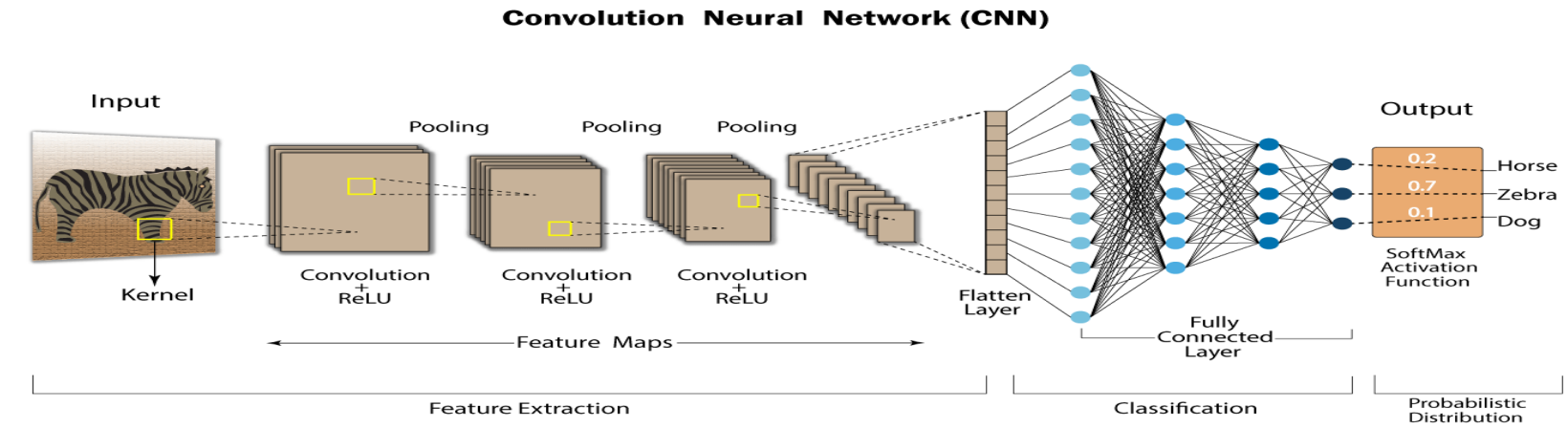
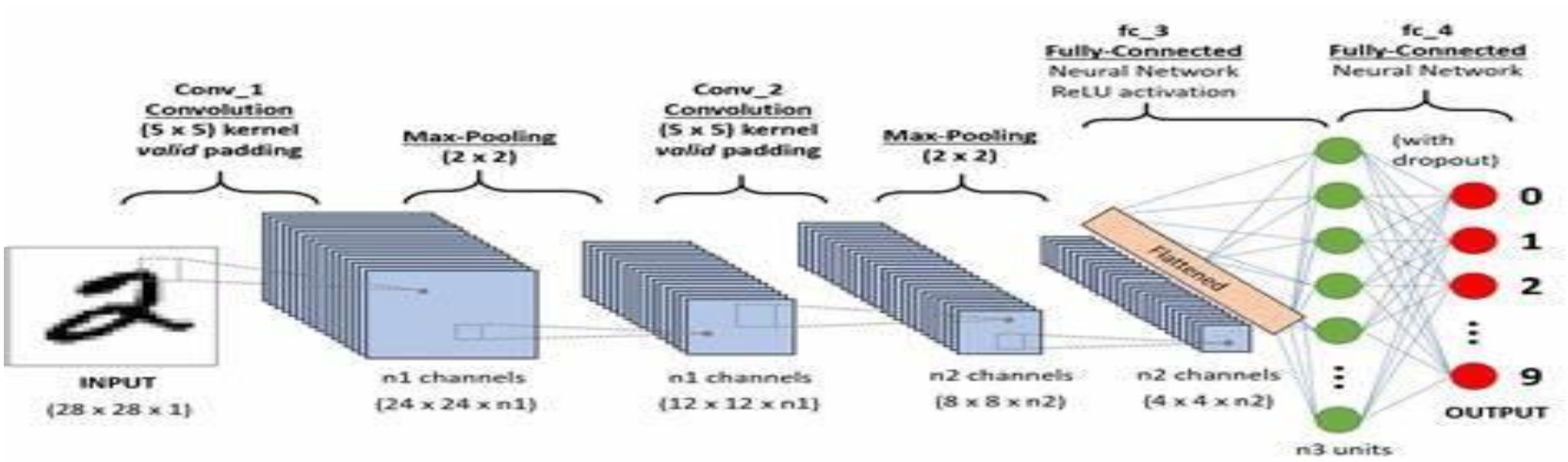
CNN Deep Learning





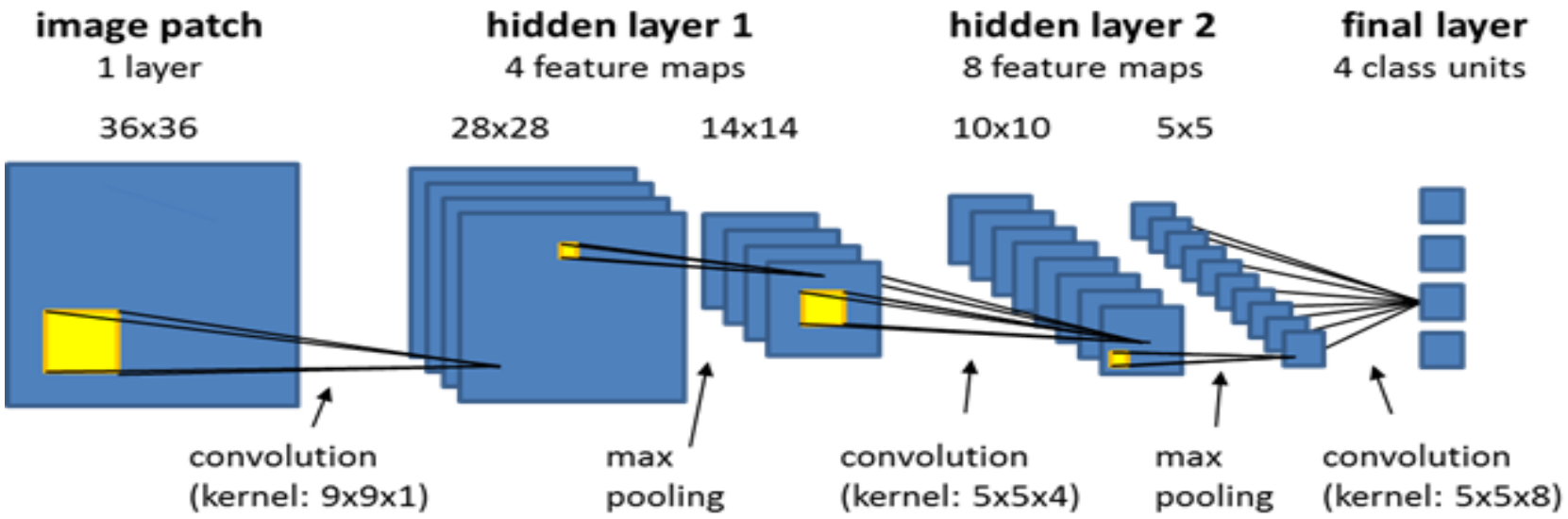
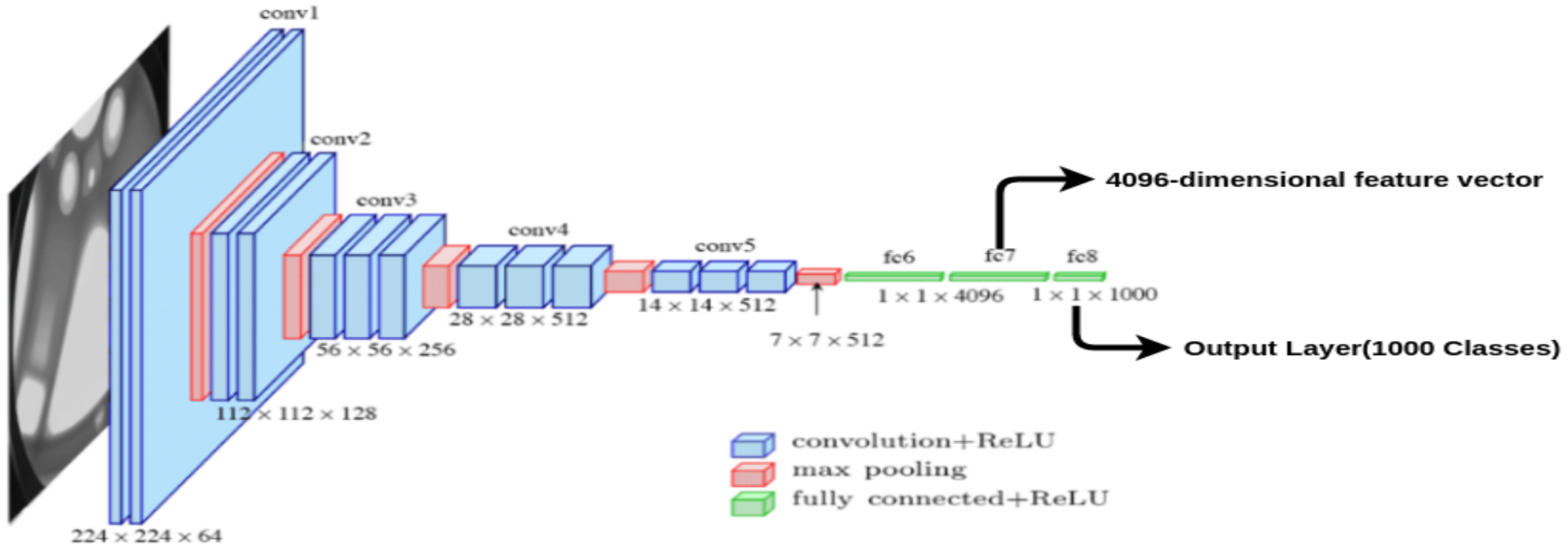


image patch
1 layer
36x36
hidden layer 1
4 feature maps
28x28
14x14
hidden layer 2
8 feature maps
10x10
5x5
final layer
4 class units
convolution
(kernel: 9x9x1)
max
pooling
convolution
(kernel: 5x5x4)
max
pooling
convolution
(kernel: 5x5x8)


conv1
conv2
conv3
conv4
conv5
fc6
fc7
fc8
4096-dimensional feature vector
1 × 1 × 4096
1 × 1 × 1000
Output Layer(1000 Classes)
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
convolution+ReLU
max pooling
fully connected+ReLU

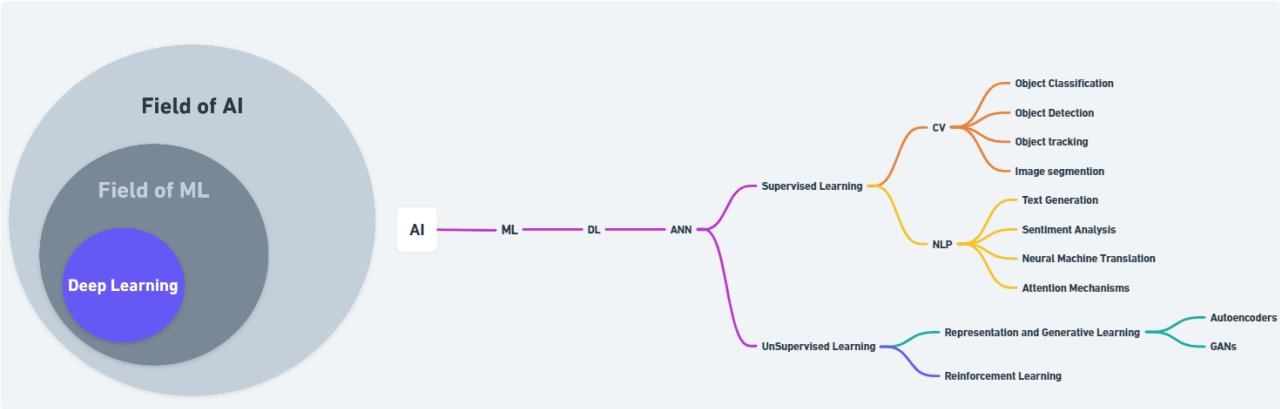
1_introduction_to_deep_learning

Introduction to Deep Learning materials

1. What is Deep Learning
2. Why Deep Learning is important
3. Neural Network Overview And Its Use Case
4. Various Neural Network Architect Overview
5. Use Case Of Neural Network In NLP And Computer Vision

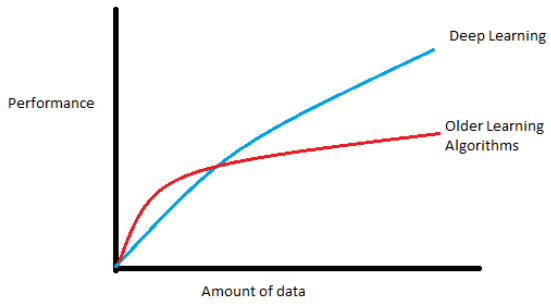
What is Deep Learning

- To understand what deep learning is, we first need to understand the relationship deep learning has with machine learning, neural networks, and artificial intelligence. The best way to think of this relationship is to visualize them as concentric circles:



- At the outer most ring you have artificial intelligence (using computers to reason). One layer inside of that is machine learning. With artificial neural networks and deep learning at the centre.
- Broadly speaking, deep learning is a more approachable name for an artificial neural network. The “deep” in deep learning refers to the depth of the network. An artificial neural network can be very shallow.
- Deep Learning is used to perform complex task which are more computationally expensive, they tend to perform better when compared to machine Learning.

Why Deep Learning is important



- The above diagram shows the relationship between amount of data vs performance of Deep Learning vs other learning algorithms.
- In the above diagram we can see that as the amount of data increases Deep learning Algorithms tend to perform better when compared to other algorithms.
- Let's look at the various use cases of Deep Learning used by companies.
- Facebook has had great success with identifying faces in photographs by using deep learning. It's not just a marginal improvement, but a game changer: "Asked whether two unfamiliar photos of faces show the same person, a human being will get it right 97.53 percent of the time. New software developed by researchers at Facebook can score 97.25 percent on the same challenge, regardless of variations in lighting or whether the person in the picture is directly facing the camera."
- Speech recognition is a another area that's felt deep learning's impact. Spoken languages are so vast and ambiguous. Baidu – one of the leading search engines of China – has developed a voice recognition system that is faster and more accurate than humans at producing text on a mobile phone. In both English and Mandarin.
- What is particularly fascinating, is that generalizing the two languages didn't require much additional design effort: "Historically, people viewed Chinese and English as two vastly different languages, and so there was a need to design very different features," Andrew Ng says, chief scientist at Baidu. "The learning algorithms are now so general that you can just learn."
- Google is now using deep learning to manage the energy at the company's data centres. They've cut their energy needs for cooling by 40%. That translates to about a 15% improvement in power usage efficiency for the company and hundreds of millions of dollars in savings

Neural Network Overview And Its Use Case

- Neural networks are a type of machine learning model that are designed to simulate the way the human brain works. They consist of interconnected nodes, called neurons, which are organized in layers. Each neuron receives input from other neurons in the previous layer, processes this information, and passes its output to the next layer of neurons until the output layer is reached.

- The main use case of neural networks is to make predictions or decisions based on input data. This is done by training the neural network on a set of data, called the training data, where the correct outputs are already known. The neural network adjusts its internal parameters, or weights, during training to minimize the difference between its predicted outputs and the correct outputs in the training data. Once the neural network is trained, it can be used to make predictions on new data that it has not seen before.

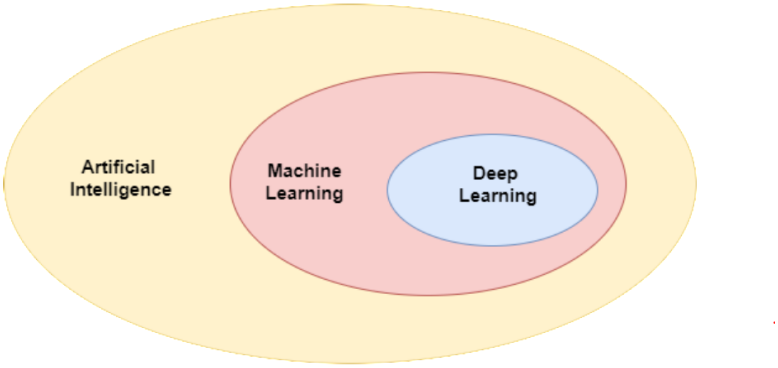
Use case Of Neural Network In NLP And Computer Vision

- Neural networks have a wide range of applications in fields such as computer vision, natural language processing, speech recognition, and autonomous vehicles.
- For example, in computer vision, neural networks are used to identify and classify objects in images or videos. In natural language processing, they are used to generate text or translate between languages.
- In speech recognition, they are used to convert spoken words into text. In autonomous vehicles, they are used to recognize and respond to road conditions and other vehicles.
- Overall, neural networks have become an important tool for machine learning and artificial intelligence, and their use cases are likely to continue expanding in the future.

Deep Learning With Computer Vision And Advanced NLP (DL_CV_NLP)

Introduction to Deep Learning:

Deep Learning is a subset of Machine Learning. Deep Learning is mostly about Neural Network. The main aim of Deep Learning or Neural Network is to mimic the human Brain. Means, It makes the machine learn like human being learn. To understand what deep learning is, we first need to understand the relationship deep learning has with machine learning, neural networks, and artificial intelligence. The best way to think of this relationship is to visualize them as concentric circles:



- At the outer most ring you have artificial intelligence. One layer inside of that is machine learning. With artificial neural networks and deep learning at the center.
- Deep learning is a more approachable name for an artificial neural network. The “deep” in deep learning refers to the depth of the network. An artificial neural network can be very shallow & deep.

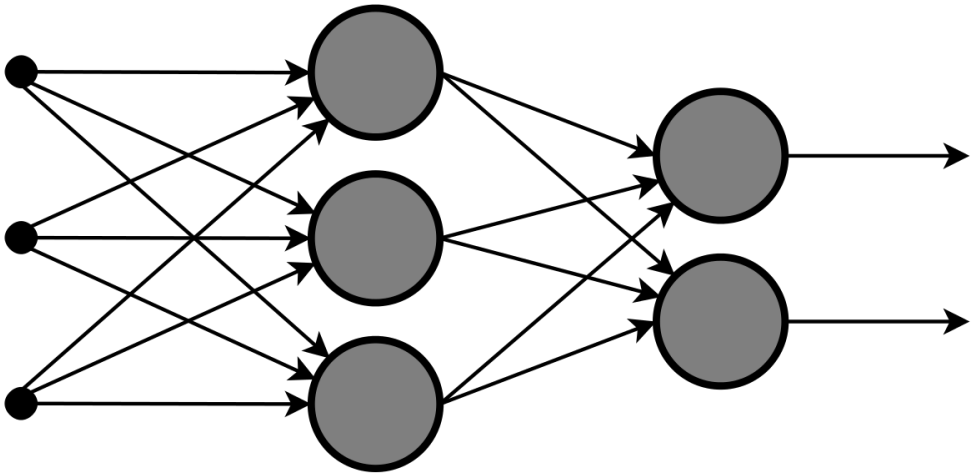
#Some requirements to learn Deep Learning:

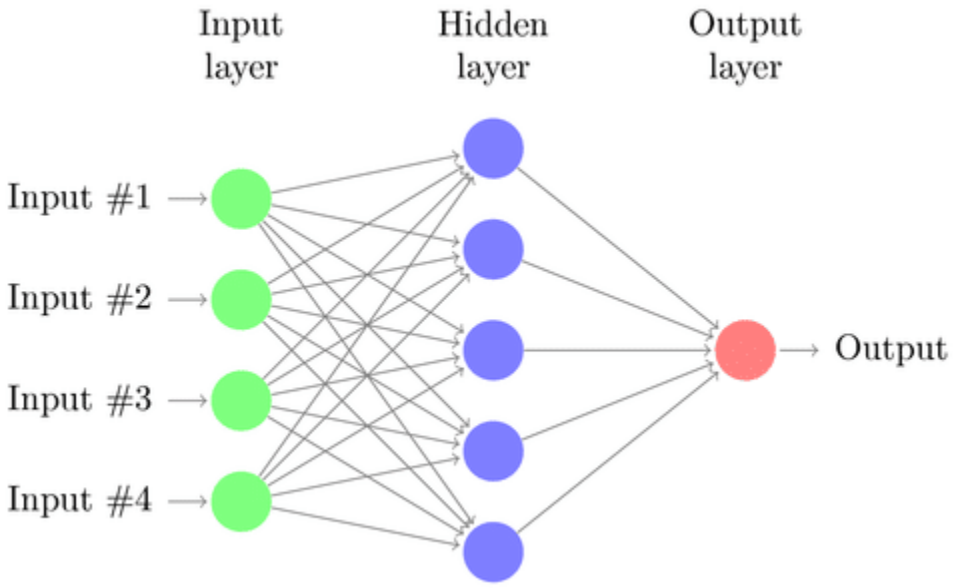
- Mathematics:
  - Functions
  - Vectors
  - Linear Algebra ->> [Metrix, Operations of Metrix]
  - Differential Calculus ->> [Gradient, Partial Derivatives, Differentiation]
  - Graphs
- Programming:
  - Python with OOPs concepts
  - Code readability
- System:
  - At least i3 or i5 processor
  - At least 8gb of RAM
  - Good internet connection
- IDE / Code Editor:
  - Google Colab
  - Paperspace
  - Pycharm
  - VS code
  - jupyter notebook

- spyder

And very last requirement is `Dedication`

Neural Neworks:

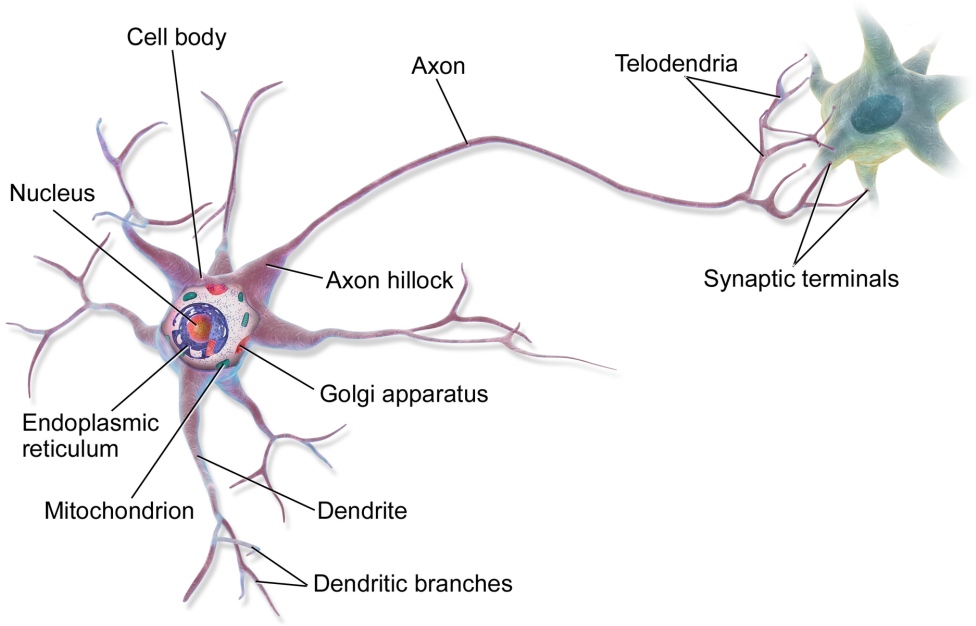




ANN: Artificial Neural Networks:-

- Neural networks are inspired by the structure of the cerebral cortex. At the basic level is the perceptron, the mathematical representation of a biological neuron. Like in the cerebral cortex, there can be several layers of interconnected perceptrons.
- ANNs are **core of deep learning**. Hence one of the most important topic to understand.
- ANNs are **versatile, scalable and powerfull**. Thus it can tackle highly complex ML tasks like classifying images, identying object, speech recognition etc.
- ANN can be single layer or Multiple layers.
- Above the ANN has just 3 layers, The green one is Input layer , Blue one is Hidden layer and the red one is output layer.
- In each neuron except input neuron we use some kinds of activation function for activating the neuron based on some threshold. Activation function also filters out the important data.
- Here each neuron passes some information to others neuron then others neuron take this information and do some calculation and pass to other , that's how it makes the neuron networks.

