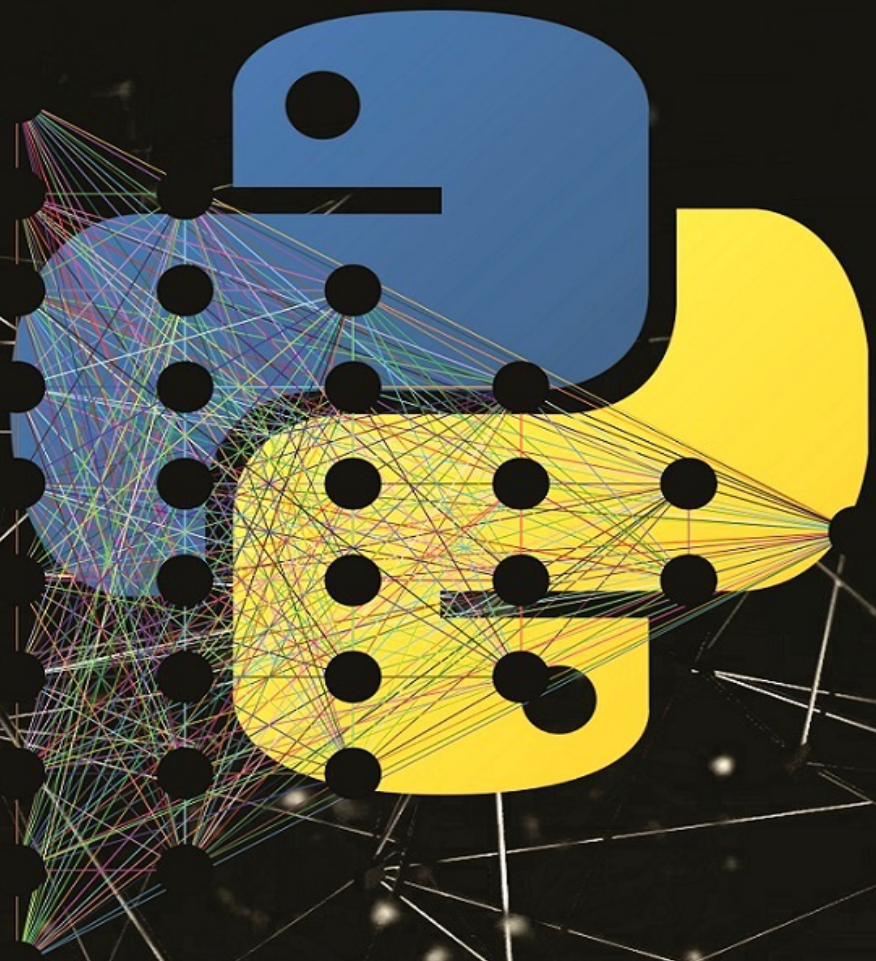
# DEEP LEARNING WITH PYTHON

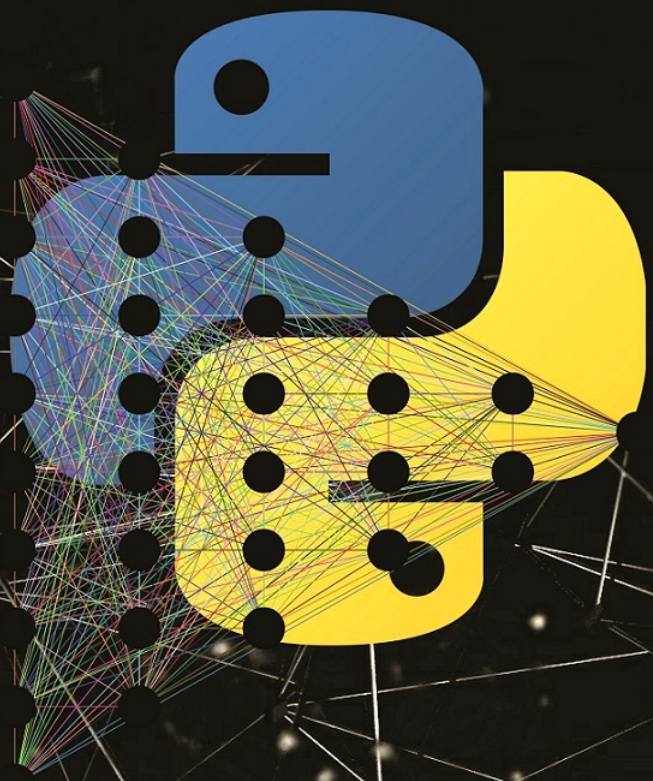
## A Crash Course to Deep Learning with illustrations in Python Programming Language



Robert Kissinger

# DEEP LEARNING WITH PYTHON

A Crash Course to
Deep Learning with illustrations
in Python Programming Language



Robert Kissinger

# Deep Learning
# With Python

A Crash Course to Deep Learning with illustrations
in Python Programming Language

Robert Kissinger

# Copyright

**Copyright©2021 Robert Kissinger**

All rights reserved. No part of this book may be reproduced or used in any manner without the prior written permission of the copyright owner, except for the use of brief quotations in a book review.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper.

Printed in the United States of America
© 2021 by Robert Kissinger

# Table of Contents

Copyright
CHAPTER ONE
ARTIFICIAL INTELLIGENCE, MACHINE LEARNING AND DEEP LEARNING: HOW THEY ALL RELATE
The History of AI and Machine Learning
Artificial Intelligence and Machine Learning
Where are we now with Artificial Intelligence?
Learning representation from data
What Deep Learning has done so far
The promise of AI
Machine Learning: Just before Deep learning
Probabilistic Modeling
Early Neural networks
Kernel Methods
Why Deep learning?
Hardware
Data
Algorithm
Making Deep Learning easy
CHAPTER TWO
UNDERSTANDING THE NEURAL NETWORK
About Keras
Keras & TensorFlow 2.0
Making your first contact with Keras
Installation & compatibility of Keras

Becoming a Keras expert
The Functional API
Training, evaluation, and inference
Custom losses
Handling losses and metrics that don't fit the standard signature
Automatically setting apart a validation holdout set
Training & evaluation from tf.data Datasets
Using a validation dataset
Other input formats supported
Deploying a keras.utils.Sequence object as input
Using class weighting and sample weighting
Class weights
Sample weights
Passing data to multi-input, multi-output models
CHAPTER THREE
Tensor Flow: Single Layer Perceptron
Single Layer Perceptron
Steps to design an algorithm for linear regression
Multi-Layer Perceptrons
Application of TensorFlow
ABOUT AUTHOR

# CHAPTER ONE

# ARTIFICIAL INTELLIGENCE, MACHINE LEARNING AND DEEP LEARNING: HOW THEY ALL RELATE

Artificial intelligence has become the hope of humanity, in an attempt to connect every dot between machines and man. With intelligent machine, it becomes very easy to for human being to relate with them just the way human relates with human. Artificial intelligence offers many promises, much of which are already manifesting in recent time. For instance;

- Artificial intelligence forms the foundation of smart and intelligent devices, which have been used in recent time to give travel alert and weather reports. For instance, say you want to embark on an urgent trip; you can use your smart devices (iPhone, iPad, Apple watch, Fitbit devices etc) to predict weather report for the city or country that you are going. This way, you are in good position to make informed decision concerning whether you can travel or not.
- Artificial intelligence, in the form of smart bots, has been used to send invitation to large group of people, and can even send customized invites to selected group of individuals.

The examples described above are just to illustrate the simple and everyday application of Artificial intelligence to regular persons who are just interested in how the seemingly talked about field – Artificial intelligence – can help them ease their everyday tasks.

## The History of AI and Machine Learning

You must have been wondering by now where the AI came from, well, it is good to say that it didn't evolve to what is today by just dreaming. There were – and currently are – realistic plans by people who had long imagined a world where things happen by mere thinking about it. Artificial intelligence is long rooted in statistics and military science, with great contribution from mathematics, philosophy, cognitive science and psychology. The original rationale behind Artificial intelligence is the quest

to make computer more capable and useful of independent reasoning. Most people who are interested in history have traced the origin of AI to a particular Dartmouth research project in 1956. The project, according to history, explored hot topics such as symbolic methods and problem solving. Few years after this, the US Defense Department took up the work on AI and thereby focused on making computers mimic the ways human being processed thoughts.

For instance, the Defense Advanced Research Projects Agency (DARPA) finished a project in the 1970s when they were able to match all the streets in the Unites States. And even long before the like of Amazon, Microsoft or Google birth the idea, the Defense Advanced Research Projects Agency (DARPA) in 2003 created intelligent personal assistance. This later opened the way for the kind of automation and formal reasoning that are observable in computers we use today.

# Artificial Intelligence and Machine Learning

While Artificial Intelligence entails the all-inclusive science of mimicking the abilities of human, machine learning is a particular subset of Artificial intelligence that trains machine how to incorporate learning. Machine learning and Deep learning are subset of Artificial intelligence

In general, artificial intelligence has many subfields, which are;

- **Machine learning :** This automates analytical model building. It deploys methods from operation research, neural networks, physics and statistics to explore hidden insights in data without being told explicitly where to look for such data or the conclusions to be drawn from such data.
- **A neural network:** This is a type of machine learning that was inspired by the way human brain works. It is actually a computing system that comprises interconnected units that help to process information just by responding to external inputs, thereby transferring information between

units. The process actually needs multiple passes at the data in order to establish connections and obtain meaning from undefined data.

- **Deep learning** : This utilizes vast neural networks with many surfaces of processing units, relying heavily on advances made in the area of computing power and enhanced training methods to learn hard patterns in huge amount of data. Common areas where this is applied include speech and image recognition.
- **Computer vision** : This relies particularly on deep learning and pattern recognition to identify what is in video or an image. When machines are created with the ability to process, analyze and give meaning to images, then they will be able to capture videos or images in real time and give interpretation to their surroundings.
- **Natural language processing:** This refers to the power of computers to scrutinize, understand and then generate human language, including the speech. The succeeding stage of Natural language processing is natural language interaction, which gives human the ability to talk to computers deploying everyday language to carry out tasks.

While machine learning thrives on the belief that machines should be able to adapt and learn through experience, Artificial intelligence talks about a broader concept where machines will be able to perform tasks in a “smart’ way.

Artificial Intelligence uses deep learning, machine learning, and some other techniques to give solutions to real life problems.

# Where are we now with Artificial Intelligence?

With Artificial Intelligence, people can now ask a machine questions, albeit loudly, and then expect a response about just anything – from inventory, sales, customer retention, fraud detection and prevention etc. AI can also give computers the power to discover details that you never even give a thought about. AI will give you a simple summary of your data while helping you suggest other ways you can use to analyze it. It can also share

details that are related to questions that have been asked before; either from you or from anyone else who asked the same question in the past. The answers are usually displayed on the screen or delivered conversationally.

