:





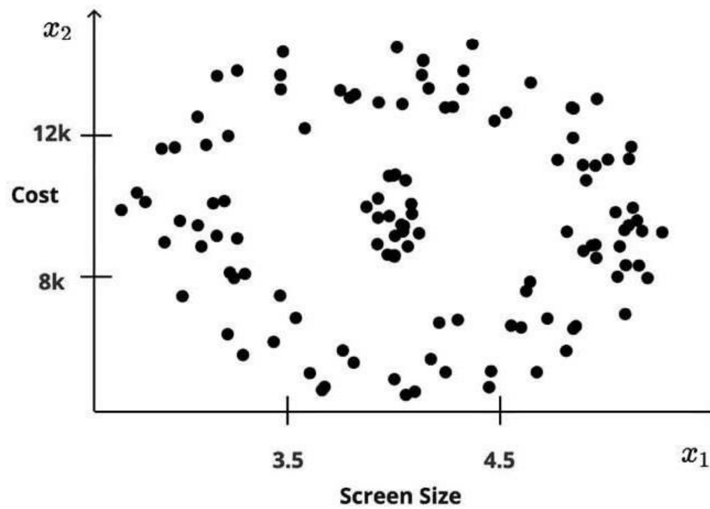
$$\hat{y} = \frac{1}{1+e^{-(2*x_1+5)}}$$

$$\hat{y} = \frac{1}{1+e^{-(-2*x_1+2*x_2+20)}}$$



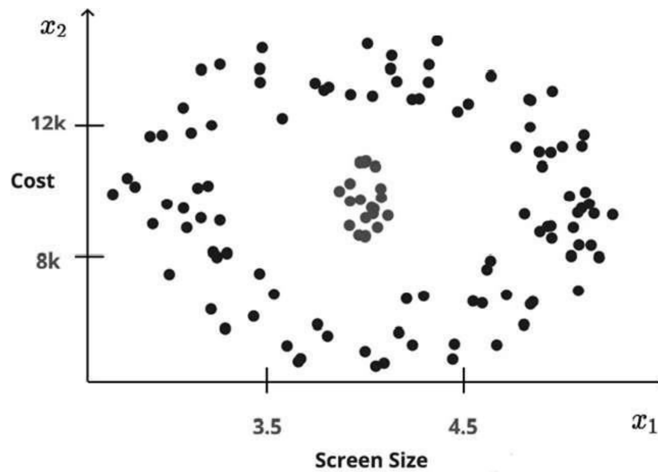
$$\hat{y} = sig_1(sig_2(x_1, x_2), sig_3(x_1, x_2), sig_4(x_1, x_2))$$

We need complex functions for modeling complex relations. Let's take an example where we have two features: Cost Price and screen size of a phone



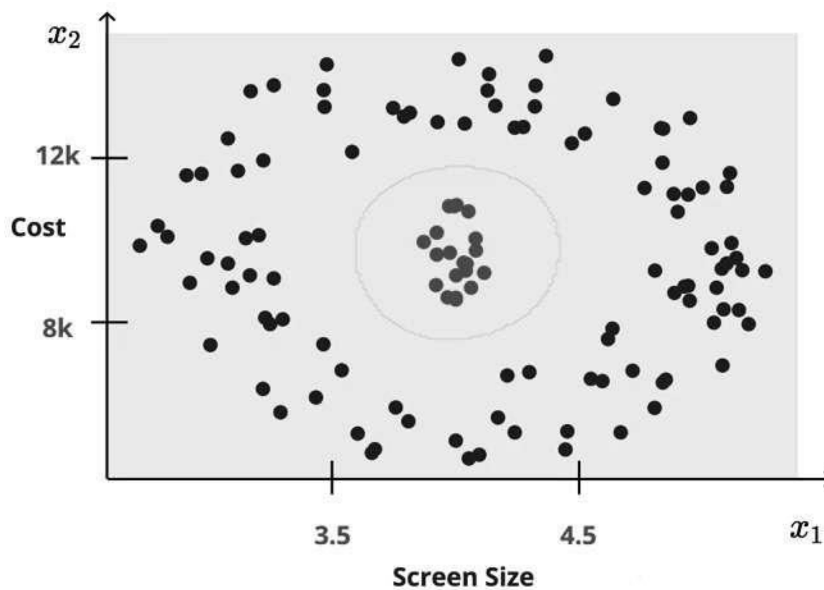
Every point corresponds to one phone.

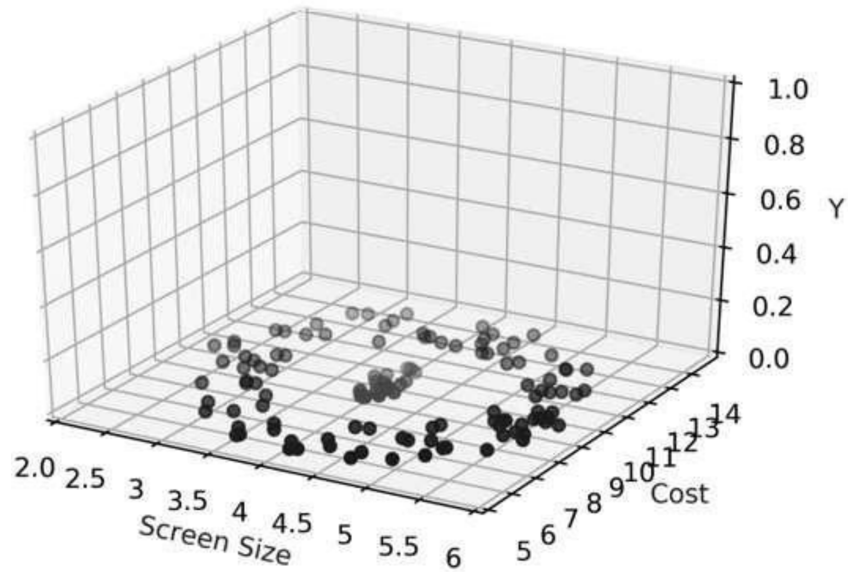
Let's say we like a phone whose screen size is between 3.5 to 4.5 and the price range is from 8k to 12k. So, data would look like:



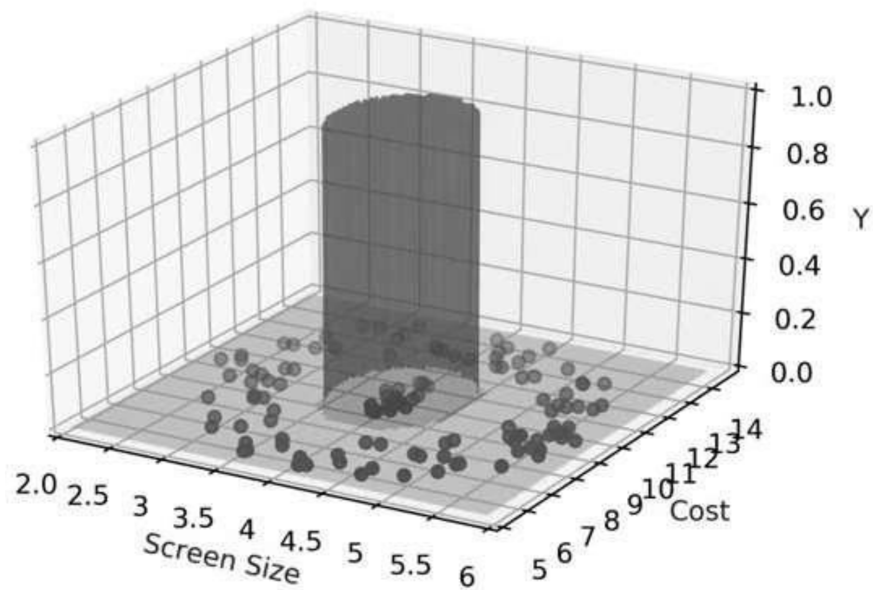
As is clear, this is non-linear data. We cannot draw a line in any way such that it can separate the red points from the blue points.

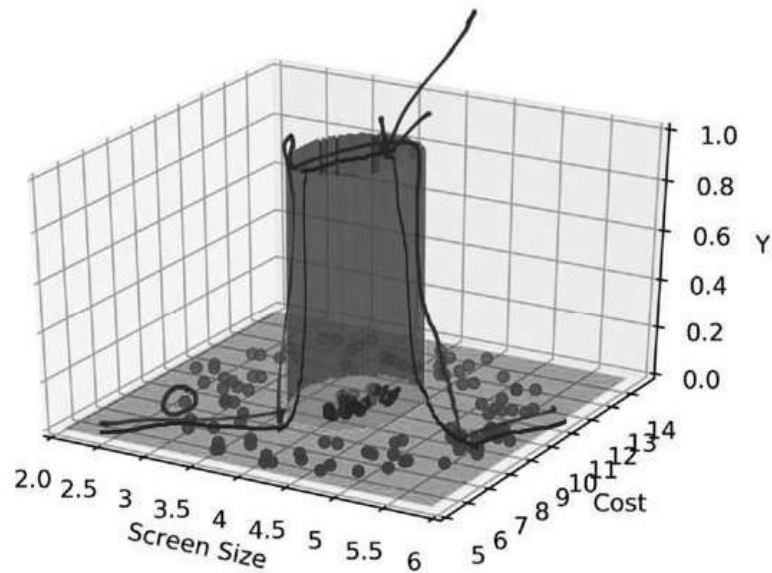
We want the model to be such that it gives an output of 0 for all the points in the green region(all the blue points) and it outputs 1 for all the points in the red region(all the red points) in the below image:



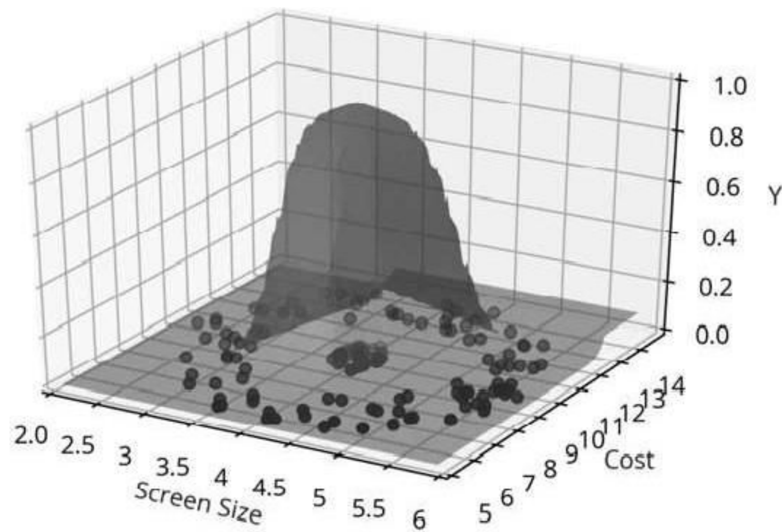


We want the function output to be 1 for the red points and the function output to be 0 for the blue points that mean we want our function to be of the following form:





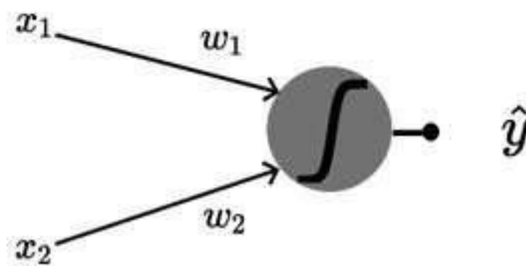
We can notice that this is not a very smooth function, it has sharp edges(in blue in the above image) which means it will not be differentiable at certain points. What we want is a very smooth function, a function of the following form:



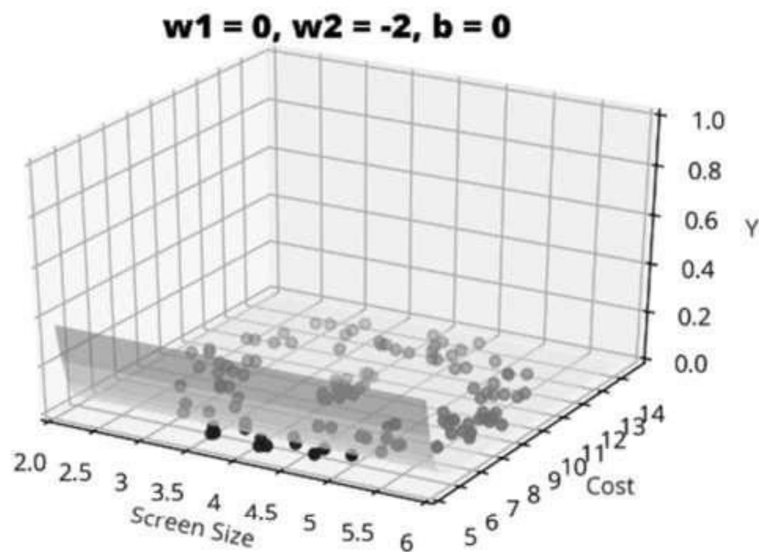
This is one of the reasons why we need complex functions because a lot of real-world data would require functions of above form and the above function is not like a Sigmoid Neuron(S-shaped), it's not like an MP Neuron or Perceptron(Linear), so we need different family of functions, we need more complex functions than what we have seen so far.

Below is the Sigmoid function that we have:

$$\hat{y} = \frac{1}{1+e^{-(w_1*x_1+w_2*x_2+b)}}$$



And if we use sigmoid to plot the data, we would get:



Now no matter what we set the values of parameters, we are not going to get a surface that can exactly fit the data.

CRASH COURSE ON MULTI-LAYER PERCEPTRON NEURAL NETWORKS

Artificial neural networks are a fascinating area of study, although they can be intimidating when just getting started. There is a lot of specialized terminology used when describing the data structures and algorithms used in the field. **Kick-start your project** with my new book Deep Learning With Python, including *step-by-step tutorials* and the *Python source code* files for all examples.

Let's get started.

Crash course in neural networks

Multi-Layer Perceptrons

The field of artificial neural networks is often just called neural networks or multi-layer perceptrons after perhaps the most useful type of neural network. A perceptron is a single neuron model that was a precursor to larger neural networks.

It is a field that investigates how simple models of biological brains can be used to solve difficult computational tasks like the predictive modeling tasks we see in machine learning. The goal is not to create realistic models of the brain but instead to develop robust algorithms and data structures that we can use to model difficult problems.

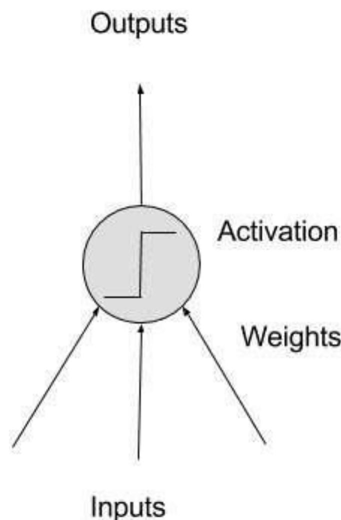
The power of neural networks comes from their ability to learn the representation in your training data and how best to relate it to the output variable you want to predict. In this sense, neural networks learn mapping. Mathematically, they are capable of learning any mapping function and have been proven to be a universal

approximation algorithm. The predictive capability of neural networks comes from the hierarchical or multi-layered structure of the networks. The data structure can pick out (learn to represent) features at different scales or resolutions and combine them into higher-order features, for example, from lines to collections of lines to shapes.

Neurons

The building blocks for neural networks are artificial neurons.

These are simple computational units that have weighted input signals and produce an output signal using an activation function.



Model of a simple neuron

NEURON WEIGHTS

You may be familiar with linear regression, where the weights on the inputs are very much like the coefficients used in a regression equation.

Like linear regression, each neuron also has a bias which can be thought of as an input that always has the value 1.0, and it, too, must be weighted.

For example, a neuron may have two inputs, which require three weights—one for each input and one for the bias.

Weights are often initialized to small random values, such as values from 0 to 0.3, although more complex initialization schemes can be used.

Like linear regression, larger weights indicate increased complexity and fragility. Keeping weights in the network is desirable, and regularization techniques can be used.

Activation

The weighted inputs are summed and passed through an activation function, sometimes called a transfer function.

An activation function is a simple mapping of summed weighted input to the output of the neuron. It is called an activation function because it governs the threshold at which the neuron is activated and the strength of the output signal.

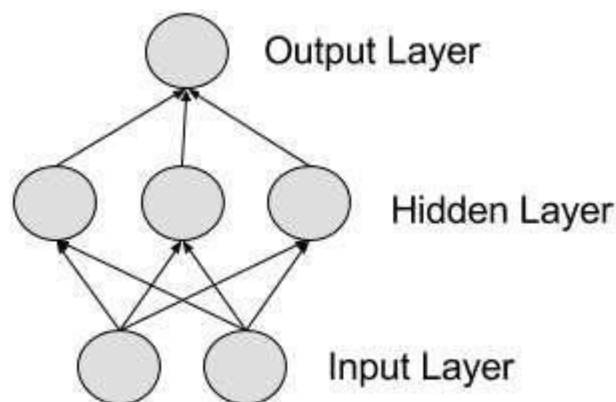
Historically, simple step activation functions were used when the summed input was above a threshold of 0.5, for example. Then the neuron would output a value of 1.0; otherwise, it would output a 0.0.

Traditionally, non-linear activation functions are used. This allows the network to combine the inputs in more complex ways and, in turn, provide a richer capability in the functions they can model. Non-linear functions like the logistic, also called the sigmoid function, were used to output a value between 0 and 1 with an s-shaped distribution. The hyperbolic tangent function, also called tanh, outputs the same distribution over the range -1 to +1.

Networks of Neurons

Neurons are arranged into networks of neurons.

A row of neurons is called a layer, and one network can have multiple layers. The architecture of the neurons in the network is often called the network topology.



Model of a simple network

INPUT OR VISIBLE LAYERS

The bottom layer that takes input from your dataset is called the visible layer because it is the exposed part of the network. Often a neural network is drawn with a visible layer with one neuron per input value or column in your dataset.

These are not neurons as described above but simply pass the input value through to the next layer.

Hidden Layers

Layers after the input layer are called hidden layers because they are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value.

Given increases in computing power and efficient libraries, very deep neural networks can be constructed. Deep learning can refer to having many hidden layers in your neural network. They are deep because they would have been unimaginably slow to train historically but may take seconds or minutes to train using modern techniques and hardware.

Output Layer

The final hidden layer is called the output layer, and it is responsible for outputting a value or vector of values that correspond to the format required for the problem.

The choice of activation function in the output layer is strongly constrained by the type of problem that you are modeling. For example:

- A regression problem may have a single output neuron, and the neuron may have no activation function.
- A binary classification problem may have a single output neuron and use a sigmoid activation function to output a value between 0 and 1 to represent the probability of predicting a value for the class 1. This can be turned into a crisp class value by using a threshold of 0.5 and snap values less than the threshold to 0, otherwise to 1.
- A multi-class classification problem may have multiple neurons in the output layer, one for each class (e.g., three neurons for the three classes in the famous iris flowers classification problem). In this case, a softmax activation function may be used to output a probability of the network predicting each of the class values. Selecting the output with the highest probability can be used to produce a crisp class classification value.

TRAINING NETWORKS

Once configured, the neural network needs to be trained on your dataset.

Data Preparation

You must first prepare your data for training on a neural network.

Data must be numerical, for example, real values. If you have categorical data, such as a sex attribute with the values "male" and "female," you can convert it to a real-valued representation called one-hot encoding. This is where one new column is added for each class value (two columns in the case of sex of male and female), and a 0 or 1 is added for each row depending on the class value for that row.

This same one-hot encoding can be used on the output variable in classification problems with more than one class. This would create a binary vector from a single column that would be easy to directly compare to the output of the neuron in the network's output layer. That, as described above, would output one value for each class.

Neural networks require the input to be scaled in a consistent way. You can rescale it to the range between 0 and 1, called normalization. Another popular technique is to standardize it so that the distribution of each column has a mean of zero and a standard deviation of 1. Scaling also applies to image pixel data. Data such as words can be converted to integers, such as the popularity rank of the word in the dataset and other encoding techniques.

Stochastic Gradient Descent

The classical and still preferred training algorithm for neural networks is called stochastic gradient descent.

This is where one row of data is exposed to the network at a time as input. The network processes the input upward, activating neurons as it goes to finally produce an output value. This is called a forward pass on the network. It is the type of pass that is also used after the network is trained in order to make predictions on new data.

The output of the network is compared to the expected output, and an error is calculated. This error is then propagated back through the network, one layer at a time, and the weights are updated according to the amount they contributed to the error. This clever bit of math is called the backpropagation algorithm.

The process is repeated for all of the examples in your training data. One round of updating the network for the entire training dataset is called an epoch. A network may be trained for tens, hundreds, or many thousands of epochs.

Weight Updates

The weights in the network can be updated from the errors calculated for each training example, and this is called online learning. It can result in fast but also chaotic changes to the network.

Alternatively, the errors can be saved across all the training examples, and the network can be updated at the end. This is called batch learning and is often more stable.

Typically, because datasets are so large and because of computational efficiencies, the size of the batch, the number of examples the network is shown before an update, is often reduced to a small number, such as tens or hundreds of examples.

The amount that weights are updated is controlled by a configuration parameter called the learning rate. It is also called the step size and controls the step or change made to a network weight for a given error. Often small weight sizes are used, such as 0.1 or 0.01 or smaller.

The update equation can be complemented with additional configuration terms that you can set.

- Momentum is a term that incorporates the properties from the previous weight update to allow the weights to continue to change in the same direction even when there is less error being calculated.
- Learning Rate Decay is used to decrease the learning rate over epochs to allow the network to make large changes to the weights at the beginning and smaller fine-tuning changes later in the training schedule.

Prediction

Once a neural network has been trained, it can be used to make predictions.

You can make predictions on test or validation data in order to estimate the skill of the model on unseen data. You can also deploy it operationally and use it to make predictions continuously. The network topology and the final set of weights are all you need to save from the model.

Predictions are made by providing the input to the network and performing a forward-pass, allowing it to generate an output you can use as a prediction.

AUTOMATIC SPEECH RECOGNITION

Large-scale automatic speech recognition is the first and most convincing successful case of deep learning. LSTM RNNs can learn "Very Deep Learning" tasks that involve multi-second intervals containing speech events separated by thousands of discrete time steps, where one time step corresponds to about 10 ms. LSTM with forget gates is competitive with traditional speech recognizers on certain tasks.

The initial success in speech recognition was based on small-scale recognition

tasks based on TIMIT. The data set contains 630 speakers from eight major dialects of American English, where each speaker reads 10 sentences. Its small size lets many configurations be tried. More importantly, the TIMIT task concerns phone-sequence recognition, which, unlike word-sequence recognition, allows weak phone bigram language models. This lets the strength of the acoustic modeling aspects of speech recognition be more easily analyzed. The error rates listed below, including these early results and measured as percent phone error rates (PER), have been summarized since 1991.

Method	Percent phoneerror rate (PER) (%)
Randomly Initialized RNN	26.1
Bayesian Triphone GMM-HMM	25.6
Hidden Trajectory (Generative) Model	24.8
Monophone Randomly Initialized DNN	23.4
Monophone DBN-DNN	22.4
Triphone GMM-HMM with BMMI Training	21.7
Monophone DBN-DNN on fbank	20.7
Convolutional DNN	20.0
Convolutional DNN w. Heterogeneous Pooling	18.7
Ensemble DNN/CNN/RNN	18.3
Bidirectional LSTM	17.8
Hierarchical Convolutional Deep Maxout Network	16.5

The debut of DNNs for speaker recognition in the late 1990s and speech recognition around 2009-2011 and of LSTM around 2003–2007, accelerated progress in eight major areas:

- Scale-up/out and accelerated DNN training and decoding
- Sequence discriminative training
- Feature processing by deep models with solid understanding of the underlying mechanisms
- Adaptation of DNNs and related deep models
- Multi-task and transfer learning by DNNs and related deep models
- CNNs and how to design them to best exploit domain knowledge of speech
- RNN and its rich LSTM variants
- Other types of deep models including tensor-based models and integrated deep generative/discriminative models.

All major commercial speech recognition systems (e.g., Microsoft Cortana, Xbox, Skype Translator, Amazon Alexa, Google Now, Apple Siri, Baidu and iFlyTek voice search, and a range of Nuance speech products, etc.) are based on deep learning.

Image recognition

A common evaluation set for image classification is the MNIST database data set. MNIST is composed of handwritten digits and includes 60,000 training examples and 10,000 test examples. As with TIMIT, its small size lets users test multiple configurations. A comprehensive list of results on this set is available.

Deep learning-based image recognition has become "superhuman", producing more accurate results than human contestants. This first occurred in 2011 in recognition of traffic signs, and in 2014, with recognition of human faces.

Deep learning-trained vehicles now interpret 360° camera views. Another example is Facial Dysmorphology Novel Analysis (FDNA) used to analyze cases of human malformation connected to a large database of genetic syndromes.