Biological Neuron:



- Biological Neuron produce short electrical impulses known as `action potentials` which travels through axons to the synapses which releases chemical signals i.e `neurotransmitters` .
- When a connected neuron recieves a sufficient amount of these neurotransmitters within a few milliseconds, it fires (or does not fires, think of a logic gate) its own action potential or elctrical impulse.
- These simple units form a strong network known as Biological Neural Network (BNN) to perform very complex computation task.

The first Artificial Neuron:

- It was in year 1943, Artificial neuron was introduced by-
  - Neurophysiologist Warren McCulloh and
  - Mathematician Walter Pitts
- They have published their work in `McCulloch, W.S., Pitts, W. A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics 5, 115–133 (1943). https://doi.org/10.1007/BF02478259` . read full paper at [link (https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf)](https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf)
- They have shown that these simple neurons can perform small logical operation like OR, NOT, AND gate etc. These neuron only fires when they get two active inputs.

Deep Learning classification: -

- (1) **ANN** --> Artificial Neural Network. This ANN works on all kinds of Tabular dataset (excel sheet data). Any kinds of Regression & Classification task of Machine Learning can be solved by ANN.
- (2) **CNN** --> Convolutional Neural Network. CNN works on computer vision like Images, Videos. CNN architecture works well on these kinds of Image & videos data. Some application of CNN are:-
  - (a) Image Classification --> CNN, Transfer Learning
  - (b) Object Detection --> RCNN, FAST RCNN, FASTER RCNN, SSD, YOLO, DETECTRON
  - (c) Segmentation
  - (d) Tracking

    - (e) GAN >> Generative Adversarial Network
- (3) **RNN** --> Recurrent Neural Network. RNN works whenever input data are text, time series or sequential data like Audio, Time series et cetra. Some techniques are used in RNN:-
    - RNN
    - LSTM RNN
    - Bidirectional LSTM RNN
    - Encoder, Decoder
    - Transformers
    - Bert
    - GPT1, GPT2, GPT3

NLP - Natural Language Processing is the part of RNN.

NLP --> Text --> Vector

- Bag of words (BOW)
- TFIDF
- word2vec
- word embedding

Why Deep Learning?

- Computers have long had techniques for recognizing features inside of images. The results weren't always great. **Computer vision** has been a main beneficiary of deep learning. Computer vision using deep learning now rivals humans on many image recognition tasks.
- **Facebook** has had great success with **identifying faces** in photographs by using deep learning. It's not just a marginal improvement, but a game changer: "Asked whether two unfamiliar photos of faces show the same person, a human being will get it right 97.53 percent of the time. New software developed by researchers at Facebook can score 97.25 percent on the same challenge, regardless of variations in lighting or whether the person in the picture is directly facing the camera."
- **Speech recognition** is a another area that's felt deep learning's impact. Spoken languages are so vast and ambiguous. Baidu – one of the leading search engines of China – has developed a voice recognition system that is faster and more accurate than humans at producing text on a mobile phone. In both English and Mandarin.
- What is particularly fascinating, is that generalizing the two languages didn't require much additional design effort: "Historically, people viewed Chinese and English as two vastly different languages, and so there was a need to design very different features," Andrew Ng says, chief scientist at Baidu. "The learning algorithms are now so general that you can just learn."
- **Google** is now using deep learning to **manage the energy** at the company's data centers. They've cut their energy needs for cooling by 40%. That translates to about a 15% improvement in power usage efficiency for the company and hundreds of millions of dollars in savings.

`To understand more about ANN (Artificial Neural Network)` Please visit [Tensorflow Playground (https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.41926&showTestData](https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.41926&showTestData) and play with the Neural Network.

Installation of Tensorflow, Keras, pytorch & Basic of Google Colab

Tensorflow:

- If you are using google colab then you don't have to install tensorflow additionaly, Colab already has tensorflow pre-installed you have to just import that. To import tensorflow just write

```
import tensorflow as tf
```

- But if you are using your local system like jupyter notebook then you have to install it. To install that first of all create a virtual Environment then activate your Environment and write to your anaconda promt `pip install tensorflow` it will install latest version of tensorflow for you. To check the version of tensorflow `tf.__version__` and for keras `tf.keras.__version__`

pytorch:

- For installing pytorch please visit their website Pytorch (https://pytorch.org)

Activate GPU on colab:

To activate GPU on colab go to the Runtime above and change the runtype type to GPU. To check your GPU just write `!nvidia-smi`

##Deep Learning Frameworks

Deep learnings is made accessible by a number of open source projects. Some of the most popular technologies include, but are not limited to, Deeplearning4j (DL4j), Theano, Torch, TensorFlow, and Caffe. The deciding factors on which one to use are the tech stack they target, and if they are low-level, academic, or application focused. Here’s an overview of each:

DL4J:

- JVM-based
- Distrubted
- Integrates with Hadoop and Spark

Theano:

- Very popular in Academia
- Fairly low level
- Interfaced with via Python and Numpy

Torch:

- Lua based
- In house versions used by Facebook and Twitter
- Contains pretrained models

TensorFlow:

- Google written successor to Theano
- Interfaced with via Python and Numpy
- Highly parallel
- Can be somewhat slow for certain problem sets

Caffe:

- Not general purpose. Focuses on machine-vision problems
- Implemented in C++ and is very fast
- Not easily extensible
- Has a Python interface

> **pip install tensorflow==2.0.0**

> To run from Anaconda Prompt

> **!pip install tensorflow==2.0.0**

> To run from Jupyter Notebook

Both Tensorflow 2.0 and Keras have been released for four years (Keras was released in March 2015, and Tensorflow was released in November of the same year). The rapid development of deep learning in the past days, we also know some problems of Tensorflow1.x and Keras:

- Using Tensorflow means programming static graphs, which is difficult and inconvenient for programs that are familiar with imperative programming
- Tensorflow api is powerful and flexible, but it is more complex, confusing and difficult to use.
- Keras api is productive and easy to use, but lacks flexibility for research

Official docs link for [DETAILED INSTALLATION STEPS (https://www.tensorflow.org/install)](https://www.tensorflow.org/install) for Tensorflow 2

In [ ]:
```
# Verify installation
import tensorflow as tf
```

In [ ]:
```
print(f"Tensorflow Version: {tf.__version__}")
print(f"Keras Version: {tf.keras.__version__}")
```

```
Tensorflow Version: 2.5.0
Keras Version: 2.5.0
```

Tensorflow2.0 is a combination design of Tensorflow1.x and Keras. Considering user feedback and framework development over the past four years, it largely solves the above problems and will become the future machine learning platform.

> Tensorflow 2.0 is built on the following core ideas:

- The coding is more pythonic, so that users can get the results immediately like they are programming in numpy
- Retaining the characteristics of static graphs (for performance, distributed, and production deployment), this makes TensorFlow fast, scalable, and ready for production.
- Using Keras as a high-level API for deep learning, making Tensorflow easy to use and efficient
- Make the entire framework both high-level features (easy to use, efficient, and not flexible) and low-level features (powerful and scalable, not easy to use, but very flexible)

> Eager execution is by default in TensorFlow 2.0 and, it needs no special setup. The following below code can be used to find out whether a CPU or GPU is in use

GPU/CPU Check

In [ ]:
```
tf.config.list_physical_devices('GPU')
```

Out[3]:
```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

In [ ]:
```
tf.config.list_physical_devices('CPU')
```

Out[4]:
```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU')]
```

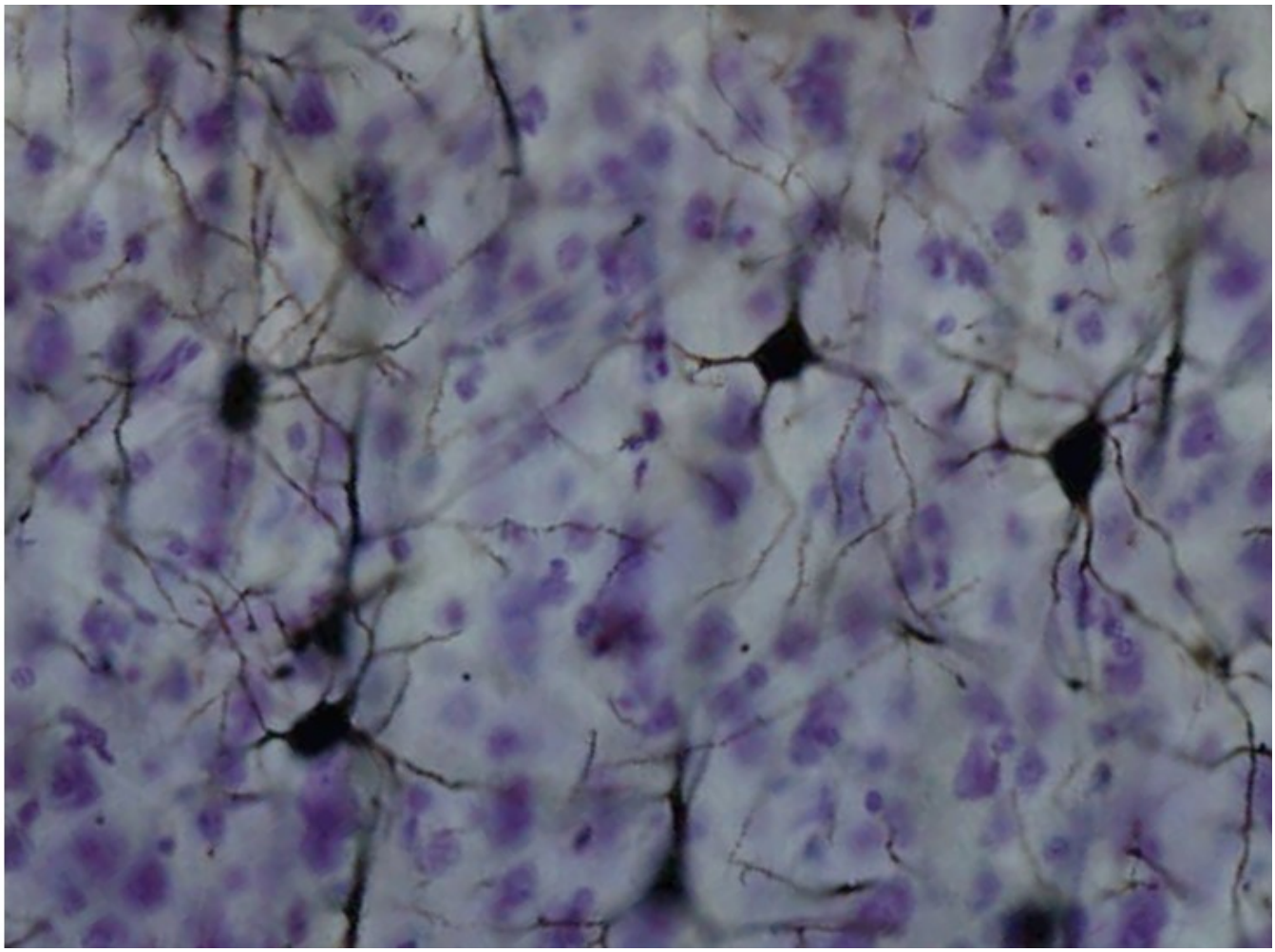
In [ ]:
```
CheckList = ["GPU", "CPU"]
for device in CheckList:
    out_ = tf.config.list_physical_devices(device)
    if len(out_) > 0:
        print(f"{device} is available")
        print("details\n",out_)
    else:
        print(f"{device} not available")
```