How can we apply all these in the real world? For instance, in health care, effectiveness of a treatment can be determined more quickly. Also, in retails, you can quickly get suggestions about add-on items. In finance, prevention of fraud becomes very easy rather than just detecting it.

In each of the cases highlighted above, the machine knows what detail is needed, checks the relationships between all of the variables, design an answer – and then give it to you letting you follow it up with further queries.

# Learning representation from data

To explain deep learning, and fully understand the discrepancy between deep learning and some other machine learning approaches, you need to understand what the algorithm of machine learning is actually doing. Already, you have seen how machine learning executes tasks given some instructions from users. To fully comprehend machine learning, you will need three things;

- **Input data points** . For example, if your task is speech recognition, the sound files of persons speaking can be the source of the input data points. But if your task is image tagging or identification, the input data point can be picture files.
- **Examples of the output you are expecting.** If the task is a speech recognition task, the expected output can be the human-generated transcript of sound files. If it the task is about image, the expected output can be results like “cat,” “dogs” etc.
- A way to know or determine if the algorithm is doing the right thing, to determine the distance between the algorithm’s current output and the output you are expecting. This is often deployed as an indicator to effectively adjust the way the algorithm is working. This adjustment is what is refers to as “Learning.”

The model for good machine learning will be able to convert all of its input data into an output that is meaningful. Hence, the main problem in deep learning and machine learning is the meaningful conversion of data, or in some other words, to understand important "representations" of the available input data, which is actually the representation that will take you closer to the expected output. Before anything at all, let us first understand what a representation is. At the core, a representation is just a different method of looking at your data – to “encode,” or “represent” your data. For example, you can encode a colored image in the HSV format (“hue-saturation-value”) or RGB format ("red-green-blue"): the same data is being represented in two ways.

If a task is difficult to carry out in one representation, then it is highly likely that such task will become easy with the second representation. For instance, the task "choose all red pixels in the image" is particularly simpler using the RBG format, while the task "let the image be less saturated" is actually a simple task in the second representation (which is the HSV format). The models of Machine learning are all about getting an appropriate representation for the various input data, conversion of the data making it more compliant to the task you are doing, like the classification task.

# What Deep Learning has done so far

Although the field is a subfield of machine learning, it only becomes widely known across the world in 2010. It has been able to solve problems involving “Hearing” and “Seeing” which are things that seem alien to machine, but only a common thing among human.

Realistically, the field of deep learning has helped in achieving the following mileage, and you can see that all of these achievements are within the difficult level of machine learning;

- Image classification: Deep learning can classify image to the level very close to what human being can do.
- Speech recognition: Deep learning can recognize speech to the level very close to what human being can do.
- Handwriting transcription: Deep learning can transcribe handwriting to the level very close to what human being can do.

- Improvement in machine translation.
- Improvement in text-to-speech conversion.
- Digital assistants like Amazon Alexa or Google Now.
- Autonomous driving: Now with the help of AI and Deep learning, we can have cars that can drive to their destinations without necessarily being controlled.
- Improvement in ad targeting, as deployed by Bing, Google, and Baidu.
- Improvement in your search results from the web.
- Giving responses to natural language questions.
- Superhuman Go playing.

## High expectations: What has Deep learning yet to achieve

It is understandable that Deep learning has helped humans solve myriads of problems, projections of what Deep learning will be able to do in years to come seem to be much higher than what will actually be obtainable, due to the fact that the field is just in its starting phase of gaining momentum and getting wide visibility. Although, some miraculous applications of Deep learning such as autonomous cars have been coming on board, many more of such applications will still be evasive for a very long period, like the ability of computers to understand natural languages much like humans, machine that can translate arbitrary language much like humans, and believable dialogue systems. To say it all, talk of anything related to “human-level general intelligence” should not be given much audience for now. The threat with high expectations like these is that, as the results fail to manifest within reasonable framework, funds earmarked for these projects will be exhausted and the progress will be slowed down, even when it could have been concentrated on the seemingly achievable projects of AI. We have seen things like these happening in times past, as AI experienced some cycles of high optimisms which were then preceded by disappointment and skepticism, and shortage of research funding as a result.

It began in the 1960s with symbolic Artificial Intelligence. In those early periods, expectations about Artificial Intelligence were on the high. One of the foremost proponents and pioneers of the symbolic Artificial Intelligence approach was Marvin Minsky, and he claimed in 1967: "Within a generation […] the problem of creating 'artificial intelligence' will substantially be solved". About three years after, in 1970, he came back with a more précised prediction: "in from three to eight years we will have a machine with the general intelligence of an average human being". In 2020, achievements like the ones he talked about are still something to be considered a little far from happening. Actually, there is no means of predicting how long it will take AI to meet these high expectations, but in the 1970s, many AI experts believed the change is just a year ahead. Some years later, as these high hopes refuse to yield results, AI researchers and investors turned their attention away from AI. This was not even the last one, as a new wave of symbolic AI, “expert system”, began to gather momentum in the 1980s among large corporations. Although, they were

able to record some successes which triggered many big corporations to start their own in-house AI departments all in the quest to work on expert systems.
By 1985, corporations were investing over one billion dollar per year on the expert system, while maintaining these systems became more difficult in the early 1990s. It seems now that the best we can do is to allow experts work more on Deep learning while giving timeframe of achievable milestones.

# The promise of AI

While there are widely circulated misconceptions about what AI is not, we cannot but still look at the seemingly bright promise of the futuristic AI. As it is now, it seems the world is just getting started in using deep learning to solve varieties of problems where deep learning could prove very useful, ranging digital assistant to medical diagnoses. From what has been observed in the last one decade, the research on AI has been moving at a progressive rate owing to huge funding earmarked by the government and corporate bodies who understand how AI could help transform their lives and businesses. Despite huge funding, little of the progress has seeped its way into the processes and designs that make up our world as a whole. This is to the fact that a large part of the research findings on AI has not been applied yet, or the findings have just not been applied to the maximum limit of the problem they can actually solve. If you care to check, your doctors in the hospital are alien to AI, the accountant that handles financial commitment of that small organization very close to your residence might probably not know how AI can make his work easier, and even you, yourself don’t really have any clue about how AI can be your number one savior in your day-to-day activities. Yes, you have started using the digital assistant on your smartphone to get conclusive answers on daunting questions. You have been using Amazon to get useful advice about new products. Still, these are still small compared to where and what AI should be, and the tools you are using should, at the very least, be normal tools that are ordinarily deployable in our everyday live. Artificial intelligence is yet to be at the focal point of the way we live our lives, eat, breath, think and procreate. And to be honest, that is where AI ought to be by now. From the look of things, it might be difficult to believe that AI can entail a huge

impact in our lives, since at this time AI has not yet been famous – just like it was difficult to believe about two decades that internet can get to where it is today. About two to three decades ago, most people were ignorant of how internet can help their business and shape their live, but you can see where we are with internet now – people are running ads on the internet reaching millions of people at a time. The same will be applicable to deep learning, as AI is just on its way. In the near future, AI will be your number one assistant, teach your kids, deliver goods to your doorstep and will effortlessly drive your cars while you only sit inside in wonder. AI has been foreseeing to help scientists in latest discoveries across all fields, ranging from mathematics to genomics. Although, there will be challenges along the way, but of course AI will always get back on track. In the future, AI will become central to every little process that constitutes our daily lives and society. The short-time timeframe might be realistically difficult, but the long-term vision is worth investing in. It can take some years before AI gets to its maximum potential, but you can be rest assured that we are getting there. AI will soon be here, and when it does, it will change the way we see things.

# Machine Learning: Just before Deep learning

Although, deep learning is now getting the much needed attention from the public, it is actually not the very first machine learning. Sometimes, deep learning might not necessarily be the right tool for a job – probably because of lack of sufficient data for deep learning to be useful. One way out of this is to learn about other approaches and then practice them as much as possible. Although, this book is not an attempt to go deep into the details of the classical machine learning approaches, but a brief history of how these machine learning approaches came into existence will be discussed so that we can know where to place deep learning and how best to approach it.

# Probabilistic Modeling

When you talk about the application of statistics principle to data analysis, you are, of course, talking about Probabilistic modeling. Probabilistic modeling was among the first type of machine learning, and it is still largely

deployed today. The Naïve Bayes algorithm is among the best used algorithm in probabilistic modeling.

Naive Bayes is just a category of machine learning classifier that is based on deploying the Bayes Theorem while making assumption that the every feature in the input data are all independent (this is a naïve assumption and that is where the name comes from). Data analysis of this type actually come before computers, and has been manually applied even decades before they start using computers to implement it. The Naïve Bayes theory and the foundation of statistics actually date as far back as 18th century, and these foundations are all that are required to start deploying Naïve Bayes classifiers. Another model that looks closely related is the Logistic regression (logreg) which is usually thought of as the "hello world" of contemporary machine learning. Logreg is actually a classification algorithm rather than an algorithm for regression. Just like the Naïve Bayes, the logical regression also comes before computer, and it is also very useful till now due to its versatile and simple nature. More often than not, it is what a data scientist will query on a datasheet in order to have an actual feel of the kind of classification at hand.

# Early Neural networks

Initial iterations of neural networks have been replaced completely by the modern versions that are now available everywhere in the field of Deep Learning. Nonetheless, it is still beneficial to have a look at where deep learning comes from. Actually, the central idea of neural networks were first studied in toy forms dated back as 1950s, the concept took some times before it really got started. For a very long period, the problem has always been lack of the wherewithal to train big neural networks. This problem was solved in the 1980s when many people rediscovered the backpropagation algorithm, which formed a means of training parametric operations by deploying the gradient descent optimization, and then began applying it to neural networks. The very first success recorded in the application of neural networks was in 1989 when Yann LeCun combined the earlier concept of convolutional neural networks and backpropagation together, and then applied this to solve the problem of handwritten digits classification. The result, titled "LeNet" was then deployed by the United States Post Office to automate reading ZIP codes on mail envelope.

# Kernel Methods

In the 1990s when neural network started gaining momentum among researchers, a new systematic method to machine learning called **kernel methods** quickly became known and then drove neural networks into obvious oblivions. Kernel methods are a set of classification algorithms, of which the **Support Vector Machine (SVM)** is the best known. The modern approach of the Support Vector Machine was designed by Cortes and Vapnik at Bell Labs in the early 1990s and was then published in 1995. Although, going far back as 1963, Vapnik and Chervonenkis published the old version of the SVM. The purpose of the SVM is to solve the problem of classification by exploring good “decision boundaries.”