Visual art processing

Closely related to the progress that has been made in image recognition is the increasing application of deep learning techniques to various visual art tasks. DNNs have proven themselves capable, for example, of

- identifying the style period of a given painting
- Neural Style Transfer – capturing the style of a given artwork and applying it in a visually pleasing manner to an arbitrary photograph or video
- generating striking imagery based on random visual input fields.

Natural language processing

Neural networks have been used for implementing language models since the early 2000s. LSTM helped to improve machine translation and language modeling.

Other key techniques in this field are negative sampling and word embedding. Word embedding, such as *word2vec*, can be thought of as a representational layer in a deep learning architecture that transforms an atomic word into a positional representation of the word relative to other words in the dataset; the position is represented as a point in a vector space. Using word embedding as an RNN input layer allows the network to parse sentences and phrases using an effective compositional vector grammar. A compositional vector grammar can be thought of as probabilistic context free grammar (PCFG) implemented by an RNN. Recursive auto-encoders built atop word embeddings can assess sentence similarity and

detect paraphrasing. Deep neural architectures provide the best results for constituency parsing, sentiment analysis, information retrieval, spoken language understanding, machine translation, contextual entity linking, writing style recognition, Text classification and others.

Recent developments generalize word embedding to sentence embedding.

Google Translate (GT) uses a large end-to-end long short-term memory (LSTM) network. Google Neural Machine Translation (GNMT) uses an example-based machine translation method in which the system "learns from millions of examples." It translates "whole sentences at a time, rather than pieces. Google Translate supports over one hundred languages. The network encodes the "semantics of the sentence rather than simply memorizing phrase-to-phrase translations". GT uses English as an intermediate between most language pairs.

Drug discovery and toxicology

A large percentage of candidate drugs fail to win regulatory approval. These failures are caused by insufficient efficacy (on-target effect), undesired interactions (off-target effects), or unanticipated toxic effects. Research has explored use of deep learning to predict the biomolecular targets, off-targets, and toxic effects of environmental chemicals in nutrients, household products and drugs.

AtomNet is a deep learning system for structure-based rational drug design. AtomNet was used to predict novel candidate biomolecules for disease targets such as the Ebola virus and multiple sclerosis.

In 2017 graph neural networks were used for the first time to predict various properties of molecules in a large toxicology data set. In 2019, generative neural networks were used to produce molecules that were validated experimentally all the way into mice.

Customer relationship management

Deep reinforcement learning has been used to approximate the value of possible direct marketing actions, defined in terms of RFM variables. The estimated value function was shown to have a natural interpretation as customer lifetime value.

Recommendation systems

Recommendation systems have used deep learning to extract meaningful features for a latent factor model for content-based music and journal recommendations. Multi-view deep learning has been applied for learning user preferences from multiple domains. The model uses a hybrid collaborative and content-based approach and enhances recommendations in multiple tasks.

Bioinformatics

An autoencoder ANN was used in bioinformatics, to predict gene ontology annotations and gene-function relationships. In medical informatics, deep learning was used to predict sleep quality based on data from wearables and predictions of health complications from electronic health record data.

Medical image analysis

Deep learning has been shown to produce competitive results in medical application such as cancer cell classification, lesion detection, organ segmentation and image enhancement. Modern deep learning tools demonstrate the high accuracy of detecting various diseases and the helpfulness of their use by specialists to improve the diagnosis efficiency.

Mobile advertising

Finding the appropriate mobile audience for mobile advertising is always challenging, since many data points must be considered and analyzed before a target segment can be created and used in ad serving by any ad server. Deep learning has been used to interpret large, many-dimensioned advertising datasets. Many data points are collected during the request/serve/click internet advertising cycle. This information can form the basis of machine learning to improve ad selection.

Image restoration

Deep learning has been successfully applied to inverse problems such as denoising, super-resolution, inpainting, and film colorization. These applications include learning methods such as "Shrinkage Fields for Effective Image Restoration" which trains on an image dataset, and Deep Image Prior, which trains on the image that needs restoration.

Financial fraud detection

Deep learning is being successfully applied to financial fraud detection, tax evasion detection, and anti-money laundering. A potentially impressive demonstration of unsupervised learning as prosecution of financial crime is required to produce training data.

Also of note is that while the state of the art model in automated financial crime detection has existed for quite some time, the applications for deep learning referred to here dramatically under perform much simpler theoretical models. One such, yet to be implemented model, the Sensor Location Heuristic and Simple Any Human Detection for Financial Crimes (SLHSAHDFC), is an example.

The model works with the simple heuristic of choosing where it gets its input data. By placing the sensors by places frequented by large concentrations of wealth and power and then simply identifying any live human being, it turns out that the automated detection of financial crime is accomplished at very high accuracies and very high confidence levels. Even better, the model has shown to be extremely effective at identifying not just crime but large, very destructive and egregious crime. Due to the effectiveness of such models it is highly likely that applications to financial crime detection by deep learning will never be able to compete.

Military

The United States Department of Defense applied deep learning to train robots in new tasks through observation.

Partial differential equations

Physics informed neural networks have been used to solve partial differential equations in both forward and inverse problems in a data driven manner. One example is the reconstructing fluid flow governed by the Navier-Stokes equations. Using physics informed neural networks does not require the often expensive mesh generation that conventional CFD methods relies on.

Image Reconstruction

Image reconstruction is the reconstruction of the underlying images from the image-related measurements. Several works showed the better and superior performance of the deep learning methods compared to analytical methods for various applications, e.g., spectral imaging and ultrasound imaging.

Epigenetic clock

An **epigenetic clock** is a biochemical test that can be used to measure age. Galkin et al. used deep neural networks to train an epigenetic aging clock of unprecedented accuracy using >6,000 blood samples. The clock uses information from 1000 CpG sites and predicts people with certain conditions older than healthy controls: IBD, frontotemporal dementia, ovarian cancer, obesity. The aging clock is planned to be released for public use in 2021 by an Insilico Medicine spinoff company Deep Longevity.

DEEP LEARNING APPLICATIONS TO KNOW

FRAUD DETECTION

Fraud is a growing problem in the digital world. In 2021, consumers reported

2.8 million cases of fraud to the Federal Trade Commission. Identify theft and imposter scams were the two most common fraud categories.

To help prevent fraud, companies like Signifyd use deep learning to detect anomalies in user transactions. Those companies deploy deep learning to collect data from a variety of sources, including the device location, length of stride and credit card purchasing patterns to create a unique user profile.

Mastercard has taken a similar approach, leveraging its Decision Intelligence and AI Express platforms to more accurately detect fraudulent credit card activity. And for companies that rely on e-commerce, Riskified is making consumer finance easier by reducing the number of bad orders and chargebacks for merchants.

CUSTOMER RELATIONSHIP MANAGEMENT

Customer relationship management systems are often referred to as the "single source of truth" for revenue teams. They contain emails, phone call records and notes about all of the company's current and former customers as well as its prospects. Aggregating that information has helped revenue teams provide a better customer experience, but the introduction of deep learning in CRM systems has unlocked another layer of customer insights.

Deep learning is able to sift through all of the scraps of data a company collects about its prospects to reveal trends about why customers buy, when they buy and what keeps them around. This includes predictive lead scoring, which helps companies identify customers they have the best chances to close; scraping data from customer notes to make it easier to identify trends; and predictions about customer support needs.

COMPUTER VISION

Deep learning aims to mimic the way the human mind digests information and detects patterns, which makes it a perfect way to train vision-based AI programs. Using deep learning models, those platforms are able to take in a series of labeled photo sets to learn to detect objects like airplanes, faces and guns.

The application for image recognition is expansive. Neurala uses an algorithm it calls Lifelong-DNN to complete manufacturing quality inspections. Others, like ZeroEyes, use deep learning to detect firearms in public places like schools and government property. When a gun is detected, the system is designed to alert police in an effort to prevent shootings. And finally, companies like Motional rely on AI technologies to reinforce their LiDAR, radar and camera systems in autonomous vehicles.

Agriculture

Agriculture will remain a key source of food production in the coming years, so people have found ways to make the process more efficient with deep learning and AI tools. In fact, a 2021 *Forbes* article revealed that the agriculture industry is expected to invest $4 billion in AI solutions by 2026. Farmers have already found various uses for the technology, wielding AI to detect intrusive wild animals, forecast crop yields and power self-driving machinery.

Blue River Technology has explored the possibilities of self-driven farm products by combining machine learning, computer vision and robotics. The results have been promising, leading to smart machines — like a lettuce bot that knows how to single out weeds for chemical spraying while leaving plants alone. In addition, companies like Taranis blend computer vision and deep learning to monitor fields and prevent crop loss due to weeds, insects and other causes.

Vocal AI

When it comes to recreating human speech or translating voice to text, deep learning has a critical role to play. Deep learning models enable tools like Google Voice Search and Siri to take in audio, identify speech patterns and translate it into text. Then there's DeepMind's WaveNet model, which employs neural networks to take text and identify syllable patterns, inflection points and more. This enables companies like Google to train their virtual assistants to sound more human. In addition, Mozilla's 2017 RRNoise Project used it to identify and suppress background noise in audio files, providing users with clearer audio.

NATURAL LANGUAGE PROCESSING

The introduction of natural language processing technology has made it possible for robots to read messages and divine meaning from them. Still, the process can be somewhat oversimplified, failing to account for the ways that words combine together to change the meaning or intent behind a sentence.

Deep learning enables natural language processors to identify more complicated patterns in sentences to provide a more accurate interpretation. Companies like Gamalon use deep learning to power a chatbot that is able to respond to a larger volume of messages and provide more accurate responses.

Other companies like Strong apply it in its NLP tool to help users translate text, categorize text to help mine data from a collection of messages and identify sentiment in text. Grammarly also uses deep learning in combination with grammatical rules and patterns to help users identify the errors and tone of their messages.

Data Refining

When large amounts of raw data are collected, it's hard for data scientists to identify patterns, draw insights or do much with it. It needs to be processed. Deep learning models are able to take that raw data and make it accessible. Companies like Descartes Labs use a cloud-based supercomputer to refine data. Making sense of swaths of raw data can be useful for disease control, disaster mitigation, food security and satellite imagery.

Virtual Assistants

The divide between humans and machines continues to blur as virtual assistants become a part of everyday life. These AI-driven tools display a mix of AI, machine learning and deep learning techniques in order to process commands. Apple's Siri and Google's Google Assistant are two prominent examples, with both being able to operate across laptops, speakers, TVs and other devices. People can expect to see more virtual assistants and chatbots in the near future as the industry is on track to undergo plenty of growth through 2028.

Autonomous Vehicles

Driving is all about taking in external factors like the cars around you, street signs and pedestrians and reacting to them safely to get from point A to B. While we're still a ways away from fully autonomous vehicles, deep learning has played a crucial role in helping the technology come to fruition. It allows autonomous vehicles to take into account where you want to go, predict what the obstacles in your environment will do and create a safe path to get you to that location.

For instance, Zoox has used AI technologies to help its fully autonomous robotaxi vehicles learn from some of the most challenging driving situations to improve their decision-making under various circumstances. Other self-driving car companies that use deep learning to power their technology include Tesla-owned DeepScale and Waymo, a subsidiary of Google.

Supercomputers

While some software uses deep learning in its solution, if you want to build your own deep learning model, you need a supercomputer. Companies like Boxx and Nvidia have built workstations that can handle the processing power needed to build deep learning models. NVIDIA's DGX Station claims to be the "equivalent of hundreds of traditional servers," and enables users to test and tweak their models. Boxx's APEXX W-class products work with deep learning frameworks to provide more powerful processing and dependable computer performance.

Investment Modeling

Investment modeling is another industry that has benefited from deep learning. Predicting the market requires tracking and interpreting dozens of data points from earning call conversations to public events to stock pricing. Companies like Aiera use an adaptive deep learning platform to provide institutional investors with real-time analysis on individual equities, content from earnings calls and public company events. Even some of the bigger names like Morgan Stanley are joining the AI movement, using AI technologies to provide sound advice on wealth management through robo-advisors.

Climate Change

Organizations are stepping up to help people adapt to quickly accelerating environmental change. One Concern has emerged as a climate intelligence leader, factoring environmental events such as extreme weather into property risk assessments. Meanwhile, NCX has expanded the carbon-offset movement to include smaller landowners by using AI technology to create an affordable carbon marketplace.

E-commerce

Online shopping is now the de-facto way people purchase goods, but it can still be frustrating to scroll through dozens of pages to find the right pair of shoes that match your style. Some e-commerce companies are turning to deep learning to make the hunt easier.

Among Clarifai's many deep learning offerings is a tool that helps brands with image labeling to boost SEO traffic and surface alternative products for users when an item is out of stock. E-commerce giant eBay also applies a suite of AI, machine learning and deep learning techniques to power its global online marketplace and further enhance its search engine capabilities.

Emotional Intelligence

While computers may not be able to replicate human emotions, they are getting better at understanding our moods thanks to deep learning. Patterns like a shift in tone, a slight frown or a huff are all valuable data signals that can help AI detect our moods.

Companies like Affectiva are using deep learning to track all of those vocal and facial reactions to provide a nuanced understanding of our moods. Others like Cogito analyze the behaviors of customer service representatives to gauge their emotional intelligence and offer real-time advice for improved interactions.

Entertainment

Ever wonder how streaming platforms seem to intuit the perfect show for you to binge-watch next? Well, you have deep learning to thank for that. Streaming platforms aggregate tons of data on what content you choose to consume and what you ignore. Take Netflix as an example. The streaming platform uses machine learning to find patterns in what its viewers watch so that it can create a personalized experience for its users.

Deep Dreaming

Introduced back in 2015 by a team of Google engineers, the concept of deep dreaming has given another dimension to the realm of deep learning. Deep dreaming involves feeding algorithms to machines, which can then mimic the process of dreaming in human neural networks. A website called Deep Dream Generator has taken advantage of these algorithms, allowing creators to produce breathtaking digital art.

Advertising

Companies can glean a lot of information from how a user interacts with its marketing. It can signal intent to buy, show that the product resonates with them or that they want to learn more information. Many marketing tech firms are using deep learning to generate even more insights into customers. Companies like 6sense use deep learning to train their software to better understand buyers based on how they engage with an app or navigate a website. This can be used to help businesses more accurately target potential buyers and create tailored ad campaigns. Other firms like Dstillery use it to understand more about a customer's consumers to help each ad campaign reach the target audience for the product.

Manufacturing

The success of a factory often hinges on machines, humans and robots working together as efficiently as possible to produce a replicable product. When one part of the production gets out of whack, it can come at a devastating cost to the company. Deep learning is being used to make that process even more efficient and eliminate those errors.

Companies like OneTrack are using it to scan factory floors for anomalies like a teetering box or an improperly used forklift and alert workers to safety risks. The goal is to prevent errors that can slow down production and cause harm. Then there's Fanuc, which uses it to train its AI Error Proofing tool to discern good parts from bad parts during the manufacturing process. Energy giant General

Electric also uses deep learning in its Predix platform to track and find all possible points of failure on a factory floor.

Healthcare

The healthcare industry contends with inefficiencies, but deep learning plays a crucial role in streamlining the patient experience. KenSci, a company under the Advata umbrella, uses AI technology that learns from past performance data to predict how much space and what resources teams need to provide proper patient care. In addition, PathAI harnesses the predictive abilities of AI to garner more accurate data from drug research, clinical trials and patient diagnostics. Deep learning has also been proven to detect skin cancer through images, according to a National Center for Biotechnology Report.

Sports

Top-performing athletes are able to be more intentional about the ways they improve their games, thanks to AI-driven data. Companies like Hawk-Eye Innovations have raised the level of professional play through advanced replay systems, ball-tracking technology and timely game data. However, this attention to detail isn't reserved for sports royalty. Nex has built the HomeCourt app that basketball players of all skill levels can consult for insights on how to fine-tune their shooting motion and more.

APPLICATIONS OF DEEP LEARNING YOU NEED TO KNOW

For us humans, the novel idea of creating machines that can mimic human intellect, and even augment it, has been and still is, of great interest. And it is thanks to the efforts made around this idea that Artificial Intelligence, Machine Learning and later, Deep Learning came into existence.

Now, these three concepts, or rather technologies, are interesting on their own. However, citing the limitations of the topic we have at hand, we will primarily discuss Deep Learning here.

Deep Learning is probably the closest we have come so far to engineering a system based on the working of the brain. It is a complicated system that endeavors to solve problems and learn concepts that we were once limited to human intelligence.

Identifying images, translating human language, conversing with humans and assisting computers to make independent decisions are some of the innumerable purposes that Deep Learning fulfils.

Many businesses and organizations across the globe are now employing Deep Learning to fuel their growth and enhance their operations. They are using it to predict consumer behaviour, detect changes in market trends, create marketing strategies and whatnot.

Furthermore, according to ReportLinker, the market for Deep Learning will be valued at a whopping $44.3 Billion by 2027.

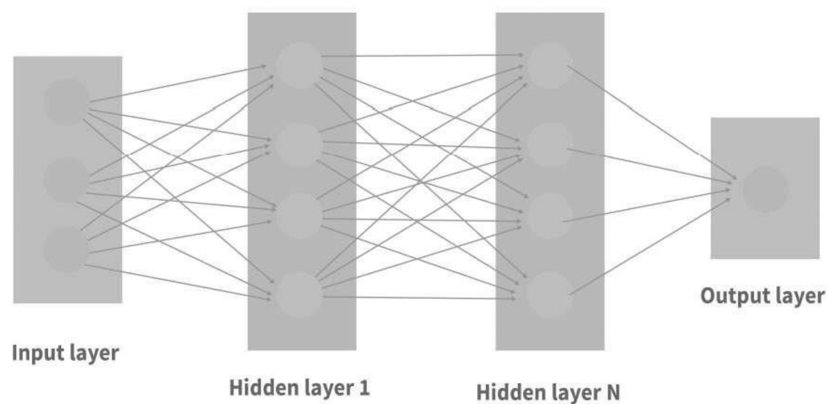
Thus, it makes sense for us to know about some of the most commonplace but significant Deep Learning applications in today's world. And that's what we are going to look at in this chapter on applications of deep learning. But before that, let us gather some insights into what deep learning is.

UNDERSTANDING DEEP LEARNING

Deep learning is fundamentally a subdiscipline of Machine Learning (hence the "learning" in the name). It bases its operations on another subset of machine learning called Neural Networks.

Neural Networks are networks of neurons or nodes – algorithmic locations where the computation of inputs takes place and output is produced. Neural networks are either biological or artificial, with the latter ones finding use in many AI-propelled applications.

These networks are essentially a somewhat less sophisticated reproduction of the biological structure of the human brain, with the nodes signifying the neurons or nerve cells.



Thus, Deep Learning is a machine learning technique composed of Artificial Neural Networks (ANNs) that are layered. Each layer comprises a certain number of neurons that receive input, compute it, and forward the output to the next layer until it reaches the final layer. Typically, except for the first layer, called the input

layer, and the last layer, the output layer, the rest of the layers stay hidden in a deep learning system.

APPLICATIONS OF DEEP LEARNING

Now, it is time we answered the million-dollar question, “which are common applications of deep learning in artificial intelligence(ai)?”

Healthcare

The healthcare sector has long been one of the prominent adopters of modern technology to overhaul itself. As such, it is not surprising to see Deep Learning finding uses in interpreting medical data for

- the diagnosis, prognosis & treatment of diseases
- drug prescription
- analysing MRIs, CT scans, ECG, X-Rays, etc., to detect and notify about medical anomalies
- personalising treatment
- monitoring the health of patients and more

One notable application of deep learning is found in the diagnosis and treatment of cancer.

Medical professionals use a CNN or Convolutional Neural Network, a Deep

learning method, to grade different types of cancer cells. They expose high-res histopathological images to deep CNN models after magnifying them 20X or 40X. The deep CNN models then demarcate various cellular features within the sample and detect carcinogenic elements.

Personalized Marketing

Personalized marketing is a concept that has seen much action in the recent few years. Marketers are now aiming their advertising campaigns at the pain points of individual consumers, offering them exactly what they need. And Deep Learning is playing a significant role in this.

Today, consumers are generating a lot of data thanks to their engagement with social media platforms, IoT devices, web browsers, wearables and the ilk. However, most of the data generated from these sources are disparate (text, audio, video, location data, etc.).

To cope with this, businesses use customisable Deep Learning models to interpret data from different sources and distil them to extract valuable customer insights. They then use this information to predict consumer behaviour and target their marketing efforts more efficiently.

So now you understand how those online shopping sites know what products to recommend to you.

Financial Fraud Detection

Virtually no sector is exempt from the evil called "fraudulent transactions" or "financial fraud". However, it is the financial corporations (banks, insurance firms, etc.) that have to bear the brunt of this menace the most. Not a day goes by when criminals attack financial institutions. There are a plethora of ways to usurp financial resources from them.

Thus, for these organizations, detecting and predicting financial fraud is critical, to say the least. And this is where Deep Learning comes into the picture.

Financial organizations are now using the concept of anomaly detection to flag inappropriate transactions. They employ deep learning algorithms, such as logistic regression (credit card fraud detection is a prime use case), decision trees, random forest, etc., to analyze the patterns common to valid transactions. Then, these models are put into action to flag financial transactions that seem potentially fraudulent.

Some examples of fraud detection being deterred by Deep Learning include:

- identity theft

- insurance fraud
- investment fraud
- fund misappropriation

Natural Language Processing

NLP or Natural Language Processing is another prominent area where Deep Learning is showing promising results.

Natural Language Processing, as the name suggests, is all about enabling machines to analyze and understand human language. The premise sounds simple, right? Well, the thing is, human language is punishingly complex for machines to interpret. It is not just the alphabet and words but also the context, the accents, the handwriting and whatnot that discourage machines from processing or generating human language accurately.

Deep Learning-based NLP is doing away with many of the issues related to understanding human language by training machines (Autoencoders and Distributed Representation) to produce appropriate responses to linguistic inputs.

One such example is the personal assistants we use on our smartphones. These applications come embedded with Deep Learning imbued NLP models to understand human speech and return appropriate output. It is, thus, no wonder why Siri and Alexa sound so much like how people talk in real life.

Another use case of Deep Learning-based NLP is how websites written in one human language automatically get translated to the user-specified language.

Autonomous Vehicles

The concept of building automated or self-governing vehicles goes back 45 years when the Tsukuba Mechanical Engineering Laboratory unveiled the world's first semi-automatic car. The car, a technological marvel then, carried a pair of cameras and an analogue computer to steer itself on a specially designed street.

However, it wasn't until 1989 when ALVINN (Autonomous Land Vehicle in a Neural Network), a modified military ambulance, used neural networks to navigate by itself on roads.

Since then, deep learning and autonomous vehicles have enjoyed a strong bond, with the former enhancing the latter's performance exponentially.

Autonomous vehicles use cameras, sensors – LiDARs, RADARs, motion sensors – and external information such as geo-mapping to perceive their environment and collect relevant data. They use this equipment both individually and in tandem for documenting the data.

This data is then fed to deep learning algorithms that direct the vehicle to perform appropriate actions such as

- accelerating, steering and braking
- identifying or planning routes
- traversing the traffic
- detecting pedestrians and other vehicles at a distance as well as in proximity
- recognising traffic signs

Deep learning is playing a huge role in realizing the perceived motives of self-driving vehicles of reducing road accidents, helping the disabled drive, eliminating traffic jams, etc.

And although still in nascent stages, the day is not far when we will see deep learning-powered vehicles form a majority of the road traffic.

Fake News Detection

The concept of spreading fake news to tip the scales in one's favour is not old. However, due to the explosive popularity of the internet, and social media platforms, in particular, fake news has become ubiquitous.

Fake news, apart from misinforming the citizens, can be used to alter political campaigns, vilify certain situations and individuals, and commit other similar morally illegible acts. As such, curbing any and all fake news becomes a priority.

Deep Learning proposes a way to deal with the menace of fake news by using complex language detection techniques to classify fraudulent news sources. This method essentially works by gathering information from trusty sources and juxtaposing them against a piece of news to verify its validity.

This paper explains how a combination of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) can validate digital news with high accuracy.

Facial Recognition

Facial Recognition is the technological method of identifying individuals from images and videos by documenting their faces. It uses advanced biometric technology to record a person's face and match it against a database to extract their identity.

Facial Recognition is an old technology, first conceptualized in the 1960s. However, it is the integration of neural networks in facial recognition that exponentially increased its detection accuracy.

Deep Learning enforced Facial Recognition works by recording face embeddings and using a trained model to map them against a huge database of millions of images.

For instance, DeepFace is a facial recognition method that uses Deep Learning (hence the name) to identify persons with a recorded 97% accuracy rate. It uses a nine-layer neural network for its purpose and has been trained using four million images of about 4000 individuals.