```
GPU is available
details
 [PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
CPU is available
details
 [PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU')]
```

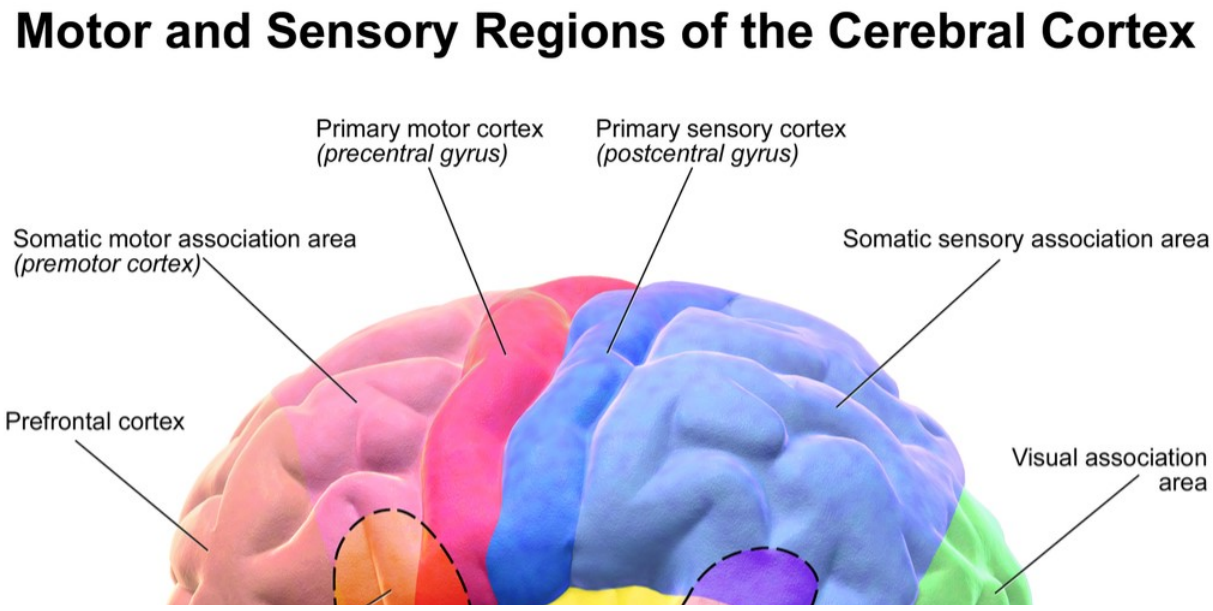
In [ ]:
```

```

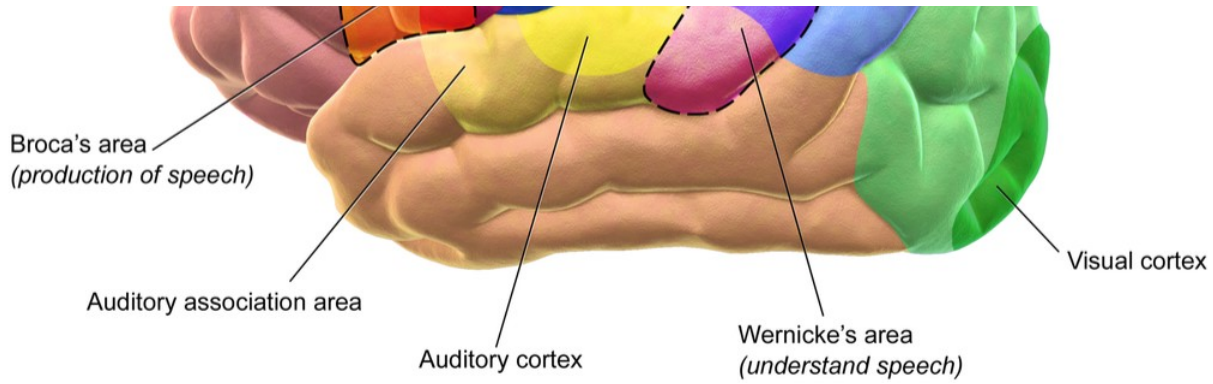
ANN: Artificial Neural Networks

- ANNs are inspired by biological neurons found in cerebral cortex of our brain.
- The cerebral cortex (plural cortices), also known as the cerebral mantle, is the outer layer of neural tissue of the cerebrum of the brain in humans and other mammals.



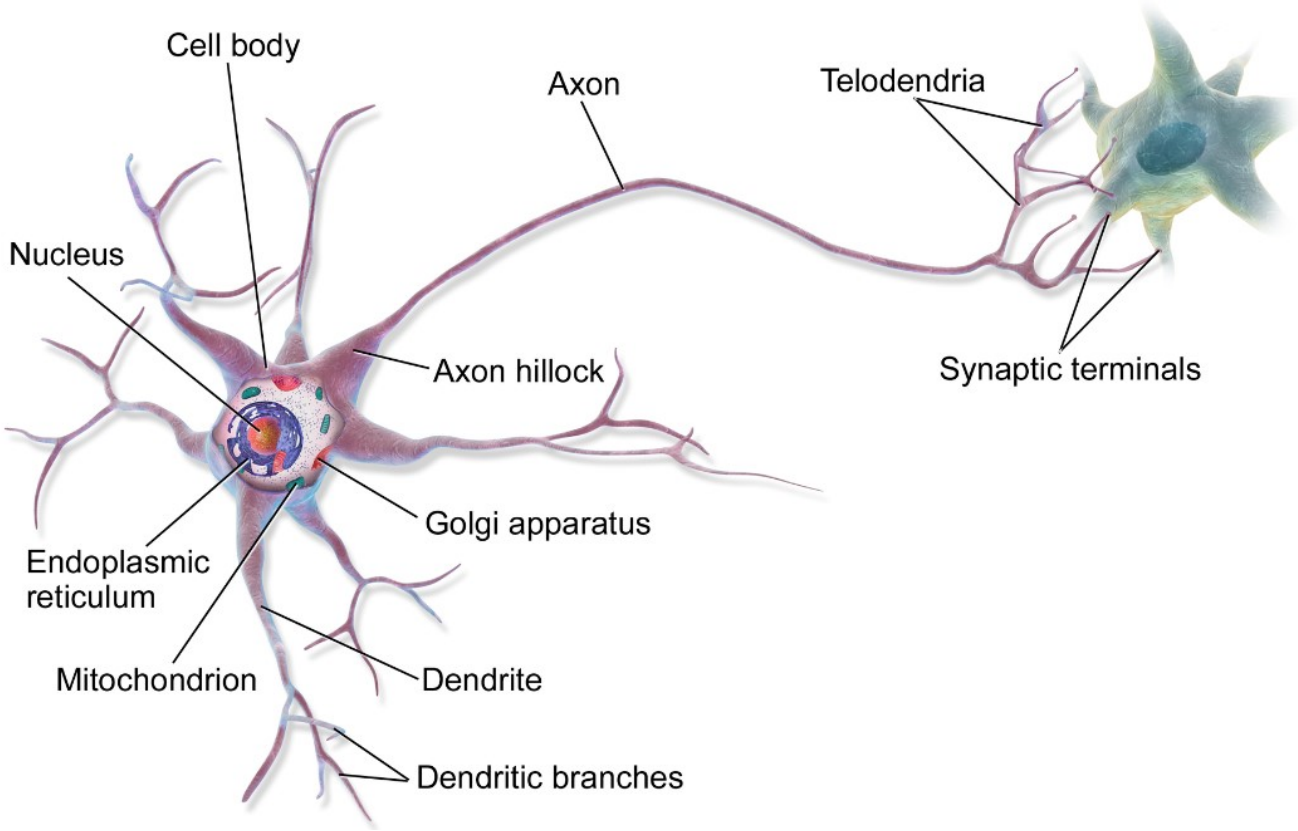
In the above diagram we can neurons of human brains, these neurons resemble the ANN.





The largest and most important part of the human brain is the cerebral cortex. Although it cannot be observed directly, various regions within the cortex are responsible for different functions, as shown in the diagram. The cortex plays a crucial role in important cognitive processes such as memory, attention, perception, thinking, language, and awareness.

Biological Neuron

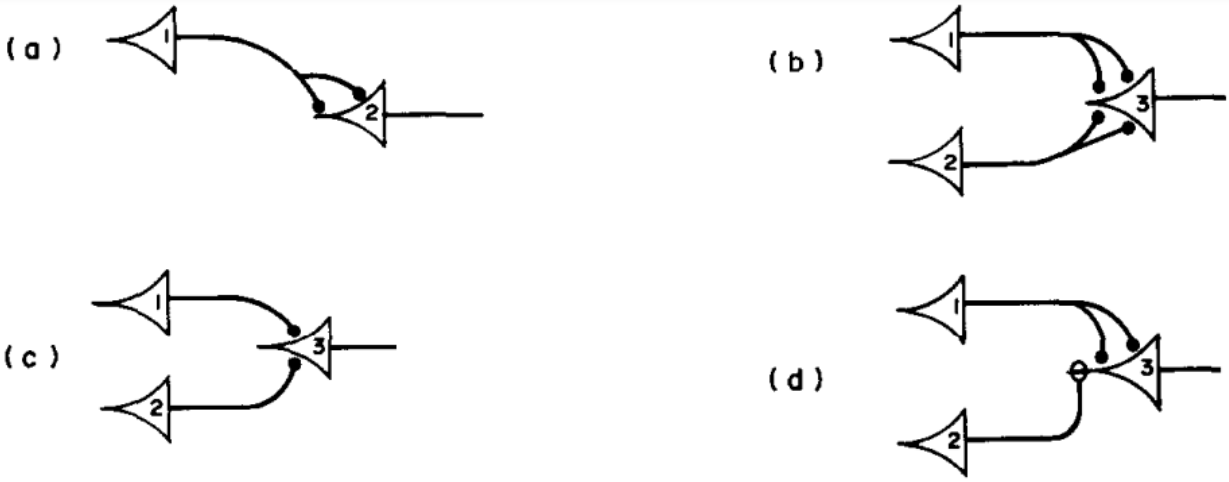


- Biological Neuron produce short electrical impulses known as action potentials which travels through axons to the synapses which releases chemical signals i.e neurotransmitters.
- When a connected neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires (or does not fires, think of a NOT gate here) its own action potential or electrical impulse.
- These simple units form a strong network known as Biological Neural Network (BNN) to perform very complex computation task.
- Similar to the Biological neuron we have artificial neuron which can be used to perform complex computation task.

The first artificial neuron

- It was in year 1943, Artificial neuron was introduced by-
  - Neurophysiologist Warren McCulloh and
  - Mathematician Walter Pitts
- They have published their work in `McCulloch, W.S., Pitts, W. A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics 5, 115–133 (1943). https://doi.org/10.1007/BF02478259` . read full paper at [this link (https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf)](https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf)
- They have shown that these simple neurons can perform small logical operation like OR, NOT, AND gate etc.
- Following figure represents these ANs which can perform (a) Buffer, (b) OR, (c) AND and (d) A-B operation

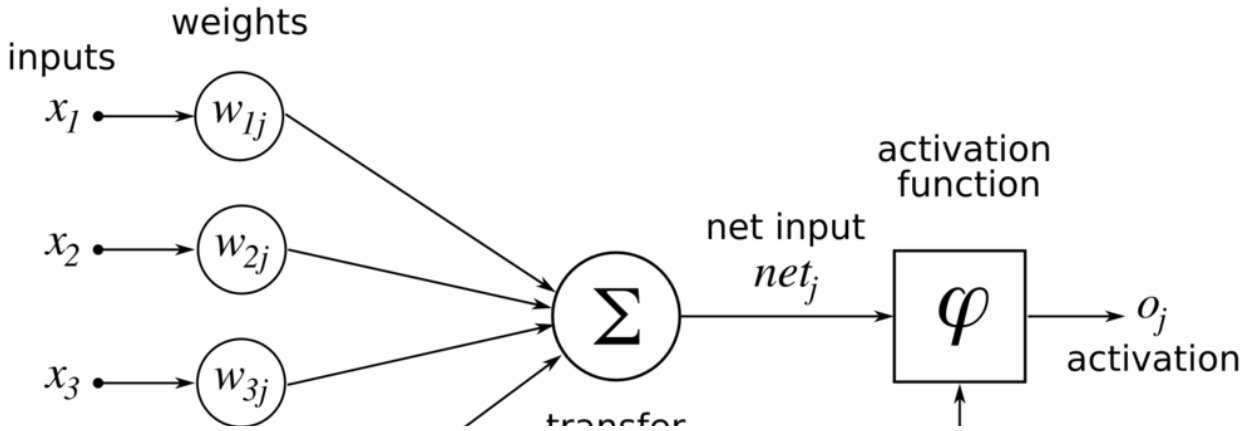
  [image source (https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf)](https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf)



- These neuron only fires when they get two active inputs.

The Perceptron

- Its the simplest ANN architecture. It was invented by Frank Rosenblatt in 1957 and published as `Rosenblatt, Frank (1958), The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Cornell Aeronautical Laboratory, Psychological Review, v65, No. 6, pp. 386–408. doi:10.1037/h0042519`
- It has different architecture then the first neuron that we have seen above. Its known as threshold logic unit(TLU) or linear threshold unit (LTU).
- Here inputs are not just binary.
- Lets see the architecture shown below -

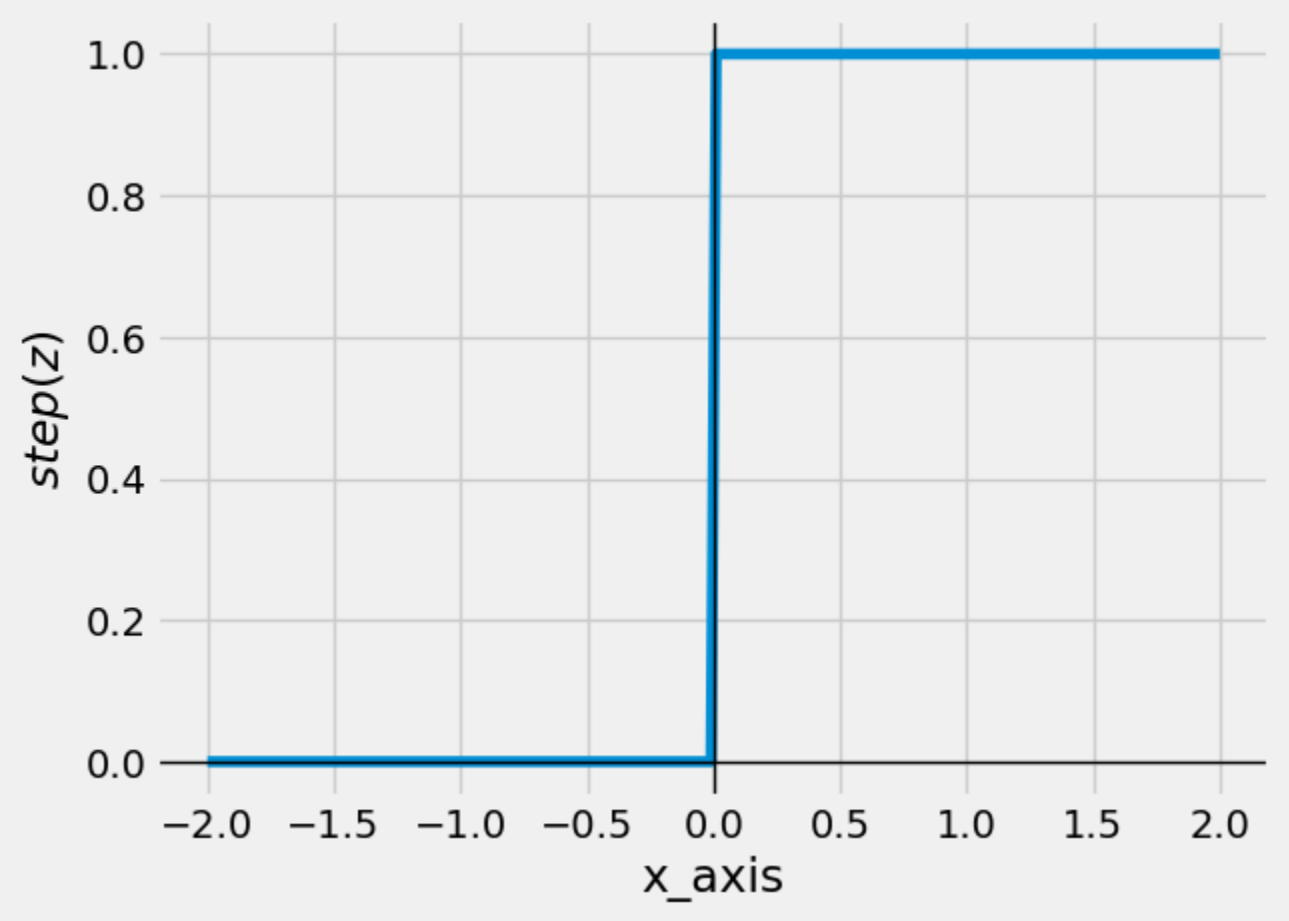


- Common activation functions used for Perceptrons are (with threshold at 0)-

$$step(z) \text{ or } heaviside(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$$

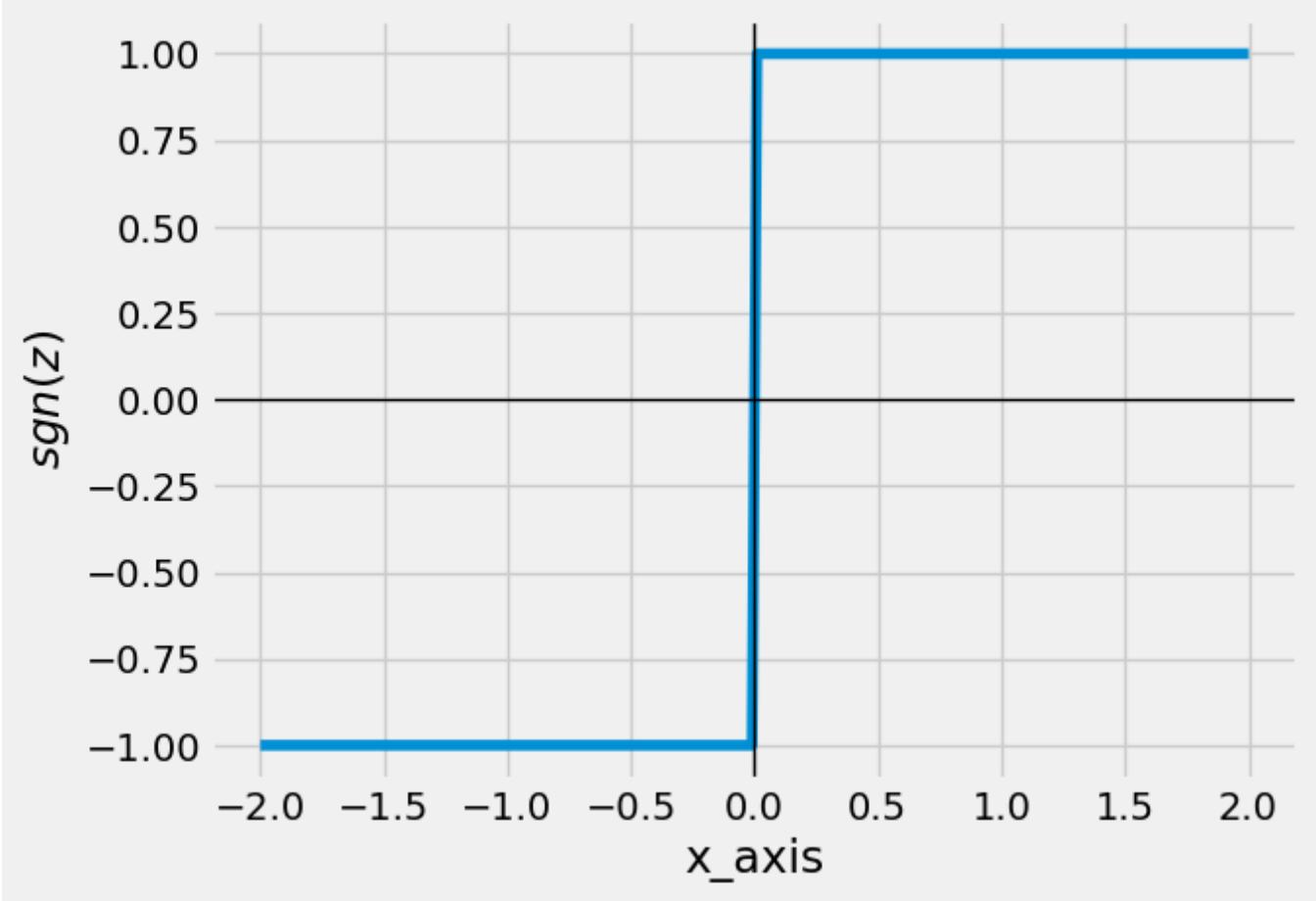
- In the further tutorials we will study more about activation functions, for now we can understand **activation functions** are mathematical equations that are applied to the output of a neural network node, in order to introduce non-linearity into the output. These functions help the neural network to learn and model complex patterns in the data. Activation functions are a key component of artificial neural networks and are used to determine the output of each neuron based on the input it receives. There are many types of activation functions, including **sigmoid, ReLU, tanh, and softmax**, each with their own unique properties and use cases.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use("fivethirtyeight")

x_axis = np.linspace(-2,2,200)
step = np.where(x_axis < 0, 0, 1)

plt.plot(x_axis, step)
plt.xlabel("x_axis")
plt.ylabel(r"$step(z)$")
plt.axhline(0, color='k', lw=1);
plt.axvline(0, color='k', lw=1);
```



- $$sgn(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$$

In [2]:
```
def sgn(x):
    if x < 0:
        return -1
    elif x > 0:
        return 1
    return 0

sgn = np.array(list(map(sgn, x_axis)))

plt.plot(x_axis, sgn)
plt.xlabel("x_axis")
plt.ylabel(r"$sgn(z)$")
plt.axhline(0, color='k', lw=1);
plt.axvline(0, color='k', lw=1);
```



- These activation function are **analogous to firing of neurotransmitter** at a specific threshold
- Perceptron learning Rule:

$$w_{i,j} \leftarrow w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

where,

- $w_{i,j}$: connection weight between i^{th} input neuron and j^{th} output neuron
- x_i : i^{th} input value.
- $\hat{y}_j$: output of j^{th} output neuron
- y_j : target output of j^{th} output neuron
- η : learning rate

- It can also be written as for jth element of w vector

$$w_j = w_j + \triangle w_j$$

- Single TLUs (Threshold Logic Unit) are **simple linear binary classifier** hence not suitable for non linear operation.
- Rosenblatt proved that if the data is **linearly separable** then only this algorithm will converge which is known as **Perceptron learning theorem**

- Some serious weaknesses of Perceptrons was revealed In 1969 by Marvin Minsky and Seymour Papert. Not able to solve some simple logic operations like XOR, EXOR etc.
- But above mentioned problem were solved by implementing multiplayer perceptron.

Derivation:-

Let's assume that you are doing a binary classification with class +1 and -1

Let there be decision function $\phi(z)$

which takes linear combination of certain inputs "$\mathbf{x}$"

corresponding weights "$\mathbf{w}$" and net input $\mathbf{z} = w_1x_1 + w_2x_2 + \ldots + w_nx_n$

So in vector form we have

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

so, $\mathbf{z} = \mathbf{w^T x}$

Now, if

for a sample $\mathbf{x}$

$$\phi(z) = \begin{cases} +1 & \text{if } z \geq \theta \\ -1 & \text{if } z < \theta \end{cases}$$

Lets simplify the above equation -

$$\phi(z) = \begin{cases} +1 & \text{if } z - \theta \geq 0 \\ -1 & \text{if } z - \theta < 0 \end{cases}$$

Suppose $w_0 = -\theta$ and $x_0 = 1$

Then,

$$\mathbf{z'} = w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n$$

and

$$\phi(z) = \begin{cases} +1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$$

here $w_0 x_0$ is usually known as bias unit

Implementation of Perceptron with Python

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import joblib
import pandas as pd
plt.style.use("fivethirtyeight")
```

In [2]:
```
class Perceptron:
  def __init__(self, eta, epochs):
    self.weights = np.random.randn(3) * 1e-4
    print(f"self.weights: {self.weights}")
    self.eta = eta
    self.epochs = epochs

  def activationFunction(self, inputs, weights):
    z = np.dot(inputs, weights)
    return np.where(z > 0 , 1, 0)

  def fit(self, X, y):
    self.X = X
    self.y = y

    X_with_bias = np.c_[self.X, -np.ones((len(self.X), 1))] # concactination
    print(f"X_with_bias: \n{X_with_bias}")

    for epoch in range(self.epochs):
      print(f"for epoch: {epoch}")
      y_hat = self.activationFunction(X_with_bias, self.weights)
      print(f"predicted value: \n{y_hat}")
      error = self.y - y_hat
      print(f"error: \n{error}")
      self.weights = self.weights + self.eta * np.dot(X_with_bias.T, error)
      print(f"updated weights: \n{self.weights}")
      print("#############\n")

  def predict(self, X):
    X_with_bias = np.c_[X, -np.ones((len(self.X), 1))]
    return self.activationFunction(X_with_bias, self.weights)
```

AND Operation:

In [3]:
```
data = {"x1": [0,0,1,1], "x2": [0,1,0,1], "y": [0,0,0,1]}

AND = pd.DataFrame(data)
AND
```

Out[3]:

	x1	x2	y
0	0	0	0
1	0	1	0
2	1	0	0
3	1	1	1

In [4]:
```
X = AND.drop("y", axis=1)
X
```

Out[4]:

	x1	x2
0	0	0
1	0	1
2	1	0
3	1	1

In [5]:
```
y = AND['y']
y.to_frame()
```

Out[5]:

	y
0	0
1	0
2	0
3	1

In [6]:
```
model = Perceptron(eta = 0.5, epochs=10)
model.fit(X,y)
```

```
self.weights: [ 4.49529150e-05  1.85654482e-04 -3.14676580e-05]
X_with_bias:
[[ 0.  0. -1.]
 [ 0.  1. -1.]
 [ 1.  0. -1.]
 [ 1.  1. -1.]]
for epoch: 0
predicted value:
[1 1 1 1]
error:
0   -1
1   -1
2   -1
3    0
Name: y, dtype: int64
updated weights:
[-0.49995505 -0.49981435  1.49996853]
##############
```

In [7]:
```
model.predict(X)
```

Out[7]:
```
array([0, 0, 0, 1])
```

In [8]:
```
model.weights
```

Out[8]:
```
array([0.50004495, 0.50018565, 0.99996853])
```

Saving and Loading model

In [9]:
```
import os

# saving model
dir_ = "Perceptron_model"
os.makedirs(dir_, exist_ok=True)
filename = os.path.join(dir_, 'AND_model.model')
joblib.dump(model, filename)
```

Out[9]:
```
['Perceptron_model\\AND_model.model']
```

In [10]:
```
# load the model from drive
loaded_model = joblib.load(filename)
result = loaded_model.predict(X)
print(result)
```

```
[0 0 0 1]
```

OR Operation:

In [12]:
```
data = {"x1": [0,0,1,1], "x2": [0,1,0,1], "y": [0,1,1,1]}

OR = pd.DataFrame(data)
OR
```

Out[12]:

	x1	x2	y
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	1

In [13]:
```
X = OR.drop("y", axis=1)
X
```

Out[13]:

	x1	x2
0	0	0
1	0	1
2	1	0
3	1	1

In [14]:
```
y = OR['y']
y.to_frame()
```

Out[14]:

	y
0	0
1	1
2	1
3	1

In [15]:
```
model = Perceptron(eta = 0.5, epochs=10)
model.fit(X,y)
```

```
self.weights: [-4.24035454e-05  1.13286067e-04  4.02537022e-05]
X_with_bias:
[[ 0.  0. -1.]
 [ 0.  1. -1.]
 [ 1.  0. -1.]
 [ 1.  1. -1.]]
for epoch: 0
predicted value:
[0 1 0 1]
error:
0    0
1    0
2    1
3    0
Name: y, dtype: int64
updated weights:
[ 4.99957596e-01  1.13286067e-04 -4.99959746e-01]
##############
```

XOR Operation:

In [16]:
```
data = {"x1": [0,0,1,1], "x2": [0,1,0,1], "y": [0,1,1,0]}

XOR = pd.DataFrame(data)
XOR
```

Out[16]:

	x1	x2	y
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

In [17]:
```
X = XOR.drop("y", axis=1) # axis = 1 >>> dropping across column
X
```

Out[17]:

	x1	x2
0	0	0
1	0	1
2	1	0
3	1	1

In [18]:
```
y = XOR['y']
y.to_frame()
```

Out[18]:

	y
0	0
1	1
2	1
3	0

In [19]:
```
model = Perceptron(eta = 0.5, epochs=50)
model.fit(X,y)
```