A"decision boundary" actually can be regarded as a surface or a line that separate your training data into two different spaces, which correspond to two different categories. Hence, for proper classification of new data point, you only have to check where they fall on the decision boundary.

The SVMs use two approaches to explore these boundaries;

- First approach is when the data is mapped to a new high-dimensional representation where the decision boundary can be expressed as an hyperplane
- Then you can compute a good decision boundary by maximizing the actual distance between the hyperplane and the data points that are close from each point. This step is called “maximizing the margin.” This enables the boundary to generalize accurately to new samples that are outside of the dataset.

This approach where data are mapped to a representation where a classification problem turn simpler is actually a good debate on paper, but the challenge is that it is particularly intractable going by computation. This is actually, where the “kernel trick” comes in, which forms the main idea in which the kernel methods are actually named after. The gist is: you don’t necessarily have to found the coordinate of your points in the new space when you want to good decision hyperplanes in the new space. What you only have to do is computing the distance between the pair of points in that space which can be done properly using a kernel function.

A kernel function is essentially an operation that is computationally tractable enough to map any two points in your starting space to the

distance between the points in the space of the target representation. Kernel functions are particularly hand crafted and rather not through learning from data. But in SVM, the only thing that is learned is the separation hyperplane. As at the period they were developed, SVMs were able to exhibit state of the art performance on classification problems that were considered simple. Also, SVMs were among the few machines learning procedures that were backed by extensive theory and passed through some serious mathematical vetting, making it easily understood and well interpreted. Owing to these important properties, SVM became very popular in the field for a very long time. The only downside was that SVM seemed difficult to big datasets and could not give reliable results to problems like image classification. Because SVM is an insubstantial method, you will need to first extract useful representations by manual method when you want to apply the method to solve perceptual problems (like the image classification). This extraction is called feature engineering and is often brittle and difficult.

# Why Deep learning?

The two central theme of the deep learning for computer vision, titled backpropagations and convolutional neural networks, were, by 1989, concepts that have already gained wide traction.

The LSTM algorithm, which is fundamental to deep learning for time series, came into light in 1997 and nothing has changed in the algorithm since then. The question to ask now is why deep learning only started getting popular in 2012? What are the things that were done differently across the decades? In essence, there are only three forces that are fueling advances in machine learning, and these are;

- Hardware
- Algorithmic advances.
- Datasets and benchmarks.

Since the field of algorithm advances is founded on experimental findings and not by theory, it only becomes something of reference when there are appropriate hardware and data to experiment new idea or to improve on old ideas. The field of machine learning, unlike physics and mathematics, is not a field where main advances can just be done on paper and pen as it is a

core combination of engineering and science field where practical and visibility are central.

Hence the real problem throughout the 1990s and 2000s was hardware and data. But what happened that period was: internet became widely accepted and graphics chips were made for games.

# Hardware

Within the space of about one decade (from 1990 to 2010), Central processing units (CPUs) have gotten faster by about 5000 times than what was obtainable at the earlier inception of computers. Hence, it is quite easy now than before to have the template of deep learning on your laptop computers, whereas this was not possible about 25 years ago. Nonetheless, the kind of deep learning model deployed in speech recognition or computer vision requires a very high order of magnitude, at least more computationally higher than what your laptop computers can do. In the 2000s, big companies like AMD and NVIDIA have been spending billions towards the development of faster GPUs (Graphic Processing Units) that are strong enough to power the graphics of video games that seem to demand higher graphics. Advanced 3D views on your screen were then rendered by cheap supercomputers in real time. The heavy investment from AMD and NVIDIA now serves as benefit to the research community. In 2007, NVIDIA instituted CUDA, which was actually an interface for its GPU.
A small fraction of Graphic Processing Units began to replace some massive clusters of Central Processing Units in many applications, beginning with physics modeling.

Deep neural networks, having high concentration of small matrix multiplications are at the same time highly parallelizable, and many researchers began to write CUDA implementation of neural network - Alex Krizhevsky and Dan Ciresan were among the first. In lieu of this, we can say that the game industry has actually subsidized supercomputing to meet the standard of the next generation Artificial intelligence. You will agree that big inventions sometimes start as game. Now, the NVIDIA Titan X, which is a gaming GPU that cost about $1000 at the end of 2015, can do a peak of 6.6 TLOPS in a single-precision, i.e. 6.6 trillion of float32

operations in one second. That is actually around 350 times more than what is obtainable in laptops you use today. On a Titan X, it takes just few days to train an ImageNet model of the type that would have won the competition some years back. Meanwhile, big companies use many GPUs of a type fashioned for the sake of deep learning to train deep learning models.

And of course, the seemingly computational capability of such cluster is something that would never have been possible without the use of modern Graphical Processing Unit. In fact, as we speak industries specializing in deep learning have started going beyond GPUs and are now investing in efficient and specialized chips for deep learning. In the year 2016, Google at its annual I/O convention, revealed its Tensor Processing Unit project which will be ten times faster than a GPU.

# Data

Artificial Intelligence has been believed times unnumbered to be the new industrial revolution. It is best to say that if deep learning is the engine of this revolution, then data is the car. Data is actually the raw materials powering intelligent machines, and nothing can be done without sufficient data. Talking about data, the biggest game changer has always been the internet, which has consistently been on the high, making it especially feasible to gather and share very large data for machine learning. Today, big organizations are working with video dataset, image dataset and natural dataset that could not have been possible without access to internet. For instance, user-generated image tags on Flickr have been a big source of data for computer vision, and same for YouTube video. Wikipedia is a big dataset for NLP (Natural Language Processing). If the increase in deep learning studies can be attributed to one factor, it will, of course, be ImageNet dataset. The ImageNet dataset consist 1.4 million images that have been hand-annotated with about 1000 images category (roughly one category per image). Actually, what drives the special nature of the ImageNet is not only its large size of image, but also the completion associated with it. According to Kaggle.com in 2010, healthy competitions are perfect way to motivate engineers and researchers do more. With common benchmark, researchers will outcompete themselves in a bid to drive the cleanest research possible.

# Algorithm

Aside from data and fast hardware, there were still some gap in designing a reliable way to train neural networks, and this was a problem up till the late 2000s.

As a result of this, there is still some dearth of knowledge when it comes to neural networks, as they were only able to come up with one or two layers of representations. Hence, neural networks were unable to shine compared to SVMs or Random Forest, which can be considered a more redefined shallow method. The main issue was that of "gradient propagation" through deep stacks of layers. The feedback signal deployed in training neural network will ordinarily fade away as the quantity of layers increased. This changed in 2009-2010 when many simple yet important algorithmic improvements, which allow improved gradient propagation, were developed. These include;

- Better "activation functions" for neural layers.
- Better "weight initialization schemes". It began with layer-wise pre-training, and was immediately abandoned.
- Better "optimization schemes", such as *RMSprop* and *Adam* .

Deep learning began to shine about the time when these improvements begin to allow the training of models using more than ten (10) layers. Finally, in 2014, 2015 and 2016, researchers were able to come up with advanced designs to even aid gradient propagation, like residual connection, batch normalization and depthwise separable convolutions. Now, training is possible from scratch models, which are many layers deep.

# Making Deep Learning easy

One main key influence that drives more people into learning more about deep learning has always been the fact that the toolsets require for deployment has been totally democratized. In the olden times, working in the area of deep learning require expertise in CUDA and C++, which only few people were good at. Now, most of what is required to carry out advanced research in deep learning is basic python expertise. This might not have been possible without the development of TensorFlow and Theano, which are the two symbolic tensor framework for Python language that work well with auto-differentiation, increase user-centric libraries like Keras, and simplification of new models to make learning attractive . Keras,

just after its early release in 2015, has actually become the learning solution of choice for many startups, researchers and graduate students dwelling on the field.

# CHAPTER TWO

# UNDERSTANDING THE NEURAL NETWORK

To effectively study the concept of neural network, you will need to have a good grasp of the **Keras Python library.**

## About Keras

Keras is a popular deep learning Application Programming Interface written in Python, using TensorFlow machine learning platform. Keras was built owing to the need to have fast experimentation with machine learning. The ability to move quickly from idea to reality is pertinent for efficient research.

## Keras & TensorFlow 2.0

The TensorFlow 2.0 is an open source, end-to-end machine learning platform. Think of TensorFlow as the infrastructure layer for differentiable programming. The TensorFlow 2.0 has four (4) main abilities;

- Tensor operations that are considered low-level are being efficiently executed on TPU, GPU, or CPU.
- Evaluating the gradient of any arbitrary differentiable functions or expressions.
- Providing a medium where scaling computation is accessible to many devices (e.g. the Summit supercomputer at Oak Ridge National Lab, spanning around 27,000 GPUs).
- Transmitting programs ("graphs") to browsers, servers, embedded devices and mobiles which are considered to be external runtimes.

Keras is actually the high-level Application Programming Interface of TensorFlow 2.0: a highly productive and approachable interface for providing solutions to problems concerning machine learning, with a thorough focus on the modern deep learning. Keras offers important building block and abstractions for building and shipping solutions that are applicable to machine learning with a particularly high iteration velocity.

Keras allows researchers and engineers to have full control of the cross-platform capabilities and the excellent scalability of the TensorFlow 2.0; meaning that Keras can be powered on large GPUs clusters or TPU, and it can even be exported to run on your mobile device or on browser.

# Making your first contact with Keras

Keras deploys **layers** and **models** as its core data structure. The Sequential model, which is a linear stack of layers, is the easiest category of models. The Keras functional API is particularly best for complex architectures. The Keras functional API enables you to construct a random graph of layers, or to use **subclassing** to write your models entirely from scratch.

The **Sequential model** is as below;

```
from tensorflow.keras.models import Sequential

model = Sequential()
```

The Stacking layers is just as easy as the .add():

```
from tensorflow.keras.layers import Dense

model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
```

Once the model is just like the way you want it, you can then use the .compile() to configure learning structure for the model;

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

If you have to, the optimizer can be configured further. The basic Kera's philosophy is making simple stuffs simple, while giving the user the ability to have total control of what they are doing whenever there is need for such. And the biggest control users can have is the easy source code extensibility via subclassing.

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True
```

Now, you can;

- Do batch iteration of your training data

```
# x_train and y_train are Numpy arrays --just like in the Scikit-Learn API.
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

- Evaluate your metrics and test loss in one line;

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

- Or produce some predictions on new set of data;

```
classes = model.predict(x_test, batch_size=128)
```

The code snippet you saw above is just the basic way of using Keras, and it actually resembles the Scikit-Learn API for those who are in the know.