Recommendation Systems

Have you ever stopped to think about how Spotify knows which genres you listen to or, how Netflix recommends shows that match your preferences exactly? The short answer is Deep Learning. And the long answer, well, it is still deep learning but with some added explanation.

As discussed earlier, Deep Learning models process user data acquired from different sources and compile them to extract consumer info. This information then goes into deep learning-based recommender systems to generate appropriate suggestions for the users.

Deep Learning empowered suggestions, although widely used by audio/video streaming services, are not just limited to them. Social media networks use similar systems to recommend relevant posts, videos, accounts and more to users in their feeds.

Smart Agriculture

Artificial Intelligence and its subsets are fortifying a lot of industries and sectors, and agriculture is no different.

Of late, smart farming has become an active agricultural movement to improve upon the various aspects of traditional agriculture. Farmers are now using IoT

devices, satellite-based soil-composition detection, GPS, remote sensing, etc., to monitor and enhance their farming methods.

Deep Learning algorithms capture and analyse agriculture data from the above sources to improve crop health and soil health, predict the weather, detect diseases, etc.

Deep learning also finds uses in the field of crop genomics. Experts use neural networks to determine the genetic makeup of different crop plants and use it for purposes like

- increasing resilience to natural phenomena and diseases
- increase crop yield per unit area
- breeding high-quality hybrids

Space Travel

For most of us, space travel is something we associate with the most advanced technology available to humankind. We think of humanoid robots, hyper-intelligent AIs, hi-tech equipment, etc., working relentlessly in space to assist the astronauts in their painstaking endeavours.

However, while most of this stuff is over-the-top, it does signal one aspect of space flight – that it is technologically demanding.

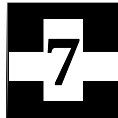
Scientists and engineers need to implement the latest and most efficient technologies – both hardware and software – to ensure the safety, integrity and success of space missions.

Thus, it goes without saying that AI, Machine Learning and Deep Learning are crucial components of everything astronomy.

For instance, ESA states that Deep Learning can be (and is, to some extent) used in

- automating the landing of rockets
- building space flight systems that can make intelligent decisions without human intervention

Also, deep learning will play an active role in helping future rovers on Mars to navigate and deduce their surroundings better and more independently.



Python Implementation of Neuron Model

We will see how to implement MP neuron model using python. The data set we will be using is breast cancer data set from sklearn. Before start building the MP Neuron Model. We will start by loading the data and separating the response and target variables.

Once we load the data, we can use the sklearn's train_test_split function to split the data into two parts — training and testing in the ratio of 80:20.

MP Neuron PreProcessing Steps

Remember from our previous discussion, MP Neuron takes only binary values as the input. So we need to convert the continuous features into binary format.

To achieve this, we will use pandas.cut function to split all the features into 0 or 1 in one single shot. Once we are ready with the inputs we need to build the model, train it on the training data and evaluate the model performance on the test data.

To create a MP Neuron Model we will create a class and inside this class, we will have three different functions:

- model function — to calculate the summation of the Binarized inputs.
- predict function — to predict the outcome for every observation in the data.
- fit function — the fit function iterates over all the possible values of the threshold **b** and find the best value of **b**, such that the loss will be minimum.

MP Neuron Model and Evaluation

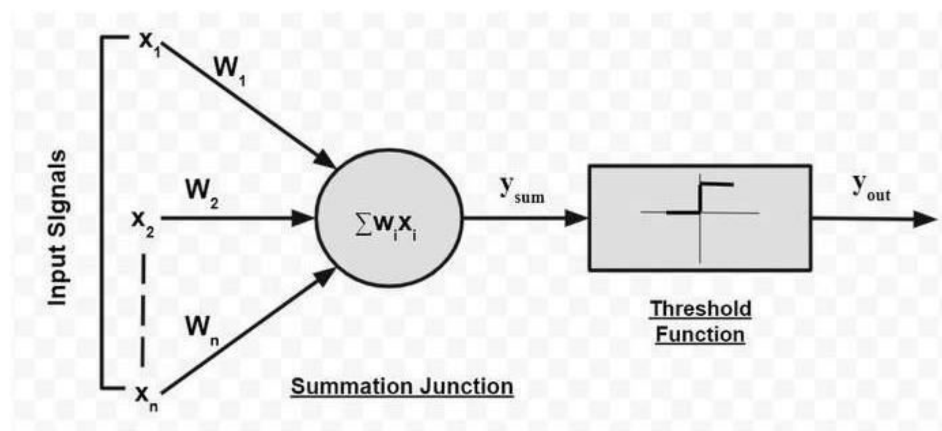
After building the model, test the model performance on the testing data and check the training data accuracy as well as the testing data accuracy.

MCCULLOCH-PITTS MODEL OF NEURON

The McCulloch-Pitts neural model, which was the earliest ANN model, has only two types of inputs — **Excitatory and Inhibitory.**

The excitatory inputs have weights of positive magnitude and the inhibitory weights have weights of negative magnitude.

The inputs of the McCulloch-Pitts neuron could be either 0 or 1. It has a threshold function as an activation function. So, the output signal y_{out} is 1 if the input y_{sum} is greater than or equal to a given threshold value, else 0. The diagrammatic representation of the model is as follows:



McCulloch-Pitts Model

Simple McCulloch-Pitts neurons can be used to design logical operations. For that purpose, the connection weights need to be correctly decided along with the threshold function (rather than the threshold value of the activation function). For better understanding purpose, let me consider an example:

John carries an umbrella if it is sunny or if it is raining. There are four given situations. I need to decide when John will carry the umbrella. The situations are as follows:

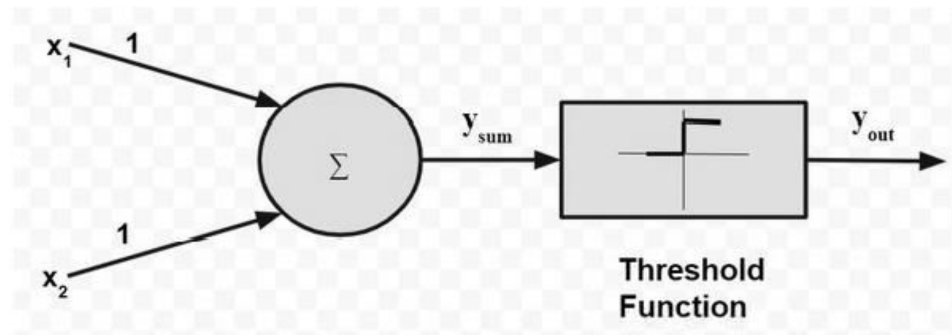
- First scenario: It is not raining, nor it is sunny
- Second scenario: It is not raining, but it is sunny

- Third scenario: It is raining, and it is not sunny
- Fourth scenario: It is raining as well as it is sunny

To analyse the situations using the McCulloch-Pitts neural model, I can consider the input signals as follows:

- X_1: Is it raining?
- X_2 : Is it sunny?

So, the value of both scenarios can be either 0 or 1. We can use the value of both weights X_1 and X_2 as 1 and a threshold function as 1. So, the neural network model will look like:



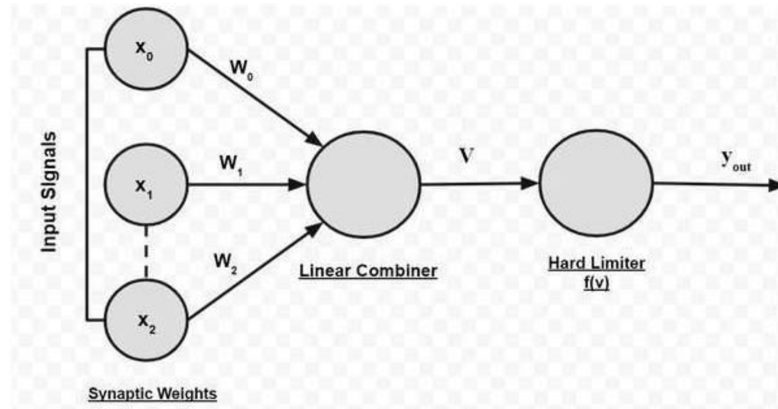
Truth Table for this case will be:

Situation	x_1	x_2	y_{sum}	y_{out}
1	0	0	0	0
2	0	1	1	1
3	1	0	1	1
4	1	1	2	1

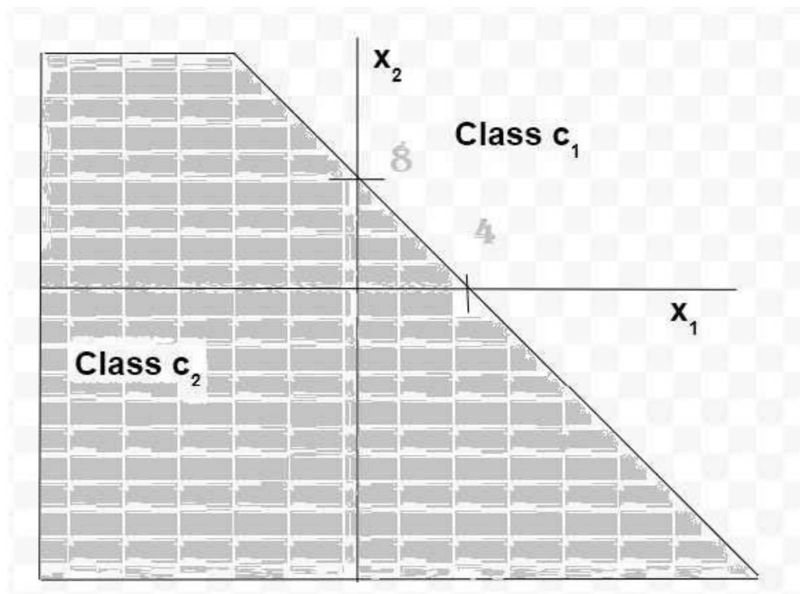
The truth table built with respect to the problem is depicted above. From the truth table, I can conclude that in the situations where the value of y_{out} is 1, John needs to carry an umbrella. Hence, he will need to carry an umbrella in scenarios 2, 3 and 4.

Rosenblatt's Perceptron

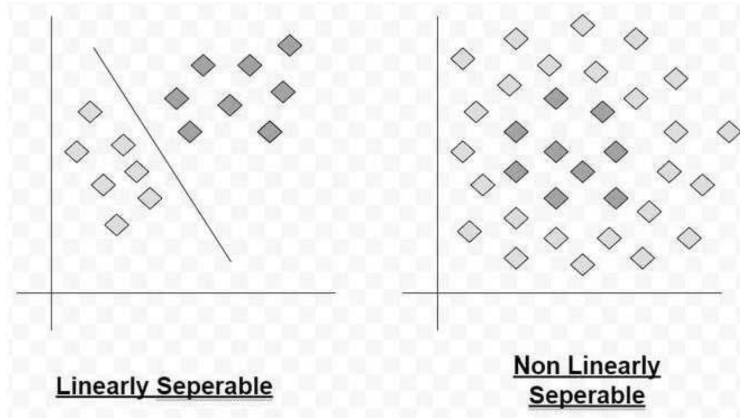
Rosenblatt's perceptron is built around the McCulloch-Pitts neural model. The diagrammatic representation is as follows:



So, any point (x_1, x_2) which lies above the decision boundary, as depicted by the graph, will be assigned to class c1 and the points which lie below the boundary are assigned to class c2.



Thus, we see that for a data set with linearly separable classes, perceptrons can always be employed to solve classification problems using decision lines (for 2-dimensional space), decision planes (for 3-dimensional space) or decision hyperplanes (for n-dimensional space). Appropriate values of the synaptic weights can be obtained by training a perceptron. However, one assumption for perceptron to work properly is that the two classes should be linearly separable i.e. the classes should be sufficiently separated from each other. Otherwise, if the classes are non-linearly separable, then the classification problem cannot be solved by perceptron.

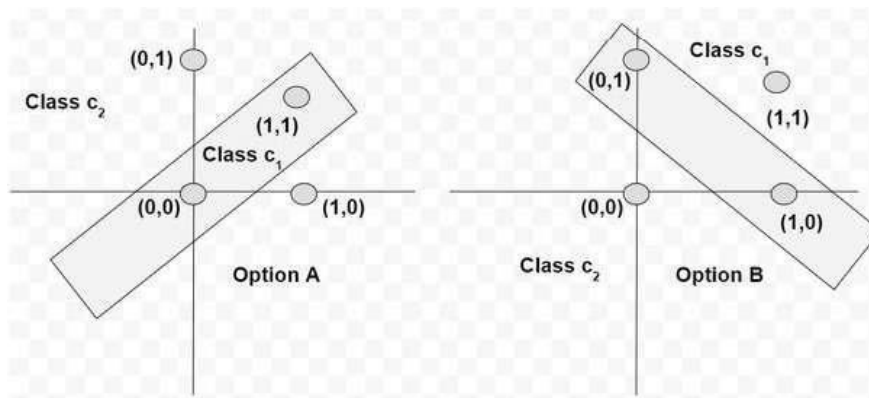


Linear Vs Non-Linearly Separable Classes

Multi-layer perceptron: A basic perceptron works very successfully for data sets which possess linearly separable patterns. However, in practical situations, that is an ideal situation to have. This was exactly the point driven by Minsky and Papert in their work in 1969. They showed that a basic perceptron is not able to learn to compute even a simple 2 bit XOR. So, let us understand the reason. Consider a truth table highlighting output of a 2 bit XOR function:

x_1	x_2	x_1 XOR x_2	Class
1	1	0	c_2
1	0	1	c_1
0	1	1	c_1
0	0	0	c_2

The data is not linearly separable. Only a curved decision boundary can separate the classes properly. To address this issue, the other option is to use two decision boundary lines in place of one.



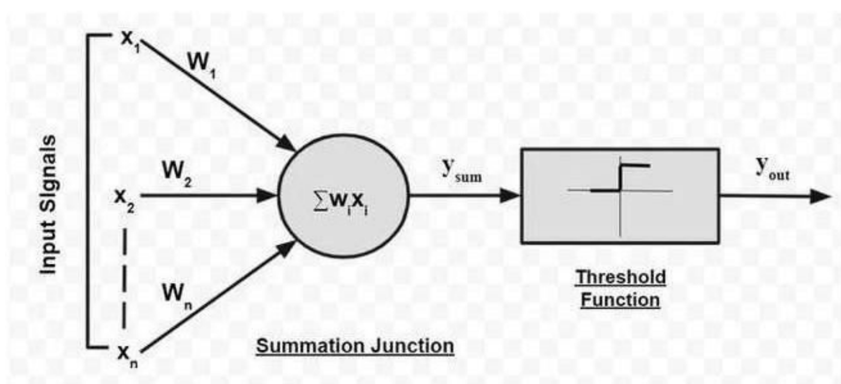
Classification with two decision lines in the XOR function output

This is the philosophy used to design the multi-layer perceptron model. The major highlights of this model are as follows:

- The neural network contains one or more intermediate layers between the input and output nodes, which are hidden from both input and output nodes
- Each neuron in the network includes a non-linear activation function that is differentiable.
- The neurons in each layer are connected with some or all the neurons in the previous layer.

ADALINE Network Model

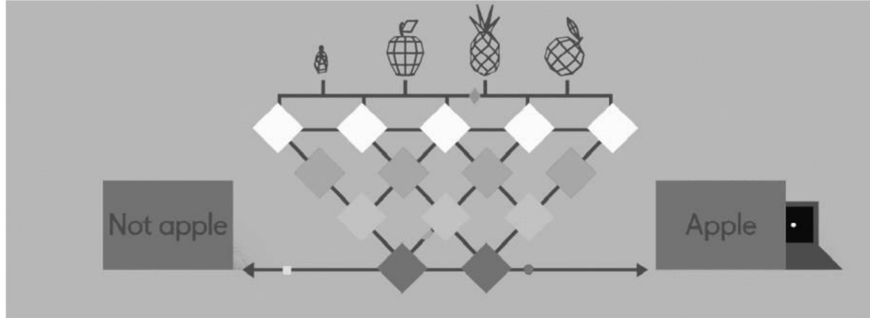
Adaptive Linear Neural Element (ADALINE) is an early single-layer ANN developed by Professor Bernard Widrow of Stanford University. As depicted in the below diagram, it has only output neurons.



The output value can be +1 or -1. A bias input x_0 (where x_0 =1) having a weight w_0 is added. The activation function is such that if weighted sum is positive or 0, the output is 1, else it is -1. Formally I can say that. The supervised learning algorithm adopted by ADALINE network is known as **Least Mean Square (LMS)** or **DELTA Rule.** A network combining a number of ADALINE is termed as **MADALINE (many ADALINE)**. MEADALINE networks can be used to solve problems related to non-linear separability.

ACTIVATION FUNCTIONS IN NEURAL NETWORKS

Activation functions play a crucial role in neural networks by introducing non-linearity into the model, enabling it to learn and model complex patterns in the data. They determine whether a neuron should be activated or not, thus influencing the output of the neural network.



ACTIVATION FUNCTION

It's just a thing function that you use to get the output of node. It is also known as **Transfer Function**.

Activation functions with Neural Networks

It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function). The Activation Functions can be basically divided into 2 types-

1. Linear Activation Function
2. Non-linear Activation Functions

Linear or Identity Activation Function

As you can see the function is a line or linear. Therefore, the output of the functions will not be confined between any range.

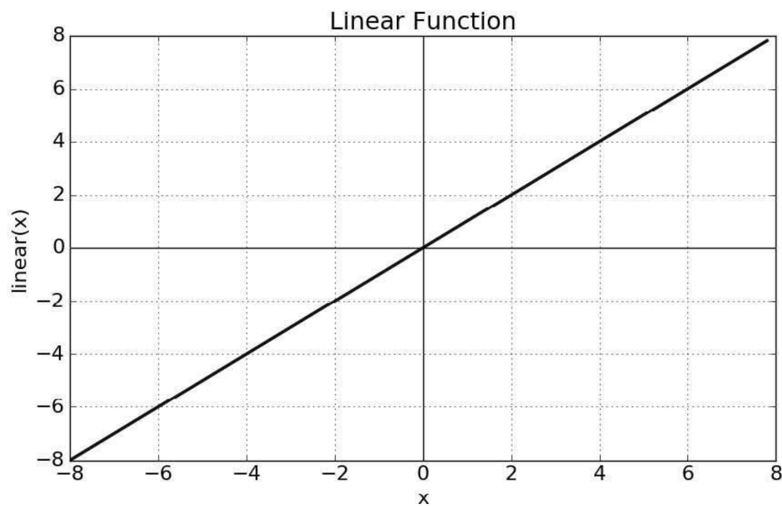


Fig: Linear Activation Function

Equation : f(x) = x

Range : (-infinity to infinity)

It doesn't help with the complexity or various parameters of usual data that is fed to the neural networks.

Non-linear Activation Function

The Nonlinear Activation Functions are the most used activation functions. Nonlinearity helps to makes the graph look something like this

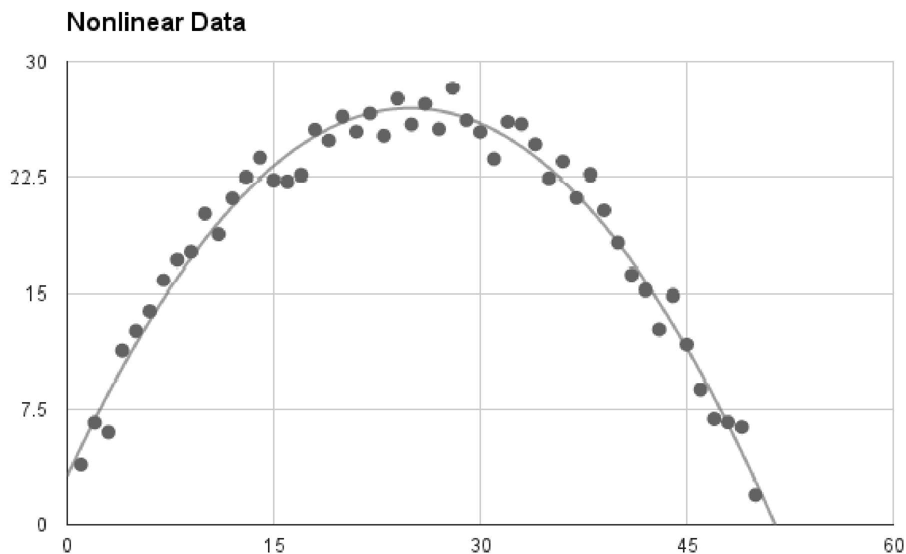


Fig: Non-linear Activation Function

It makes it easy for the model to generalize or adapt with variety of data and to differentiate between the output.

The main terminologies needed to understand for nonlinear functions are:

Derivative or Differential

Change in y-axis w.r.t. change in x-axis.It is also known as slope.

Monotonic function

A function which is either entirely non-increasing or non-decreasing.

The Nonlinear Activation Functions are mainly divided on the basis of their **range or curves**-

Sigmoid or Logistic Activation Function

The Sigmoid Function curve looks like a S-shape.

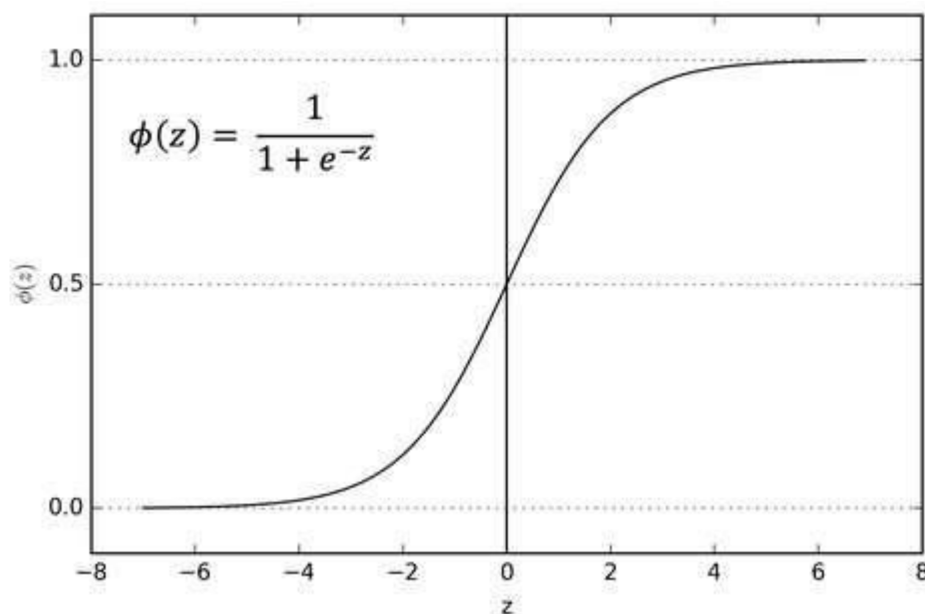


Fig: Sigmoid Function

The main reason why we use sigmoid function is because it exists between **(0 to 1).** Therefore, it is especially used for models where we have to **predict the probability** as an output.Since probability of anything exists only between the range of **0 and 1,** sigmoid is the right choice.

The function is **differentiable**.That means, we can find the slope of the sigmoid curve at any two points.

The function is **monotonic** but function's derivative is not.

The logistic sigmoid function can cause a neural network to get stuck at the training time.

The **softmax function** is a more generalized logistic activation function which is used for multiclass classification.

Tanh or hyperbolic tangent Activation Function

tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped).

The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.

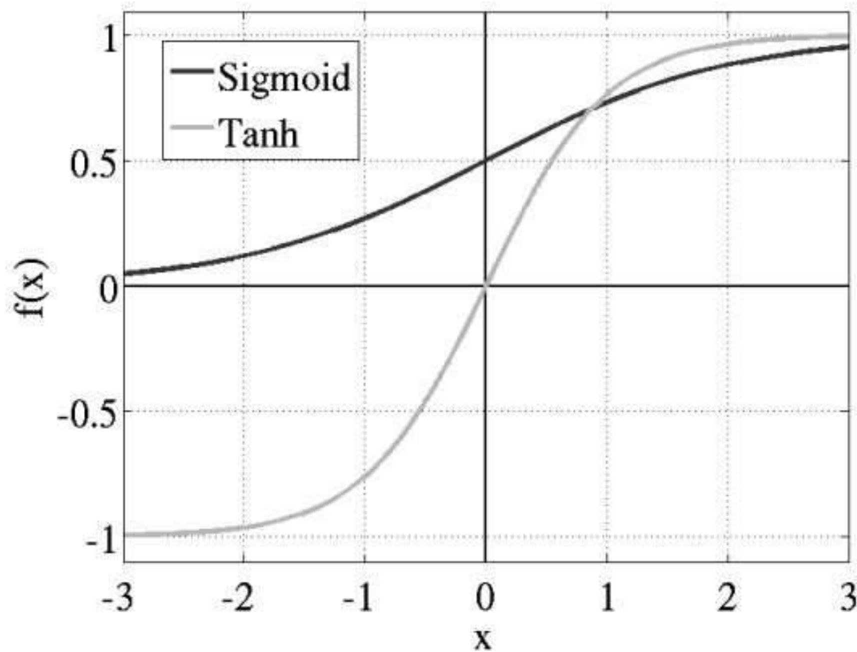


Fig: tanh v/s Logistic Sigmoid

The function is **differentiable**.