```
self.weights: [-1.51414242e-04  1.54706413e-04  4.27810246e-05]
X_with_bias:
[[ 0.  0. -1.]
 [ 0.  1. -1.]
 [ 1.  0. -1.]
 [ 1.  1. -1.]]
for epoch: 0
predicted value:
[0 1 0 0]
error:
0    0
1    0
2    1
3    0
Name: y, dtype: int64
updated weights:
[ 4.99848586e-01  1.54706413e-04 -4.99957219e-01]
##############
```

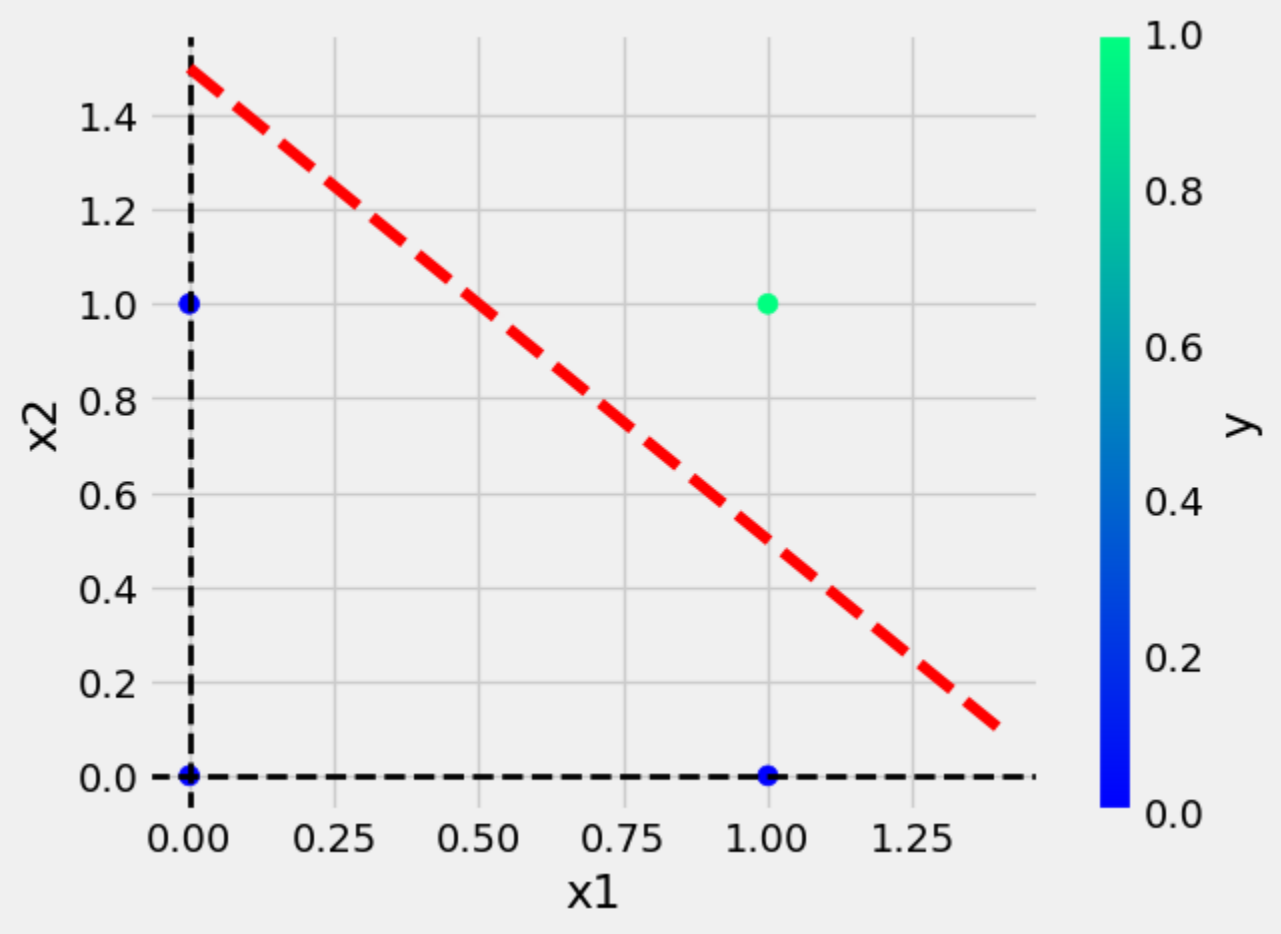
Conclusion:

Here we can see Perceptron can only classify the linear problem like AND, OR operation because they were linear problem. But in the case of XOR it couldn't classify correctly because it was a non-linear problem. Lets see graphically.

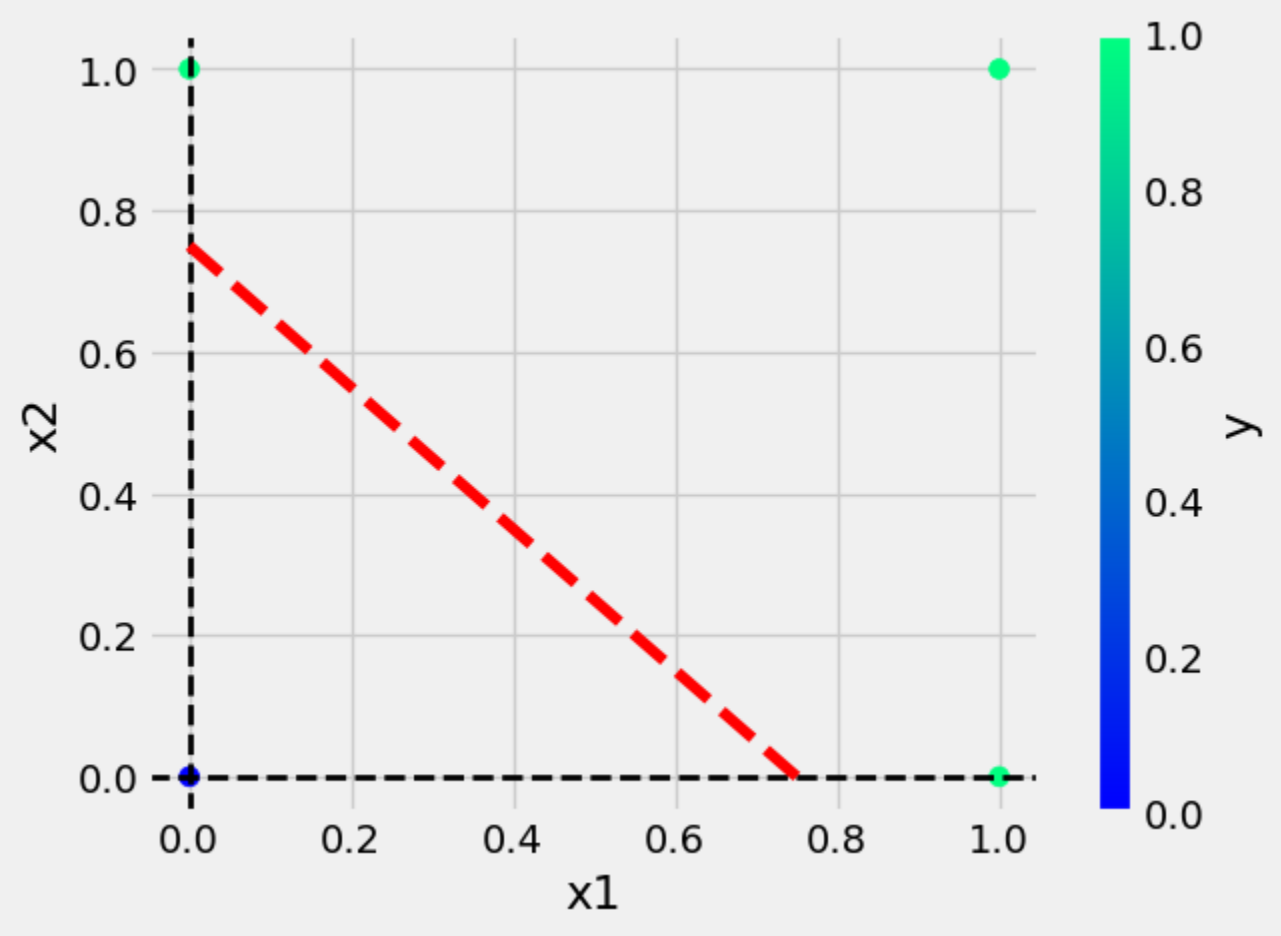
Analysis with the graph

In [20]:
```
AND.plot(kind="scatter", x="x1", y="x2", c="y", s=50, cmap="winter")
plt.axhline(y=0, color="black", linestyle="--", linewidth=2)
plt.axvline(x=0, color="black", linestyle="--", linewidth=2)

x = np.linspace(0, 1.4) # >>> 50
y = 1.5 - 1*np.linspace(0, 1.4) # >>> 50

plt.plot(x, y, "r--")
```

Out[20]:
```
[<matplotlib.lines.Line2D at 0x22459924790>]
```

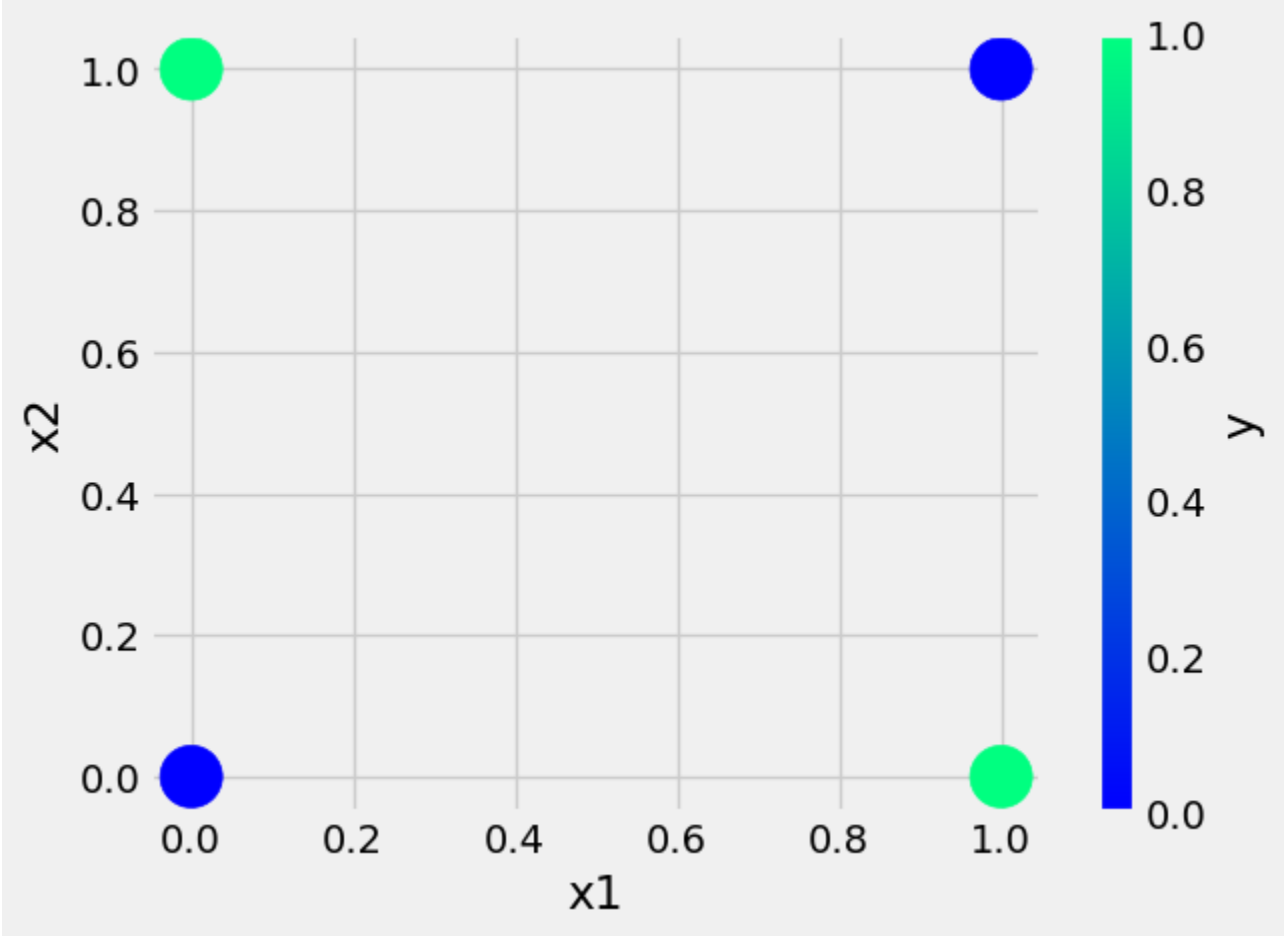


In [21]:
```
OR.plot(kind="scatter", x="x1", y="x2", c="y", s=50, cmap="winter")
plt.axhline(y = 0, color ="black", linestyle ="--", linewidth=2)
plt.axvline(x = 0, color ="black", linestyle ="--", linewidth=2)
plt.plot(np.linspace(0,0.75), 0.75 - 1*np.linspace(0,0.75), 'r--');
```



In [22]:
```
XOR.plot(kind="scatter", x="x1", y="x2", c="y", s=500, cmap="winter")
```

Out[22]:
```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```



What is learning then?

forward pass + backward pass, we will learn more about this later

When does the prediction happen?

The prediction happens during the forward pass.

Drawbacks of Perceptron:-

- It cannot be used if the data is non linear.

Solution to this problem:-

- This can be solved by stacking the layers of perceptron, Also called as Multilayer Perceptron (MLP).

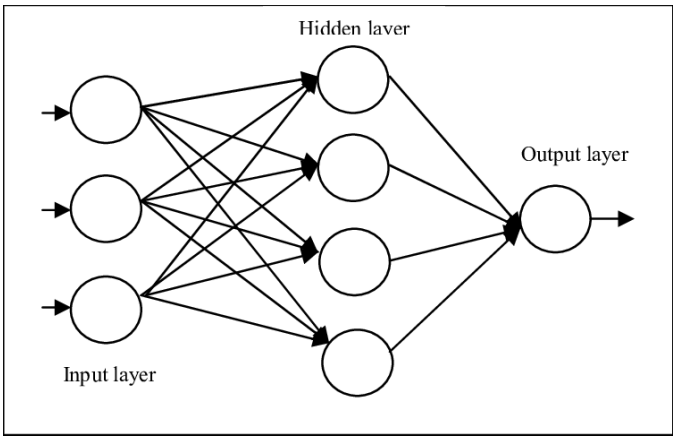
Deep Learning With Computer Vision And Advanced NLP (DL_CV_NLP)

Basic Understanding of ANN (Artificial Neural Network)

The solution to fitting more complex (*i.e.* non-linear) models with neural networks is to use a more complex network that consists of more than just a single perceptron. The take-home message from the perceptron is that all of the learning happens by adapting the synapse weights until prediction is satisfactory. Hence, a reasonable guess at how to make a perceptron more complex is to simply **add more weights**.

There are two ways to add complexity:

1. Add backward connections, so that output neurons feed back to input nodes, resulting in a **recurrent network**
2. Add neurons between the input nodes and the outputs, creating an additional ("hidden") layer to the network, resulting in a **multi-layer perceptron**



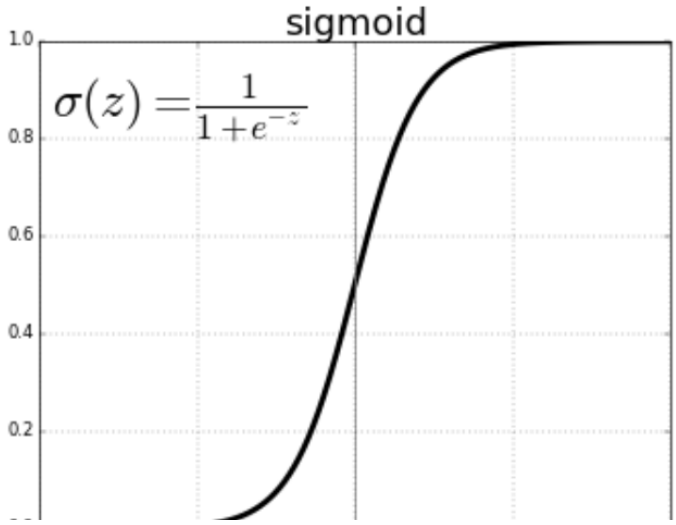
How to train a multilayer network is not intuitive. Propagating the inputs forward over two layers is straightforward, since the outputs from the hidden layer can be used as inputs for the output layer. However, the process for updating the weights based on the prediction error is less clear, since it is difficult to know whether to change the weights on the input layer or on the hidden layer in order to improve the prediction.

Updating a multi-layer perceptron (MLP) is a matter of:

1. moving forward through the network, calculating outputs given inputs and current weight estimates
2. moving backward updating weights according to the resulting error from forward propagation(using backpropagation method).

In this sense, it is similar to a single-layer perceptron, except it has to be done twice, once for each layer.

##Activation Function of ANN: In ANN we use sigmoid as an activation function in each layer instead of step function.Because ANN can solve non-linear problem so the output can be varied. Sigmoid outputs numbers 0 to 1. On the other hand step function outputs just 0 or 1.



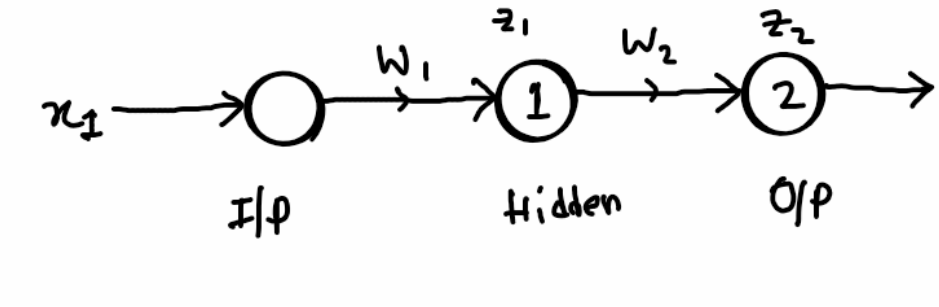
- Formula of Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- It can take number between $-\infty, +\infty$ and gives output 0 to 1.

Need of Activation Function:

- An activation function added into an artificial neural network in order to help the network learn complex patterns in the data.
- It also scales the data
- It filters out the important portion of data
- Without activation function Deep stack of network will behave like a single linear transformation.
  - **Example:** without activation function



$z_1 = x_1 . w_1, z_2 = w_2 . z_1$

$$z_2 = w_2 . z_1$$
$$z_2 = w_2 . x_1 w_1$$
$$z_2 = W x_1$$

  - So, you can see it has been a single neuron.Behave like a single linear transformation.
- Without activation function all the continuous function cannot be approximated.

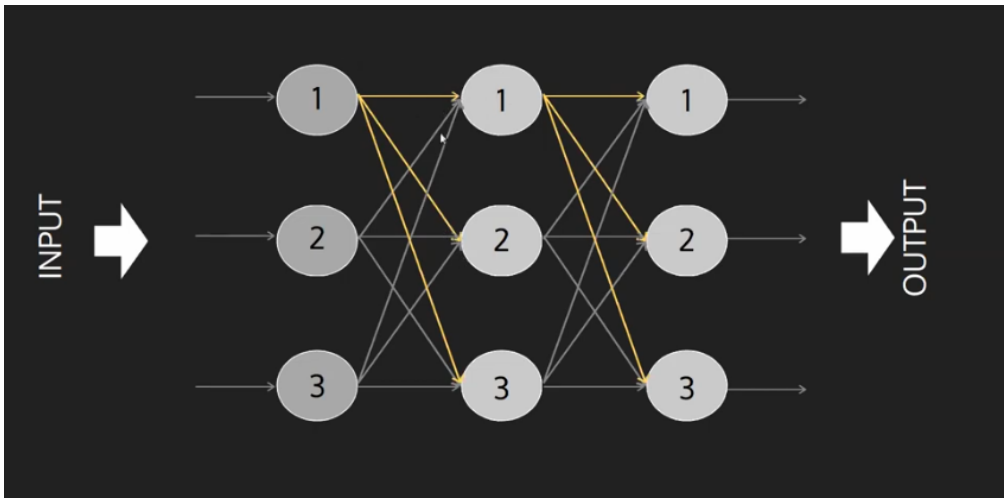
Weight update of ANN (Backpropagation intiution):

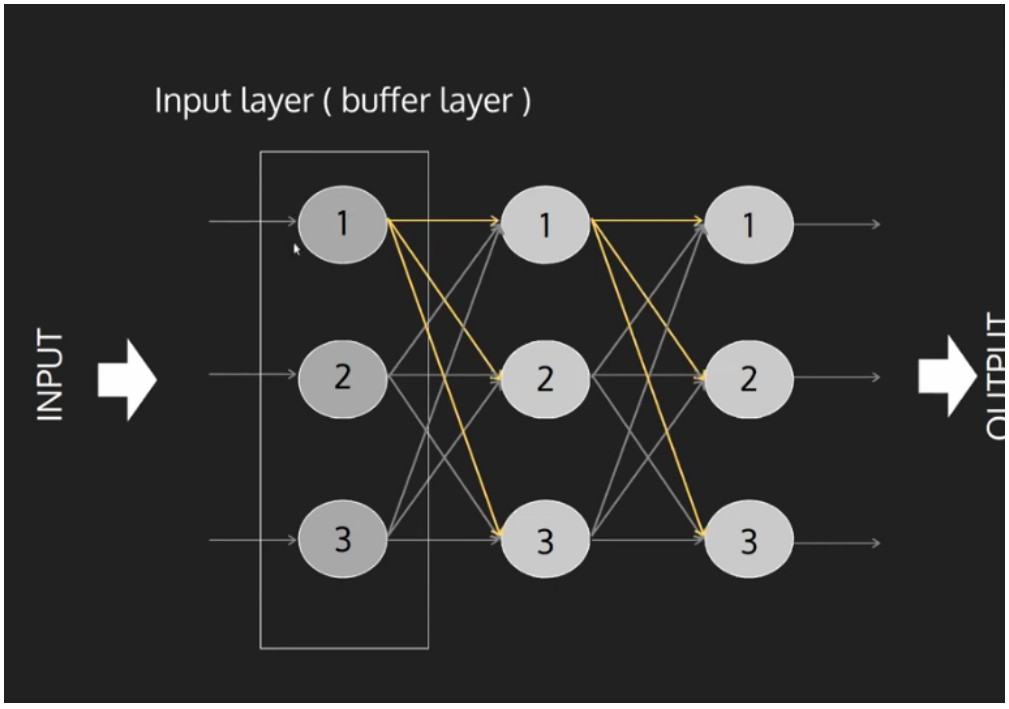
- In the year 1986 a groundbreaking paper "Learning Internal Representation by Error Propagation" was published by -