Keras, however, is a typical highly-flexible framework that is particularly effective in iterating research ideas that are of state-of-the-art. Keras uses the idea of **progressive disclosure of complexity** : it is easier to begin, and users can scale this up to handle more advanced use cases, while the only requirement is progressive learning at every stage.

Just the same way you have used a few line to evaluate and train a neural network above, Keras can also be used to develop advanced model architecture or some exotic training procedures. Look below to see how a simple training loop looks like, using TensorFlow GradientTape with Keras functionality;

```python
import tensorflow as tf

# Prepare an optimizer.
optimizer = tf.keras.optimizers.Adam()
# Prepare a loss function.
loss_fn = tf.keras.losses.kl_divergence

# Iterate over the batches of a dataset.
for inputs, targets in dataset:
    # Open a GradientTape.
    with tf.GradientTape() as tape:
        # Forward pass.
        predictions = model(inputs)
        # Compute the loss value for this batch.
        loss_value = loss_fn(targets, predictions)

    # Get gradients of loss wrt the weights.
    gradients = tape.gradient(loss_value, model.trainable_weights)
    # Update the weights of the model.
    optimizer.apply_gradients(zip(gradients, model.trainable_weights))
```

# Installation & compatibility of Keras

The TensorFlow 2.0 is loaded with Keras as tensorflow.keras. If you want to start with Keras, you will need to install the TensorFlow 2.0. The TensorFlow 2.0/Keras is compatible with;

- The Python version 3.5–3.8
- Ubuntu 16.04 or higher version
- Windows 7, Windows 8 and Windows 10.
- macOS 10.12.6 (Sierra) or later.

**To install the TensorFlow 2.0 with pip package,** follow the steps below;

- Install the Python IDE on your system: The required python is version 3.5–3.8 and the **pip** and venv should be version equal to or greater than 19.0. If you already have Pyhton, pip and venv installed, simply move to the next step.

```
$ python3 --version
$ pip3 --version
```

To install Python, pip and venv on Ubuntu,

```
$ sudo apt update
$ sudo apt install python3-dev python3-pip python3-venv
```

To install To install Python, pip and venv on macOS,

/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
export PATH="/usr/local/opt/python/libexec/bin:$PATH"
# if you are on macOS 10.12 (Sierra) use `export PATH="/usr/local/bin:/usr/local/sbin:$PATH"`
brew update
brew install python  # Python 3

To install Python, pip and venv on Windows;

You will have to Install the *Microsoft Visual C++ Redistributable for Visual Studio 2015, 2017, and 2019* . Beginning with the TensorFlow 2.1.0 version, you will need the msvcp140_1.dll file from the Visual Studio packages (which may not be available in the older redistributable packages). The redistributable actually comes with the *Visual Studio 2019, though will be installed separately;*

1. Navigate to the Microsoft Visual C++ downloads ,
2. Move down the page and scroll to the *Visual Studio 2015, 2017 and 2019* area.
3. Download and then install the *Microsoft Visual C++ Redistributable for Visual Studio 2015, 2017 and 2019* for your platform.

# Becoming a Keras expert

# The Functional API

## Introduction

The functional API in Keras is a route to build models that are particularly more flexible than the API of the tf.keras.Sequential. The functional API can handle models with shared layers, multiple inputs or outputs or non-linear topology.

The key idea is from the very fact that the deep learning model is normally a directed acyclic graph (DAG) of layers, hence the functional API is a method to make *graphs of layers* .

Check the model below;

```
(input: 784-dimensional vectors)
       ↧
[Dense (64 units, relu activation)]
       ↧
[Dense (64 units, relu activation)]
       ↧
[Dense (10 units, softmax activation)]
       ↧
(output: logits of a probability distribution over 10 classes)
```

You can agree that the case above is a graph having three layers. To construct this model using the functional API, you will begin by making an input node;

```
inputs = keras.Input(shape=(784,))
```

A 784-dimensional vector is set as the shape of the data. You will normally omit the batch size, because of the fact that it is only the shape of each sample that is specified.

Let us say, for instance, that there is an input image that has a shape of (32, 32, 3), you can use;

```
# Just for demonstration purposes.
img_inputs = keras.Input(shape=(32, 32, 3))
```

The returned input has information about the dtype and the shape of the input data that was fed into your model. The shape is of;

```
inputs.shape
```

```
TensorShape([None, 784])
```

And the dtype is like;

```
inputs.dtype
```

```
tf.float32
```

You can create another node in the graph of layers just by calling a layer on this inputs object:

```
dense = layers.Dense(64, activation="relu")
x = dense(inputs)
```

The "layer call" action just like making an arrow from "inputs" to the created layer. You are actually "passing" the inputs directly to the dense layer, and x is returned as the output.

See below how you can add more layers to the graph of layers;

```
x = layers.Dense(64, activation="relu")(x)
outputs = layers.Dense(10)(x)
```

At this point, a Model can then be created just by indicating the inputs and outputs of the model in the graph of layers:

```
model = keras.Model(inputs=inputs, outputs=outputs, name="mnist_model")
```

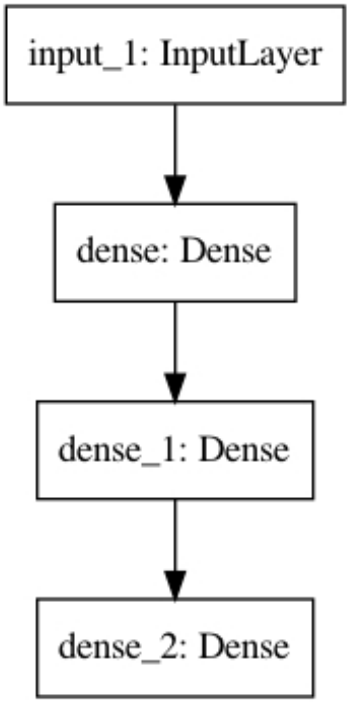
See what the summary of the model look like below;

```
model.summary()
```

```
Model: "mnist_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 784)]             0
_________________________________________________________________
dense (Dense)                (None, 64)                50240
_________________________________________________________________
dense_1 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```

The model can also be plotted as graph;

```
keras.utils.plot_model(model, "my_first_model.png")
```




# Training, evaluation, and inference

Training, evaluation, and inference actually work just exactly in the same manner for models constructed using the functional API as for the Sequential models.

But in this case, you will load the MNIST image data, reshape the image data into vectors, allow the model to fit properly on the data (while you continue to monitor the performance on a validation split), and then estimate the model on the test data;

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

x_train = x_train.reshape(60000, 784).astype("float32") / 255
x_test = x_test.reshape(10000, 784).astype("float32") / 255

model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True)
,
    optimizer=keras.optimizers.RMSprop(),
```

```
    metrics=["accuracy"],
)

history = model.fit(x_train, y_train, batch_size=64, epochs=2,
validation_split=0.2)

test_scores = model.evaluate(x_test, y_test, verbose=2)
print("Test loss:", test_scores[0])
print("Test accuracy:", test_scores[1])
```

```
Epoch 1/2
750/750 [==============================] - 1s 1ms/step - loss:
0.5778 - accuracy: 0.8391 - val_loss: 0.1893 - val_accuracy: 0.9440
Epoch 2/2
750/750 [==============================] - 1s 832us/step - loss:
0.1739 - accuracy: 0.9476 - val_loss: 0.1564 - val_accuracy: 0.9528
313/313 - 0s - loss: 0.1497 - accuracy: 0.9528
Test loss: 0.14971688389778137
Test accuracy: 0.9527999758720398
```

When you want to pass data to the built-in training loop of any specific model, you can either deploy the **NumPy arrays** (in case your data is little and fits in memory) or use the **tf.data Dataset objects** . You will see how the MNIST dataset will be used as NumPy array in order to properly show how to utilize losses, optimizers and metrics. See the model below (although the Functional API was used to build the code, but it can as well be a subclassed or a Sequential model as well);

```
inputs = keras.Input(shape=(784,), name="digits")
x = layers.Dense(64, activation="relu", name="dense_1")(inputs)
x = layers.Dense(64, activation="relu", name="dense_2")(x)
outputs = layers.Dense(10, activation="softmax", name="predictions")(x)

model = keras.Model(inputs=inputs, outputs=outputs)
```

The usual end-to-end workflow looks like;

- Training

- Validation on a holdout set generated from the original training data
- Evaluation on the test data

The MNIST data will be used for this example;

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Preprocess the data (these are NumPy arrays)
x_train = x_train.reshape(60000, 784).astype("float32") / 255
x_test = x_test.reshape(10000, 784).astype("float32") / 255

y_train = y_train.astype("float32")
y_test = y_test.astype("float32")

# Reserve 10,000 samples for validation
x_val = x_train[-10000:]
y_val = y_train[-10000:]
x_train = x_train[:-10000]
y_train = y_train[:-10000]
```

The training configuration consisting of optimizer, metrics and losses is specified below;

```
model.compile(
    optimizer=keras.optimizers.RMSprop(),  # Optimizer
    # Loss function to minimize
    loss=keras.losses.SparseCategoricalCrossentropy(),
    # List of metrics to monitor
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
)
```

The fit() will be called and it is the one that will train the model, and this is done when it slices the data into various “batches” of size "batch_size", and repeatedly iterating over the entire dataset for a given number of "epochs".