The function is **monotonic** while its **derivative is not monotonic**.

The tanh function is mainly used classification between two classes.

Both tanh and logistic sigmoid activation functions are used in feed-forward nets.

ReLU (Rectified Linear Unit) Activation Function

The ReLU is the most used activation function in the world right now.Since, it is used in almost all the convolutional neural networks or deep learning.

As you can see, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

Range: [0 to infinity)

The function and its derivative **both are monotonic**.

But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero

immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately.

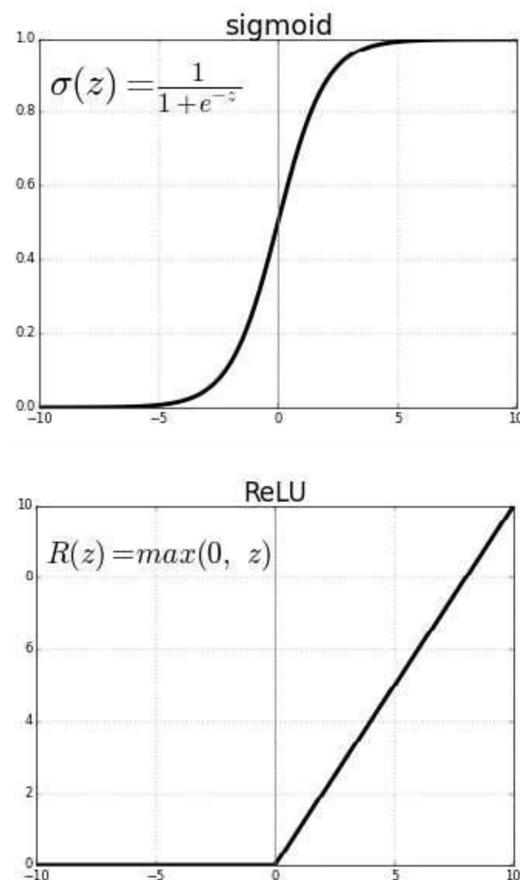


Fig: ReLU v/s Logistic Sigmoid

Leaky ReLU

It is an attempt to solve the dying ReLU problem

Can you see the Leak? =

The leak helps to increase the range of the ReLU function. Usually, the value of **a** is 0.01 or so.

When **a is not 0.01** then it is called **Randomized ReLU**.

Therefore the **range** of the Leaky ReLU is (-infinity to infinity).

Both Leaky and Randomized ReLU functions are monotonic in nature. Also, their derivatives also monotonic in nature.

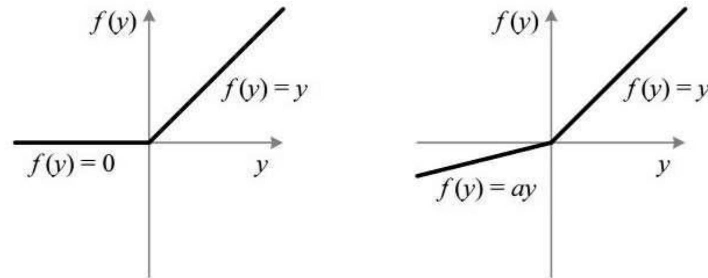


Fig : ReLU v/s Leaky ReLU

Why derivative/differentiation is used ?

When updating the curve, to know in **which direction** and **how much** to change or update the curve depending upon the slope.That is why we use differentiation in almost every part of Machine Learning and Deep Learning.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Fig: Activation Function Cheetsheet

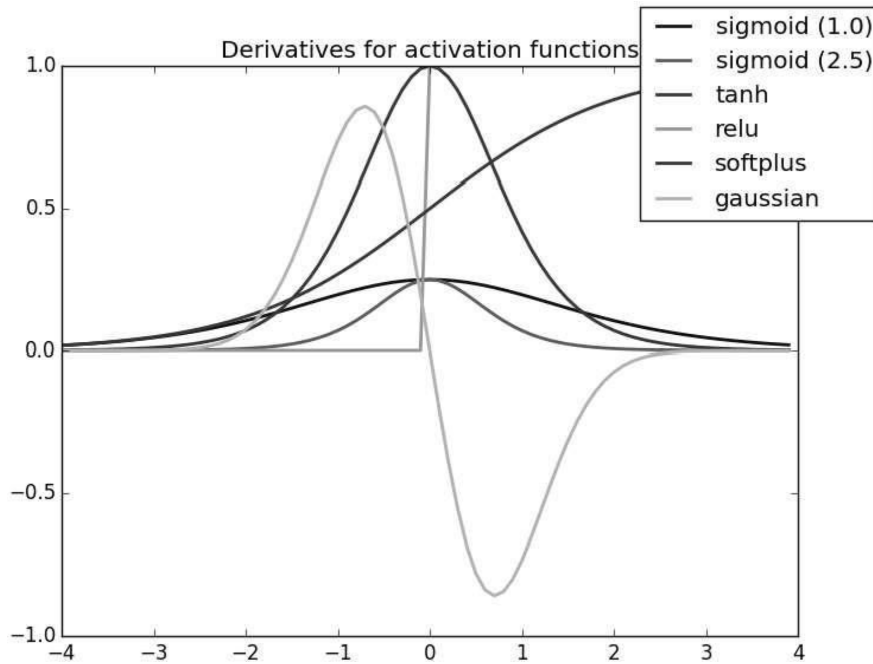


Fig: Derivative of Activation Functions

COMPONENTS OF NEURAL NETWORKS

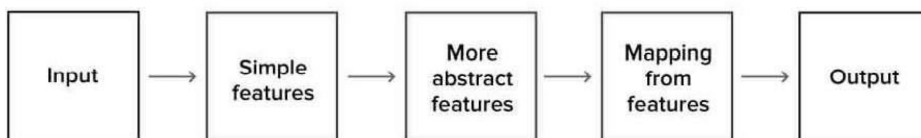
There are different types of neural networks but they always consist of the same components: neurons, synapses, weights, biases, and functions.

Neurons

A neuron or a node is a basic unit of neural networks that receives information, performs simple calculations, and passes it further.

All neurons in a net are divided into three groups:

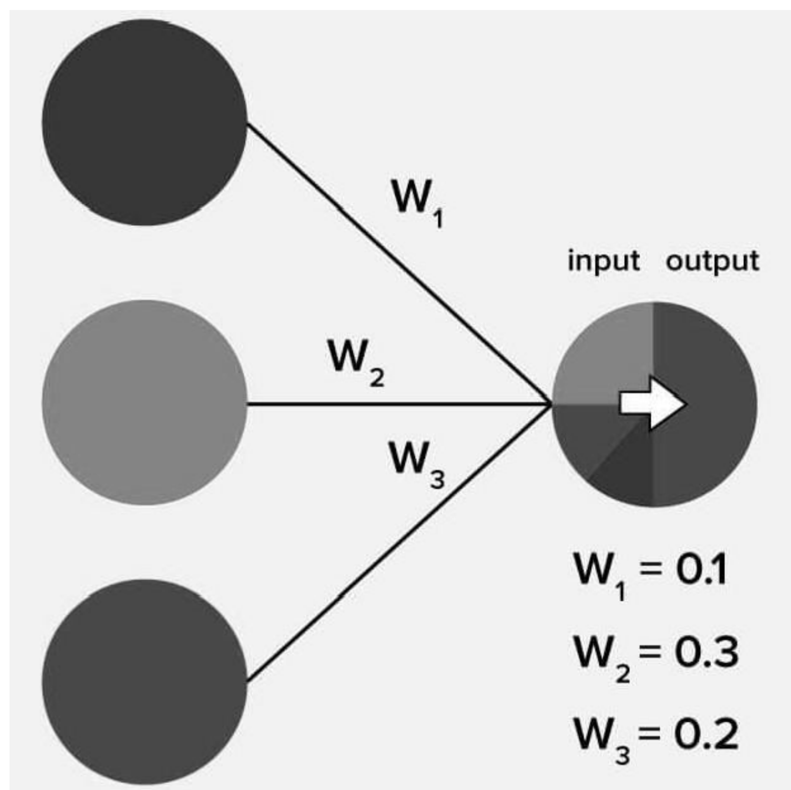
- Input neurons that receive information from the outside world;
- Hidden neurons that process that information;
- Output neurons that produce a conclusion.



In a large neural network with many neurons and connections between them, neurons are organized in layers. There is an input layer that receives information, a number of hidden layers, and the output layer that provides valuable results. Every neuron performs transformation on the input information. Neurons only operate numbers in the range [0,1] or [-1,1]. In order to turn data into something that a neuron can work with, we need normalization. We talked about what it is in the post about regression analysis.

Synapses and weights

A synapse is what connects the neurons like an electricity cable. Every synapse has a weight. The weights also add to the changes in the input information. The results of the neuron with the greater weight will be dominant in the next neuron, while information from less 'weighty' neurons will not be passed over. One can say that the matrix of weights governs the whole neural system.

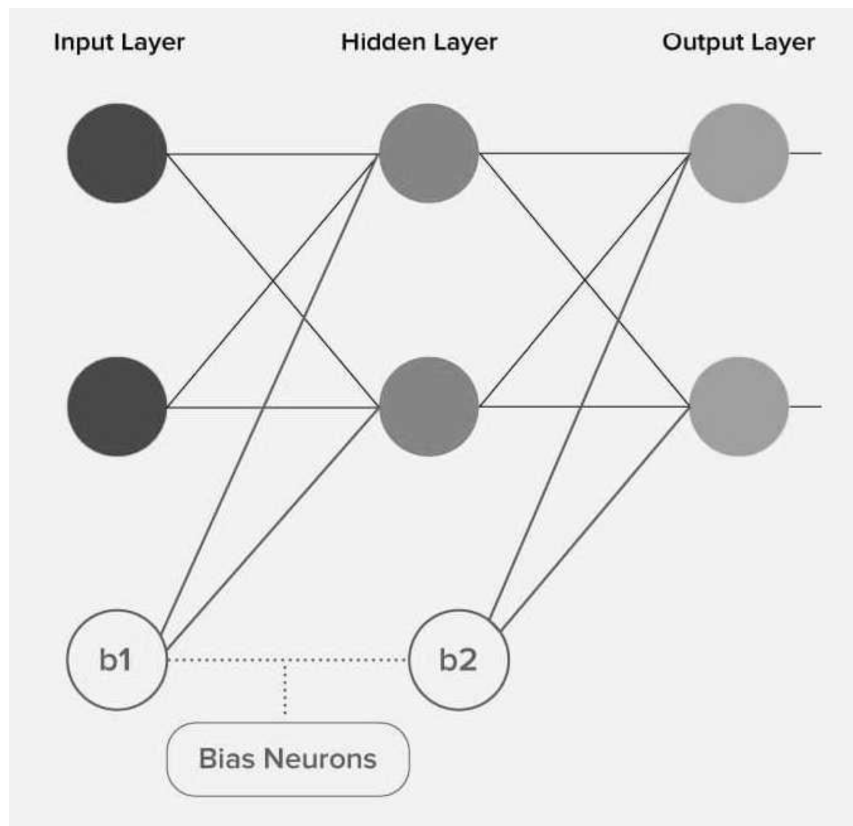


How do you know which neuron has the biggest weight? During the initialization (first launch of the NN), the weights are randomly assigned but then you will have to optimize them.

Bias

A bias neuron allows for more variations of weights to be stored. Biases add richer representation of the input space to the model's weights.

In the case of neural networks, a bias neuron is added to every layer. It plays a vital role by making it possible to move the activation function to the left or right on the graph.



It is true that ANNs can work without bias neurons. However, they are almost always added and counted as an indispensable part of the overall model.

How ANNs work

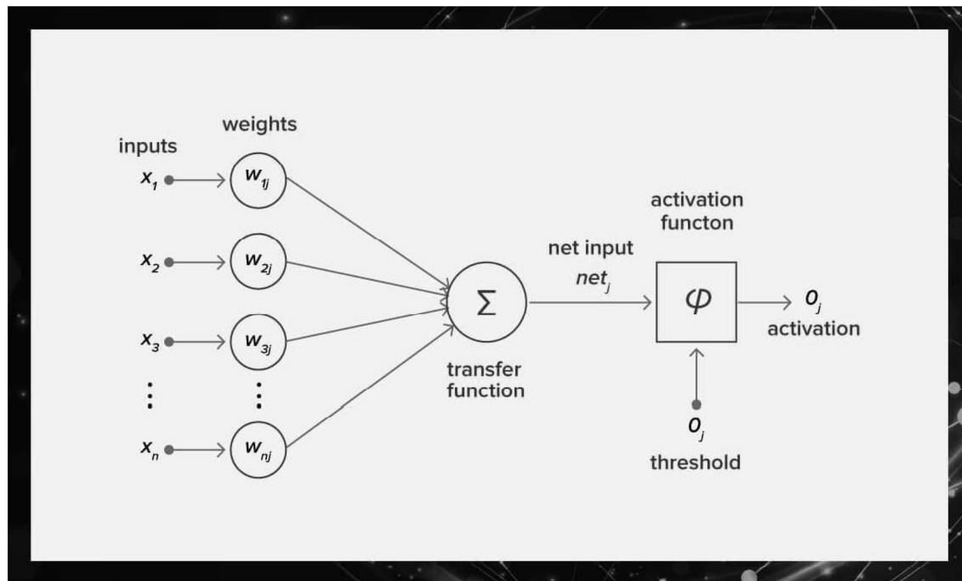
Every neuron processes input data to extract a feature. Let's imagine that we have three features and three neurons, each of which is connected with all these features.

Each of the neurons has its own weights that are used to weight the features.

During the training of the network, you need to select such weights for each

of the neurons that the output provided by the whole network would be true-to-life.

To perform transformations and get an output, every neuron has an activation function. This combination of functions performs a transformation that is described by a common function F — this describes the formula behind the NN's magic.



There are a lot of activation functions. The most common ones are linear, sigmoid, and hyperbolic tangent. Their main difference is the range of values they work with.

How do you train an algorithm?

Neural networks are trained like any other algorithm. You want to get some results and provide information to the network to learn from. For example, we want our neural network to distinguish between photos of cats and dogs and provide plenty of examples.

Delta is the difference between the data and the output of the neural network. We use calculus magic and repeatedly optimize the weights of the network until the delta is zero. Once the delta is zero or close to it, our model is correctly able to predict our example data.

Iteration

This is a kind of counter that increases every time the neural network goes through one training set. In other words, this is the total number of training sets completed by the neural network.

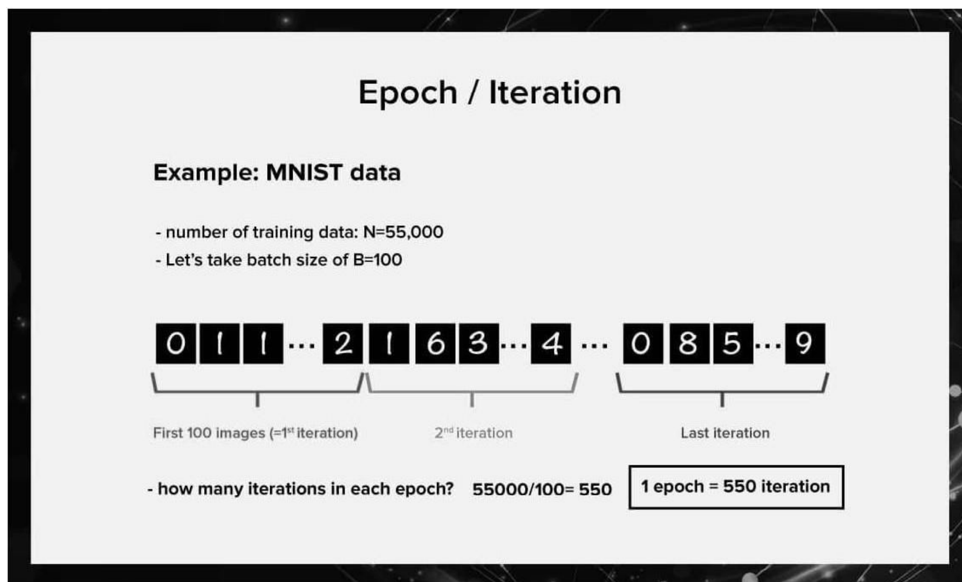
Epoch

The epoch increases each time we go through the entire set of training sets. The more epochs there are, the better is the training of the model.

Batch

Batch size is equal to the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.

What is the difference between an iteration and an epoch?



- one epoch is one forward pass and one backward pass of all the training examples;
- number of iterations is a number of passes, each pass using [batch size] number of examples. To be clear, one pass equals one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

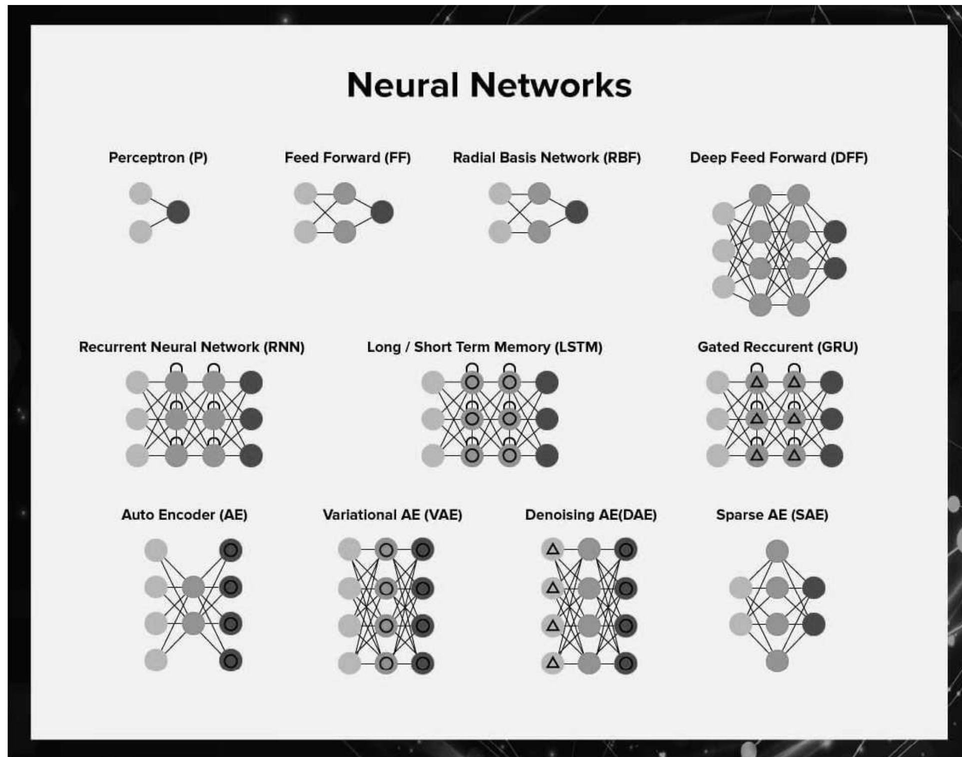
And what about errors?

Error is a deviation that reflects the discrepancy between expected and received output. The error should become smaller after every epoch. If this does not happen, then you are doing something wrong.

The error can be calculated in different ways, but we will consider only two main ways: Arctan and Mean Squared Error.

What kinds of neural networks exist?

There are so many different neural networks out there that it is simply impossible to mention them all. If you want to learn more about this variety, visit the neural network zoo where you can see them all represented graphically.



Feed-forward neural networks

This is the simplest neural network algorithm. A feed-forward network doesn't have any memory. That is, there is no going back in a feed-forward network.

In many tasks, this approach is not very applicable. For example, when we work with text, the words form a certain sequence, and we want the machine to understand it.

Feedforward neural networks can be applied in supervised learning when the data that you work with is not sequential or time-dependent. You can also use it if you don't know how the output should be structured but want to build a relatively fast and easy NN.

Recurrent neural networks

A recurrent neural network can process texts, videos, or sets of images and

become more precise every time because it remembers the results of the previous iteration and can use that information to make better decisions.

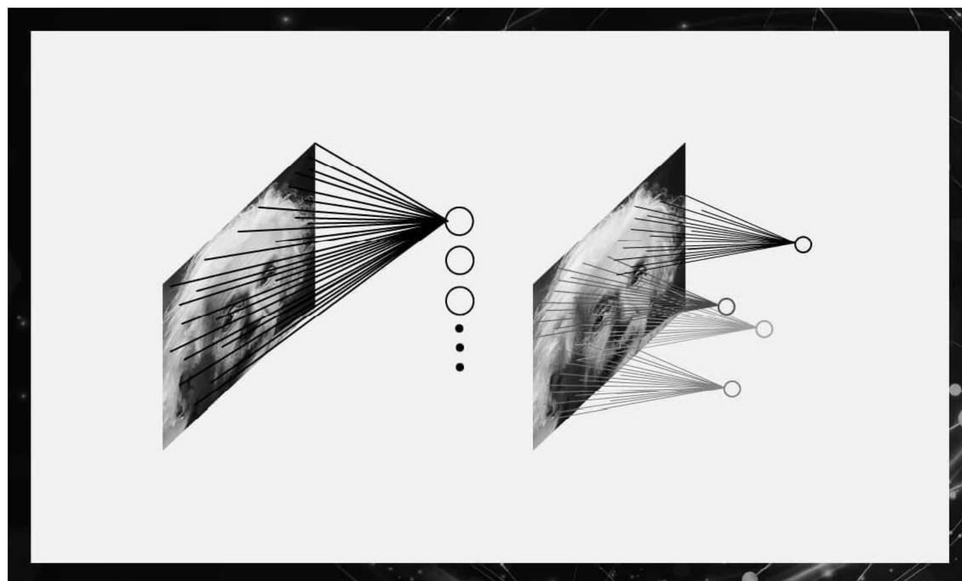
Recurrent neural networks are widely used in natural language processing and speech recognition.

Convolutional neural networks

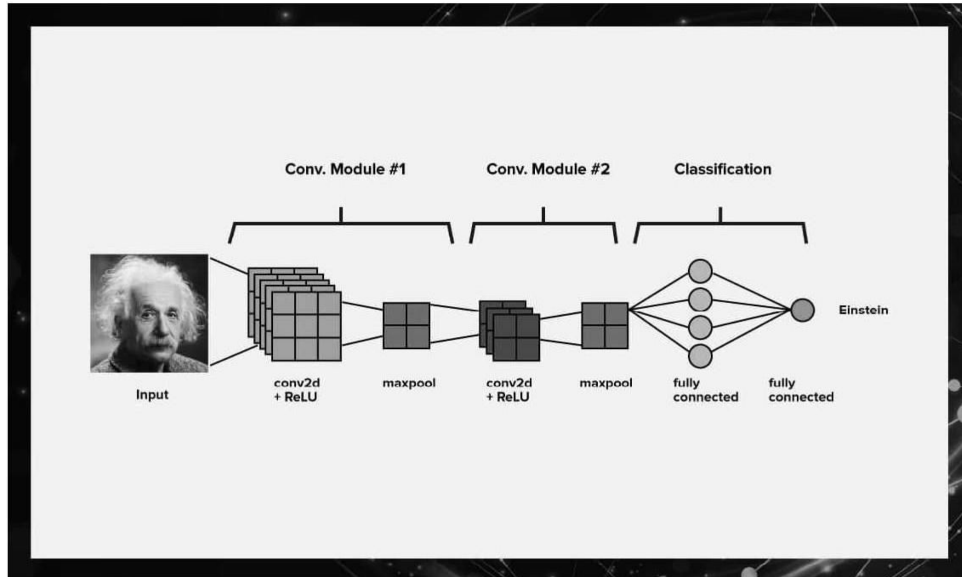
Convolutional neural networks are the standard of today's deep machine learning and are used to solve the majority of problems. Convolutional neural networks can be either feed-forward or recurrent.

Let's see how they work. Imagine we have an image of Albert Einstein. We can assign a neuron to all pixels in the input image.

But there is a big problem here: if you connect each neuron to all pixels, then, firstly, you will get a lot of weights. Hence, it will be a very computationally intensive operation and take a very long time. Then, there will be so many weights that this method will be very unstable to overfitting. It will predict everything well on the training example but work badly on other images.