    - David Rumelhart,
    - Geoffrey Hinton, &
    - Ronald Williams
- It depicted an efficient way to update weights and biases of the network based on the error/loss function by passing twice through the network i.e forward and backward pass.
    - forward pass: data is passed through the input layer to the hidden layer and it calculates ouput. Its nothing but making prediction.
    - error calculation: Based on loss function error is calculated to check how much deviation is there from the ground truth or actual value and predicted value.
    - error contribution from the each connection of the output layer is calculated.
    - Then algo goes a layer deep and calculates how much previous layer contributed into the error of present layer and this way it propagates till the input layer.
    - This reverse pass measures the error gradient accross all the connection.
    - At last by using these error gradients a gradient step is performed to update the weights.
- In MLP key changes were to introduce a sigmoid activation function

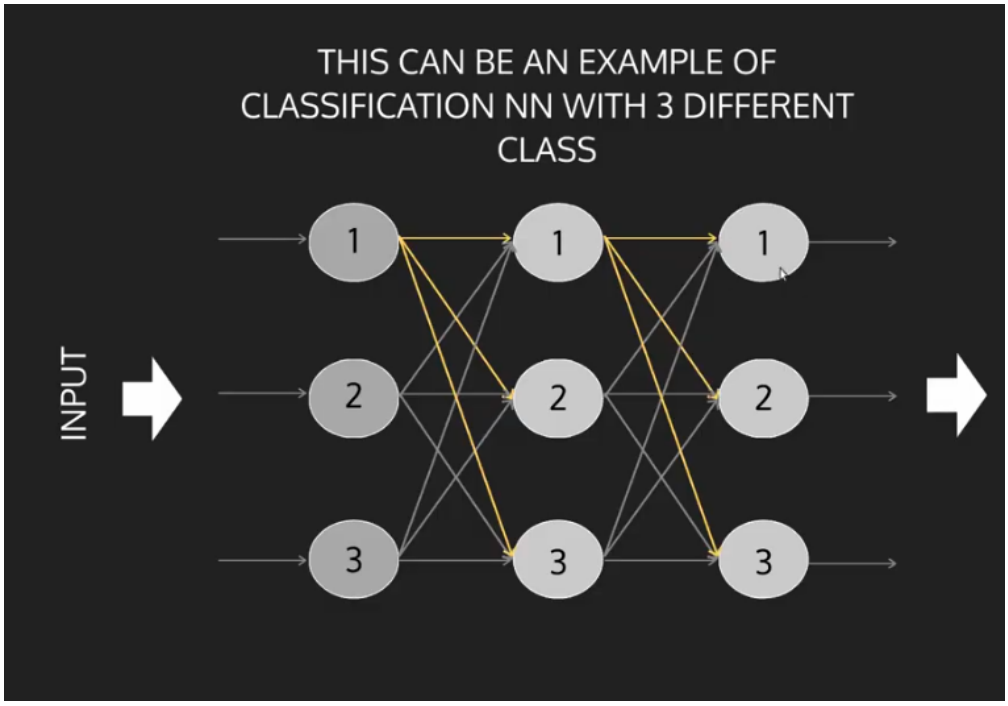
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

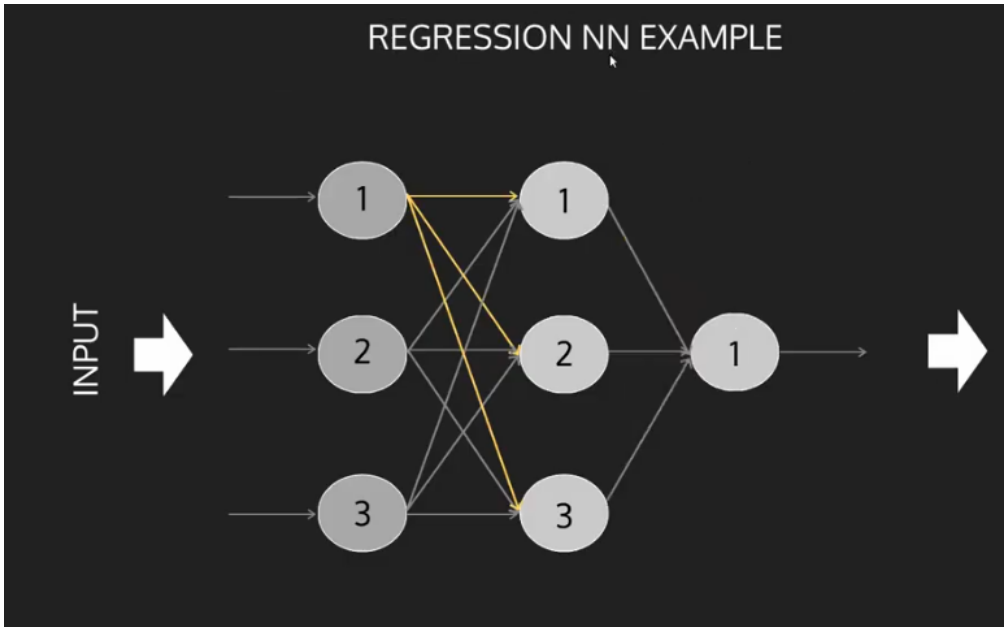
#Now lets get some intuition of ANN.





At the first layer is Input or Buffer layer. Second layer called hidden layer & the 3rd layer called output layer.

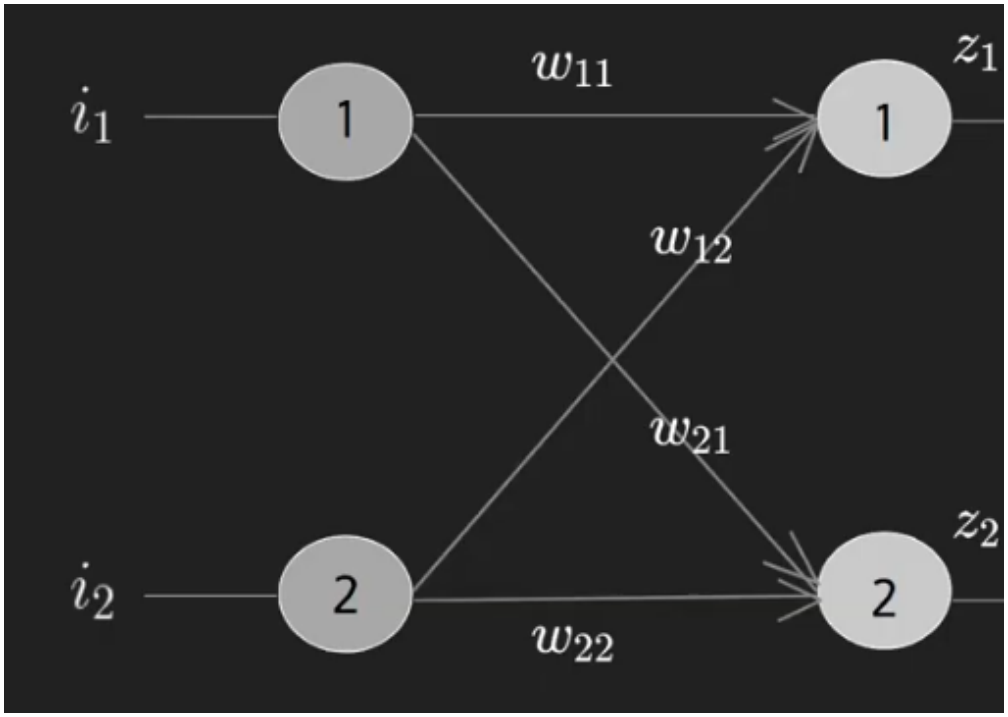




In the classification outputs neuron can be multiple but in the case of Regression output neuron might be one.

Simple Example:

Lets take a simple neuron network , Here consider bias = 0



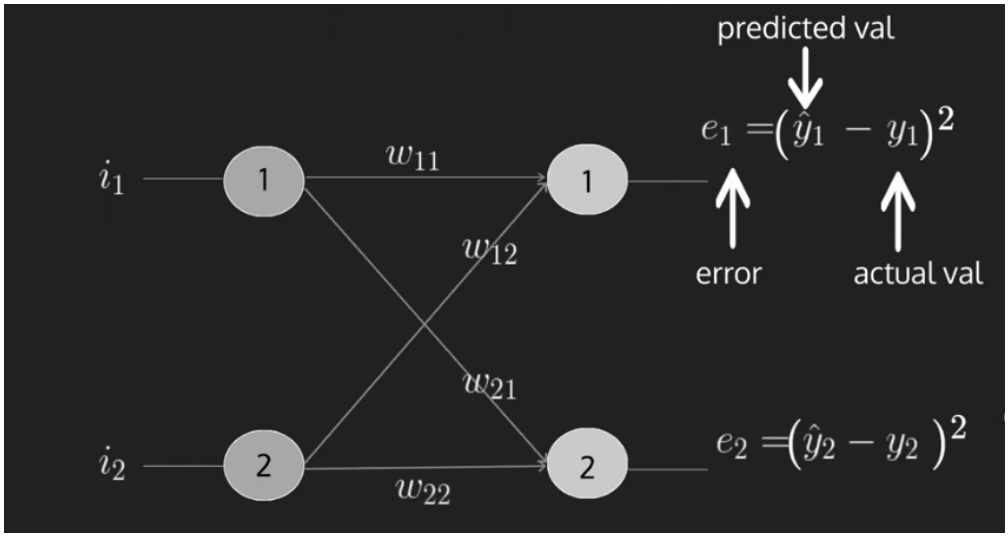
- So,

$$z_1 = (w_{11}.i_1) + (w_{12}.i_2)$$
$$\therefore \hat{y_1} = act(z_1)$$
$$z_2 = (w_{21}.i_1) + (w_{22}.i_2)$$
$$\therefore \hat{y_2} = act(z_2)$$

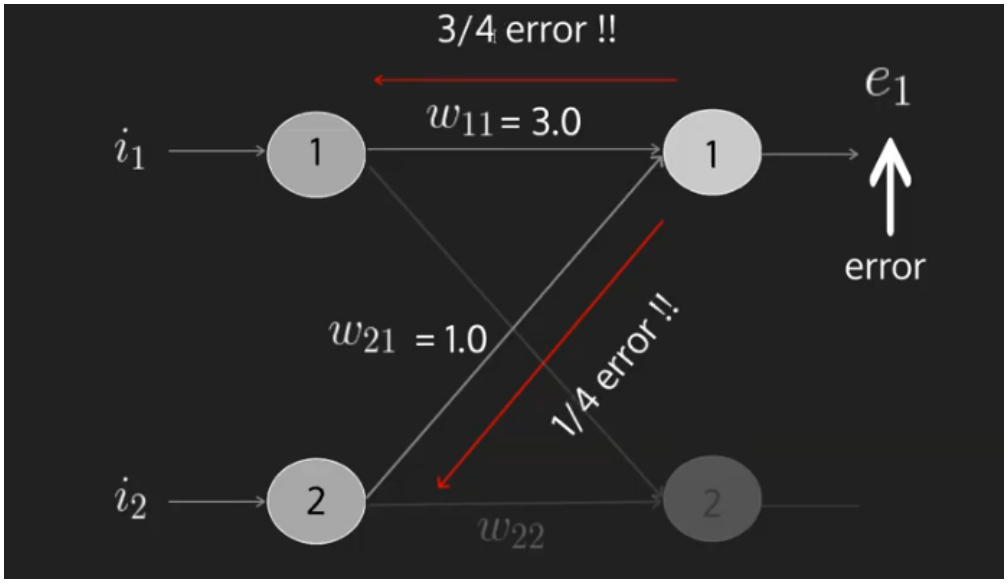
Let's define it as a metrix form,

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} * \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} (w_{11}.i_1) + (w_{12}.i_2) \\ (w_{21}.i_1) + (w_{22}.i_2) \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

Error Calculation:



Now weight will be updated based on the proportional error!



In order to do that weight update rule (Backpropagration) can be used, that will be discuss further.

Difference between Perceptron & Neural Network:

Perceptron:

- Perceptron is a single layer neural network, It can be also multi layer.
- Perceptron is a linear classifier (binary).
- It cann't solve non-linear problem.
- Perceptron use step function as an activation function.

###Neural Network:

- A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates.
- A neuron network is consisted by single layer or multiple layer
- It can be very depth
- It can solve non-linear problems.
- It can have many hidden layers.
- It use sigmoid, ReLu, softmax etc. as an activation function.

In [ ]:

Basic Introduction ¶

LeNet-5, from the paper Gradient-Based Learning Applied to Document Recognition, is a very efficient convolutional neural network for handwritten character recognition.

Paper: Gradient-Based Learning Applied to Document Recognition (http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf)

Authors: Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

Published in: Proceedings of the IEEE (1998)

Structure of the LeNet network

LeNet5 is a small network, it contains the basic modules of deep learning: convolutional layer, pooling layer, and full link layer. It is the basis of other deep learning models. Here we analyze LeNet5 in depth. At the same time, through example analysis, deepen the understanding of the convolutional layer and pooling layer.

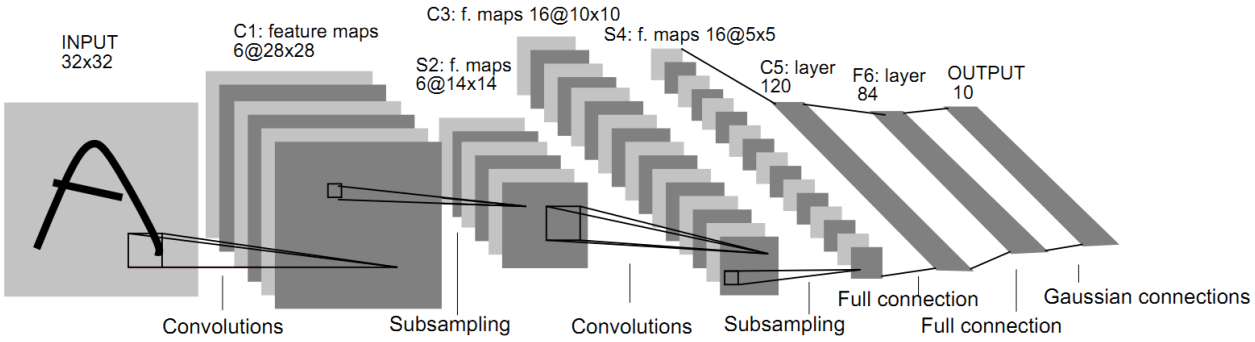


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

LeNet-5 Total seven layer , does not comprise an input, each containing a trainable parameters; each layer has a plurality of the Map the Feature , a characteristic of each of the input FeatureMap extracted by means of a convolution filter, and then each FeatureMap There are multiple neurons.

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	32x32	-	-	-
1	Convolution	6	28x28	5x5	1	tanh
2	Average Pooling	6	14x14	2x2	2	tanh
3	Convolution	16	10x10	5x5	1	tanh
4	Average Pooling	16	5x5	2x2	2	tanh
5	Convolution	120	1x1	5x5	1	tanh
6	FC	-	84	-	-	tanh
Output	FC	-	10	-	-	softmax

Detailed explanation of each layer parameter:

INPUT Layer

The first is the data INPUT layer. The size of the input image is uniformly normalized to 32 * 32.

> Note: This layer does not count as the network structure of LeNet-5. Traditionally, the input layer is not considered as one of the network hierarchy.

C1 layer-convolutional layer

> **Input picture**: 32 * 32

> **Convolution kernel size**: 5 * 5

> **Convolution kernel types**: 6

> **Output featuremap size**: 28 * 28 (32-5 + 1) = 28

> **Number of neurons**: 28 * 28 * 6

> **Trainable parameters**: (5 * 5 + 1) * 6 (5 * 5 = 25 unit parameters and one bias parameter per filter, a total of 6 filters)

> **Number of connections**: (5 * 5 + 1) * 6 * 28 * 28 = 122304

Detailed description:

1. The first convolution operation is performed on the input image (using 6 convolution kernels of size 5 * 5) to obtain 6 C1 feature maps (6 feature maps of size 28 * 28, 32-5 + 1 = 28).
2. Let's take a look at how many parameters are needed. The size of the convolution kernel is 5 * 5, and there are 6 * (5 * 5 + 1) = 156 parameters in total, where +1 indicates that a kernel has a bias.
3. For the convolutional layer C1, each pixel in C1 is connected to 5 * 5 pixels and 1 bias in the input image, so there are 156 * 28 * 28 = 122304 connections in total. There are 122,304 connections, but we only need to learn 156 parameters, mainly through weight sharing.

S2 layer-pooling layer (downsampling layer)

> **Input**: 28 * 28

> **Sampling area**: 2 * 2

> **Sampling method**: 4 inputs are added, multiplied by a trainable parameter, plus a trainable offset. Results via sigmoid

> **Sampling type**: 6

> **Output featureMap size**: 14 * 14 (28/2)

> **Number of neurons**: 14 * 14 * 6

> **Trainable parameters**: 2 * 6 (the weight of the sum + the offset)

> **Number of connections**: (2 * 2 + 1) * 6 * 14 * 14

> The size of each feature map in S2 is 1/4 of the size of the feature map in C1.

Detailed description:

The pooling operation is followed immediately after the first convolution. Pooling is performed using 2 * 2 kernels, and S2, 6 feature maps of 14 * 14 (28/2 = 14) are obtained.

The pooling layer of S2 is the sum of the pixels in the 2 * 2 area in C1 multiplied by a weight coefficient plus an offset, and then the result is mapped again.

So each pooling core has two training parameters, so there are 2x6 = 12 training parameters, but there are 5x14x14x6 = 5880 connections.

C3 layer-convolutional layer

> **Input**: all 6 or several feature map combinations in S2

> **Convolution kernel size**: 5 * 5

> **Convolution kernel type**: 16

> **Output featureMap size**: 10 * 10 (14-5 + 1) = 10

> Each feature map in C3 is connected to all 6 or several feature maps in S2, indicating that the feature map of this layer is a different combination of the feature maps extracted from the previous layer.

One way is that the first 6 feature maps of C3 take 3 adjacent feature map subsets in S2 as input. The next 6 feature maps take 4 subsets of neighboring feature maps in S2 as input. The next three take the non-adjacent 4 feature map subsets as input. The last one takes all the feature maps in S2 as input.

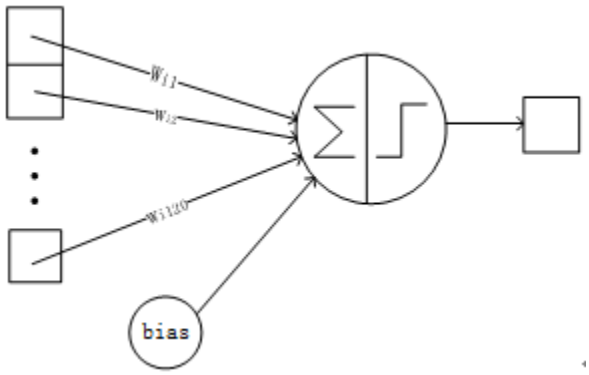
The trainable parameters are: 6 * (3 * 5 * 5 + 1) + 6 * (4 * 5 * 5 + 1) + 3 * (4 * 5 * 5 + 1) + 1 * (6 * 5 * 5 +1) = 1516

Number of connections: 10 * 10 * 1516 = 151600

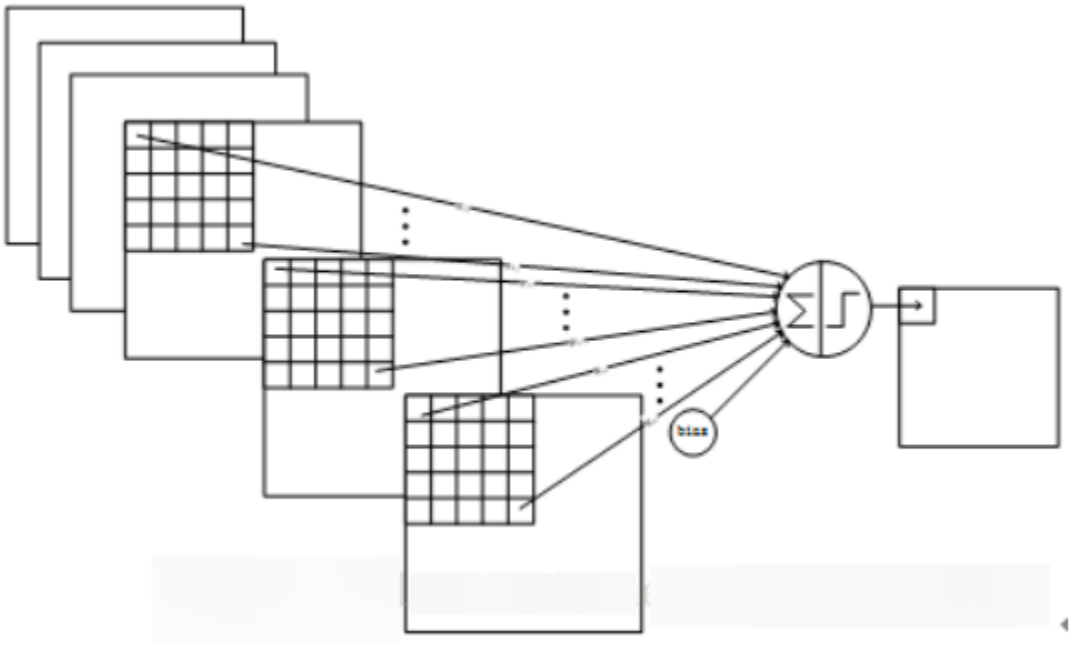
Detailed description:

After the first pooling, the second convolution, the output of the second convolution is C3, 16 10x10 feature maps, and the size of the convolution kernel is 5 * 5. We know that S2 has 6 14 * 14 feature maps, how to get 16 feature maps from 6 feature maps? Here are the 16 feature maps calculated by the special combination of the feature maps of S2. details as follows:

The first 6 feature maps of C3 (corresponding to the 6th column of the first red box in the figure above) are connected to the 3 feature maps connected to the S2 layer (the first red box in the above figure), and the next 6 feature maps are connected to the S2 layer The 4 feature maps are connected (the second red box in the figure above), the next 3 feature maps are connected with the 4 feature maps that are not connected at the S2 layer, and the last is connected with all the feature maps at the S2 layer. The convolution kernel size is still 5 * 5, so there are 6 * (3 * 5 * 5 + 1) + 6 * (4 * 5 * 5 + 1) + 3 * (4 * 5 * 5 + 1) +1 * (6 * 5 * 5 + 1) = 1516 parameters. The image size is 10 * 10, so there are 151600 connections.