```
print("Fit model on training data")
history = model.fit(
    x_train,
    y_train,
    batch_size=64,
    epochs=2,
    # We pass some validation for
    # monitoring validation loss and metrics
    # at the end of each epoch
    validation_data=(x_val, y_val),
)
```

```
Fit model on training data
Epoch 1/2
782/782 [==============================] - 1s 1ms/step - loss: 0.5879 - sparse_categorical_a
Epoch 2/2
782/782 [==============================] - 1s 824us/step - loss: 0.1732 - sparse_categorical
```

The "history" object returned contains a record of the metric value and the loss values during training;

```
history.history
```

```
{'loss': [0.34767386317253113, 0.1630229949951172],
 'sparse_categorical_accuracy': [0.9023799896240234, 0.951479971408844],
 'val_loss': [0.19535799324512482, 0.12865379452705383],
 'val_sparse_categorical_accuracy': [0.9452999830245972, 0.9617999792098999]}
```

The model will be evaluated on the test data via evaluate();

**# Evaluate the model on the test data using `evaluate`**
**print("Evaluate on test data")**
**results = model.evaluate(x_test, y_test, batch_size=128)**
**print("test loss, test acc:", results)**

**# Generate predictions (probabilities -- the output of the last layer)**
**# on new data using `predict`**
**print("Generate predictions for 3 samples")**

**predictions = model.predict (x_test [ :3])**

**print("predictions shape:" predictions.shape**

Evaluate on test data
79/79 [==========] - 0s 736us/step - loss: 0.1365 - sparse_categorical_accuracy: 0.9590
test loss, test acc: [0.13652312755584717, 0.9589999914169312]
Generate predictions for 3 samples
predictions shape: (3, 10)

**The compile() method: specifying a loss, metrics, and an optimizer**

To efficiently train any specific model with fit(), there is the need to indicate an optimizer, a loss function, and some metrics (though optionally).

The optimizer, loss function and the metrics are passed to the model as argument to the model using the compile() method:

```
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=1e-3),
    loss=keras.losses.SparseCategoricalCrossentropy(),
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
)
```

The argument of the metrics must be a list -- the model can contain any number of metrics.

If the model contains multiple outputs, different metrics and losses can be specified for each output, and the contribution of each output can be modulated to the total loss of the model. Details about this will be found in the **"Passing data to multi-input, multi-output models"** section. Be aware that if you are okay with the default settings, the loss, metrics and the optimizer can actually be specified using identifier as a shortcut;

```
model.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["sparse_categorical_accuracy"],
)
```

You can put everything inside a function as shown below;

```
def get_uncompiled_model():
    inputs = keras.Input(shape=(784,), name="digits")
    x = layers.Dense(64, activation="relu", name="dense_1")(inputs)
    x = layers.Dense(64, activation="relu", name="dense_2")(x)
    outputs = layers.Dense(10, activation="softmax", name="predictions")(x)
    model = keras.Model(inputs=inputs, outputs=outputs)
    return model


def get_compiled_model():
    model = get_uncompiled_model()
    model.compile(
        optimizer="rmsprop",
        loss="sparse_categorical_crossentropy",
        metrics=["sparse_categorical_accuracy"],
    )
    return model
```

**Many built-in optimizers, losses, and metrics are available**

In short, you are not going to be building your own optimizers, losses and metrics from scratch, because these are part of the Keras API already;

For the Optimizers:

- SGD() (with momentum or without momentum)
- RMSprop()
- Adam()
- etc.

Losses:

- MeanSquaredError()
- KLDivergence()
- CosineSimilarity()
- etc.

Metrics:

- AUC()
- Precision()
- Recall()
- etc.

# Custom losses

Custom losses can be provided in two ways in Keras. The first case builds a function which will be able to take inputs y_true and y_pred. The case below depicts a loss function that finds the mean squared error between predictions and the real data;

```
def custom_mean_squared_error(y_true, y_pred):
    return tf.math.reduce_mean(tf.square(y_true - y_pred))


model = get_uncompiled_model()
model.compile(optimizer=keras.optimizers.Adam(), loss=custom_mean_squared_error)

# We need to one-hot encode the labels to use MSE
y_train_one_hot = tf.one_hot(y_train, depth=10)
model.fit(x_train, y_train_one_hot, batch_size=64, epochs=1)
```

```
782/782 [==============================] - 1s 661us/step - loss: 0.0269

<tensorflow.python.keras.callbacks.History at 0x14fc6af90>
```

If you want a loss function that accepts parameters aside from the y_true and the y_pred, then you can actually subclass

the tf.keras.losses.Loss class and then execute the two methods below;

- __init__(self): take parameters to pass when you call the loss function.
- call(self, y_true, y_pred): utilize the (y_true) targets and the (y_pred) model predictions to compute the loss of the model.

Let us assume that you want to deploy the mean squared error, but you want to insert an additional term that will be able to de-incentivize the values of prediction far from 0.5 (this is based on the assumption that the categorical targets are one-hot encoded and accept values between 0 and 1). This enables any incentive for your model not to be too confident, which may particularly limit or decrease overfitting.

See below to check out it should be done;

```
class CustomMSE(keras.losses.Loss):
    def __init__(self, regularization_factor=0.1, name="custom_mse"):
        super().__init__(name=name)
        self.regularization_factor = regularization_factor

    def call(self, y_true, y_pred):
        mse = tf.math.reduce_mean(tf.square(y_true - y_pred))
        reg = tf.math.reduce_mean(tf.square(0.5 - y_pred))
        return mse + reg * self.regularization_factor


model = get_uncompiled_model()
model.compile(optimizer=keras.optimizers.Adam(), loss=CustomMSE())

y_train_one_hot = tf.one_hot(y_train, depth=10)
model.fit(x_train, y_train_one_hot, batch_size=64, epochs=1)
```

```
782/782 [==============================] - 1s 736us/step - loss: 0.0487

<tensorflow.python.keras.callbacks.History at 0x14f571150>
```

If you require a metric that is not part of the API, custom metrics can easily be created you by just subclassing the tf.keras.metrics.Metric class. Here,

four methods will be implemented;

- __init__(self), this is where state variables will be created for the metric.
- update_state(self, y_true, y_pred, sample_weight=None), which deploys the model predictions y_pred and the targets y_true to update your state variables.
- result(self), which utilizes the state variables to find the final results.
- reset_states(self), which is the one that will reinitialize the state of your metric.

Results computation and State update are kept separate (in update_ state() and result(), respectively). This is because in a few cases, results computation might be hugely expensive, and would only need to be done periodically.

Here is a simple case depicting how you can implement a Categorical TruePositives metric that will help you to count how many samples were classified correctly as belonging to a particular class:

```python
class CategoricalTruePositives(keras.metrics.Metric):
    def __init__(self, name="categorical_true_positives", **kwargs):
        super(CategoricalTruePositives, self).__init__(name=name, **kwargs)
        self.true_positives = self.add_weight(name="ctp", initializer="zeros")

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_pred = tf.reshape(tf.argmax(y_pred, axis=1), shape=(-1, 1))
        values = tf.cast(y_true, "int32") == tf.cast(y_pred, "int32")
        values = tf.cast(values, "float32")
        if sample_weight is not None:
            sample_weight = tf.cast(sample_weight, "float32")
            values = tf.multiply(values, sample_weight)
        self.true_positives.assign_add(tf.reduce_sum(values))

    def result(self):
        return self.true_positives

    def reset_states(self):
        # The state of the metric will be reset at the start of each epoch.
        self.true_positives.assign(0.0)


model = get_uncompiled_model()
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=1e-3),
    loss=keras.losses.SparseCategoricalCrossentropy(),
    metrics=[CategoricalTruePositives()],
)
model.fit(x_train, y_train, batch_size=64, epochs=3)
```

# Handling losses and metrics that don't fit the standard signature

The staggering majority of metrics and losses can be computed from the y_true and the y_pred, where the y_pred is your model's output, though not all of them can be computed this way. For example, a regularization loss may only need the activation of a layer (no targets in a case like this), and an activation like this one may not necessarily be a model output.

In such examples, the self.add_loss(loss_value) can be called from inside by utilizing the call method of a custom layer. Losses that are added in this

way will be added to the "main" loss during training (the one passed to compile()). The below snippet is a simple instance where an activity regularization has been added (you should understand that activity regularization is at the core of all Keras layers – the layer is just for the purpose of giving a concrete instance);

```
class ActivityRegularizationLayer(layers.Layer):
    def call(self, inputs):
        self.add_loss(tf.reduce_sum(inputs) * 0.1)
        return inputs  # Pass-through layer.


inputs = keras.Input(shape=(784,), name="digits")
x = layers.Dense(64, activation="relu", name="dense_1")(inputs)

# Insert activity regularization as a layer
x = ActivityRegularizationLayer()(x)

x = layers.Dense(64, activation="relu", name="dense_2")(x)
outputs = layers.Dense(10, name="predictions")(x)

model = keras.Model(inputs=inputs, outputs=outputs)
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=1e-3),
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
)

# The displayed loss will be much higher than before
# due to the regularization component.
model.fit(x_train, y_train, batch_size=64, epochs=1)
```

The add_metric() can also be used to carry out the same action for logging metric values.

```python
class MetricLoggingLayer(layers.Layer):
    def call(self, inputs):
        # The `aggregation` argument defines
        # how to aggregate the per-batch values
        # over each epoch:
        # in this case we simply average them.
        self.add_metric(
            keras.backend.std(inputs), name="std_of_activation", aggregation="mean"
        )
        return inputs  # Pass-through layer.


inputs = keras.Input(shape=(784,), name="digits")
x = layers.Dense(64, activation="relu", name="dense_1")(inputs)

# Insert std logging as a layer.
x = MetricLoggingLayer()(x)

x = layers.Dense(64, activation="relu", name="dense_2")(x)
outputs = layers.Dense(10, name="predictions")(x)

model = keras.Model(inputs=inputs, outputs=outputs)
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=1e-3),
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
)
model.fit(x_train, y_train, batch_size=64, epochs=1)
```

# Automatically setting apart a validation holdout set

You can observed that the validation_data argument was utilized in the first end-to-end scenario to pass a tuple of NumPy arrays (x_val, y_val) to the model for assessing a validation metrics and validation loss at the end of each epoch.

Consider a new option here; the argument validation_split enables you to reserve some part of the training data for validation automatically. The argument value stands for the part of the data that are to be reserved for validation; hence, you will have to set this to a value that is higher than zero and lower than one. For example, validation_split=0.4 implies "use 40% of your data for the purpose of validation", and validation_split=0.5 means "use 50% of your data for the purpose of validation".

The validation above is computed by using the last x% samples of any particular arrays gotten by the fit call, before any shuffling.

Be reminded that validation_split can only be used when you are training with NumPy data.

```
model = get_compiled_model()
model.fit(x_train, y_train, batch_size=64, validation_split=0.2, epochs=1)
```

```
625/625 [==============================] - 1s 1ms/step - loss: 0.6143 - sparse_categorical_a

<tensorflow.python.keras.callbacks.History at 0x150468390>
```

# Training & evaluation from tf.data Datasets

From the last few paragraphs, you have been shown how you can work with optimizers, metrics and losses, and you have also seen how the validation_split and validation_split arguments in fit, when the data is passed as the NumPy arrays.

Now, let us consider a case where your data is in the form of a tf.data.Dataset object.

The tf.data API is actually a set of utilities in the TensorFlow 2.0 that is used for loading data and then preprocessing the data in a fast and scalable way.

A Dataset instance can also be passed directly to the methods evaluate(), fit(), and predict():

```python
model = get_compiled_model()

# First, let's create a training Dataset instance.
# For the sake of our example, we'll use the same MNIST data as before.
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
# Shuffle and slice the dataset.
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(64)

# Now we get a test dataset.
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.batch(64)

# Since the dataset already takes care of batching,
# we don't pass a `batch_size` argument.
model.fit(train_dataset, epochs=3)

# You can also evaluate or predict on a dataset.
print("Evaluate")
result = model.evaluate(test_dataset)
dict(zip(model.metrics_names, result))
```

In order for the dataset to be reusable for the preceding epoch, the data is reset after each epoch.

If you plan to run your own training using only some specific number of batches from the Dataset, the steps_per_epoch argument can be passed, which indicates the exact number of training steps that the model should run using the dataset before it moves on to another epoch.

If you actually do this, your dataset will not be reset at the end of each epoch, rather the next batch will be drawn. The disadvantage is that your dataset will run out of data eventually, unless you are actually working with an infinitely-looping dataset.