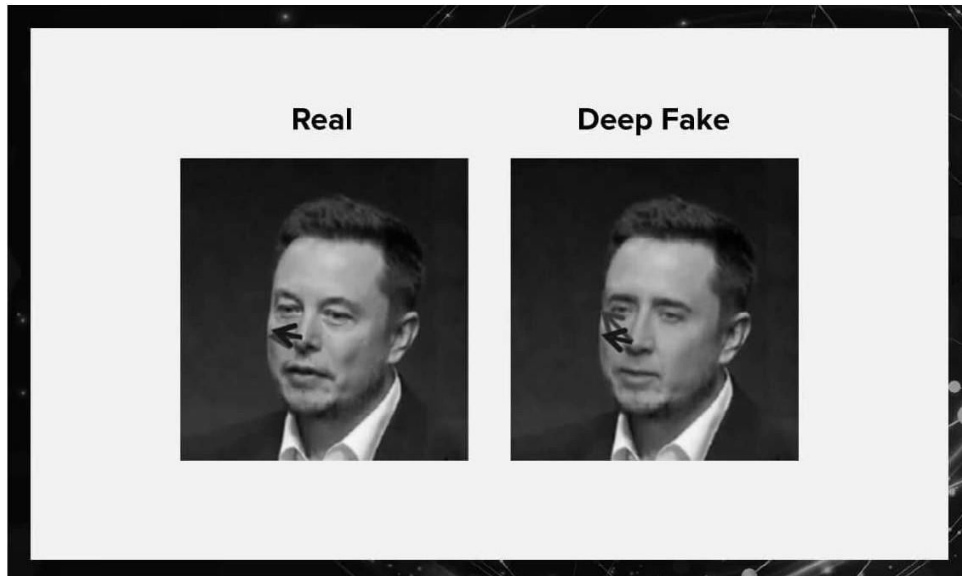


Therefore, programmers came up with a different architecture where each of the neurons is connected only to a small square in the image. All these neurons will have the same weights, and this design is called image convolution. We can say that we have transformed the picture, walked through it with a filter simplifying the process. Fewer weights, faster to count, less prone to overfitting.



For an awesome explanation of how convolutional neural networks work.

Generative adversarial neural networks



GANs are used, for example, to generate photographs that are perceived by the human eye as natural images or deepfakes.

A generative adversarial network is an unsupervised machine learning algorithm that is a combination of two neural networks, one of which (network G) generates

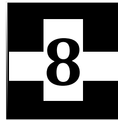
patterns and the other (network A) tries to distinguish genuine samples from the fake ones. Since networks have opposite goals – to create samples and reject samples – they start an antagonistic game that turns out to be quite effective.

What kind of problems do NNs solve?

Neural networks are used to solve complex problems that require analytical calculations similar to those of the human brain. The most common uses for neural networks are:

- **Classification.** NNs label the data into classes by implicitly analyzing its parameters. For example, a neural network can analyse the parameters of a bank client such as age, solvency, credit history and decide whether to loan them money.
- **Prediction.** The algorithm has the ability to make predictions. For example, it can foresee the rise or fall of a stock based on the situation in the stock market.
- **Recognition.** This is currently the widest application of neural networks. For example, a security system can use face recognition to only let authorized people into the building.

Deep learning and neural networks are useful technologies that expand human intelligence and skills. Neural networks are just one type of deep learning architecture. However, they have become widely known because NNs can effectively solve a huge variety of tasks and cope with them better than other algorithms.

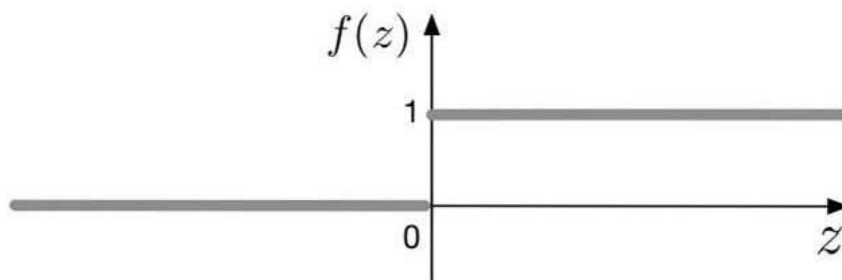


Sigmoid Neurons Function in Deep Learning

SIGMOID NEURONS: AN INTRODUCTION

So now we have a more sophisticatedly structured neural network with hidden layers. But we haven't solved the activation problem with the step function.

In the last post, we talked about the limitations of the linearity of step function. One thing to remember is: **If the activation function is linear, then you can stack as many hidden layers in the neural network as you wish, and the final output is still a linear combination of the original input data**. Please **make sure you read this link** for an explanation if the concept is difficult to follow. This linearity means that it cannot really grasp the complexity of non-linear problems like XOR logic or patterns separated by curves or circles.

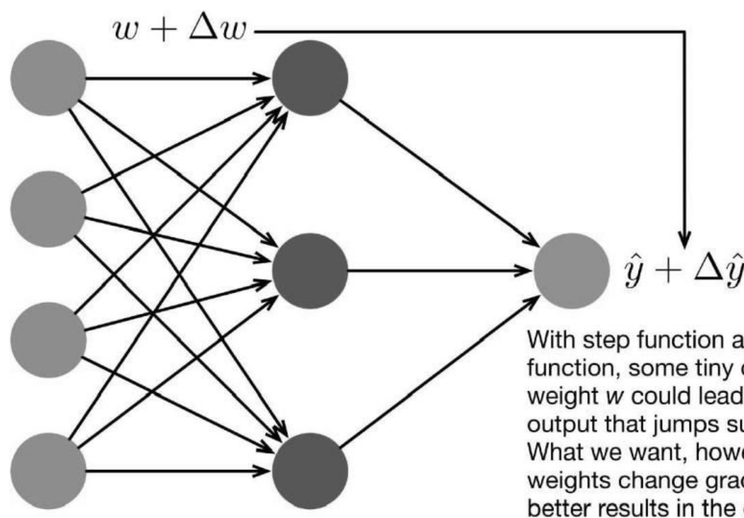


Meanwhile, step function also has no useful derivative (its derivative is 0

everywhere or undefined at the 0 point on x-axis). It doesn't work for **backpropagation**, which we will definitely talk about in the next post!

Graph 7: Step Function

Well, here's another problem: Perceptron with step function isn't very "stable" as a "relationship candidate" for neural networks. Think about it: this girl (or boy) has got some serious bipolar issues! One day (for $z < 0$), (s)he's all "quiet" and "down", giving you zero response. Then another day (for *z e" 0*), (s)he's suddenly "talkative" and "lively", speaking to you nonstop. Heck of a drastic change! There's no transition for her/his mood, and you don't know when it's going down or up. Yeah…that's step function.



Graph 8: We Want Gradual Change in Weights to Gradually Change Outputs

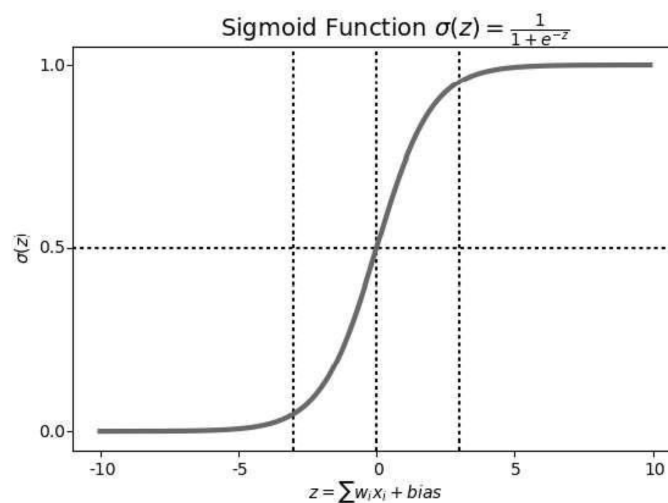
So basically, a small change in any weight in the input layer of our perceptron network could possibly lead to one neuron to suddenly flip from 0 to 1, which could again affect the hidden layer's behavior, and then affect the final outcome. Like we said already, we want a learning algorithm that could improve our neural network by gradually changing the weights, not by flat-no-response or sudden jumps. If we can't use step function to gradually change the weights, then it shouldn't be the choice.

Say goodbye to perceptron with step function now. We are finding a new partner for our neural network, the **sigmoid neuron**, which comes with sigmoid function (duh). But no worries: The only thing that will change is the activation function, and everything else we've learned so far about neural networks still works for this new type of neuron!

$$z = \sum_{i=1}^{m} w_i x_i + bias$$

Sigmoid Function is: $\sigma(z) = \frac{1}{1+e^{-z}}$

Sigmoid Function



Graph 9: Sigmoid Function using Matplotlib

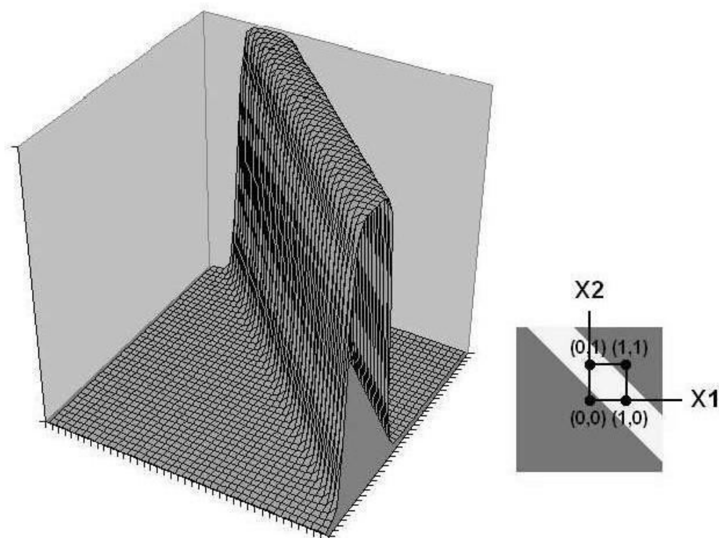
If the function looks very abstract or strange to you, don't worry too much about the details like Euler's number ***e*** or how someone came up with this crazy function in the first place. For those who aren't math-savvy, the only important thing about sigmoid function in **Graph 9** is first, its curve, and second, its derivative. Here are some more details:

1. Sigmoid function produces similar results to step function in that the output is between 0 and 1. The curve crosses 0.5 at ***z=0***, which we can set up rules for the activation function, such as: If the sigmoid neuron's output is larger than or equal to 0.5, it outputs 1; if the output is smaller than 0.5, it outputs 0.
2. Sigmoid function does not have a jerk on its curve. It is smooth and it has a very nice and simple derivative of ***□(z) * (1-□(z)),*** which is differentiable everywhere on the curve. The calculus derivation of the derivative can be found on Stack Overflow here if you want to see it. But you don't have to know how to derive it. No stress here.

3. If z is very negative, then the output is approximately 0; if z is very positive, the output is approximately 1; but around $z=0$ where z is neither too large or too small (in between the two outer vertical dotted grid lines in **Graph 9**), we have relatively more deviation as z changes.

Now that seems like a dating material for our neural network :) Sigmoid function, unlike step function, introduces non-linearity into our neural network model. Non-linear just means that the output we get from the neuron, which is the dot product of some inputs ***x (x1, x2, …, xm)*** and weights ***w (w1, w2, …,wm)*** plus bias and then put into a sigmoid function, cannot be represented by a linear combination of the input ***x (x1, x2, …,xm)***.

This non-linear activation function, when used by each neuron in a multi-layer neural network, produces a new "representation" of the original data, and ultimately allows for non-linear decision boundary, such as XOR. So in the case of XOR, if we add two sigmoid neurons in a hidden layer, we could, in another space, reshape the original 2D graph into something like the 3D image in the left side of **Graph 10** below. This ridge thus allows for classification of the XOR gate and it represents the light yellowish region of the 2D XOR gate in the right side of **Graph 10**. So if our output value is on the higher area of the ridge, then it should be a true or 1 (like the weather is cold but not hot, or the weather is hot but not cold); if our output value is on the lower flat area on the two corners, then it's false or 0 since it's not right to say the weather is both hot and cold or neither hot or cold (ok, I guess the weather could be neither hot or cold…you get what I mean though…right?).



Graph 10. Representation of Neural Networks with Hidden Layers to Classify XOR Gate.

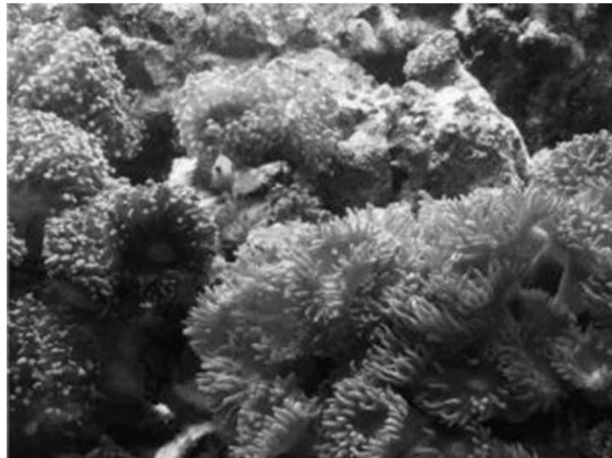
I know these talks on non-linearity can be confusing, so please read more about linearity & non-linearity here (intuitive post with animation from a great blog by Christopher Olah), here (by

Problem solved……for now ;) We will see some different types of activation function in the near future because sigmoid function has its own issues, too! Some popular ones include ***tanh*** and ***ReLU***. That, however, is for another post.

A GENTLE INTRODUCTION TO SIGMOID FUNCTION

Whether you implement a neural network yourself or you use a built in library for neural network learning, it is of paramount importance to understand the significance of a sigmoid function. The sigmoid function is the key to understanding how a neural network learns complex problems. This function also served as a basis for discovering other functions that lead to efficient and good solutions for supervised learning in deep learning architectures.

In this chapter, you will discover the sigmoid function and its role in learning from examples in neural networks.



A Gentle Introduction to sigmoid function. Photo by Mehreen Saeed, some rights reserved.

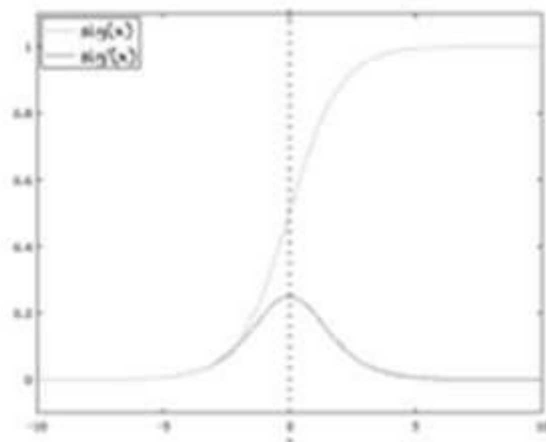
SIGMOID FUNCTION

The sigmoid function is a special form of the logistic function and is usually denoted by ▪(x) or sig(x). It is given by:

▪(x) = 1/(1+exp(-x))

PROPERTIES AND IDENTITIES OF SIGMOID FUNCTION

The graph of sigmoid function is an S-shaped curve as shown by the green line in the graph below. The figure also shows the graph of the derivative in pink color. The expression for the derivative, along with some important properties are shown on the right.



Graph of the sigmoid function and its derivative. Some important properties are also shown.

A few other properties include:

1. Domain: (-”, +”)
2. Range: (0, +1)
3. ▪(0) = 0.5
4. The function is monotonically increasing.
5. The function is continuous everywhere.
6. The function is differentiable everywhere in its domain.
7. Numerically, it is enough to compute this function’s value over a small range of numbers, e.g., [-10, +10]. For values less than -10, the function’s value is almost zero. For values greater than 10, the function’s values are almost one.

THE SIGMOID AS A SQUASHING FUNCTION

The sigmoid function is also called a squashing function as its domain is the set of all real numbers, and its range is (0, 1). Hence, if the input to the function

is either a very large negative number or a very large positive number, the output is always between 0 and 1. Same goes for any number between -" and +".

Sigmoid As An Activation Function In Neural Networks

The sigmoid function is used as an activation function in neural networks. Just to review what is an activation function, the figure below shows the role of an activation function in one layer of a neural network. A weighted sum of inputs is passed through an activation function and this output serves as an input to the next layer.

When the activation function for a neuron is a sigmoid function it is a guarantee that the output of this unit will always be between 0 and 1. Also, as the sigmoid is a non-linear function, the output of this unit would be a non-linear function of the weighted sum of inputs. Such a neuron that employs a sigmoid function as an activation function is termed as a sigmoid unit.

Linear Vs. Non-Linear Separability?

Suppose we have a typical classification problem, where we have a set of points in space and each point is assigned a class label. If a straight line (or a hyperplane in an n-dimensional space) can divide the two classes, then we have a linearly separable problem. On the other hand, if a straight line is not enough to divide the two classes, then we have a non-linearly separable problem. The figure below shows data in the 2 dimensional space. Each point is assigned a red or blue class label. The left figure shows a linearly separable problem that requires a linear boundary to distinguish between the two classes. The right figure shows a non-linearly separable problem, where a non-linear decision boundary is required.

For three dimensional space, a linear decision boundary can be described via the equation of a plane. For an n-dimensional space, the linear decision boundary is described by the equation of a hyperplane.

Why The Sigmoid Function Is Important In Neural Networks?

If we use a linear activation function in a neural network, then this model can only learn linearly separable problems. However, with the addition of just one hidden layer and a sigmoid activation function in the hidden layer, the neural network can easily learn a non-linearly separable problem. Using a non-linear function produces non-linear boundaries and hence, the sigmoid function can be used in neural networks for learning complex decision functions.

The only non-linear function that can be used as an activation function in a neural network is one which is monotonically increasing. So for example, sin(x) or cos(x) cannot be used as activation functions. Also, the activation function

should be defined everywhere and should be continuous everywhere in the space of real numbers. The function is also required to be differentiable over the entire space of real numbers.

Typically a back propagation algorithm uses gradient descent to learn the weights of a neural network. To derive this algorithm, the derivative of the activation function is required.

The fact that the sigmoid function is monotonic, continuous and differentiable everywhere, coupled with the property that its derivative can be expressed in terms of itself, makes it easy to derive the update equations for learning the weights in a neural network when using back propagation algorithm.

MULTI-LAYER NEURAL NETWORKS: AN INTUITIVE APPROACH

Alright. So we've introduced hidden layers in a neural network and replaced perceptron with sigmoid neurons. We also introduced the idea that non-linear activation function allows for classifying non-linear decision boundaries or patterns in our data. You can memorize these takeaways since they're facts, but I encourage you to google a bit on the internet and see if you can understand the concept better (it is natural that we take some time to understand these concepts). Now, we've never talked about one very important point: Why on earth do we want hidden layers in neural networks in the first place? How do hidden layers magically help us to tackle complicated problems that single-layer neurons cannot do?

From the XOR example above, you've seen that adding two hidden neurons in 1 hidden layer could reshape our problem into a different space, which magically created a way for us to classify XOR with a ridge. So hidden layers somehow twist the problem in a way that makes it easy for the neural network to classify the problem or pattern. Now we'll use a classic textbook example: Recognition of hand-written digits, to help you intuitively understand what hidden layers do.

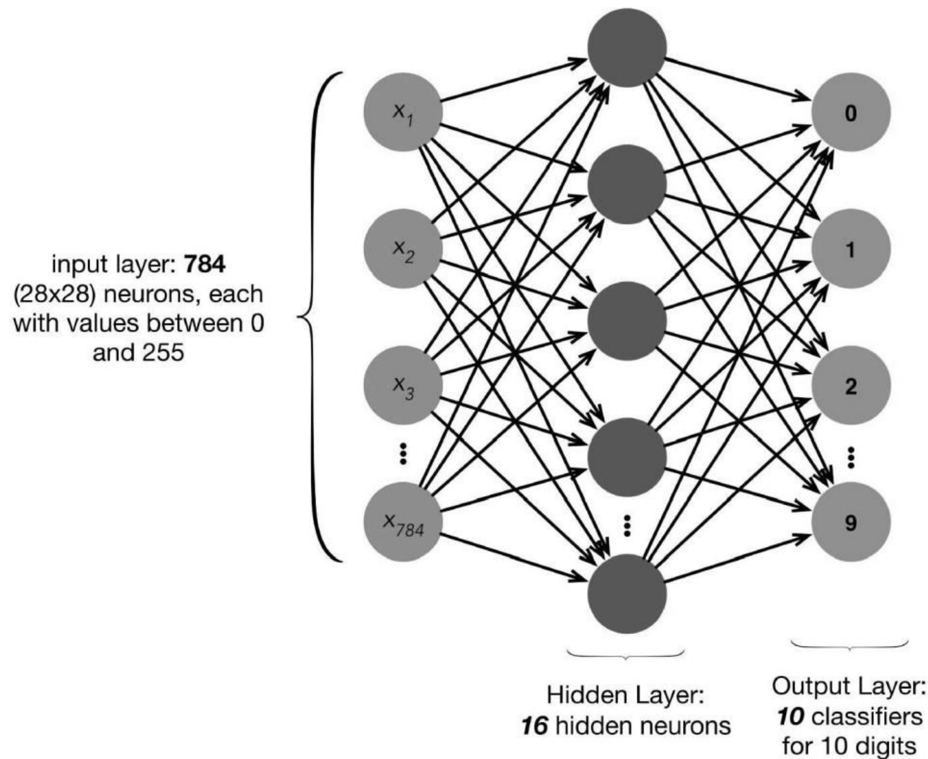
Graph 11. MNIST dataset of Hand-written Digits.

The digits in **Graph 11** belong to a dataset called MNIST. It contains 70,000 examples of digits written by human hands. Each of these digits is a picture of 28x28 pixels. So in total each image of a digit has 28*28=784 pixels. Each pixel takes a value beween 0 and 255 (RGB color code). 0 means the color is white and 255 means the color black.

Graph 12. MNIST digit 5, which consist of 28x28 pixel values between 0 and 255.

Now, the computer can't really "see" a digit like we humans do, but if we dissect the image into an array of 784 numbers like [0, 0, 180, 16, 230, …, 4, 77, 0, 0, 0], then we can feed this array into our neural network. The computer can't understand an image by "seeing" it, but it can understand and analyze the pixel numbers that represent an image.

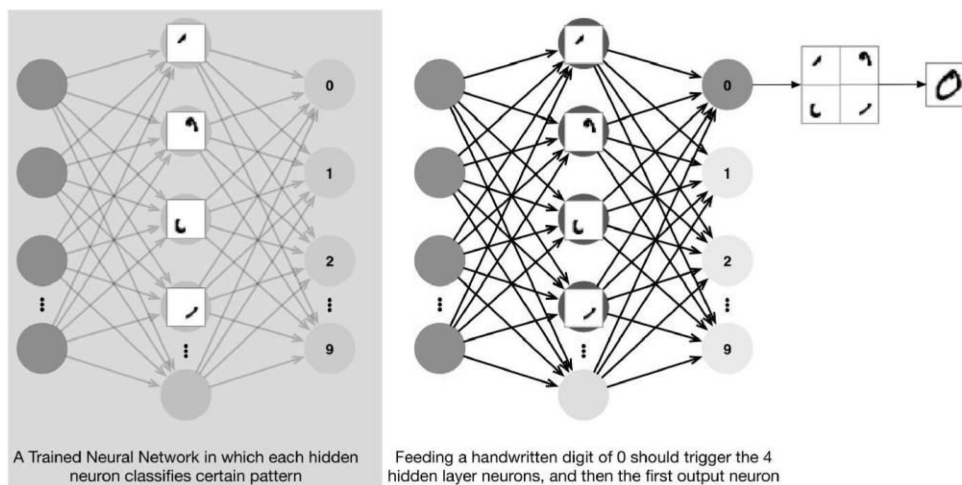
So, let's set up a neural network like above in **Graph 13**. It has 784 input neurons for 28x28 pixel values. Let's assume it has 16 hidden neurons and 10 output neurons. The 10 output neurons, returned to us in an array, will each be in charge to classify a digit from 0 to 9. So if the neural network thinks the handwritten digit is a zero, then we should get an output array of [1, 0, 0, 0, 0, 0, 0, 0, 0, 0], the first output in this array that senses the digit to be a zero is "fired" to be 1 by our neural network, and the rest are 0. If the neural network thinks the handwritten digit is a 5, then we should get [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]. The 6th element that is in charge to classify a five is triggered while the rest are not. So on and so forth.