The convolution structure of C3 and the first 3 graphs in S2 is shown below:



S4 layer-pooling layer (downsampling layer)

> **Input**: 10 * 10

> **Sampling area**: 2 * 2

> **Sampling method**: 4 inputs are added, multiplied by a trainable parameter, plus a trainable offset. Results via sigmoid

> **Sampling type**: 16

> **Output featureMap size**: 5 * 5 (10/2)

> **Number of neurons**: 5 * 5 * 16 = 400

> **Trainable parameters**: 2 * 16 = 32 (the weight of the sum + the offset)

> **Number of connections**: 16 * (2 * 2 + 1) * 5 * 5 = 2000

> The size of each feature map in S4 is 1/4 of the size of the feature map in C3

Detailed description:

S4 is the pooling layer, the window size is still 2 * 2, a total of 16 feature maps, and the 16 10x10 maps of the C3 layer are pooled in units of 2x2 to obtain 16 5x5 feature maps. This layer has a total of 32 training parameters of 2x16, 5x5x5x16 = 2000 connections.

The connection is similar to the S2 layer.

C5 layer-convolution layer

> **Input**: All 16 unit feature maps of the S4 layer (all connected to s4)

> **Convolution kernel size**: 5 * 5

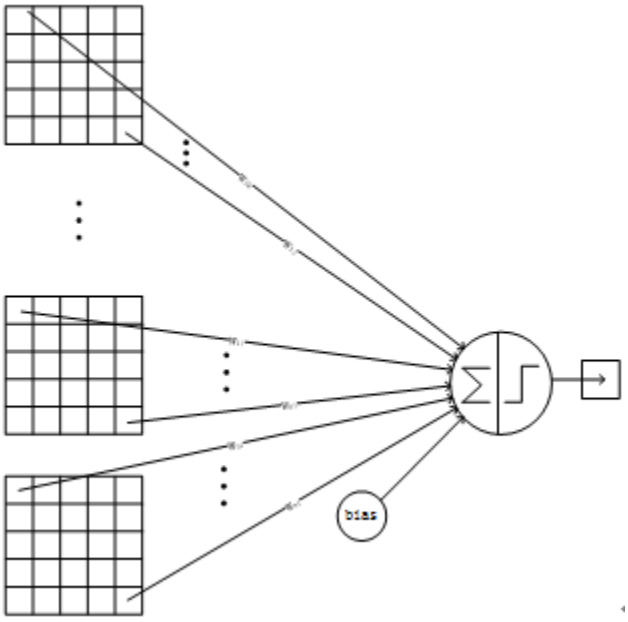
> **Convolution kernel type**: 120

> **Output featureMap size**: 1 * 1 (5-5 + 1)

> **Trainable parameters / connection**: 120 * (16 * 5 * 5 + 1) = 48120

Detailed description:

The C5 layer is a convolutional layer. Since the size of the 16 images of the S4 layer is 5x5, which is the same as the size of the convolution kernel, the size of the image formed after convolution is 1x1. This results in 120 convolution results. Each is connected to the 16 maps on the previous level. So there are (5x5x16 + 1) x120 = 48120 parameters, and there are also 48120 connections. The network structure of the C5 layer is as follows:



F6 layer-fully connected layer

> **Input**: c5 120-dimensional vector

> **Calculation method**: calculate the dot product between the input vector and the weight vector, plus an offset, and the result is output through the sigmoid function.

Trainable parameters: 84 * (120 + 1) = 10164

Detailed description:

Layer 6 is a fully connected layer. The F6 layer has 84 nodes, corresponding to a 7x12 bitmap, -1 means white, 1 means black, so the black and white of the bitmap of each symbol corresponds to a code. The training parameters and number of connections for this layer are (120 + 1) x84 = 10164. The ASCII encoding diagram is as follows:

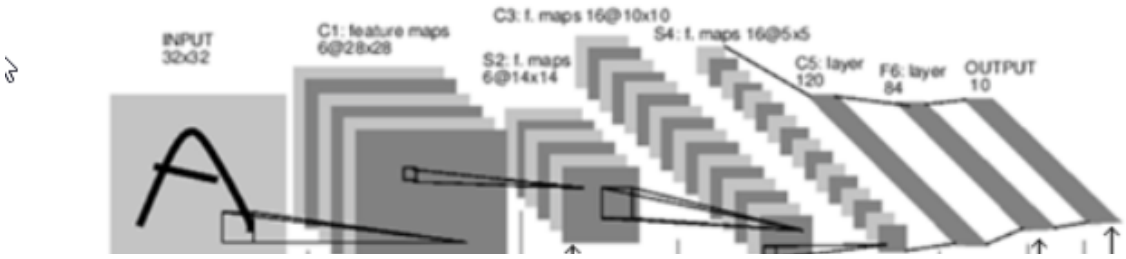
The connection method of the F6 layer is as follows:

Output layer-fully connected layer

The output layer is also a fully connected layer, with a total of 10 nodes, which respectively represent the numbers 0 to 9, and if the value of node i is 0, the result of network recognition is the number i. A radial basis function (RBF) network connection is used. Assuming x is the input of the previous layer and y is the output of the RBF, the calculation of the RBF output is:

$$y_i = \sum_j (x_j - w_{ij})^2$$

The value of the above formula w_ij is determined by the bitmap encoding of i, where i ranges from 0 to 9, and j ranges from 0 to 7 * 12-1. The closer the value of the RBF output is to 0, the closer it is to i, that is, the closer to the ASCII encoding figure of i, it means that the recognition result input by the current network is the character i. This layer has 84x10 = 840 parameters and connections.



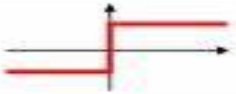
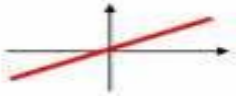
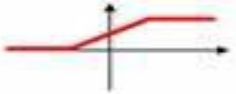
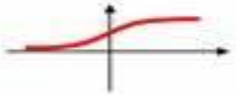
Code Implementation

In [5]:
```
from tensorflow import keras
from keras.datasets import mnist
from keras.layers import Conv2D, MaxPooling2D,AveragePooling2D
from keras.layers import Dense, Flatten
from keras.models import Sequential

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Normalize pixel values between 0 and 1
x_train = x_train / 255.0
x_test = x_test / 255.0

# Convert labels to one-hot encoding
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)


# Building the Model Architecture

model = Sequential()

model.add(Conv2D(6, kernel_size = (5,5), padding = 'valid', activation='tanh', input_sha
model.add(AveragePooling2D(pool_size= (2,2), strides = 2, padding = 'valid'))

model.add(Conv2D(16, kernel_size = (5,5), padding = 'valid', activation='tanh'))
model.add(AveragePooling2D(pool_size= (2,2), strides = 2, padding = 'valid'))

model.add(Flatten())

model.add(Dense(120, activation='tanh'))
model.add(Dense(84, activation='tanh'))
model.add(Dense(10, activation='softmax'))

model.summary()


model.compile(loss=keras.metrics.categorical_crossentropy, optimizer=keras.optimizers.Ad
model.fit(x_train, y_train, batch_size=128, epochs=2, verbose=1, validation_data=(x_test
score = model.evaluate(x_test, y_test)

print('Test Loss:', score[0])
print('Test accuracy:', score[1])
```

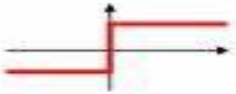
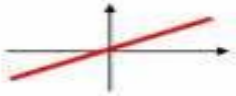
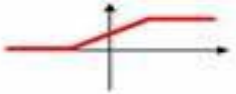
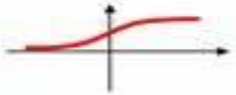
```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_5 (Conv2D)           (None, 28, 28, 6)         456

 average_pooling2d_4 (Averag  (None, 14, 14, 6)        0
 ePooling2D)

 conv2d_6 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_5 (Averag  (None, 5, 5, 16)         0
 ePooling2D)

 flatten_2 (Flatten)         (None, 400)               0

 dense_6 (Dense)             (None, 120)               48120

 dense_7 (Dense)             (None, 84)                10164

 dense_8 (Dense)             (None, 10)                850

=================================================================
Total params: 62,006
Trainable params: 62,006
Non-trainable params: 0
_________________________________________________________________
Epoch 1/2
391/391 [==============================] - 14s 8ms/step - loss: 1.8395 - accuracy: 0.34
66 - val_loss: 1.7231 - val_accuracy: 0.3949
Epoch 2/2
391/391 [==============================] - 2s 5ms/step - loss: 1.6719 - accuracy: 0.411
2 - val_loss: 1.6083 - val_accuracy: 0.4258
313/313 [==============================] - 1s 3ms/step - loss: 1.6083 - accuracy: 0.425
8
Test Loss: 1.6083446741104126
Test accuracy: 0.42579999566078186
```

In [ ]:

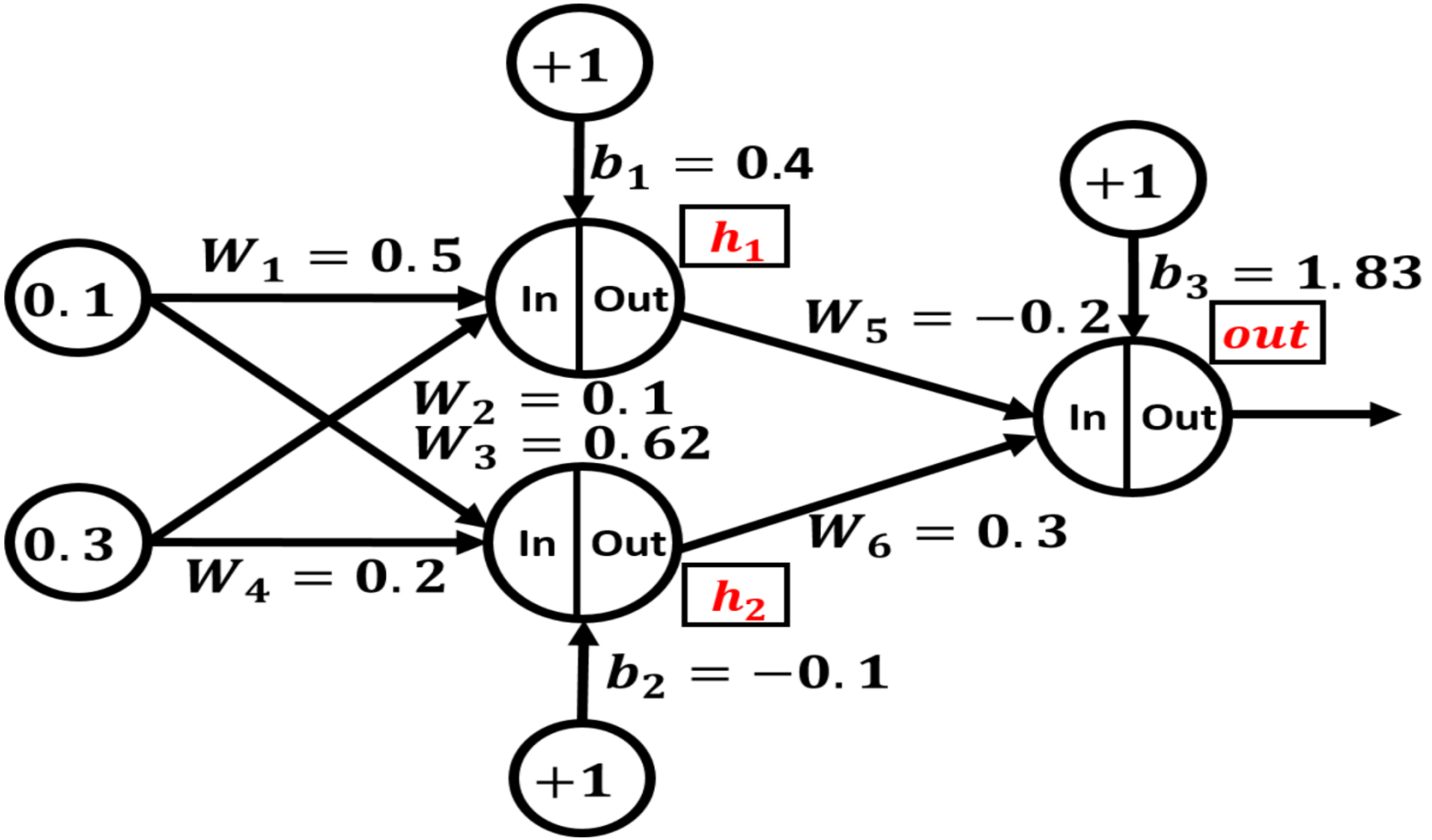
Activation Function

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Activation Function

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Back Propagation

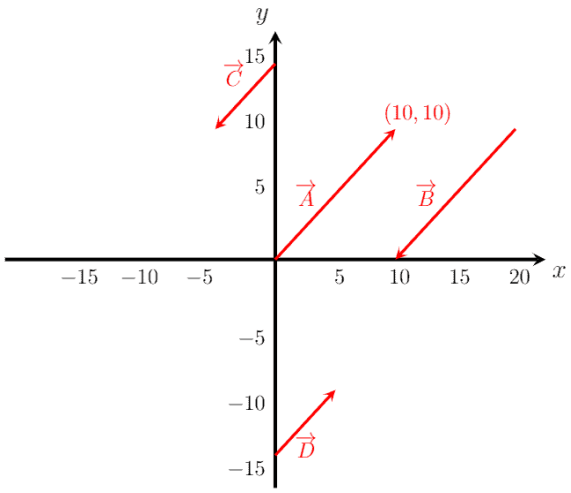


Some derivation of necessary mathematics:

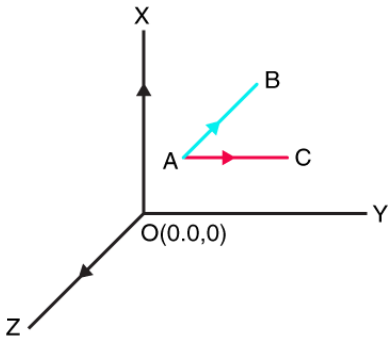
- Vectors
- Differentiation
- Partial differentiation
- Gradient of a Function
- Maxima & Minima

Vectors:

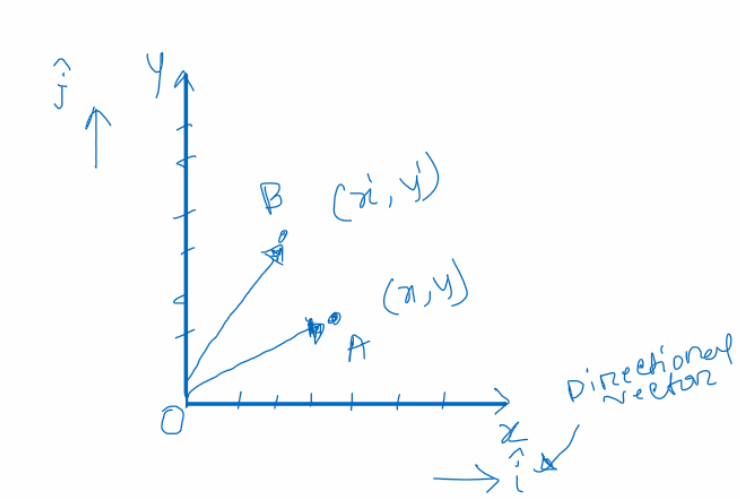
A vector is an object that has both a magnitude and a direction (i.e. 5km/m in north). Geometrically, we can picture a vector as a directed line segment, whose length is the magnitude of the vector and with an arrow indicating the direction. The direction of the vector is from its tail to its head.Two vectors are the same if they have the same magnitude and direction. This means that if we take a vector and translate it to a new position (without rotating it), then the vector we obtain at the end of this process is the same vector we had in the beginning.



Vector in 3D:



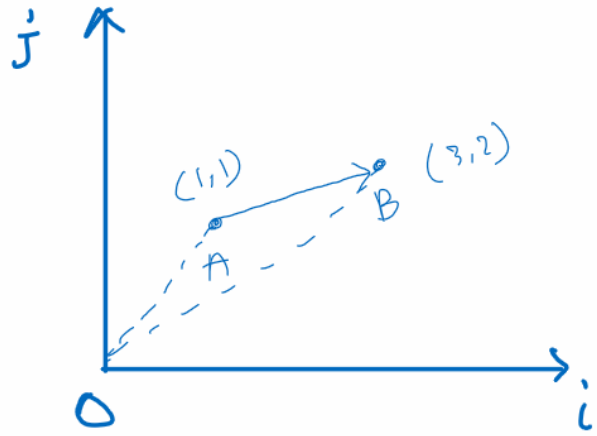
- **Now lets derive some derivation:**



  - $OA = x_{\hat{i}} + y_{\hat{j}}$
  - $OB = x'_{\hat{i}} + y'_{\hat{j}}$

We can also represent vectors like that,

- $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix}$



$\overrightarrow{AB}$ = Length of AB

- $OA = \hat{i} + \hat{j}$
- $OB = 3\hat{i} + 2\hat{j}$

$\overrightarrow{AB} = \overrightarrow{OB} - \overrightarrow{OA}$

$$= (3-1)i + (2-1)j$$

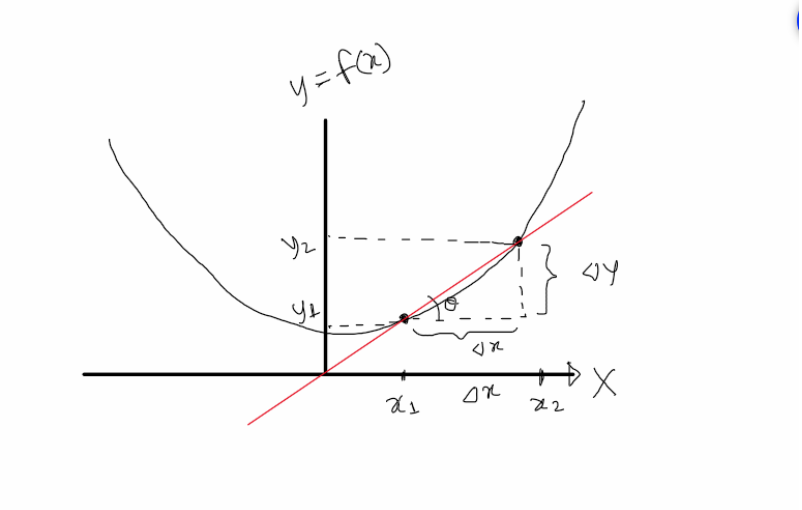
$\therefore \overrightarrow{AB} = 2i + j$

Differentiation:

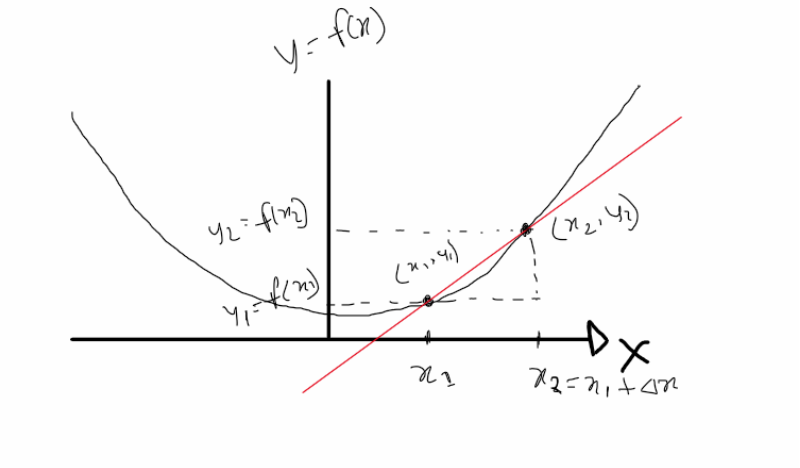
Differentiation generally refers to the rate of change of a function with respect to one of its variables. Here its similar to finding the tangent line slope of a function at some specific point.

- Connecting f,f' and f'' graphically

Lets derive some derivation:



- $slope = \frac{\Delta y}{\Delta x}$
- $slope = tan\theta = \frac{perpendicular}{base}$
- $slope = \frac{y_2 - y_1}{x_2 - x_1}$



$\lim_{x \to 0} \frac{\Delta y}{\Delta x} = \frac{dy}{dx}$

$\lim_{x \to 0} \frac{y_2 - y_1}{\Delta x}$

$\lim_{x \to 0} \frac{f(x_2) - f(x_1)}{\Delta x}$

$\therefore \frac{dy}{dx} = \lim_{x \to 0} \frac{f(x_1 + \Delta x) - f(x_1)}{\Delta x}$

This is the final differentiation equation.If you put any x here you will get the slope at your x point.