```python
model = get_compiled_model()

# Prepare the training dataset
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(64)

# Only use the 100 batches per epoch (that's 64 * 100 samples)
model.fit(train_dataset, epochs=3, steps_per_epoch=100)
```

# Using a validation dataset

You can also pass a Dataset case as the validation_data argument in fit():

```
model = get_compiled_model()

# Prepare the training dataset
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(64)

# Prepare the validation dataset
val_dataset = tf.data.Dataset.from_tensor_slices((x_val, y_val))
val_dataset = val_dataset.batch(64)

model.fit(train_dataset, epochs=1, validation_data=val_dataset)
```

After each epoch, your model will proceed to iterate over the dataset of the validation and then go ahead to evaluate the validation metrics and validation loss.

If you plan to run your own training using only some specific number of batches from the Dataset, the validation_steps argument can be passed, which indicates the exact number of training steps that the model should run using the dataset before it gets it interrupts validation and move on to another epoch;

```python
model = get_compiled_model()

# Prepare the training dataset
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(64)

# Prepare the validation dataset
val_dataset = tf.data.Dataset.from_tensor_slices((x_val, y_val))
val_dataset = val_dataset.batch(64)

model.fit(
    train_dataset,
    epochs=1,
    # Only run validation using the first 10 batches of the dataset
    # using the `validation_steps` argument
    validation_data=val_dataset,
    validation_steps=10,
)
```

# Other input formats supported

Aside from the NumPy arrays, TensorFlow Datasets and eager tensors, it is actually possible to train your Keras model with Pandas dataframes, or from Python generators that are able to produce batches of data & labels.

In particular, the keras.utils.Sequence class gives an easy interface to construct Python data generators that can be shuffled and are multiprocessing-aware.

In general, it is advisable that you use;

- A NumPy input data in case you have a small set of data and your data fits in memory.
- Dataset objects in case your dataset is large and you have to carry out distributed training.
- Sequence objects in case your dataset is large and you have to carry out custom Python-side processing that you will not be able to do in TensorFlow (e.g. if your focus is only on external libraries for data preprocessing or data loading).

# Deploying a keras.utils.Sequence object as input

keras.utils.Sequence is actually a utility that you can use to get a Python generator having two important properties when you subclass it;

- It performs very well with multiprocessing.
- It can also be shuffled (e.g. when you pass shuffle=True in fit()).

A Sequence must be able to implement two methods:

- __getitem__
- __len__

The __getitem__ method should be able to return a complete batch. If you plan to modify your dataset totally or slightly between epochs, the on_epoch_end method can be implemented; see below;

```
from skimage.io import imread
from skimage.transform import resize
import numpy as np

# Here, `filenames` is list of path to the images
# and `labels` are the associated labels.

class CIFAR10Sequence(Sequence):
    def __init__(self, filenames, labels, batch_size):
        self.filenames, self.labels = filenames, labels
        self.batch_size = batch_size

    def __len__(self):
        return int(np.ceil(len(self.filenames) / float(self.batch_size)))

    def __getitem__(self, idx):
        batch_x = self.filenames[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = self.labels[idx * self.batch_size:(idx + 1) * self.batch_size]
        return np.array([
            resize(imread(filename), (200, 200))
               for filename in batch_x]), np.array(batch_y)

sequence = CIFAR10Sequence(filenames, labels, batch_size)
model.fit(sequence, epochs=10)
```

# Using class weighting and sample weighting

Going by the default settings, the weight of any sample is usually a factor of the frequency of the sample in the dataset. Two methods are available for weighing the data, and these two methods are not dependent on frequency of the sample. These two methods include;

- Class weights, and
- Sample weights

Class weights

The class weight is often set when a dictionary is passed to the class_ weight argument to Model.fit(). The dictionary will map the class indices to the weight that should ordinarily be utilized for samples that belong to this class.

What this means is that you will be able to balance classes without necessarily resampling, or even train any model that focuses on a particular class.

For example, if the class "0" is just half as represented as class "1" in your set of data, the Model.fit(..., class_weight={0: 1., 1: 0.5}) can be used.

The one below is an example of a NumPy where sample weights or class weight are deployed to provide more to the actual classification of class #5 (which is the digit "5" in the MNIST dataset).

```python
import numpy as np

class_weight = {
    0: 1.0,
    1: 1.0,
    2: 1.0,
    3: 1.0,
    4: 1.0,
    # Set weight "2" for class "5",
    # making this class 2x more important
    5: 2.0,
    6: 1.0,
    7: 1.0,
    8: 1.0,
    9: 1.0,
}

print("Fit with class weight")
model = get_compiled_model()
model.fit(x_train, y_train, class_weight=class_weight, batch_size=64, epochs=1)
```

# Sample weights

The “sample weight” can be deployed when you are not making a classifier. This is necessary for good grain control.

- When you are training from the NumPy data: You should pass the sample_weight argument to the Model.fit().
- When you are training from tf.data or any other iterator: Yield (input_batch, label_batch, sample_weight_batch) tuples.

A "sample weights" array is any array of numbers that indicates the amount of weight that each sample in your batch can have in order to evaluate the total loss.

```python
sample_weight = np.ones(shape=(len(y_train),))
sample_weight[y_train == 5] = 2.0

print("Fit with sample weight")
model = get_compiled_model()
model.fit(x_train, y_train, sample_weight=sample_weight, batch_size=64, epochs=1)
```

# Passing data to multi-input, multi-output models

In the past examples, model having a simple input (a tensor of shape (764,)) and a single output (a prediction tensor of shape (10,)) were considered. But how about looking at models with multiple inputs or outputs?

Let us take the following models having an image output of shape (32, 32, 3) ((height, width, channels)) with a timeseries input of shape (None, 10) ((timesteps, features)). The model will feature two outputs evaluated from the blending of these inputs: a "score" (of shape (1,)) with a probability distribution over five classes (of shape (5,)).

```
image_input = keras.Input(shape=(32, 32, 3), name="img_input")
timeseries_input = keras.Input(shape=(None, 10), name="ts_input")

x1 = layers.Conv2D(3, 3)(image_input)
x1 = layers.GlobalMaxPooling2D()(x1)

x2 = layers.Conv1D(3, 3)(timeseries_input)
x2 = layers.GlobalMaxPooling1D()(x2)

x = layers.concatenate([x1, x2])

score_output = layers.Dense(1, name="score_output")(x)
class_output = layers.Dense(5, name="class_output")(x)

model = keras.Model(
    inputs=[image_input, timeseries_input], outputs=[score_output, class_output]
)
```

When you are compiling, different losses can be specified to different outputs by just passing the loss function as a list;

```
model.compile(
    optimizer=keras.optimizers.RMSprop(1e-3),
    loss=[keras.losses.MeanSquaredError(), keras.losses.CategoricalCrossentropy()],
)
```

If only one function is being passed to the model, then every output will have a share in the loss function, which is not particularly appropriate in

this case.

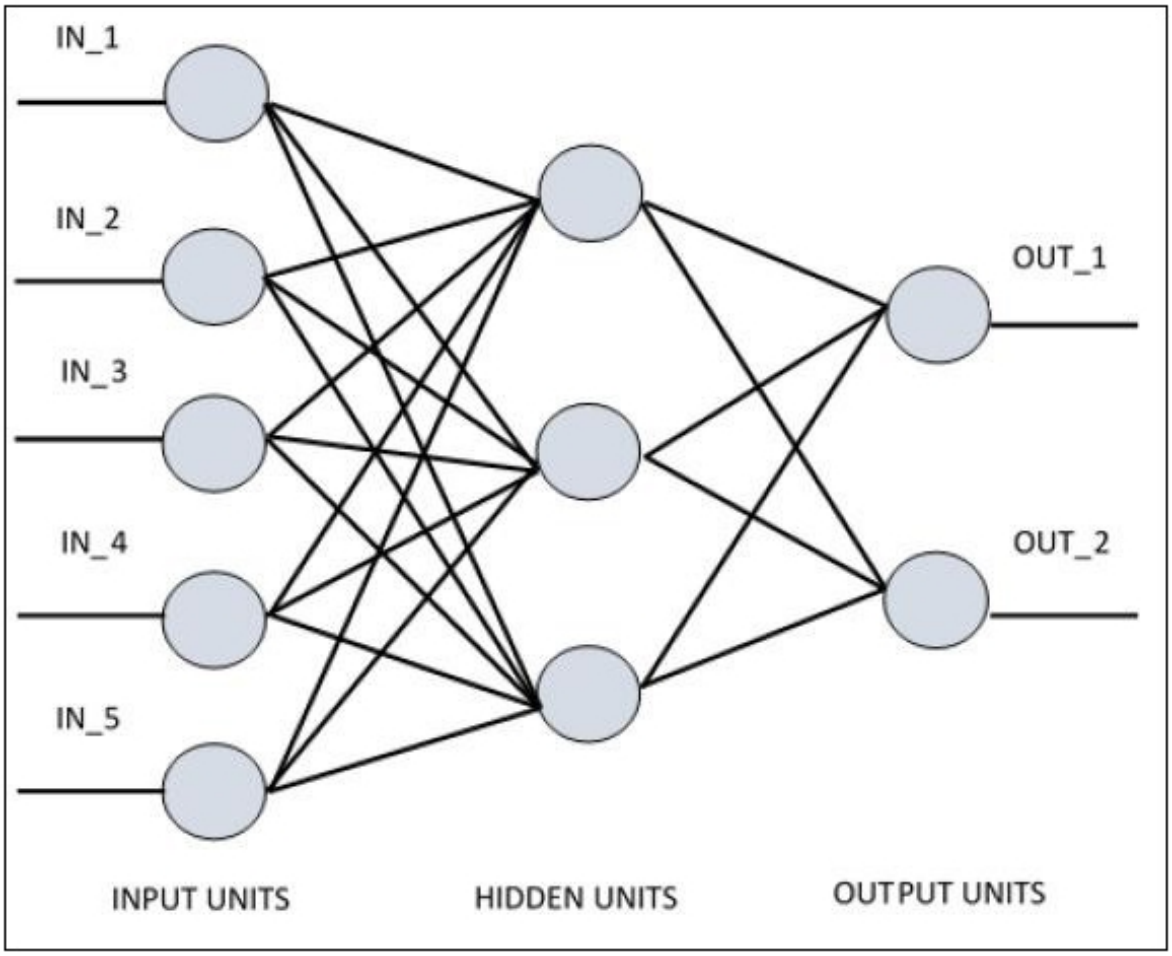
Also for metrics:

```
model.compile(
    optimizer=keras.optimizers.RMSprop(1e-3),
    loss=[keras.losses.MeanSquaredError(), keras.losses.CategoricalCrossentropy()],
    metrics=[
        [
            keras.metrics.MeanAbsolutePercentageError(),
            keras.metrics.MeanAbsoluteError(),
        ],
        [keras.metrics.CategoricalAccuracy()],
    ],
)
```

# CHAPTER THREE

## Tensor Flow: Single Layer Perceptron

For proper evaluation of the single layer perceptron, it becomes necessary to know what Artificial Neural Network (ANN) is. Artificial Neural Network (ANN) refers to the information processing system whose mechanism was stimulated with the biological neural circuits work. An Artificial Neural Network (ANN) has more than one processing units that are connected to each other. See the schematic representation of the artificial neural network below;




The above diagram indicates that the hidden unit is communicating intricately with the external network, but the input and the output unit only communicate via the network's hidden layer.

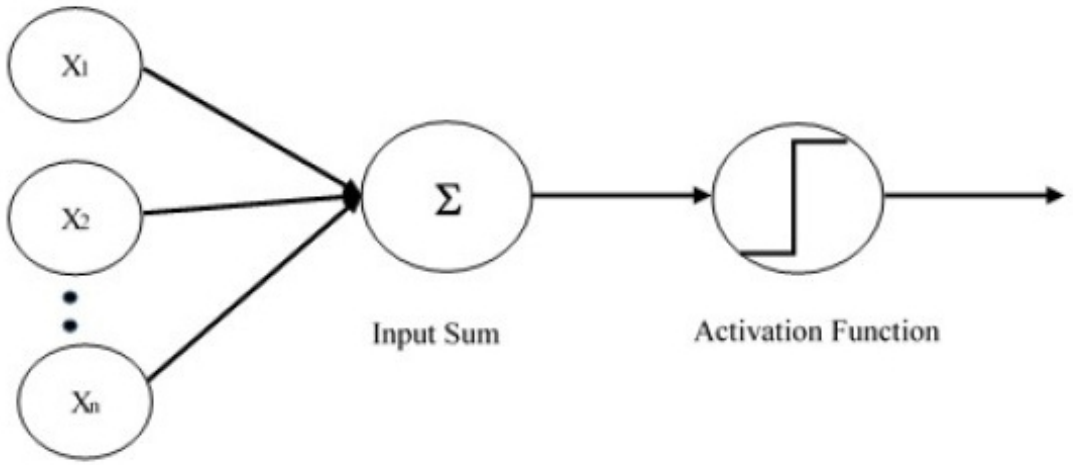
The system of connection with nodes, the actual number of the layers and the level of nodes between outputs and inputs coupled with the number of neurons per layer are what explain a neural network's architecture.

Two types of architectures are possible, and the two types actually concentrate on the artificial neural network. These include;

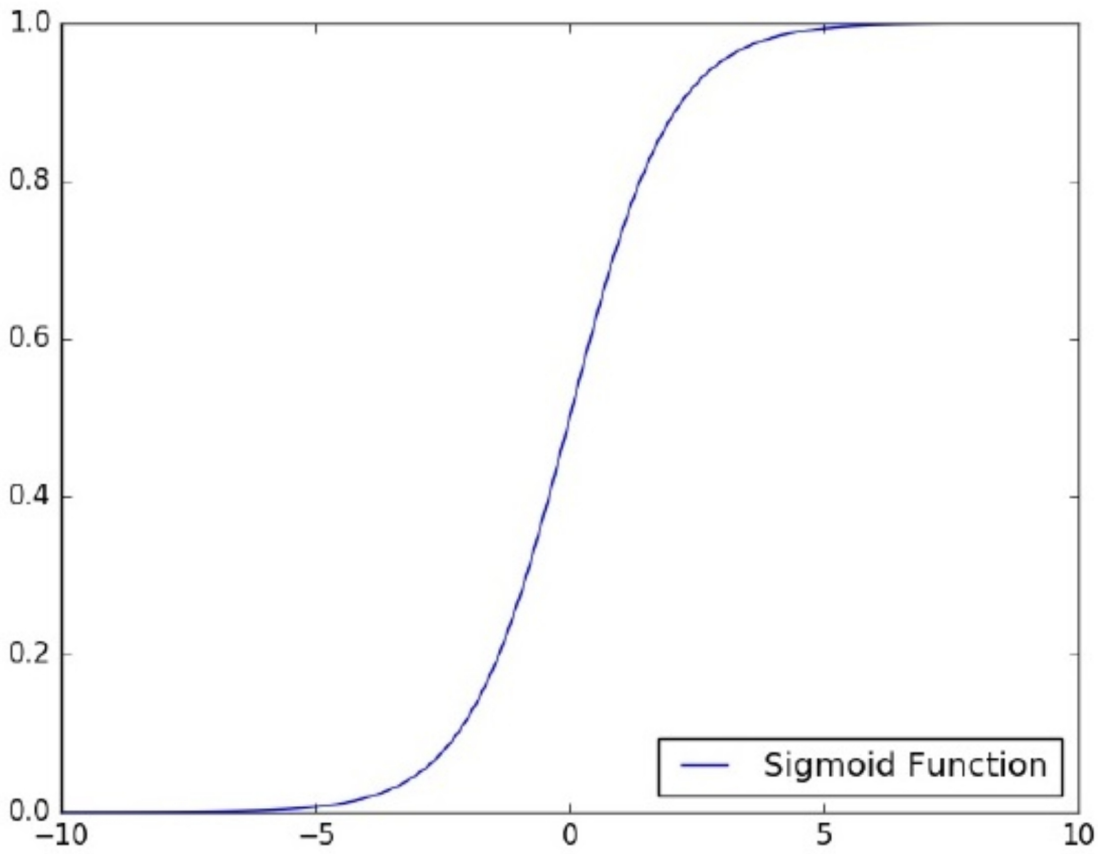
- Single Layer Perceptron, and
- Multi-Layer Perceptron

# Single Layer Perceptron

The very first proposed neural model that was created is the Single layer perceptron. The neuron's local memory has a vector of weights, and the evaluation of a single layer perceptron is done over the estimation of the addition of each of the vectors with the value multiplied with the corresponding element of the vector. The displayed output will be the input of the activation function.




Let us work on how we can implement the single layer perceptron to solve an image classification case by deploying TensorFlow. The best case to show the single layer perceptron is via the characterization of "Logistic Regression".




Now, you should access the basic procedures of training logistic regression highlighted below;

- Random values are used to initialize the weight at the start of the training.
- For every training set element, the error will be estimated variation between the actual output and the desired output. And you can then utilize the calculated error to adjust the weights.
- The entire process is continually repeated until the error from the whole training set is not actually less than the threshold that has been specified, until you reach the limit of iteration.

The logistic regression is a tool for efficient predictive analysis. Logistic regression can be used to properly describe data and also to explain the connection between one dependent binary variable and one or more independent or nominal variables.

Now, we will concentrate on the simple case of implementing linear regression by deploying TensorFlow. Linear regression or Logistic regression is a managed machine learning method that can be used for classifying order discrete categories. Here, you are going to construct a model that can allow a user to predict the connection between one or more independent variables and predictor variables.

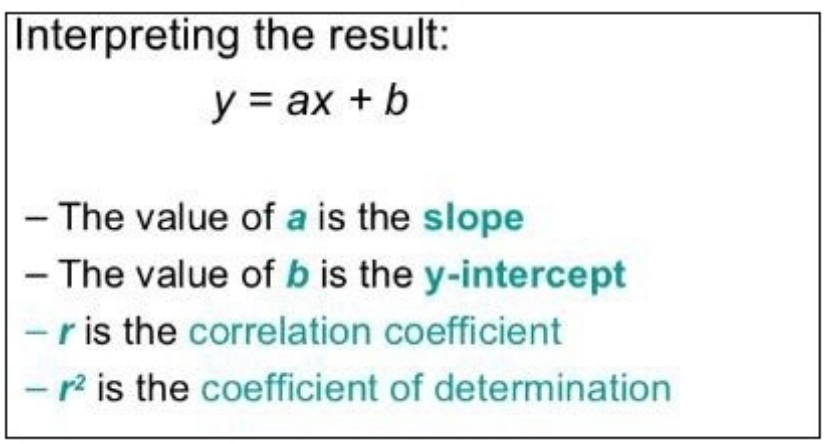
The association between the two variables is considered to be linear. Let us say that X is the independent variable and Y is the independent variable, then you will have the linear regression connection of the two variables looking like;

$$\mathbf{Y = Ax+b}$$

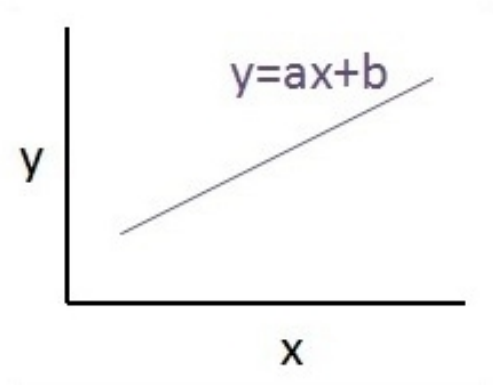
An algorithm will be designed for linear regression that can allow you to fully comprehend the two very important concepts below;

- Cost Function, and
- Gradient descent algorithms

You can see the schematic representation of the linear regression below;




You can see the graph of the regression below;




# Steps to design an algorithm for linear regression

The steps below are important steps that will help you design a feasible algorithm for linear regression;

Step 1

The first thing to do is to import the required modules that will be used to plot your linear regression module. The Python NumPy library and Matplotlib will be imported.

```
import numpy as np
import matplotlib.pyplot as plt
```

Step 2

Estimate and indicate the actual number of coefficients you will require for the logistic regression.