Graph 13: Multi-Layer Sigmoid Neural Network with 784 input neurons, 16 hidden neurons, and 10 output neurons

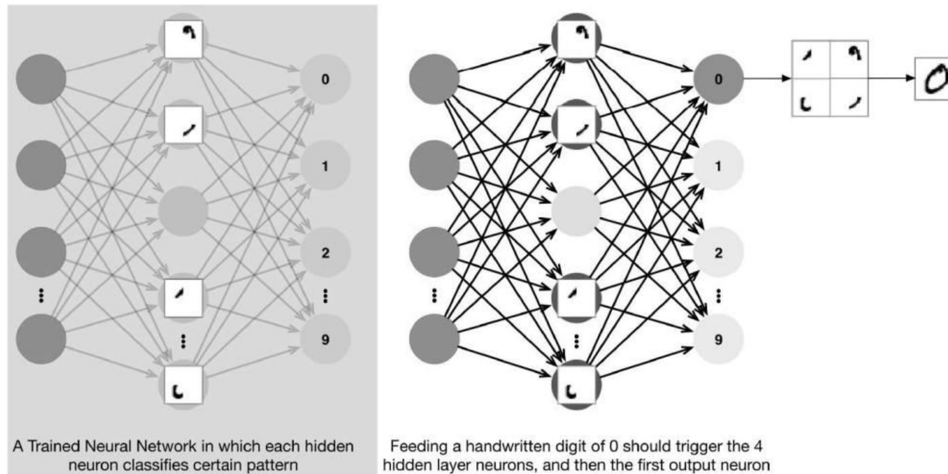
Remember we mentioned that neural networks become better by repetitively training themselves on data so that they can adjust the weights in each layer of the network to get the final results/actual output closer to the desired output? So when we actually train this neural network with all the training examples in MNIST dataset, we don't know what weights we should assign to each of the layers. So we just randomly ask the computer to assign weights in each layer. (We don't want all the weights to be 0, which I'll explain in the next post if space allows).

This concept of randomly initializing weights is important because each time you train a deep learning neural network, you are initializing different numbers to the weights. So essentially, you and I have no clue what's going on in the neural network until after the network is trained. A trained neural network has weights which are optimized at certain values that make the best prediction or classification on our problem. It's a black box, literally. And each time the trained network will have different sets of weights. For the sake of argument, let's imagine the following case in **Graph 14**, which I borrow from Michael Nielsen's online book:



Graph 14. An Intuitive Example to Understand Hidden Layers

After training the neural network with rounds and rounds of labeled data in supervised learning, assume the first 4 hidden neurons learned to recognize the patterns above in the left side of **Graph 14**. Then, if we feed the neural network an array of a handwritten digit zero, the network should correctly trigger the top 4 hidden neurons in the hidden layer while the other hidden neurons are silent, and then again trigger the first output neuron while the rest are silent.



Graph 15. Neural Networks are Black Boxes. Each Time is Different.

If you train the neural network with a new set of randomized weights, it might produce the following network instead (**compare Graph 15 with Graph 14**), since the weights are randomized and we never know which one will learn which or what pattern. But the network, if properly trained, should still trigger the correct hidden neurons and then the correct output.

One last thing to mention: In a multi-layer neural network, the first hidden layer will be able to learn some very simple patterns. Each additional hidden layer will somehow be able to learn progressively more complicated patterns. Check out **Graph 16** from Scientific American with an example of face ecognition :)

ACTIVATION FUNCTIONS

SO WHY DO WE NEED ACTIVATION FUNCTIONS IN OUR NEURAL NETWORKS?

Welcome to my first post! I am a Data Scientist and have been an active reader of Medium blogs. Now, I am planning to use Medium blog to document my journey in learning Deep Learning and share my experiences through the projects I have been working on. Hopefully, by sharing my views on the subjects I can also learn from the fantastic data science/deep learning community on Medium! I would love to hear your feedback on my first post here. With that said, let's get started …

The basic idea of how a neural network learns is — We have some input data that we feed it into the network and then we perform a series of linear operations layer by layer and derive an output. In a simple case for a particular layer is that

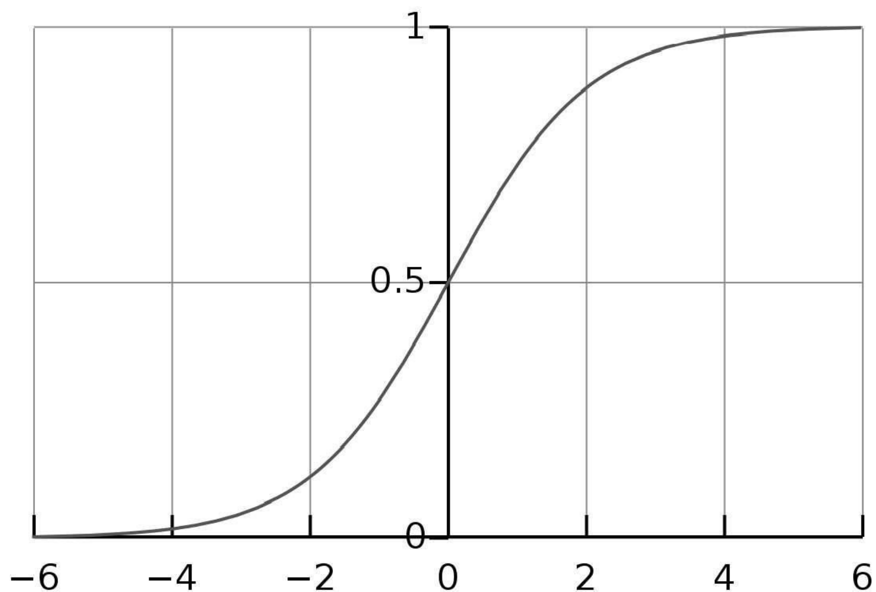
we multiply the input by the weights, add a bias and apply an activation function and pass the output to the next layer. We keep repeating the process until we reach the last layer. The final value is our output. We then compute the error between the "calculated output" and the "true output" and then calculate the partial derivatives of this error with respect to the parameters in each layer going backwards and keep updating the parameters accordingly!

Neural networks are said to be universal function approximators. The main underlying goal of a neural network is to learn complex non-linear functions. If we do not apply any non-linearity in our multi-layer neural network, we are simply trying to seperate the classes using a linear hyperplane. As we know, in the real-world nothing is linear!

Also, imagine we perform simple linear operation as described above, namely; multiply the input by weights, add a bias and sum them across all the inputs arriving to the neuron. It is likely that in certain situations, the output derived above, takes a large value. When, this output is fed into the further layers, they can be transformed to even larger values, making things computationally uncontrollable. This is where the activation functions play a major role i.e. squashing a real-number to a fix interval (e.g. between -1 and 1).

Let us see different types of activation functions and how they compare against each other:

Sigmoid



The sigmoid activation function has the mathematical form ‘sig(z) = 1/ (1 + e^-z)‘. As we can see, it basically takes a real valued number as the input and squashes it between 0 and 1. It is often termed as a squashing function as well. It aims to introduce non-linearity in the input space. The non-linearity is where we get the wiggle and the network learns to capture complicated relationships. As we can see from the above mathematical representation, a large negative number passed through the sigmoid function becomes 0 and a large positive number becomes 1. Due to this property, sigmoid function often has a really nice interpretation associated with it as the firing rate of the neuron; from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1). However, sigmoid activation functions have become less popular over the period of time due to the following two major drawbacks:

Killing gradients

- Sigmoid neurons get saturated on the boundaries and hence the local gradients at these regions is almost zero. To give you a more intuitive example to understand this, consider the inputs to the sigmoid function to be +15 and -15. The derivative of sigmoid function is ‘sig(z) * (1 — sig(z))‘. As mentioned above, the large positive values are squashed near 1 and large negative values are squashed near 0. Hence, effectively making the local gradient to near 0. As a result, during backpropagation, this gradient gets multiplied to the gradient of this neurons’ output for the final objective function, hence it will effectively “kill” the gradient and no signal will flow through the neuron to its weights. Also, we have to pay attention to initializing the weights of sigmoid neurons to avoid saturation, because, if the initial weights are too large, then most neurons will get saturated and hence the network will hardly learn.

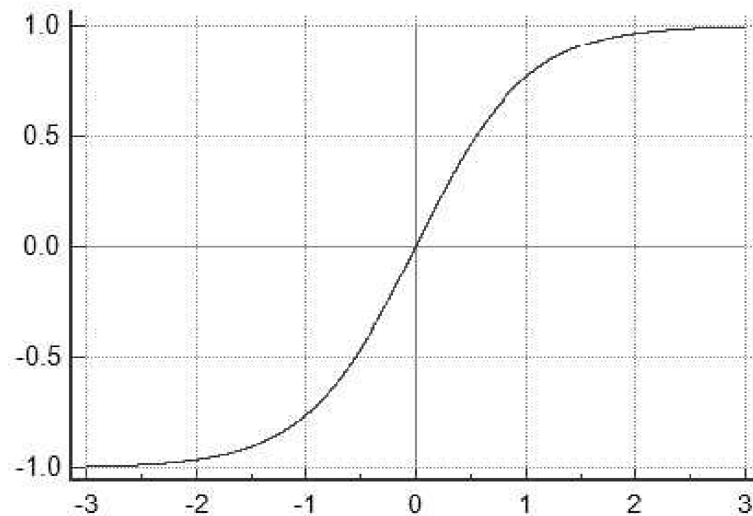
Non zero-centered outputs:

- The output is always between 0 and 1, that means that the output after applying sigmoid is always positive hence, during gradient-descent, the gradient on the weights during backpropagation will always be either positive or negative depending on the output of the neuron. As a result, the gradient updates go too far in different directions which makes optimization harder.

The python implementation looks something similar to:

```
import numpy as npdef sigmoid(z):
return 1 / (1 + np.exp(-z))
```

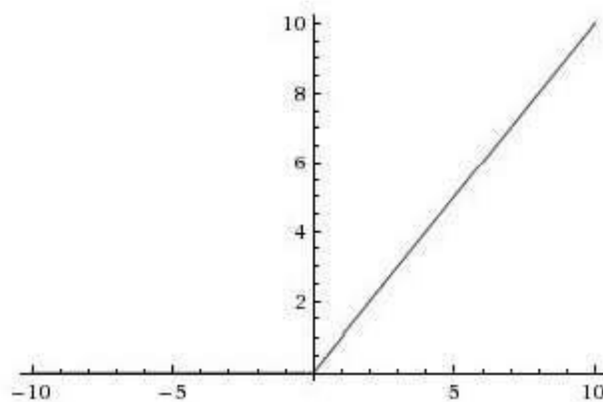
Tanh



The tanh or hyperbolic tangent activation function has the mathematical form 'tanh(z) = (e^z — e^-z) / (e^z + e^-z)'. It is basically a shifted sigmoid neuron. It basically takes a real valued number and squashes it between -1 and +1. Similar to sigmoid neuron, it saturates at large positive and negative values. However, its output is always zero-centered which helps since the neurons in the later layers of the network would be receiving inputs that are zero-centered. Hence, in practice, tanh activation functions are preffered in hidden layers over sigmoid.

```
import numpy as npdef tanh(z):
return np.tanh(z)
```

ReLU:



The ReLU or Rectified Linear Unit is represented as 'ReLU(z) = max(0, z)'. It basically thresholds the inputs at zero, i.e. all negative values in the input to the ReLU neuron are set to zero. Fairly recently, it has become popular as it was found that it greatly accelerates the convergence of stochastic gradient descent as compared to Sigmoid or Tanh activation functions. Just to give an intuition, the gradient is either 0 or 1 depending on the sign of the input. Let us discuss some of the advantages of ReLU:

Sparsity of Activations:

- As we studied above, ReLU and Tanh activation functions would almost always get fired in the neural network, resulting in the almost all the activations getting processed in calculating the final output of the network. Now surely this is a good thing but only if our network is small or we had unlimited computational power. Imagine we have a very deep neural network with a lot of neurons, we would ideally want only a section of neurons to fire and contribute to the final output of the network and hence, we want a section of the neurons in the network to be passive. ReLU gives us this benefit. Hence, due to the characteristics of ReLU, there is a possibility that 50% of neurons to give 0 activations and thus leading to fewer neurons to fire as a result of which the network becomes lighter and we can compute the output faster.

However, it has a drawback in terms of a problem called as dying neurons.

Dead Neurons:

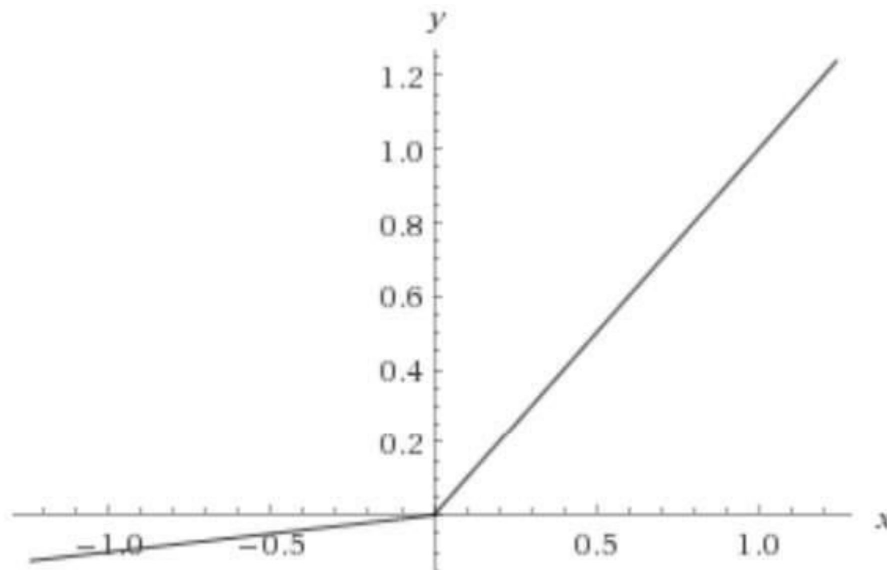
- ReLU units can be fragile during training and can "die". That is, if the units are not activated initially, then during backpropagation zero gradients

flow through them. Hence, neurons that "die" will stop responding to the variations in the output error because of which the parameters will never be updated/updated during backpropagation. However, there are concepts such as Leaky ReLU that can be used to overcome this problem. Also, having a proper setting of the learning rate can prevent causing the neurons to be dead.

```
import numpy as npdef relu(z):
return z * (z > 0)
```

Leaky ReLU:

The Leaky ReLU is just an extension of the traditional ReLU function. As we saw that for values less than 0, the gradient is 0 which results in "Dead Neurons" in those regions. To address this problem, Leaky ReLU comes in handy.



That is, instead of defining values less than 0 as 0, we instead define negative values as a small linear combination of the input. The small value commonly used is 0.01. It is represented as `LeakyReLU(z) = max(0.01 * z, z)`. The idea of Leaky ReLU can be extended even further by making a small change. Instead of multiplying `z` with a constant number, we can learn the multiplier and treat it as an additional hyperparameter in our process. This is known as Parametric ReLU. In practice, it is believed that this performs better than Leaky ReLU.

```
import numpy as npdef leaky_relu(z):
return np.maximum(0.01 * z, z)
```

ACTIVATION FUNCTIONS AND THEIR DERIVATIVES

Function Type	***Equation***	***Derivative***
Linear	$f(x) = ax + c$	$f'(x) = a$
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)\,(1- f(x))$
TanH	$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ReLU	$f(x) = \begin{cases} 0 \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$
Parametric ReLU	$f(x) = \begin{cases} \alpha x \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$
ELU	$f(x) = \begin{cases} \alpha(e^{x} - 1) \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$

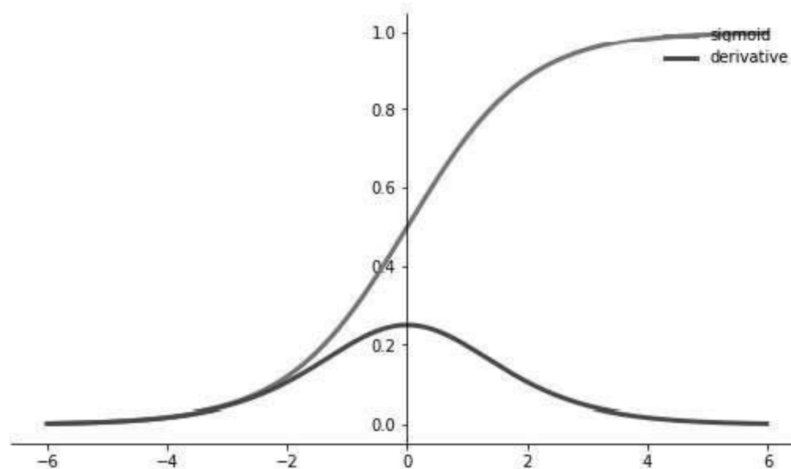
IMPLEMENTATION USING PYTHON

Having learned the types and significance of each activation function, it is also essential to implement some basic (non-linear) activation functions using python code and observe the output for more clear understanding of the concepts:

Sigmoid Activation Function

```
import matplotlib.pyplot as plt
import numpy as np
```

```
def sigmoid(x):
   s=1/(1+np.exp(-x))
   ds=s*(1-s)
   return s,ds
x=np.arange(-6,6,0.01)
sigmoid(x)
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.plot(x,sigmoid(x)[0], color="#307EC7", linewidth=3,
label="sigmoid")
ax.plot(x,sigmoid(x)[1], color="#9621E2", linewidth=3,
label="derivative")
ax.legend(loc="upper  right", frameon=False)
fig.show()
```



Observations:

- The sigmoid function has values between 0 to 1.
- The output is not zero-centered.
- Sigmoids saturate and kill gradients.
- At the top and bottom level of sigmoid functions, the curve changes slowly, the derivative curve above shows that the slope or gradient it is zero.

Tanh Activation Function

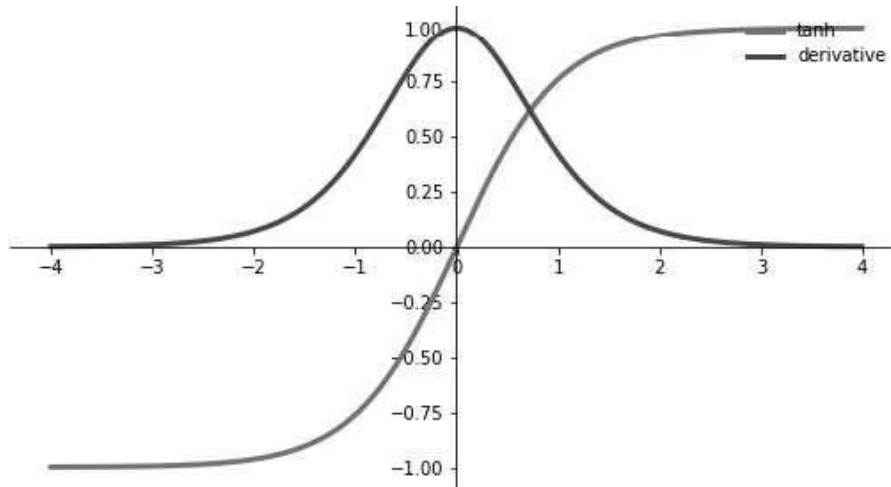
```
import matplotlib.pyplot as plt
import numpy as np
def tanh(x):
   t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
   dt=1-t**2
   return t,dt
z=np.arange(-4,4,0.01)
tanh(z)[0].size,tanh(z)[1].size
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.plot(z,tanh(z)[0], color="#307EC7", linewidth=3,
label="tanh")
ax.plot(z,tanh(z)[1], color="#9621E2", linewidth=3,
label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.show()
```

Observations:

- Its output is zero-centered because its range is between -1 to 1. i.e. -1 < output < 1.
- Optimization is easier in this method hence in practice it is always preferred over the Sigmoid function.

When all of this is said and done, the actual purpose of an activation function is to feature some reasonably non-linear property to the function, which could be a neural network. A neural network, without the activation functions, might perform solely linear mappings from the inputs to the outputs, and also the mathematical operation throughout the forward propagation would be the dot-products between an input vector and a weight matrix.

Since one dot product could be a linear operation, sequent dot products would be nothing more than multiple linear operations repeated one after another. And sequent linear operations may be thought of as mutually single learn operations. To be able to work out extremely attention-grabbing stuff, the neural networks should be able to approximate the nonlinear relations from input features to the output labels.



The more complicated the information, the more non-linear the mapping of features to the bottom truth label will usually be. If there is no activation function in a neural network, the network would in turn not be able to understand such complicated mappings mathematically and wouldn't be able to solve tasks that the network is really meant to resolve.

TYPES OF ACTIVATION FUNCTIONS

The different kinds of activation functions include:

Linear Activation Functions

A linear function is also known as a straight-line function where the activation is proportional to the input i.e. the weighted sum from neurons. It has a simple function with the equation:

$f(x) = ax + c$

The problem with this activation is that it cannot be defined in a specific range. Applying this function in all the nodes makes the activation function work like linear regression. The final layer of the Neural Network will be working as a linear function of the first layer. Another issue is the gradient descent when differentiation is done, it has a constant output which is not good because during backpropagation the rate of change of error is constant that can ruin the output and the logic of backpropagation.

Non-Linear Activation Functions

The non-linear functions are known to be the most used activation functions.

It makes it easy for a neural network model to adapt with a variety of data and to differentiate between the outcomes.

These functions are mainly divided basis on their range or curves:

Sigmoid Activation Functions

Sigmoid takes a real value as the input and outputs another value between 0 and 1. The sigmoid activation function translates the input ranged in (-”,”) to the range in (0,1)

Tanh Activation Functions

The tanh function is just another possible function that can be used as a non-linear activation function between layers of a neural network. It shares a few things in common with the sigmoid activation function. Unlike a sigmoid function that will map input values between 0 and 1, the Tanh will map values between -1 and 1. Similar to the sigmoid function, one of the interesting properties of the tanh function is that the derivative of tanh can be expressed in terms of the function itself.

ReLU Activation Functions

The formula is deceptively simple: *max(0,z)*. Despite its name, Rectified Linear Units, it’s not linear and provides the same benefits as Sigmoid but with better performance.

Leaky Relu

Leaky Relu is a variant of ReLU. Instead of being 0 when z<0, a leaky ReLU allows a small, non-zero, constant gradient ▪ (normally, ▪=0.01). However, the consistency of the benefit across tasks is presently unclear. Leaky ReLUs attempt to fix the “dying ReLU” problem.

Parametric Relu

PReLU gives the neurons the ability to choose what slope is best in the negative region. They can become ReLU or leaky ReLU with certain values of ▪.

Maxout

The Maxout activation is a generalization of the ReLU and the leaky ReLU functions. It is a piecewise linear function that returns the maximum of inputs, designed to be used in conjunction with the dropout regularization technique. Both ReLU and leaky ReLU are special cases of Maxout. The Maxout neuron, therefore,

enjoys all the benefits of a ReLU unit and does not have any drawbacks like dying ReLU. However, it doubles the total number of parameters for each neuron, and hence, a higher total number of parameters need to be trained.

ELU

The Exponential Linear Unit or ELU is a function that tends to converge faster and produce more accurate results. Unlike other activation functions, ELU has an extra alpha constant which should be a positive number. ELU is very similar to ReLU except for negative inputs. They are both in the identity function form for non-negative inputs. On the other hand, ELU becomes smooth slowly until its output equal to -▪ whereas ReLU sharply smoothes.

Softmax Activation Functions

Softmax function calculates the probabilities distribution of the event over 'n' different events. In a general way, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will help determine the target class for the given inputs.

When to use which Activation Function in a Neural Network?

Specifically, it depends on the problem type and the value range of the expected output. For example, to predict values that are larger than 1, tanh or sigmoid are not suitable to be used in the output layer, instead, ReLU can be used.

On the other hand, if the output values have to be in the range (0,1) or (-1, 1) then ReLU is not a good choice, and sigmoid or tanh can be used here. While performing a classification task and using the neural network to predict a probability distribution over the mutually exclusive class labels, the softmax activation function should be used in the last layer. However, regarding the hidden layers, as a rule of thumb, use ReLU as an activation for these layers.

In the case of a binary classifier, the Sigmoid activation function should be used. The sigmoid activation function and the tanh activation function work terribly for the hidden layer. For hidden layers, ReLU or its better version leaky ReLU should be used. For a multiclass classifier, Softmax is the best-used activation function. Though there are more activation functions known, these are known to be the most used activation functions.

ACTIVATION FUNCTIONS IN NEURAL NETWORKS

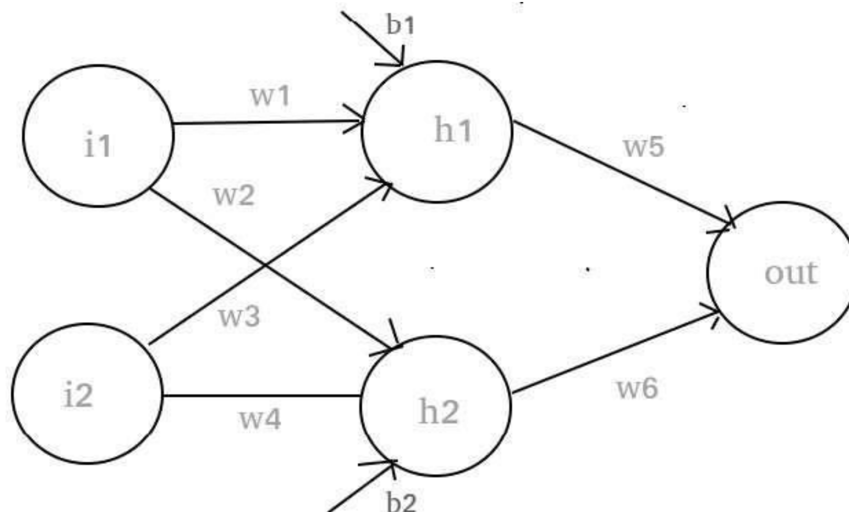
It is recommended to understand what is a neural network before reading this chapter. In The process of building a neural network, one of the choices you get

to make is what activation function to use in the hidden layer as well as at the output layer of the network.