Lets see some formulas / rules of Differentiation:

Power Rule:

Here we use the power rule in order to calculate the derivative and it's pretty simple though.

$$if,\ \mathbf{f(x) = x^n}$$

$$then,\ \mathbf{f'(x) = n.x^{n-1}}$$

Examples

The considered function f(x) is equal to x to the fifth.

$$f(x) = x^5$$
$$f'(x) = 5x^{(5-1)}$$
$$f'(x) = 5x^4$$

Product Rule:

If a(x) and b(x) are two differentiable functions, then the product rule is used, where at first time it compute derivative of first function and at second time it compute derivative of second function.

$$\mathbf{f(x) = f(x).g(x)}$$

$$\mathbf{f'(x) = f'(x).g(x) + f(x).g'(x)}$$

Example

$$f(x) = (x^4 + 2).cosx$$
$$\Rightarrow f'(x) = 4x^3.cosx + (x^4 + 2).(-sinx)$$
$$\Rightarrow f'(x) = 4x^3cosx - x^4sinx - 2sinx$$

Partial Differentiation:

Now we will see the Partial Differentiation. Here, the same rules apply while obtaining derivatives with respect to one variable while keeping others constant. This term is used for Multi-Variable Functions.

Example

$$f(x, y) = x^4y$$

Obtaining partial derivative w.r.t x

$$\frac{\partial(x^4y)}{\partial x} = 4x^3y$$

Obtaining partial derivative w.r.t y

$$\frac{\partial(x^4y)}{\partial y} = x^4$$

Gradient of Function:

- Let's say there's a function of two variable x and y $\Rightarrow f(x, y)$
- then $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ is partial derivative w.r.t x and y respectively
- Now Gradient '∇' of f is defined as -

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix}^T = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

- Its nothing but vector of partial derivatives

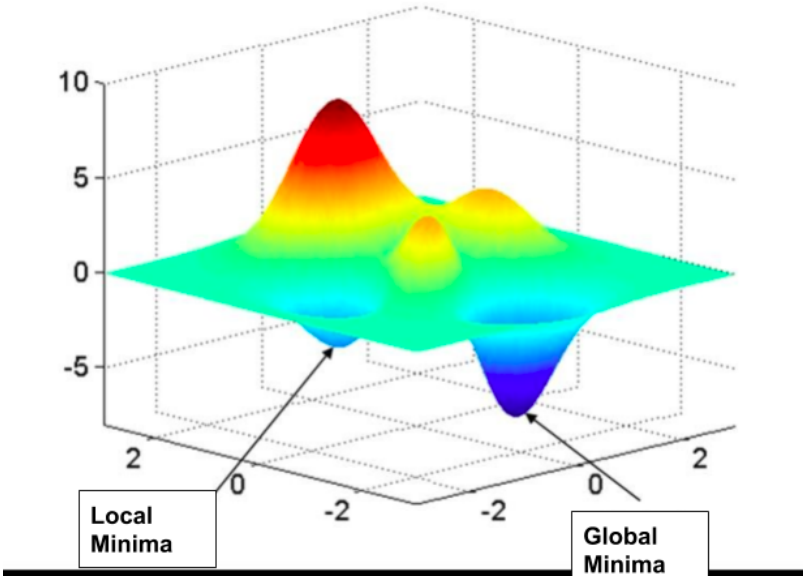
EXAMPLE

$$f(x, y) = 2.x^2 + 4y$$

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 4x \\ 4 \end{bmatrix}$$

Local and Global Minima:

- For any function their can be many minimum points and among those minimum there can exist an minima point where function has the least value and this point is known as **global minima** and other minimum points are known as **local minima** point.
- In ML/DL finding the global minima of a loss function using optimization techniques like gradient descent plays a very important role.
-



Finding Maxima and Minima of a Function:

Calculating Maxima and Minima for a function with a single variable(univariate):

- Find points that satisfies the equation $f'(x) = 0$. These points are known as critical points. Let's say you get two points c1 and c2.
- Find double derivative of $f''(x)$ and find its value at c1 and c2
  - for c1 if -
    - f''(c1) > 0 $\Rightarrow$ its a point of minima and f(c1) is the minimum value
    - f''(c1) < 0 $\Rightarrow$ its a point of maxima and f(c1) is the maximum value

Calculating Maxima and Minima for a function with two variable(multivariate):

- Let $f(x, y)$ is a bivariate function whose local minima or maxima point needs to be calculated.
- Find -
  - $f_x = p = \frac{\partial f}{\partial x}$ and
  - $f_y = q = \frac{\partial f}{\partial y}$.
- Solve $f_x = 0$ and $f_y = 0$ and find stationary or critical points.
- Find -
  - $r = f_{xx} = \frac{\partial f^2}{\partial x^2}$,
  - $s = f_{xy} = \frac{\partial f^2}{\partial xy}$ and
  - $t = f_{yy} = \frac{\partial f^2}{\partial y^2}$
- Lets do the analysis for the critical points that we have obtained. Lets take a critical point (a,b)
  - if $r.t - s^2 > 0$ and
    - if $r > 0 \Rightarrow f(a, b)$ has local minimum at that critical point
    - if $r < 0 \Rightarrow f(a, b)$ has local maximum at that critical point
  - if $r.t - s^2 = 0 \Rightarrow$ test fails.
  - if $r.t - s^2 < 0 \Rightarrow$ its a sadal point at the critical point (i.e. neither max nor minimum)

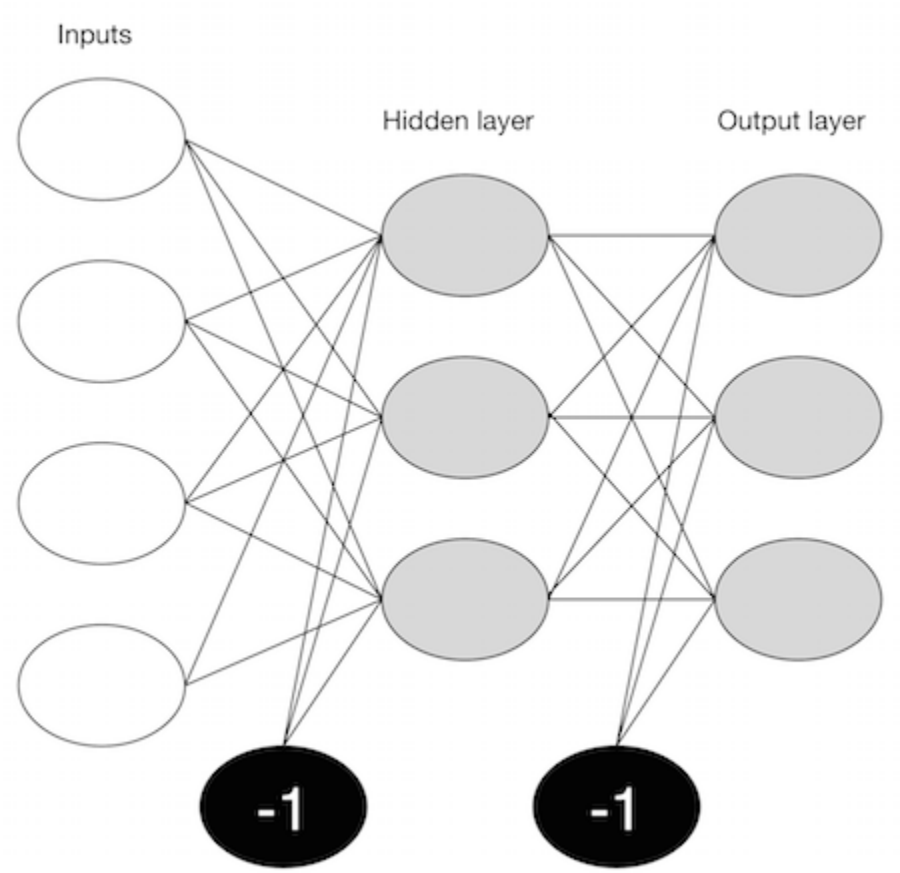
Multi-layer Perceptron

The solution to fitting more complex (*i.e.* non-linear) models with neural networks is to use a more complex network that consists of more than just a single perceptron. The take-home message from the perceptron is that all of the learning happens by adapting the synapse weights until prediction is satisfactory. Hence, a reasonable guess at how to make a perceptron more complex is to simply **add more weights**.

There are two ways to add complexity:

1. Add backward connections, so that output neurons feed back to input nodes, resulting in a **recurrent network**
2. Add neurons between the input nodes and the outputs, creating an additional ("hidden") layer to the network, resulting in a **multi-layer perceptron**

The latter approach is more common in applications of neural networks.



How to train a multilayer network is not intuitive. Propagating the inputs forward over two layers is straightforward, since the outputs from the hidden layer can be used as inputs for the output layer. However, the process for updating the weights based on the prediction error is less clear, since it is difficult to know whether to change the weights on the input layer or on the hidden layer in order to improve the prediction.

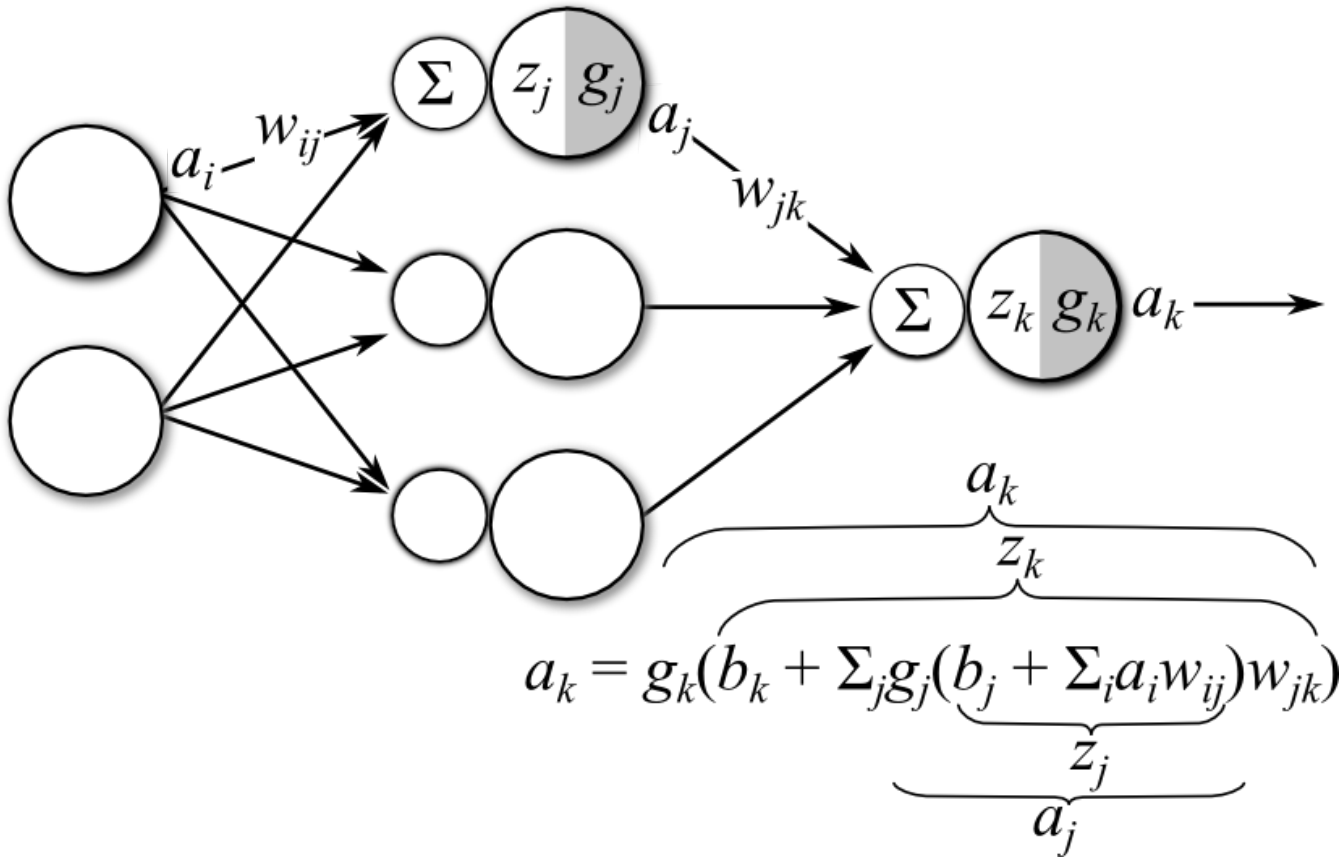
Updating a multi-layer perceptron (MLP) is a matter of:

1. moving forward through the network, calculating outputs given inputs and current weight estimates
2. moving backward updating weights according to the resulting error from forward propagation.

In this sense, it is similar to a single-layer perceptron, except it has to be done twice, once for each layer.

Backpropagation

Backpropagation is a method for efficiently computing the gradient of the cost function of a neural network with respect to its parameters. These partial derivatives can then be used to update the network's parameters using, e.g., gradient descent. This may be the most common method for training neural networks. Deriving backpropagation involves numerous clever applications of the chain rule for functions of vectors.



Review: The chain rule

The chain rule is a way to compute the derivative of a function whose variables are themselves functions of other variables. If C is a scalar-valued function of a scalar z and z is itself a scalar-valued function of another scalar variable w, then the chain rule states that

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial z}\frac{\partial z}{\partial w}$$

For scalar-valued functions of more than one variable, the chain rule essentially becomes additive. In other words, if C is a scalar-valued function of N variables $z_1, \ldots, z_N$, each of which is a function of some variable w, the chain rule states that

$$\frac{\partial C}{\partial w} = \sum_{i=1}^{N} \frac{\partial C}{\partial z_i}\frac{\partial z_i}{\partial w}$$

Notation

In the following derivation, we'll use the following notation:

L - Number of layers in the network.

N^n - Dimensionality of layer $n \in \{0, \dots, L\}$. N^0 is the dimensionality of the input; N^L is the dimensionality of the output.

$W^m \in \mathbb{R}^{N^m \times N^{m-1}}$ - Weight matrix for layer $m \in \{1, \dots, L\}$. W_{ij}^m is the weight between the i^{th} unit in layer m and the j^{th} unit in layer $m-1$.

$b^m \in \mathbb{R}^{N^m}$ - Bias vector for layer m.

σ^m - Nonlinear activation function of the units in layer m, applied element wise.

$z^m \in \mathbb{R}^{N^m}$ - Linear mix of the inputs to layer m, computed by $z^m = W^m a^{m-1} + b^m$.

$a^m \in \mathbb{R}^{N^m}$ - Activation of units in layer m, computed by $a^m = \sigma^m(h^m) = \sigma^m(W^m a^{m-1} + b^m)$. a^L is the output of the network. We define the special case a^0 as the input of the network.

$y \in \mathbb{R}^{N^L}$ - Target output of the network.

C - Cost/error function of the network, which is a function of a^L (the network output) and y (treated as a constant).

Backpropagation in general

In order to train the network using a gradient descent algorithm, we need to know the gradient of each of the parameters with respect to the cost/error function C; that is, we need to know $\frac{\partial C}{\partial W^m}$ and $\frac{\partial C}{\partial b^m}$. It will be sufficient to derive an expression for these gradients in terms of the following terms, which we can compute based on the neural network's architecture:

- $\frac{\partial C}{\partial a^L}$: The derivative of the cost function with respect to its argument, the output of the network
- $\frac{\partial a^m}{\partial z^m}$: The derivative of the nonlinearity used in layer m with respect to its argument

To compute the gradient of our cost/error function C to W_{ij}^m (a single entry in the weight matrix of the layer m), we can first note that C is a function of a^L, which is itself a function of the linear mix variables z_k^m, which are themselves functions of the weight matrices W^m and biases b^m. With this in mind, we can use the chain rule as follows:

$$\frac{\partial C}{\partial W_{ij}^m} = \sum_{k=1}^{N^m} \frac{\partial C}{\partial z_k^m} \frac{\partial z_k^m}{\partial W_{ij}^m}$$

Note that by definition

$$z_k^m = \sum_{l=1}^{N^m} W_{kl}^m a_l^{m-1} + b_k^m$$

It follows that $\frac{\partial z_k^m}{\partial W_{ij}^m}$ will evaluate to zero when $i \neq k$ because z_k^m does not interact with any elements in W^m except for those in the k^{th} row, and we are only considering the entry W_{ij}^m. When $i = k$, we have

$$\begin{aligned}\frac{\partial z_i^m}{\partial W_{ij}^m} &= \frac{\partial}{\partial W_{ij}^m}\left(\sum_{l=1}^{N^m} W_{il}^m a_l^{m-1} + b_i^m\right) \\ &= a_j^{m-1} \\ \rightarrow \frac{\partial z_k^m}{\partial W_{ij}^m} &= \begin{cases} 0 & k \neq i \\ a_j^{m-1} & k = i \end{cases}\end{aligned}$$

The fact that $\frac{\partial C}{\partial a_k^m}$ is 0 unless $k = i$ causes the summation above to collapse, giving

$$\frac{\partial C}{\partial W_{ij}^m} = \frac{\partial C}{\partial z_i^m} a_j^{m-1}$$

or in vector form

$$\frac{\partial C}{\partial W^m} = \frac{\partial C}{\partial z^m} a^{m-1\top}$$

Similarly for the bias variables b^m, we have

$$\frac{\partial C}{\partial b_i^m} = \sum_{k=1}^{N^m} \frac{\partial C}{\partial z_k^m} \frac{\partial z_k^m}{\partial b_i^m}$$

As above, it follows that $\frac{\partial z_k^m}{\partial b_i^m}$ will evaluate to zero when $i \neq k$ because z_k^m does not interact with any element in b^m except b_k^m. When $i = k$, we have

$$\begin{aligned}\frac{\partial z_i^m}{\partial b_i^m} &= \frac{\partial}{\partial b_i^m}\left(\sum_{l=1}^{N^m} W_{il}^m a_l^{m-1} + b_i^m\right) \\ &= 1 \\ \rightarrow \frac{\partial z_i^m}{\partial b_i^m} &= \begin{cases} 0 & k \neq i \\ 1 & k = i \end{cases}\end{aligned}$$

The summation also collapses to give

$$\frac{\partial C}{\partial b_i^m} = \frac{\partial C}{\partial z_i^m}$$

or in vector form

$$\frac{\partial C}{\partial b^m} = \frac{\partial C}{\partial z^m}$$

Now, we must compute $\frac{\partial C}{\partial z_k^m}$. For the final layer ($m = L$), this term is straightforward to compute using the chain rule:

$$\frac{\partial C}{\partial z_k^L} = \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_k^L}$$

or, in vector form

$$\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L} \frac{\partial a^L}{\partial z^L}$$

The first term $\frac{\partial C}{\partial a^L}$ is just the derivative of the cost function with respect to its argument, whose form depends on the cost function chosen. Similarly, $\frac{\partial a^m}{\partial z^m}$ (for any layer m includling L) is the derivative of the layer's nonlinearity with respect to its argument and will depend on the choice of nonlinearity. For other layers, we again invoke the chain rule:

$$\begin{aligned}
\frac{\partial C}{\partial z_k^m} &= \frac{\partial C}{\partial a_k^m}\frac{\partial a_k^m}{\partial z_k^m} \\
&= \left(\sum_{l=1}^{N^{m+1}} \frac{\partial C}{\partial z_l^{m+1}}\frac{\partial z_l^{m+1}}{\partial a_k^m}\right)\frac{\partial a_k^m}{\partial z_k^m} \\
&= \left(\sum_{l=1}^{N^{m+1}} \frac{\partial C}{\partial z_l^{m+1}}\frac{\partial}{\partial a_k^m}\left(\sum_{h=1}^{N^m} W_{lh}^{m+1} a_h^m + b_l^{m+1}\right)\right)\frac{\partial a_k^m}{\partial z_k^m} \\
&= \left(\sum_{l=1}^{N^{m+1}} \frac{\partial C}{\partial z_l^{m+1}} W_{lk}^{m+1}\right)\frac{\partial a_k^m}{\partial z_k^m} \\
&= \left(\sum_{l=1}^{N^{m+1}} W_{kl}^{m+1\top}\frac{\partial C}{\partial z_l^{m+1}}\right)\frac{\partial a_k^m}{\partial z_k^m}
\end{aligned}$$

where the last simplification was made because by convention $\frac{\partial C}{\partial z_l^{m+1}}$ is a column vector, allowing us to write the following vector form:

$$\frac{\partial C}{\partial z^m} = \left(W^{m+1\top}\frac{\partial C}{\partial z^{m+1}}\right) \circ \frac{\partial a^m}{\partial z^m}$$

Backpropagation in practice

As discussed above, the exact form of the updates depends on both the chosen cost function and each layer's chosen nonlinearity. The following two table lists the some common choices for non-linearities and the required partial derivative for deriving the gradient for each layer:

Nonlinearity	$a^m = \sigma^m(z^m)$	$\frac{\partial a^m}{\partial z^m}$	**Notes**
Sigmoid	$\frac{1}{1+e^{z^m}}$	$\sigma^m(z^m)(1-\sigma^m(z^m)) = a^m(1-a^m)$	"Squashes" any input to the range $[0, 1]$
Tanh	$\frac{e^{z^m}-e^{-z^m}}{e^{z^m}+e^{-z^m}}$	$1-(\sigma^m(z^m))^2 = 1-(a^m)^2$	Equivalent, up to scaling, to the sigmoid function
ReLU	$\max(0, z^m)$	$0, z^m < 0;\ \ 1, z^m \geq 0$	Commonly used in neural networks with many layers

Similarly, the following table collects some common cost functions and the partial derivative needed to compute the gradient for the final layer:

Cost Function	C	$\frac{\partial C}{\partial a^L}$	**Notes**
Squared Error	$\frac{1}{2}(y-a^L)^\top(y-a^L)$	$y-a^L$	Commonly used when the output is not constrained to a specific range
Cross-Entropy	$(y-1)\log(1-a^L) - y\log(a^L)$	$\frac{a^L-y}{a^L(1-a^L)}$	Commonly used for binary classification tasks; can yield faster convergence

In practice, backpropagation proceeds in the following manner for each training sample:

1. Forward pass: Given the network input a^0, compute a^m recursively by
$$a^1 = \sigma^1(W^1 a^0 + b^1), \ldots, a^L = \sigma^L(W^L a^{L-1} + b^L)$$
2. Backward pass: Compute
$$\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}$$
for the final layer based on the tables above, then recursively compute
$$\frac{\partial C}{\partial z^m} = \left(W^{m+1\top}\frac{\partial C}{\partial z^{m+1}}\right) \circ \frac{\partial a^m}{\partial z^m}$$
for all other layers. Plug these values into
$$\frac{\partial C}{\partial W^m} = \frac{\partial C}{\partial z_i^m} a^{m-1\top}$$
and
$$\frac{\partial C}{\partial b^m} = \frac{\partial C}{\partial z^m}$$
to obtain the updates.