```
number_of_points = 500
x_point = []
y_point = []
a = 0.22
b = 0.78
```

Step 3

Repeat the variables that you can use for generating about 300 random points within the line of regression;

Y = 0.22x+0.78

```
for i in range(number_of_points):
   x = np.random.normal(0.0,0.5)
   y = a*x + b +np.random.normal(0.0,0.1) x_point.append([x])
   y_point.append([y])
```

Step 4

You can view the points generated using Matplotlib.

```
fplt.plot(x_point,y_point, 'o', label = 'Input Data') plt.legend() plt.show()
```

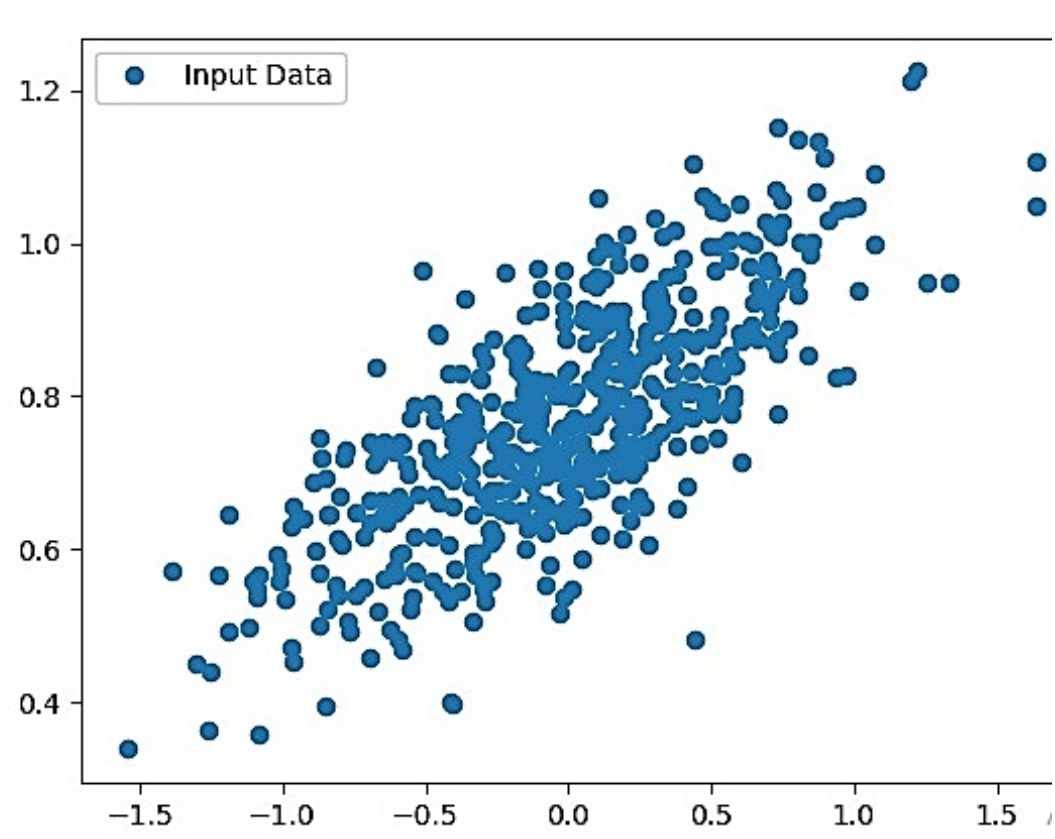
The full codes for the regression will look like;

```
import numpy as np
import matplotlib.pyplot as plt

number_of_points = 500
x_point = []
y_point = []
a = 0.22
b = 0.78

for i in range(number_of_points):
   x = np.random.normal(0.0,0.5)
   y = a*x + b +np.random.normal(0.0,0.1) x_point.append([x])
   y_point.append([y])

plt.plot(x_point,y_point, 'o', label = 'Input Data') plt.legend()
plt.show()
```




# Multi-Layer Perceptrons

A perceptron can be considered as a single model neuron which serves as a precursor to bigger neural networks.

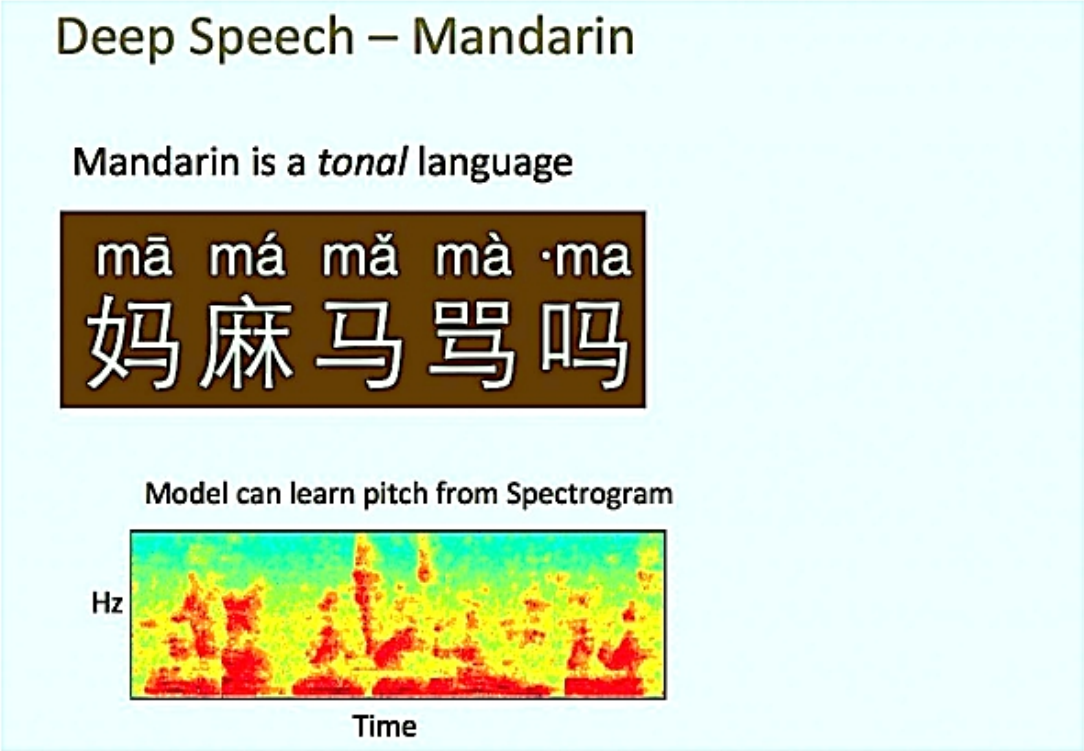
It is an area of specialization that explores how we can utilize simple models of biological brains to offer solutions to difficult computational problems, such as the predicting modeling problem in machine learning. The purpose is not to build realistic models of the human brain, but to, instead, develop strong data structures and algorithms that can be used to model complex problems.

The capability of neural networks stems from their power to understand the representation in your training data and visualize how to relate it to the output variable that you plan to predict. The predictive power of neural networks stems from the multi-layered structure or hierarchical structure of the networks. The data structure can learn to represent concepts at different levels and then combine them to give higher-order features.

# Application of TensorFlow

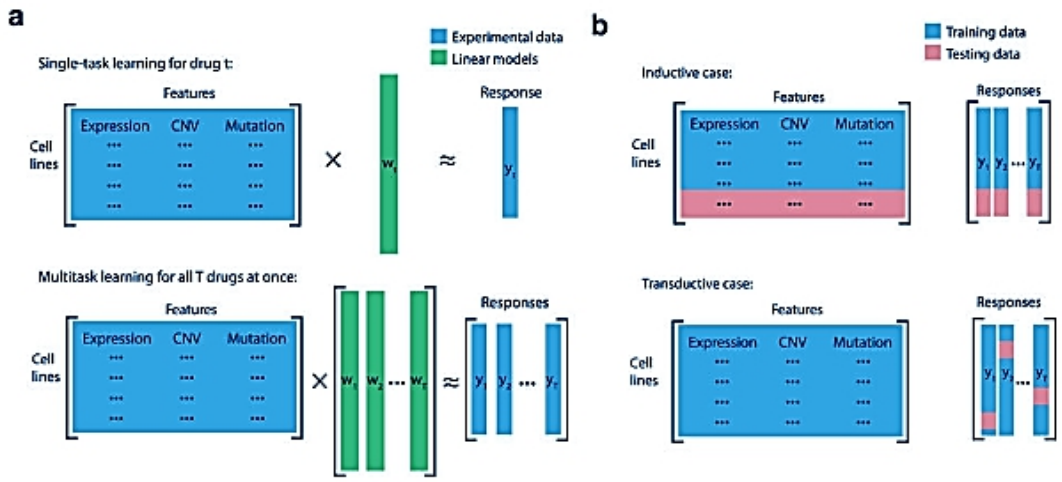
TensorFlow has enormous application as a tool in deep learning. The major areas where it has found its most benefits are in the area of classification, understanding, creation, prediction and perception. Few of the main applications of TensorFlow and many other great projects carried out by the open-source communities are;

- Deep Speech created by *Mozilla* is an implementation of TensorFlow that was motivated by the architecture of Baidu's Deep Speech. The code can be downloaded and worked upon under the license https://github.com/ mozilla/DeepSpeech




- **RankBrain:** This was created by Google, it is a wide-reaching application of deep neural networks for ranking search on www.google.com. It forms an essential component of the search algorithm utilized to sort through the many billions of pages it knows and then locate the ones that are most relevant. The RankBrain is as well the most discussed application of TensorFlow.

- **Inception Image Classifier:** This was also created by Google, and it is a baseline model. It works on achieving a very accurate computer vision model. The model that won the 2014 Imagenet image classification is a typical example of this application.
- Massive multitask for Drug Discovery from the Stanford University, is a deep neural network model that can be used for the purpose of identifying potential drug candidates.




*Massive Multitask*

Real time translation is enabled through optical character recognition from On Device Computer Vision.

- With TensorFlow, you can design algorithms to visualize objects in a photograph or paint an image. You can as well train your pc to recognize objects in image and deploy the data to handle new and interesting behaviors, knowing the differences and similarities in large data sets for self-organizing, to understanding how to generate new contents entirely or even correlate the appearance of other images. You can also teach your computers to create and read new phrases, which are actually part of Natural Language Processing.
- TensorFlow can equally be deployed with containerization tools like docker, for example, it could be utilized to deploy a sentiment analysis model using character level ConvNet networks for the purpose of classifying texts.

# ABOUT AUTHOR

***Robert Kissinger*** is an Apple developer with five years of experience coding Apple apps. He also takes active interest in reviewing gadgets and software. He is also a programmer with incredible expertise in machine learning. Robert has trained a lot of individuals in the field.

***Robert*** studied Computer Science and Engineering from the University of Michigan, USA.