Elements of a Neural Network :- Input Layer :- This layer accepts input features. It provides information from the outside world to the network, no computation is performed at this layer, nodes here just pass on the information(features) to the hidden layer. **Hidden Layer :-** Nodes of this layer are not exposed to the outer world, they are the part of the abstraction provided by any neural network. Hidden layer performs all sort of computation on the features entered through the input layer and transfer the result to the output layer. **Output Layer :-** This layer bring up the information learned by the network to the outer world.

What is an activation function and why to use them? Definition of activation function:- Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to **introduce non-linearity** into the output of a neuron.

Explanation :- We know, neural network has neurons that work in correspondence of *weight, bias* and their respective activation function. In a neural network, we would update the weights and biases of the neurons on the basis of the error at the output. This process is known as *back-propagation*. Activation functions make the back-propagation possible since the gradients are supplied along with the error to update the weights and biases.



Why do we need Non-linear activation functions :- A neural network without an activation function is essentially just a linear regression model. The

activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

Mathematical proof :- *Suppose we have a Neural net like this :-*

Elements of the diagram are as follows:

Hidden layer i.e. layer 1:

z(1) = W(1)X + b(1) a(1) = z(1) Here,

- *z(1) is the vectorized output of layer 1*
- *W(1) be the vectorized weights assigned to neurons of hidden layer i.e. w1, w2, w3 and w4*
- *X be the vectorized input features i.e. i1 and i2*
- *b is the vectorized bias assigned to neurons in hidden layer i.e. b1 and b2*
- *a(1) is the vectorized form of any linear function.*

*(***Note:** *We are not considering activation function here)*

Layer 2 i.e. output layer :-

// **Note :** Input for layer

// 2 is output from layer 1

z(2) = W(2)a(1) + b(2)

a(2) = z(2)

Calculation at Output layer:

// Putting value of z(1) here

z(2) = (W(2) * [W(1)X + b(1)]) + b(2)

z(2) = [W(2) * W(1)] * X + [W(2)*b(1) + b(2)]

Let,

[W(2) * W(1)] = W

[W(2)*b(1) + b(2)] = b

Final output : z(2) = W*X + b

Which is again a linear function

This observation results again in a linear function even after applying a hidden layer, hence we can conclude that, doesn't matter how many hidden layer we attach in neural net, all layers will behave same way because **the composition of two linear function is a linear function itself**. Neuron can not learn with just a linear function attached to it. A non-linear activation function will let it learn as per the

difference w.r.t error. **Hence we need activation function. VARIANTS OF ACTIVATION FUNCTION :-**

1). Linear Function :-

- **Equation :** Linear function has the equation similar to as of a straight line i.e. **y = x**
- No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer.
- **Range :** -inf to +inf
- **Uses : Linear activation function** is used at just one place i.e. output layer.
- **Issues :** If we will differentiate linear function to bring non-linearity, result will no more depend on *input "x"* and function will become constant, it won't introduce any ground-breaking behavior to our algorithm.

For example : Calculation of price of a house is a regression problem. House price may have any big/small value, so we can apply linear activation at output layer. Even in this case neural net must have any non-linear function at hidden layers.

2). Sigmoid Function :-

- It is a function which is plotted as **'S'** shaped graph.
- **Equation :** $A = 1/(1 + e^{-x})$
- **Nature :** Non-linear. Notice that X values lies between -2 to 2, Y values are very steep. This means, small changes in x would also bring about large changes in the value of Y.
- **Value Range :** 0 to 1
- **Uses :** Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be **1** if value is greater than **0.5** and **0** otherwise.

3). **Tanh Function :-** The activation that works almost always better than sigmoid function is Tanh function also knows as **Tangent Hyperbolic function**. It's actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other.

- **Equation :-**

$f(x) = \tanh(x) = 2/(1 + e^{-2x}) - 1$

OR

tanh(x) = 2 * sigmoid(2x) - 1

- **Value Range :-** -1 to +1
- **Nature :-** non-linear
- **Uses :-** Usually used in hidden layers of a neural network as it's values lies between **-1 to 1** hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in *centering the data* by bringing mean close to 0. This makes learning for the next layer much easier.
- **Equation :- A(x) = max(0,x)**. It gives an output x if x is positive and 0 otherwise.
- **Value Range :-** [0, inf)
- **Nature :-** non-linear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.
- **Uses :-** ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

In simple words, RELU learns *much faster* than sigmoid and Tanh function.

5). Softmax Function :- The softmax function is also a type of sigmoid function but is handy when we are trying to handle mult- class classification problems.

- **Nature :-** non-linear
- **Uses :-** Usually used when trying to handle multiple classes. the softmax function was commonly found in the output layer of image classification problems.The softmax function would squeeze the outputs for each class between 0 and 1 and would also divide by the sum of the outputs.
- **Output:-** The softmax function is ideally used in the output layer of the classifier where we are actually trying to attain the probabilities to define the class of each input.
- The basic rule of thumb is if you really don't know what activation function to use, then simply use *RELU* as it is a general activation function in hidden layers and is used in most cases these days.
- If your output is for binary classification then, *sigmoid function* is very natural choice for output layer.
- If your output is for multi-class classification then, Softmax is very useful to predict the probabilities of each classes.

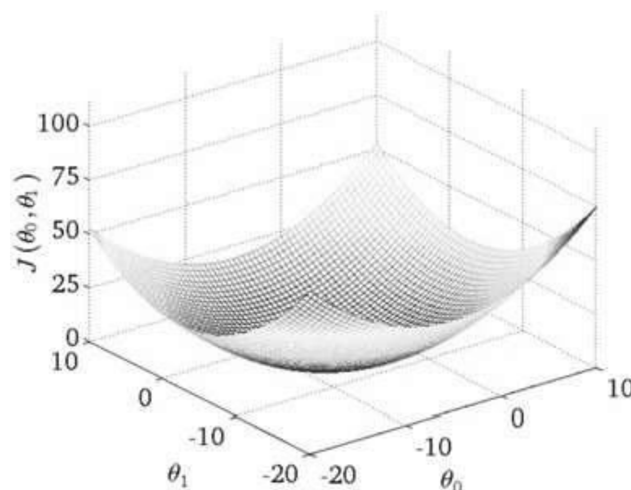
INTRO TO OPTIMIZATION IN DEEP LEARNING: GRADIENT DESCENT

Deep Learning, to a large extent, is really about solving massive nasty optimization problems. A Neural Network is merely a very complicated function, consisting of millions of parameters, that represents a mathematical solution to a problem. Consider the task of image classification. AlexNet is a mathematical function that takes an array representing RGB values of an image, and produces the output as a bunch of class scores.

By training neural networks, we essentially mean we are minimising a loss function. The value of this loss function gives us a measure how far from perfect is the performance of our network on a given dataset.

THE LOSS FUNCTION

Let us, for sake of simplicity, let us assume our network has only two parameters. In practice, this number would be around a billion, but we'll still stick to the two parameter example throughout the post so as not drive ourselves nuts while trying to visualise things. Now, the countour of a *very nice* loss function may look like this.



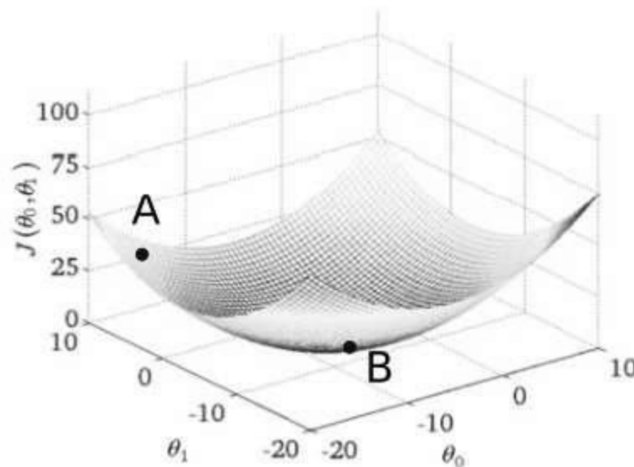
Contour of a Loss Function

Why do I say a *very nice* loss function? Because a loss function having a contour like above is like Santa, it doesn't exist. However, it still serves as a decent pedagogical tool to get some of the most important ideas about gradient descent

across the board. So, let's get to it! The *x* and *y* axes represent the values of the two weights. The *z* axis represents the value of the loss function for a particular value of two weights. Our goal is to find the particular value of weight for which the loss is minimum. Such a point is called a **minima** for the loss function.

You have randomly initialized weights in the beginning, so your neural network is probably behaving like a drunk version of yourself, classifying images of cats as humans. Such a situation correspond to point A on the contour, where the network is performing badly and consequently the loss is high.

We need to find a way to somehow navigate to the bottom of the "valley" to point B, where the loss function has a minima? So how do we do that?

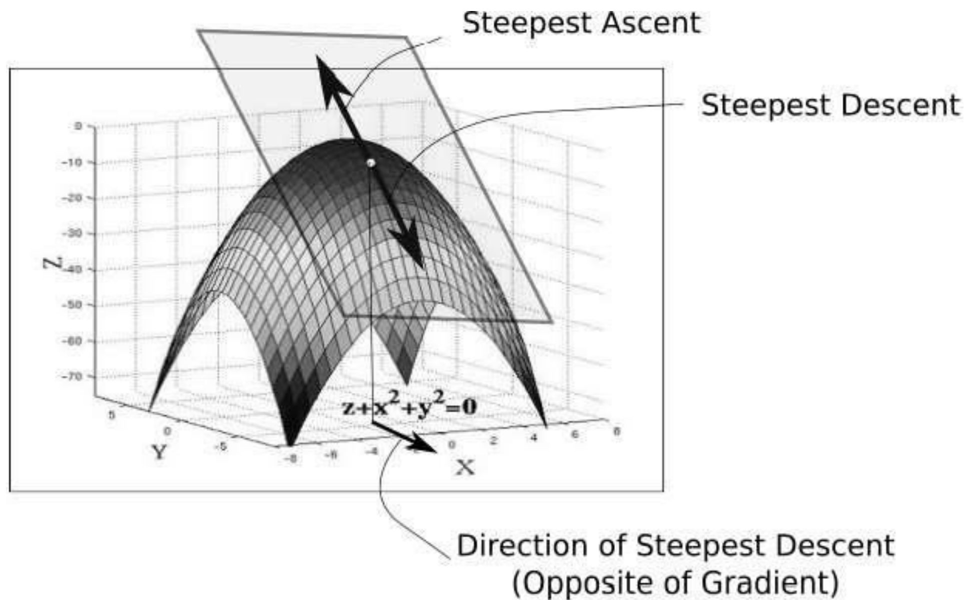


Gradient Descent

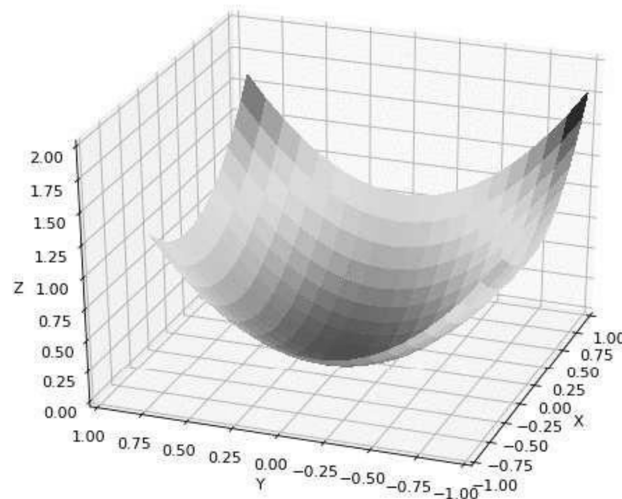
When we initialize our weights, we are at point A in the loss landscape. The first thing we do is to check, out of all possible directions in the x-y plane, **moving along which direction brings about the steepest decline in the value of the loss function**. This is the direction we have to move in. This direction is given by the direction exactly opposite to the direction of the gradient. The gradient, the higher dimensional cousin of derivative, gives us the direction with the steepest ascent.

To wrap your head around it, consider the following figure. At any point of our curve, we can define a plane that is tangential to the point. In higher dimensions, we can always define a hyperplane, but let's stick to 3-D for now. Then, we can have infinite directions on this plane. Out of them, precisely one direction will give us the direction in which the function has the steepest ascent. This direction

is given by the gradient. The direction opposite to it is the direction of steepest descent. This is how the algorithm gets it's name. We perform descent along the direction of the gradient, hence, it's called Gradient Descent.



Now, once we have the direction we want to move in, we must decide the size of the step we must take. The the size of this step is called the **learning rate**. We must chose it carefully to ensure we can get down to the minima.



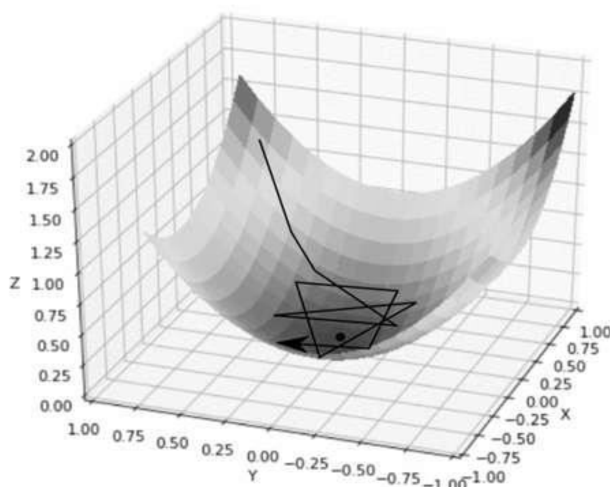
If we go too fast, we might overshoot the minima, and keep bouncing along the ridges of the "valley" without ever reaching the minima. Go too slow, and

the training might turn out to be too long to be feasible at all. Even if that's not the case, very slow learning rates make the algorithm more prone to get stuck in a minima, something we'll cover later in this post.

Once we have our gradient and the learning rate, we take a step, and recompute the gradient at whatever position we end up at, and repeat the process.

While the direction of the gradient tells us which direction has the steepest ascent, it's magnitude tells us how steep the steepest ascent/descent is. So, at the minima, where the contour is almost flat, you would expect the gradient to be almost zero. In fact, it's precisely zero for the point of minima.

Gradient Descent in Action



Using too large a learning rate

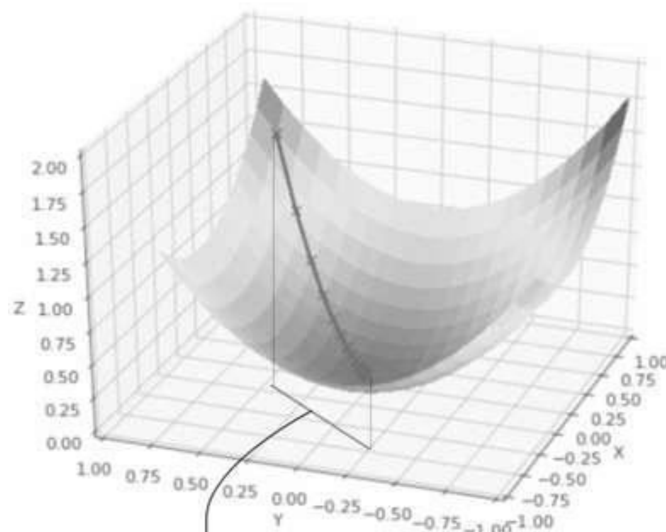
In practice, we might never *exactly* reach the minima, but we keep oscillating in a flat region in close vicinity of the minima. As we oscillate our this region, the loss is almost the minimum we can achieve, and doesn't change much as we just keep bouncing around the actual minimum. Often, we stop our iterations when the loss values haven't improved in a pre-decided number, say, 10, or 20 iterations. When such a thing happens, we say our training has converged, or convergence has taken place.

A COMMON MISTAKE

Let me digress for a moment. If you google for visualizations of gradient descent, you'll probably see a trajectory that starts from a point and heads to a minima, just like the animation presented above. However, this gives you a very

inaccurate picture of what gradient descent really is. The trajectory we take is entire confined to the x-y plane, the plane containing the weights.

As depicted in the above animation, gradient descent doesn't involve moving in *z* direction at all. This is because only the weights are the free parameters, described by the *x* and *y* directions. The actual trajectory that we take is defined in the x-y plane as follows.



Real Gradient Descent Trajectory

Each point in the x-y plane represents a unique combination of weights, and we want have a sets of weights described by the minima.

BASIC EQUATIONS

The basic equation that describes the update rule of gradient descent is.

Repeat Until Convergence {

$$\omega \leftarrow \omega - \alpha * \nabla_w \sum_{1}^{m} L_m(w)$$

}

This update is performed during every iteration. Here, *w* is the weights vector, which lies in the x-y plane. From this vector, we subtract the gradient of the loss function with respect to the weights multiplied by *alpha*, **the learning rate**. The gradient is a vector which gives us the direction in which loss function has the steepest ascent.

The direction of steepest descent is the direction exactly opposite to the gradient, and that is why we are subtracting the gradient vector from the weights vector.

If imagining vectors is a bit hard for you, almost the same update rule is applied to every weight of the network simultaneously. The only change is that since we are performing the update individually for each weight now, the gradient in the above equation is replaced the the projection of the gradient vector along the direction represented by the particular weight.

$$\text{Repeat Until Convergence } \{$$

$$\omega_j \leftarrow \omega_j - \alpha * \nabla_w \sum_{1}^{m} L_m(w).\hat{w_j}$$

$$\}$$

This update is simultaneously done for all the weights.

Before subtracting we multiply the gradient vector by the learning rate. This represents the step that we talked about earlier. Realise that even if we keep the learning rate constant, the size of step can change owing to changes in magnitude of the gradient, ot the steepness of the loss contour. As we approach a minima, the gradient approaches zero and we take smaller and smaller steps towards the minima.

In theory, this is good, since we want the algorithm to take smaller steps when it approaches a minima. Having a step size too large may cause it to overshoot a minima and bounce between the ridges of the minima.

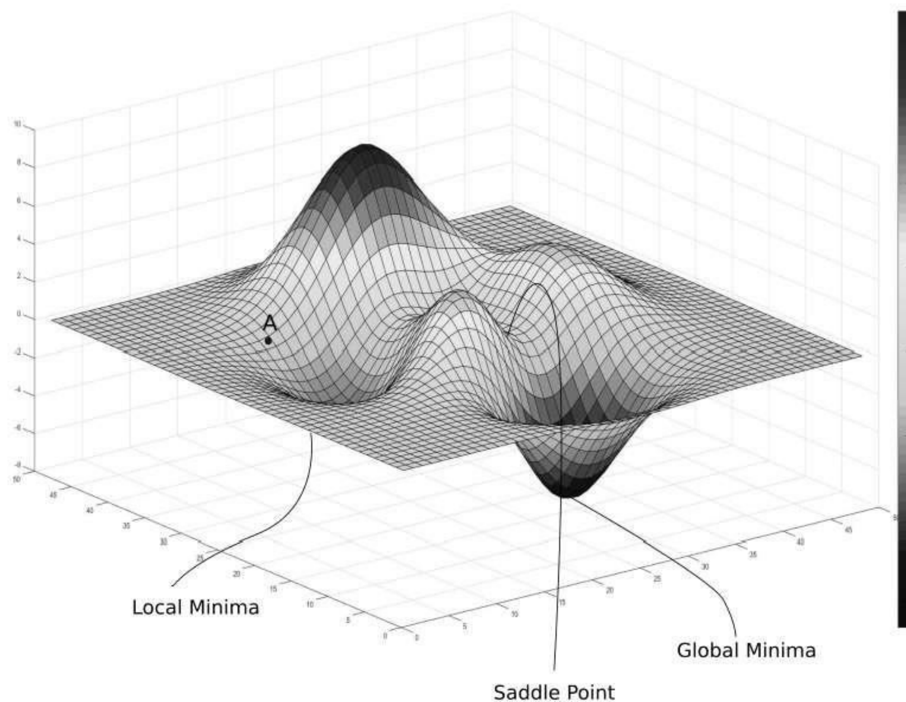
A widely used technique in gradient descent is to have a variable learning rate, rather than a fixed one. Initially, we can afford a large learning rate. But later on, we want to slow down as we approach a minima. An approach that implements this strategy is called **Simulated annealing**, or decaying learning rate. In this, the learning rate is decayed every fixed number of iterations.

CHALLENGES WITH GRADIENT DESCENT #1: LOCAL MINIMA

Okay, so far, the tale of Gradient Descent seems to be a really happy one. Well. Let me spoil that for you. Remember when I said our loss function is *very nice*, and such loss functions don't really exists? They don't.

First, neural networks are complicated functions, with lots of non-linear transformations thrown in our hypothesis function. The resultant loss function doesn't look a nice bowl, with only one minima we can converge to. In fact, such nice santa-like loss functions are called **convex** functions (functions for which are always curving upwards) , and the loss functions for deep nets are hardly convex. In fact, they may look like this.

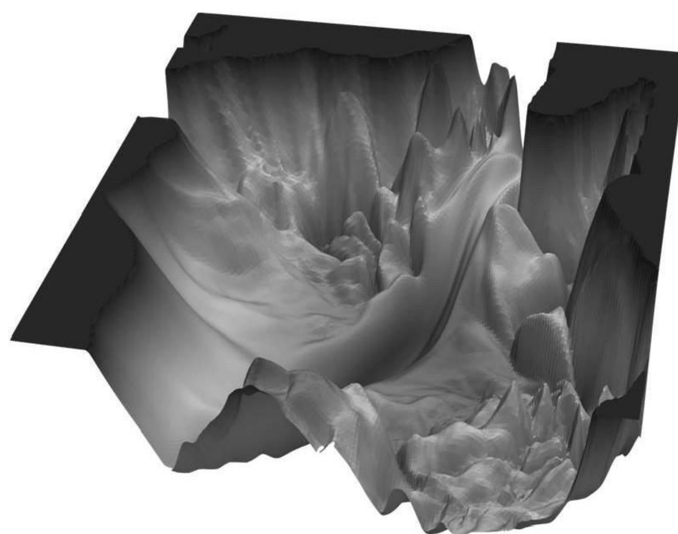
In the above image, there exists a local minima where the gradient is zero. However, we know that they are not the lowest loss we can achieve, which is the point corresponding to the global minima. Now, if you initialze your weights at point A, then you're gonna converge to the local minima, and there's no way gradient descent will get you out of there, once you converge to the local minima.



Gradient descent is driven by the gradient, which will be zero at the base of any minima. Local minimum are called so since the value of the loss function is

minimum at that point in a local region. Whereas, a global minima is called so since the value of the loss function is minimum there, *globally* across the entire domain the loss function.

Only to make things worse, the loss contours even may be more complicated, given the fact that 3-d contours like the one we are considering never actually happen in practice. In practice, our neural network may have about, give or take, 1 billion weights, given us a roughly (1 billion + 1) dimensional function. I don't even know the number of zeros in that figure.

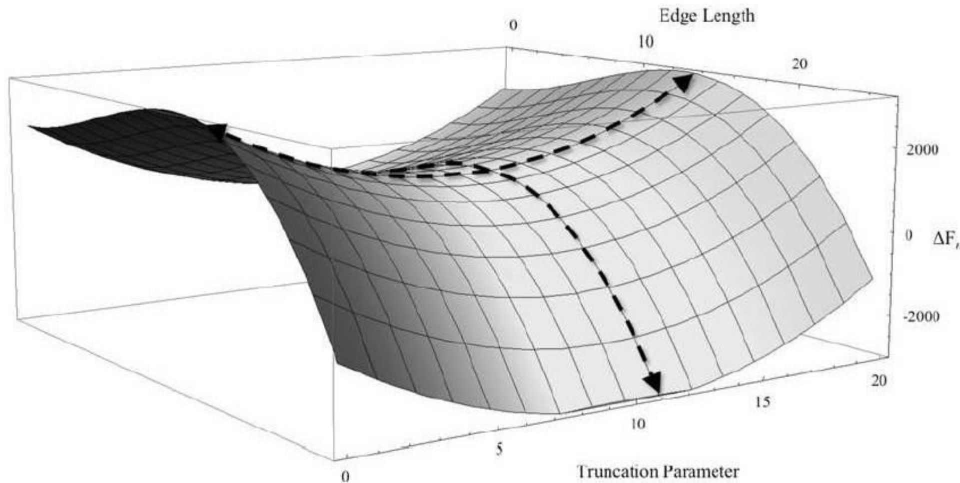


In fact, it's even hard to visualize what such a high dimensional function. However, given the sheer talent in the field of deep learning these days, people have come up with ways to visualize, the contours of loss functions in 3-D. A recent paper pioneers a technique called **Filter Normalization**, explaining which is beyond the scope of this post.

However, it does give to us a view of the underlying complexities of loss functions we deal with. For example, the following contour is a constructed 3-D representation for loss contour of a VGG-56 deep network's loss function on the CIFAR-10 dataset.

CHALLENGES WITH GRADIENT DESCENT #2: SADDLE POINTS

The basic lesson we took away regarding the limitation of gradient descent was that once it arrived at a region with gradient zero, it was almost impossible for it to escape it regardless of the quality of the minima. Another sort of problem we face is that of saddle points, which look like this.



A Saddle Point

You can also see a saddle point in the earlier pic where two "mountains" meet.

A saddle point gets it's name from the saddle of a horse with which it resembles.

While it's a minima in one direction (x), it's a local maxima in another direction, and if the contour is flatter towards the x direction, GD would keep oscillating to and fro in the y - direction, and give us the illusion that we have converged to a minima.

RANDOMNESS TO THE RESCUE!

So, how do we go about escaping local minima and saddle points, while trying to converge to a global minima. The answer is randomness.

Till now we were doing gradient descent with the loss function that had been created by summing loss over all possible examples of the training set. If we get into a local minima or saddle point, we are stuck. A way to help GD escape these is to use what is called Stochastic Gradient Descent.

In stochastic gradient descent, instead of taking a step by computing the gradient of the loss function creating by summing all the loss functions, we take a step by computing the gradient of the loss of only one randomly sampled (without replacement) example.

In contrast to **Stochastic Gradient Descent**, where each example is stochastically chosen, our earlier approach processed all examples in one single batch, and therefore, is known as **Batch Gradient Descent.**

The update rule is modified accordingly.

$$
\begin{aligned}
&\text{Repeat Until Convergence } \{ \\
&\quad \text{for i} = 1...\text{m } \{ \\
&\quad\quad \omega \leftarrow \omega - \alpha * \nabla_w L_m(w) \\
&\quad \} \\
&\}
\end{aligned}
$$

Update Rule For Stochastic Gradient Descent

This means, at every step, we are taking the gradient of a loss function, which is different from our actual loss function (which is a summation of loss of every example). The gradient of this "one-example-loss" at a particular may actually point in a direction slighly different to the gradient of "all-example-loss".

This also means, that while the gradient of the "all-example-loss" may push us down a local minima, or get us stuck at a saddle point, the gradient of "one-example-loss" might point in a different direction, and might help us steer clear of these.

One could also consider a point that is a local minima for the "all-example-loss". If we're doing Batch Gradient Descent, we will get stuck here since the gradient will always point to the local minima. However, if we are using Stochastic Gradient Descent, this point may not lie around a local minima in the loss contour of the "one-example-loss", allowing us to move away from it.

Even if we get stuck in a minima for the "one-example-loss", the loss landscape for the "one-example-loss" for the next randomly sampled data point might be different, allowing us to keep moving.

When it does converge, it converges to a point that is a minima for almost all the "one-example-losses". It's also been emperically shown the saddle points are extremely unstable, and a slight nudge may be enough to escape one.

So, does this mean in practice, should be always perform this one-example stochastic gradient descent?

BATCH SIZE

The answer is no. Though from a theoretical standpoint, stochastic gradient descent might give us the best results, it's not a very viable option from a

computational stand point. When we perform gradient descent with a loss function that is created by summing all the individual losses, the gradient of the individual losses can be calculated in parallel, whereas it has to calculated sequentially step by step in case of stochastic gradient descent.

So, what we do is a balancing act. Instead of using the entire dataset, or just a single example to construct our loss function, we use a fixed number of examples say, 16, 32 or 128 to form what is called a mini-batch. The word is used in contrast with processing all the examples at once, which is generally called Batch Gradient Descent.

The size of the mini-batch is chosen as to ensure we get enough stochasticity to ward off local minima, while leveraging enough computation power from parallel processing.

LOCAL MINIMA REVISITED: THEY ARE NOT AS BAD AS YOU THINK

Before you antagonise local minima, recent research has shown that local minima is not neccasarily bad. In the loss landscape of a neural network, there are just way too many minimum, and a “good” local minima might perform just as well as a global minima.

Why do I say “good”? Because you could still get stuck in “bad” local minima which are created as a result of erratic training examples. “Good” local minima, or often referred to in literature as optimal local minima, can exist in considerable numbers given a neural network’s high dimensional loss function.

It might also be noted that a lot of neural networks perform classification. If a local minima corresponds to it producing scores between 0.7-0.8 for the correct labels, while the global minima has it producing scores between 0.95-0.98 for the correct labels for same examples, the output class prediction is going to be same for both.

A desirable property of a minima should be it that it should be on the flatter side. Why? Because flat minimum are easy to converge to, given there’s less chance to overshoot the minima, and be bouncing between the ridges of the minima.

More importantly, we expect the loss surface of the test set to be slightly different from that of the training set, on which we do our training. For a flat and wide minima, the loss won’t change much due to this shift, but this is not the case for narrow minima. The point that we are trying to make is flatter minima generalise better and are thus desirable.

DEEP LEARNING APPLICATIONS USED ACROSS INDUSTRIES

Deep Learning is a part of Machine Learning used to solve complex problems and build intelligent solutions. The core concept of Deep Learning has been derived from the structure and function of the human brain. Deep Learning uses artificial neural networks to analyze data and make predictions. It has found its application in almost every sector of business.

DEEP LEARNING APPLICATIONS

Virtual Assistants

Virtual Assistants are cloud-based applications that understand natural language voice commands and complete tasks for the user. Amazon Alexa, Cortana, Siri, and Google Assistant are typical examples of virtual assistants. They need internet-connected devices to work with their full capabilities. Each time a command is fed to the assistant, they tend to provide a better user experience based on past experiences using Deep Learning algorithms.

Chatbots

Chatbots can solve customer problems in seconds. A chatbot is an AI application to chat online via text or text-to-speech. It is capable of communicating and performing actions similar to a human.

Chatbots are used a lot in customer interaction, marketing on social network sites, and instant messaging the client. It delivers automated responses to user inputs. It uses machine learning and deep learning algorithms to generate different types of reactions.

Healthcare

Deep Learning has found its application in the Healthcare sector. Computer-aided disease detection and computer-aided diagnosis have been possible using Deep Learning. It is widely used for medical research, drug discovery, and diagnosis of life-threatening diseases such as cancer and diabetic retinopathy through the process of medical imaging.

Entertainment

Companies such as Netflix, Amazon, YouTube, and Spotify give relevant movies, songs, and video recommendations to enhance their customer experience. This is all thanks to Deep Learning. Based on a person's browsing history, interest,

and behavior, online streaming companies give suggestions to help them make product and service choices. Deep learning techniques are also used to add sound to silent movies and generate subtitles automatically.

News Aggregation and Fake News Detection

Deep Learning allows you to customize news depending on the readers' persona. You can aggregate and filter out news information as per social, geographical, and economic parameters and the individual preferences of a reader. Neural Networks help develop classifiers that can detect fake and biased news and remove it from your feed. They also warn you of possible privacy breaches.

Composing Music

A machine can learn the notes, structures, and patterns of music and start producing music independently. Deep Learning-based generative models such as WaveNet can be used to develop raw audio. Long Short Term Memory Network helps to generate music automatically. Music21 Python toolkit is used for computer-aided musicology. It allows us to train a system to develop music by teaching music theory fundamentals, generating music samples, and studying music.

Image Coloring

Image colorization has seen significant advancements using Deep Learning. Image colorization is taking an input of a grayscale image and then producing an output of a colorized image. ChromaGAN is an example of a picture colorization model. A generative network is framed in an adversarial model that learns to colorize by incorporating a perceptual and semantic understanding of both class distributions and color.

Robotics

Deep Learning is heavily used for building robots to perform human-like tasks. Robots powered by Deep Learning use real-time updates to sense obstacles in their path and pre-plan their journey instantly. It can be used to carry goods in hospitals, factories, warehouses, inventory management, manufacturing products, etc.

Boston Dynamics robots react to people when someone pushes them around, they can unload a dishwasher, get up when they fall, and do other tasks as well.

Now, let's understand our next deep learning application, i.e. Image captioning.

Image Captioning

Image Captioning is the method of generating a textual description of an image. It uses computer vision to understand the image's content and a language model to turn the understanding of the image into words in the right order. A

recurrent neural network such as an LSTM is used to turn the labels into a coherent sentence. Microsoft has built its caption bot where you can upload an image or the URL of any image, and it will display the textual description of the image. Another such application that suggests a perfect caption and best hashtags for a picture is Caption AI.

Advertising

In Advertising, Deep Learning allows optimizing a user's experience. Deep Learning helps publishers and advertisers to increase the significance of the ads and boosts the advertising campaigns. It will enable ad networks to reduce costs by dropping the cost per acquisition of a campaign from $60 to $30. You can create data-driven predictive advertising, real-time bidding of ads, and target display advertising.

Self Driving Cars

Deep Learning is the driving force behind the notion of self-driving automobiles that are autonomous. Deep Learning technologies are actually "learning machines" that learn how to act and respond using millions of data sets and training. To diversify its business infrastructure, Uber Artificial Intelligence laboratories are powering additional autonomous cars and developing self-driving cars for on-demand food delivery. Amazon, on the other hand, has delivered their merchandise using drones in select areas of the globe.

The perplexing problem about self-driving vehicles that the bulk of its designers are addressing is subjecting self-driving cars to a variety of scenarios to assure safe driving. They have operational sensors for calculating adjacent objects. Furthermore, they manoeuvre through traffic using data from its camera, sensors, geo-mapping, and sophisticated models. Tesla is one popular example.

Natural Language Processing

Another important field where Deep Learning is showing promising results is NLP, or Natural Language Processing. It is the procedure for allowing robots to study and comprehend human language.

However, keep in mind that human language is excruciatingly difficult for robots to understand. Machines are discouraged from correctly comprehending or creating human language not only because of the alphabet and words, but also because of context, accents, handwriting, and other factors.

Many of the challenges associated with comprehending human language are being addressed by Deep Learning-based NLP by teaching computers (Autoencoders and Distributed Representation) to provide suitable responses to linguistic inputs.

Visual Recognition

Just assume you're going through your old memories or photographs. You may choose to print some of these. In the lack of metadata, the only method to achieve this was through physical labour. The most you could do was order them by date, but downloaded photographs occasionally lack that metadata. Deep Learning, on the other hand, has made the job easier. Images may be sorted using it based on places recognised in pictures, faces, a mix of individuals, events, dates, and so on. To detect aspects when searching for a certain photo in a library, state-of-the-art visual recognition algorithms with various levels from basic to advanced are required.

Fraud Detection

Another attractive application for deep learning is fraud protection and detection; major companies in the payment system sector are already experimenting with it. PayPal, for example, uses predictive analytics technology to detect and prevent fraudulent activity. The business claimed that examining sequences of user behaviour using neural networks' long short-term memory architecture increased anomaly identification by up to 10%. Sustainable fraud detection techniques are essential for every fintech firm, banking app, or insurance platform, as well as any organisation that gathers and uses sensitive data. Deep learning has the ability to make fraud more predictable and hence avoidable.

Personalisations

Every platform is now attempting to leverage chatbots to create tailored experiences with a human touch for its users. Deep Learning is assisting e-commerce behemoths such as Amazon, E-Bay, and Alibaba in providing smooth tailored experiences such as product suggestions, customised packaging and discounts, and spotting huge income potential during the holiday season. Even in newer markets, reconnaissance is accomplished by providing goods, offers, or plans that are more likely to appeal to human psychology and contribute to growth in micro markets. Online self-service solutions are on the increase, and dependable procedures are bringing services to the internet that were previously only physically available.

Detecting Developmental Delay in Children

Early diagnosis of developmental impairments in children is critical since early intervention improves children's prognoses. Meanwhile, a growing body of research suggests a link between developmental impairment and motor competence, therefore motor skill is taken into account in the early diagnosis of developmental

disability. However, because of the lack of professionals and time restrictions, testing motor skills in the diagnosis of the developmental problem is typically done through informal questionnaires or surveys to parents. This is progressively becoming achievable with deep learning technologies. Researchers at MIT's Computer Science and Artificial Intelligence Laboratory and the Institute of Health Professions at Massachusetts General Hospital have created a computer system that can detect language and speech impairments even before kindergarten.

Colourisation of Black and White images

The technique of taking grayscale photos in the form of input and creating colourized images for output that represent the semantic colours and tones of the input is known as image colourization. Given the intricacy of the work, this technique was traditionally done by hand using human labour. However, using today's Deep Learning Technology, it is now applied to objects and their context inside the shot - in order to colour the image, in the same way that a human operator would. In order to reproduce the picture with the addition of color, high-quality convolutional neural networks are utilized in supervised layers.

Adding Sounds to Silent Movies

In order to make a picture feel more genuine, sound effects that were not captured during production are frequently added. This is referred to as "Foley." Deep learning was used by researchers at the University of Texas to automate this procedure. They trained a neural network on 12 well-known film incidents in which filmmakers commonly used Foley effects. Their neural network identifies the sound to be generated, and they also have a sequential network that produces the sound. They employed neural networks to transition from temporally matched visuals to sound creation, a completely another medium!

Automatic Machine Translation

Deep learning has changed several disciplines in recent years. In response to these advancements, the field of Machine Translation has switched to the use of deep-learning neural-based methods, which have supplanted older approaches such as rule-based systems or statistical phrase-based methods. Neural MT (NMT) models can now access the whole information accessible anywhere in the source phrase and automatically learn which piece is important at which step of synthesising the output text, thanks to massive quantities of training data and unparalleled processing power. The elimination of previous independence assumptions is the primary cause for the remarkable improvement in translation quality. This resulted in neural translation closing the quality gap between human and neural translation.

Automatic Handwriting Generation

This Deep Learning application includes the creation of a new set of handwriting for a given corpus of a word or phrase. The handwriting is effectively presented as a series of coordinates utilised by a pen to make the samples. The link between pen movement and letter formation is discovered, and additional instances are developed.

Automatic Game Playing

A corpus of text is learned here, and fresh text has created word for word or character for character. Using deep learning algorithms, it is possible to learn how to spell, punctuate, and even identify the style of the text in corpus phrases. Large recurrent neural networks are typically employed to learn text production from objects in sequences of input strings. However, LSTM recurrent neural networks have lately shown remarkable success in this challenge by employing a character-based model that creates one character at a time.

Demographic and Election Predictions

Gebru et al used 50 million Google Street View pictures to see what a Deep Learning network might accomplish with them. As usual, the outcomes were amazing. The computer learned to detect and pinpoint automobiles and their specs. It was able to identify approximately 22 million automobiles, as well as their make, model, body style, and year. The explorations did not end there, inspired by the success story of these Deep Learning capabilities. The algorithm was shown to be capable of estimating the demographics of each location based just on the automobile makeup.

Deep Dreaming

DeepDream is an experiment that visualises neural network taught patterns. DeepDream, like a toddler watching clouds and attempting to decipher random forms, over-interprets and intensifies the patterns it finds in a picture.

It accomplishes this by sending an image across the network and then calculating the gradient of the picture in relation to the activations of a certain layer. The image is then altered to amplify these activations, improving the patterns perceived by the network and producing a dream-like visual. This method was named "Inceptionism" (a reference to InceptionNet, and the movie Inception).

Bibliography

Bertsekas and Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, 1996.

Bird, S. ; E. Klein, and E. Loper. *Natural language processing with Python*. O'Reilly Media, Inc.", 2009.

Bishop, *Pattern Recognition and Machine Learning*, 2006

David Scott, *Multivariate Density Estimation*, 1992

Devroye and Lugosi, *Combinatorial Methods in Density Estimation*, 2001

Devroye, Gyorfi and Lugosi, *A Probabilistic Theory of Pattern Recognition*, 1996

Domingos, P., *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. Basic Books, 2015.

Duda, Hart, and Stork, *Pattern Classification*, 2nd Ed., 2002

Edward A.: *Computers and Thought*, New York: McGraw-Hill, 1963.

Edwards, Paul N: *The Closed World: Computers and the Politics of Discourse in Cold War America,* Cambridge, MA, The MIT Press, 1996.

Gilligan, C.: *In a Different Voice: Psychological Theory and Women's Development,* Cambridge, Harvard University Press, 1993.

Glass, Robert L.: *Software Creativity*, Prentice Hall PTR, 1995.

Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron, *Deep Learning*. MIT Press, 2016.

Hagle, T. : *Basic Math for Social Scientists - Problems and Solutions*, CA: Sage Publications, 1996.

Han, Jiawei, Micheline Kamber, and Jian Pei: *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

Hardin, J. : *Common Errors in Statistics (and How to Avoid Them)*, New Jersey: Wiley-Interscience, 2003.

Hastie, Friedman, and Tibshirani, *The Elements of Statistical Learning*, 2001

Hastie, T., Tibshirani, R., & Friedman, J., *The Elements of Statistical Learning*, Springer, 2020.

Hawkes, Nigel: *The Computer Revolution*, New York: E.P. Dutton, 1972.

Hawkins, Jeff; Blakeslee, Sandra (2004), *On Intelligence*, New York, NY: Owl Books.

Heims, Steve J.: *The Cybernetics Group*, Cambridge, MA, The MIT Press, 1991.

Henricson, M.: *Industrial Strength C++, Rules and Recommendations*, Hertfordshire: Prentice Hall, Inc, 2000.

Hollingdale, S. H., and G. C. Tootill: *Electronic Computers*, Baltimore: Penguin Books, 1967.

Hopkins, D. : *Personal Growth Through Adventure*, London, David Fulton Publishers, 1993.

Jack Murrin, *Valuation: Measuring and Managing the Value of Companies,* New York, Wiley, 1994.

James Keller: *Public Access to the Internet*, Cambridge, MA: The MIT Press, 1995.

John R. : *Computer Programs for Literary Analysis*, Philadelphia: University of Pennsylvania Press, 1984.

Katherine Davis: *The Computer Establishment*, New York: McGraw-Hill, 1981.

Kawasaki, Guy : *The Computer Curmudgeon,* Indianapolis: Hayden Books, 1992.

Kolb, D.A.: *Experiential Learning: Experience as the Source of Learning and Development,* New Jersey, Prentice-Hall Inc., 1984.

Kraft, Robert A. : *Computer Assisted Tools for Septuagint Analysis*. 1986.

Lawler E.L.: "*Combinatorial Optimization: Networks and Matroids*", New York: Rinehart and Winston, 2003.

Lippman S.B.: "*C++Primer*", London: Addison-Wesley, Publishing Company, 2004.

Mehlhorn K.: "*The Implementation of Geometric Algorithms*", North-Holland, Amsterdam Press, 2003.

Michael S. : *The Microprocessor: A Biography,* Santa Clara, CA: TELOS, 1995.

Murphy, Kevin P., *Machine learning: a probabilistic perspective*, MIT Press 2012.

Ottosen, T.: *Proposal to add Design by Contract to C++,* New York: Ellis Horwood, 1998.

Popper, K.: *The logic of scientific discovery*. Routledge, 2005.

Quinlan, J. R.: *Programs for machine learning*. Elsevier, 2014.

Ripley, *Pattern Recognition and Neural Networks*, 1996

Rockafellar, R. T.: *Convex Analysis*. Princeton, NJ: Princeton University Press, 1970.

Rudin, W.: *Functional Analysis*. New York: McGraw-Hill, 1973.

Russell, S. J., & Norvig, P.: *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

Scholkopf and Smola, *Learning with Kernels*, 2002

Scholkopf, B., & Smola, A. J.: *Learning with kernels: support vector machines, regularization, optimization, and beyond*. Adaptive Computation and Machine Learning Series, 2002.

Speelpenning, B.: *Compiling fast partial derivatives of functions given by algorithms,* University of Illinois at Urbana-Champaign, 1980.

Stanley, J. : *Experimental and Quasi-Experimental Designs for Research*, Boston, MA: Houghton Mifflin Company, 1963.

Sutton and Barto, *Reinforcement Learning: An Introduction*, MIT Press, 1998.

Tan, Steinbach, and Kumar, *Introduction to Data Mining*, Addison-Wesley, 2005.

Van Loan, C. F., & Golub, G. H.: *Matrix computations*. Johns Hopkins University Press, 1983.

Vapnik, V.: *The Nature of Statistical Learning Theory*. New York: Springer, 1995.

Index

A

Activation Functions, 4, 5, 6, 11, 13, 14, 15, 18, 35, 67, 150, 156, 162, 195, 203, 230, 231, 232, 234, 240, 252, 256, 257, 258, 259, 260, 262, 264, 265, 267, 268.

Artificial Deep Neural Networks, 177.

Artificial Neural Network, 158, 161, 178, 180, 182, 184, 187.

Automatic Speech Recognition, 8, 179, 206.

B

Biological Neurons, 73, 177, 184.

C

Components of Neural Networks, 237.

D

Deep Learning, 1, 2, 3, 6, 8, 12, 13, 14, 22, 33, 36, 63, 71, 72, 73, 74, 75, 116, 148, 149, 195, 201, 206, 211, 217, 218, 219, 220, 221, 222, 223, 224, 236, 246, 256, 272, 283, 284, 285, 286, 287, 288.

Descent Algorithm, 59, 127, 135, 136.

G

Geometric Interpretation, 41.

Gradient Descent, 4, 53, 124, 127, 128, 130, 131, 132, 133, 134, 135, 136, 138, 139, 141, 144, 195, 205, 272, 273, 274, 275, 276, 278, 279, 280, 281, 282.

I

Image Processing, 13, 25, 27, 29, 31, 63, 90.

L

Learning Applications, 211, 283.

M

Machine Learning, 12, 25, 71, 91, 135, 150, 153, 159, 168, 174, 175, 183, 186, 283, 284, 289.

N

Natural Language Processing, 2, 75, 152, 213, 221, 285.

Neural Network, 1, 13, 33, 63, 69, 150, 219, 221, 255, 265, 267, 268, 272.

Neuron Model, 37, 38, 39, 225, 226.

P

Principles of Deep Learning, 1.
Programming Tasks, 118.
Python Implementation, 225.
Python Performance, 71.

R

Representation Power, 195.

S

Sigmoid Neuron, 57, 59, 62, 195, 199.
Steepest Descent, 127, 128, 141, 274, 277.

❑❑❑

ABOUT THE AUTHOR

Ahmad Ali AlZubi is a full Professor at Computer Science Department, King Saud University, Saudi Arabia. He obtained his PhD from National Technical University of Ukraine (NTUU) in Computer Networks Engineering in 1999. His current research interests include but not limited to Computer Networks, Grid Computing, Cloud Computing, AI, Machine learning and Deep Learning and their applications in various fields, and services automation. He has also gained valuable industry experience, having worked as a consultant and a member of the Saudi National Team for E-Government in Saudi Arabia. He has author a book title Heart Disease Prediction Using Machine Learning having ISBN: 978-81-19477-42-5

ABOUT THE BOOK

Deep learning using Python involves leveraging libraries like TensorFlow, PyTorch, and Keras to implement neural networks for complex tasks such as image recognition, natural language processing, and reinforcement learning. Python's simplicity and extensive libraries make it ideal for deep learning projects, offering frameworks that simplify the creation, training, and deployment of neural networks. In practice, deep learning projects in Python typically start with data preprocessing and feature extraction to prepare datasets for training. Neural network architectures, such as convolutional neural networks (CNNs) for image data or recurrent neural networks (RNNs) for sequential data, are then designed and implemented using Python frameworks. These frameworks provide high-level APIs that facilitate building and tuning models, making it easier to experiment with different architectures and hyperparameters. Moreover, Python's ecosystem supports visualization tools like Matplotlib and Seaborn, which aid in understanding model performance and data distributions. The integration of deep learning with Python also benefits from the availability of pre-trained models through libraries like TensorFlow Hub and Hugging Face Transformers, accelerating development and improving accuracy for various applications. Overall, Python's versatility and the robustness of deep learning libraries empower developers to explore and implement cutting-edge AI solutions efficiently across diverse domains. Deep Learning Using Python guides readers through implementing advanced neural network models for real-world applications using Python's powerful libraries.

India | UAE | Nigeria | Malaysia | Montenegro | Iraq | Egypt | Thailand | Uganda | Philippines | Indonesia

Parab Publications || www.parabpublications.com || info@parabpublications.com