Example: Sigmoid network with cross-entropy loss using gradient descent

A common network architecture is one with fully connected layers where each layer's nonlinearity is the sigmoid function $a^m = \frac{1}{1+e^{z^m}}$ and the cost function is the cross-entropy loss $(y-1)\log(1-a^L) - y\log(a^L)$. To compute the updates for gradient descent, we first compute (based on the tables above)

$$\begin{aligned}\frac{\partial C}{\partial z^L} &= \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L} \\ &= \left(\frac{a^L - y}{a^L(1-a^L)}\right) a^L(1-a^L) \\ &= a^L - y\end{aligned}$$

From here, we can compute

$$\begin{aligned}\frac{\partial C}{\partial z^{L-1}} &= \left(W^{L\top}\frac{\partial C}{\partial z^L}\right) \circ \frac{\partial a^{L-1}}{\partial z^{L-1}} \\ &= W^{L\top}(a^L - y) \circ a^{L-1}(1-a^{L-1}) \\ \frac{\partial C}{\partial z^{L-2}} &= \left(W^{L-1\top}\frac{\partial C}{\partial z^{L-1}}\right) \circ \frac{\partial a^{L-2}}{\partial z^{L-2}} \\ &= W^{L-1\top}\left(W^{L\top}(a^L - y) \circ a^{L-1}(1-a^{L-1})\right) \circ a^{L-2}(1-a^{L-2})\end{aligned}$$

and so on, until we have computed $\frac{\partial C}{\partial z^m}$ for $m \in \{1, \ldots, L\}$. This allows us to compute $\frac{\partial C}{\partial W_{ij}^m}$ and $\frac{\partial C}{\partial b_i^m}$, e.g.

$$\begin{aligned}\frac{\partial C}{\partial W^L} &= \frac{\partial C}{\partial z^L} a^{L-1\top} \\ &= (a^L - y)a^{L-1\top} \\ \frac{\partial C}{\partial W^{L-1}} &= \frac{\partial C}{\partial z^{L-1}} a^{L-2\top} \\ &= W^{L\top}(a^L - y) \circ a^{L-1}(1-a^{L-1})a^{L-2\top}\end{aligned}$$

and so on. Standard gradient descent then updates each parameter as follows:

$$W^m = W^m - \lambda\frac{\partial C}{\partial W^m}$$
$$b^m = b^m - \lambda\frac{\partial C}{\partial b^m}$$

Toy Python example

Due to the recursive nature of the backpropagation algorithm, it lends itself well to software implementations. The following code implements a multi-layer perceptron which is trained using backpropagation with user-supplied non-linearities, layer sizes, and cost function.

In [1]:

```
# Ensure python 3 forward compatibility
from __future__ import print_function
import numpy as np

def sigmoid(x):
    return 1/(1 + np.exp(-x))

class SigmoidLayer:
    def __init__(self, n_input, n_output):
        self.W = np.random.randn(n_output, n_input)
        self.b = np.random.randn(n_output, 1)
    def output(self, X):
        if X.ndim == 1:
            X = X.reshape(-1, 1)
        return sigmoid(self.W.dot(X) + self.b)

class SigmoidNetwork:

    def __init__(self, layer_sizes):
        '''
        :parameters:
            - layer_sizes : list of int
                List of layer sizes of length L+1 (including the input dimensionality)
        '''
        self.layers = []
        for n_input, n_output in zip(layer_sizes[:-1], layer_sizes[1:]):
            self.layers.append(SigmoidLayer(n_input, n_output))

    def train(self, X, y, learning_rate=0.2):
        X = np.array(X)
        y = np.array(y)
        if X.ndim == 1:
            X = X.reshape(-1, 1)
        if y.ndim == 1:
            y = y.reshape(1, -1)

        # Forward pass - compute a^n for n in {0, ... L}
        layer_outputs = [X]
        for layer in self.layers:
            layer_outputs.append(layer.output(layer_outputs[-1]))

        # Backward pass - compute \partial C/\partial z^m for m in {L, ..., 1}
        cost_partials = [layer_outputs[-1] - y]
        for layer, layer_output in zip(reversed(self.layers), reversed(layer_outputs[:-1
            cost_partials.append(layer.W.T.dot(cost_partials[-1])*layer_output*(1 - laye
        cost_partials.reverse()

        # Compute weight gradient step
        W_updates = []
        for cost_partial, layer_output in zip(cost_partials[1:], layer_outputs[:-1]):
            W_updates.append(cost_partial.dot(layer_output.T)/X.shape[1])
        # and biases
        b_updates = [cost_partial.mean(axis=1).reshape(-1, 1) for cost_partial in cost_p

        for W_update, b_update, layer in zip(W_updates, b_updates, self.layers):
            layer.W -= W_update*learning_rate
            layer.b -= b_update*learning_rate

    def output(self, X):
```

```
        a = np.array(X)
        if a.ndim == 1:
            a = a.reshape(-1, 1)
        for layer in self.layers:
            a = layer.output(a)
        return a
```

In [2]:
```
nn = SigmoidNetwork([2, 2, 1])
X = np.array([[0, 1, 0, 1],
              [0, 0, 1, 1]])
y = np.array([0, 1, 1, 0])
for n in range(int(1e3)):
    nn.train(X, y, learning_rate=1.)
print("Input\tOutput\tQuantized")
for i in [[0, 0], [1, 0], [0, 1], [1, 1]]:
    print("{}\t{:.4f}\t{}".format(i, nn.output(i)[0, 0], 1*(nn.output(i)[0] > .5)))
```

```
Input   Output  Quantized
[0, 0]  0.0148  [0]
[1, 0]  0.9825  [1]
[0, 1]  0.9825  [1]
[1, 1]  0.0280  [0]
```

In [4]:
```
import ipywidgets as widgets
from ipywidgets import *
import matplotlib.pyplot as plt
logistic = lambda h, beta: 1./(1 + np.exp(-beta * h))

@interact(beta=(-1, 25))
def logistic_plot(beta=5):
    hvals = np.linspace(-2, 2)
    plt.plot(hvals, logistic(hvals, beta))
```

```
interactive(children=(IntSlider(value=5, description='beta', max=25, min=-1), Output
()), _dom_classes=('widget…
```

This has the advantage of having a simple derivative:

$$\frac{dg}{dh} = \beta g(h)(1 - g(h))$$

Alternatively, the hyperbolic tangent function is also sigmoid:

$$g(h) = \tanh(h) = \frac{\exp(h) - \exp(-h)}{\exp(h) + \exp(-h)}$$

In [5]:
```
hyperbolic_tangent = lambda h: (np.exp(h) - np.exp(-h)) / (np.exp(h) + np.exp(-h))

@interact(theta=(-1, 25))
def tanh_plot(theta=5):
    hvals = np.linspace(-2, 2)
    h = hvals*theta
    plt.plot(hvals, hyperbolic_tangent(h))
```

```
interactive(children=(IntSlider(value=5, description='theta', max=25, min=-1), Output
()), _dom_classes=('widge…
```

Gradient Descent

The simplest algorithm for iterative minimization of differentiable functions is known as just **gradient descent**. Recall that the gradient of a function is defined as the vector of partial derivatives:

$$\nabla f(x) = [\partial f x_1, \partial f x_2, \ldots, \partial f x_n]$$

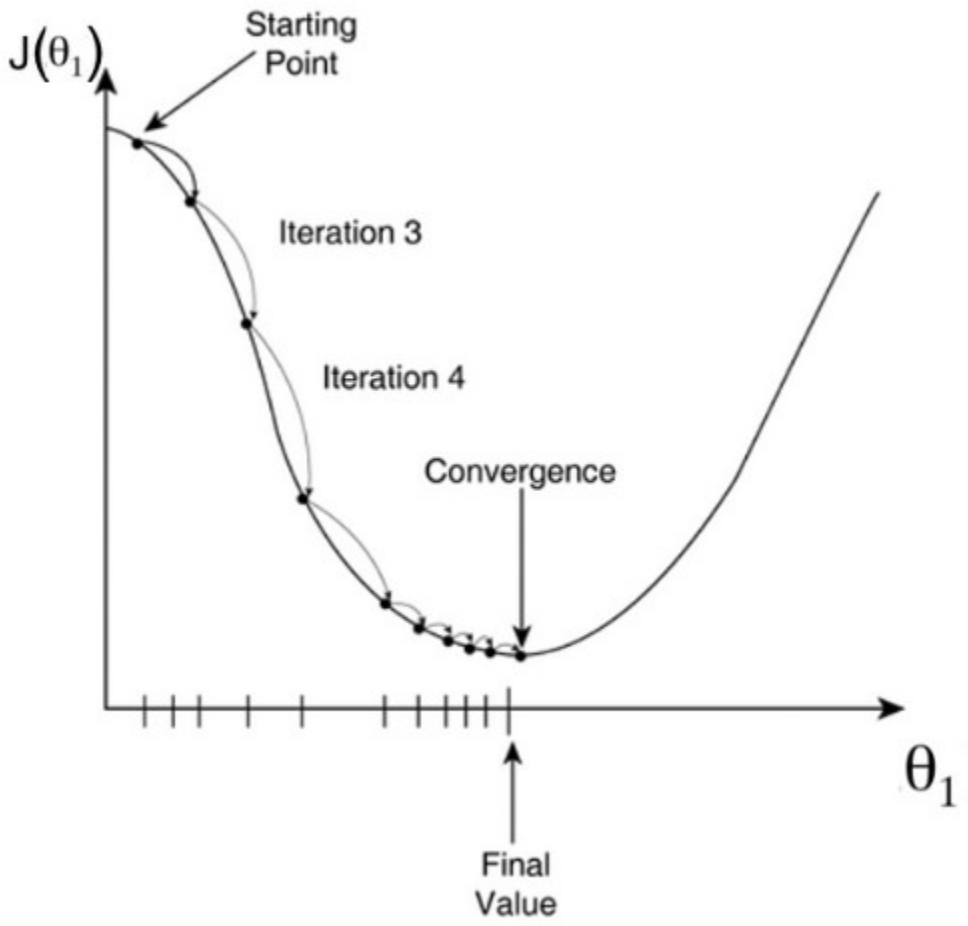
and that the gradient of a function always points towards the direction of maximal increase at that point.

Equivalently, it points *away* from the direction of maximum decrease - thus, if we start at any point, and keep moving in the direction of the negative gradient, we will eventually reach a local minimum.

This simple insight leads to the Gradient Descent algorithm. Outlined algorithmically, it looks like this:

1. Pick a point x_0 as your initial guess.
2. Compute the gradient at your current guess: $v_i = \nabla f(x_i)$
3. Move by α (your step size) in the direction of that gradient: $x_{i+1} = x_i + \alpha v_i$
4. Repeat steps 1-3 until your function is close enough to zero (until $f(x_i) < \varepsilon$ for some small tolerance ε)

Note that the step size, α, is simply a parameter of the algorithm and has to be fixed in advance.



Notice that the hyperbolic tangent function asymptotes at -1 and 1, rather than 0 and 1, which is sometimes beneficial, and its derivative is simple:

$$\frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

Performing gradient descent will allow us to change the weights in the direction that optimally reduces the error. The next trick will be to employ the **chain rule** to decompose how the error changes as a function of the input weights into the change in error as a function of changes in the inputs to the weights, multiplied by the changes in input values as a function of changes in the weights.

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial h}\frac{\partial h}{\partial w}$$

This will allow us to write a function describing the activations of the output weights as a function of the activations of the hidden layer nodes and the output weights, which will allow us to propagate error backwards through the network.

The second term in the chain rule simplifies to:

$$\begin{aligned}\frac{\partial h_k}{\partial w_{jk}} &= \frac{\partial \sum_l w_{lk}a_l}{\partial w_{jk}} \\ &= \sum_l \frac{\partial w_{lk}a_l}{\partial w_{jk}} \\ &= a_j\end{aligned}$$

where a_j is the activation of the jth hidden layer neuron.

For the first term in the chain rule above, we decompose it as well:

$$\frac{\partial E}{\partial h_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial h_k} = \frac{\partial E}{\partial g(h_k)}\frac{\partial g(h_k)}{\partial h_k}$$

The second term of this chain rule is just the derivative of the activation function, which we have chosen to have a convenient form, while the first term simplifies to:

$$\frac{\partial E}{\partial g(h_k)} = \frac{\partial}{\partial g(h_k)}\left[\frac{1}{2}\sum_k (t_k - y_k)^2\right] = t_k - y_k$$

Combining these, and assuming (for illustration) a logistic activation function, we have the gradient:

$$\frac{\partial E}{\partial w} = (t_k - y_k)y_k(1 - y_k)a_j$$

Which ends up getting plugged into the weight update formula that we saw in the single-layer perceptron:

$$w_{jk} \leftarrow w_{jk} - \eta(t_k - y_k)y_k(1 - y_k)a_j$$

Note that here we are *subtracting* the second term, rather than adding, since we are doing gradient descent.

We can now outline the MLP learning algorithm:

1. Initialize all w_{jk} to small random values
2. For each input vector, conduct forward propagation:
   - compute activation of each neuron j in hidden layer (here, sigmoid):

$$h_j = \sum_i x_i v_{ij}$$

$$a_j = g(h_j) = \frac{1}{1 + \exp(-\beta h_j)}$$

- when the output layer is reached, calculate outputs similarly:

$$h_k = \sum_k a_j w_{jk}$$

$$y_k = g(h_k) = \frac{1}{1 + \exp(-\beta h_k)}$$

3. Calculate loss for resulting predictions:
   - compute error at output:

$$\delta_k = (t_k - y_k) y_k (1 - y_k)$$

4. Conduct backpropagation to get partial derivatives of cost with respect to weights, and use these to update weights:
   - compute error of the hidden layers:

$$\delta_{hj} = \left[\sum_k w_{jk} \delta_k \right] a_j (1 - a_j)$$

   - update output layer weights:

$$w_{jk} \leftarrow w_{jk} - \eta \delta_k a_j$$

   - update hidden layer weights:

$$v_{ij} \leftarrow v_{ij} - \eta \delta_{hj} x_i$$

Return to (2) and iterate until learning completes. Best practice is to shuffle input vectors to avoid training in the same order.

Its important to be aware that because gradient descent is a hill-climbing (or descending) algorithm, it is liable to be caught in local minima with respect to starting values. Therefore, it is worthwhile training several networks using a range of starting values for the weights, so that you have a better chance of discovering a globally-competitive solution.

One useful performance enhancement for the MLP learning algorithm is the addition of **momentum** to the weight updates. This is just a coefficient on the previous weight update that increases the correlation between the current weight and the weight after the next update. This is particularly useful for complex models, where falling into local minima is an issue; adding momentum will give some weight to the previous direction, making the resulting weights essentially a weighted average of the two directions. Adding momentum, along with a smaller learning rate, usually results in a more stable algorithm with quicker convergence. When we use momentum, we lose this guarantee, but this is generally seen as a small price to pay for the improvement momentum usually gives.

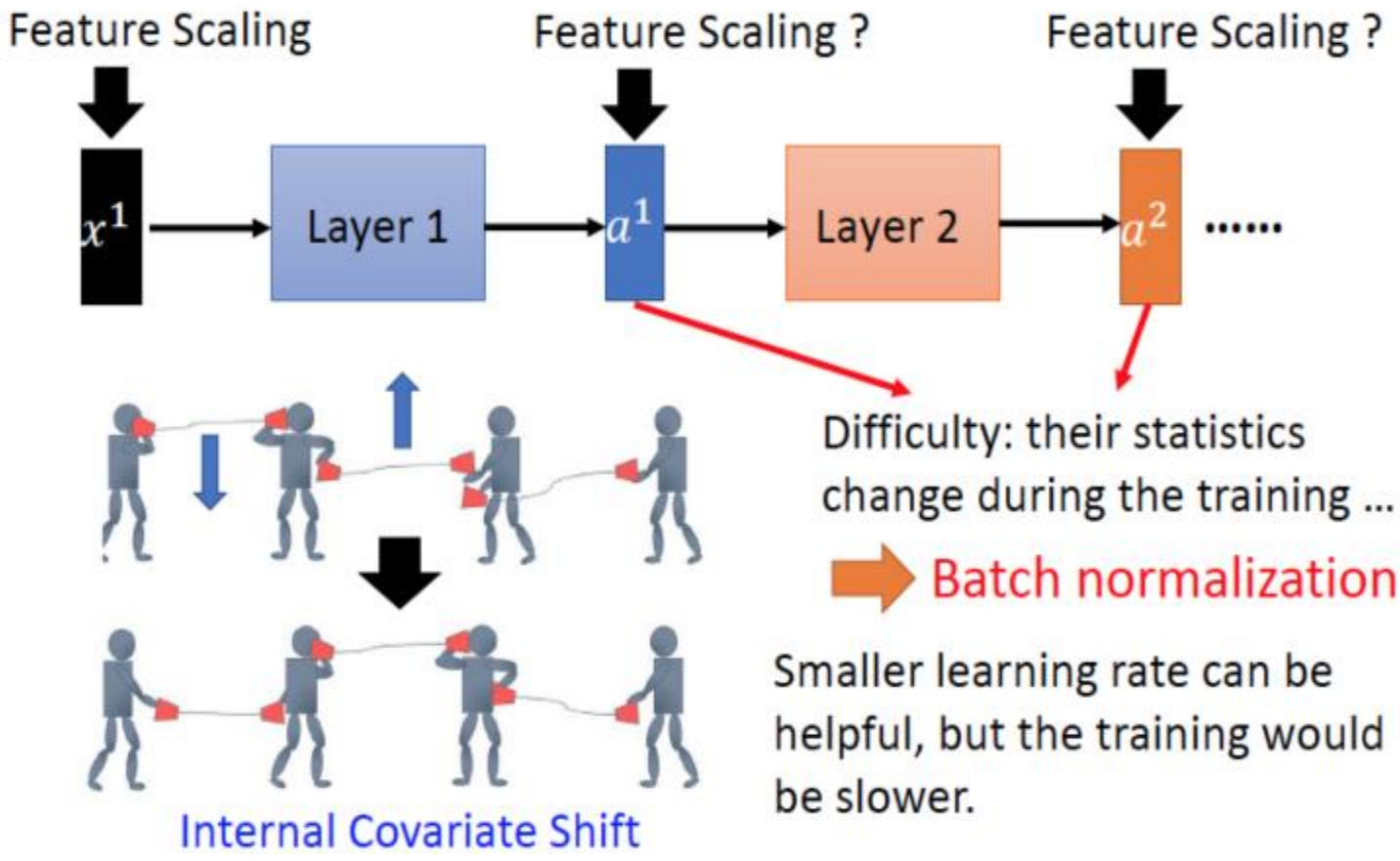
A weight update with momentum looks like this:

$$w_{jk} \leftarrow w_{jk} - \eta \delta_k a_j + \alpha \Delta w_{jk}^{t-1}$$

where α is the momentum (regularization) parameter and Δw_{jk}^{t-1} the update from the previous iteration.

The multi-layer pereptron is implemented below in the `MLP` class. The implementation uses the scikit-learn interface, so it is uses in the same way as other supervised learning algorithms in that package.

Batch Normalization



Batch normalization

Issues with training Deep Neural Networks

- There are 2 major issues 1) Internal Covariate shift, 2) Vanishing Gradient

Internal Covariate shift

- The concept of covariate shift pertains to the change that occurs in the distribution of the input to a learning system. In deep networks, this distribution can be influenced by parameters across all input layers. Consequently, even minor changes in the network can have a significant impact on its output. This effect gets magnified as the signal propagates through the network, which can result in a shift in the distribution of the inputs to internal layers. This phenomenon is known as internal covariate shift.
- When inputs are whitened (i.e., have zero mean and unit variance) and are uncorrelated, they tend to converge faster during training. However, internal covariate shift can have the opposite effect, as it introduces changes to the distribution of inputs that can slow down convergence. Therefore, to mitigate this effect, techniques like batch normalization have been developed to normalize the inputs to each layer in the network based on statistics of the current mini-batch.

Vanishing Gradient

- Saturating non-linearities such as sigmoid or tanh are not suitable for deep networks, as the signal tends to get trapped in the saturation region as the network grows deeper. This makes it difficult for the network to learn and can result in slow convergence during training. To overcome this problem we can use the following.
- Non-linearities like ReLU which do not saturate.
- Smaller learning rates
-

Careful initializations

What is Normalization?

-

Normalization in deep learning refers to the process of transforming the input or output of a layer in a neural network to improve its performance during training. The most common type of normalization used in deep learning is batch normalization, which normalizes the activations of a layer for each mini-batch during training.

What is batch normalization?

- Batch normalization is a technique in deep learning that helps to standardize and normalize the input to each layer of a neural network by adjusting and scaling the activations. The idea behind batch normalization is to normalize the inputs to a layer to have zero mean and unit variance across each mini-batch of the training data.

Steps involved in batch normalization

1) During training, for each mini-batch of data, compute the mean and variance of the activations of each layer. This can be done using the following formulas:

- Mean: $\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i$
- Variance: $\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$
- Here, m is the size of the mini-batch, and x_i is the activation of the i-th neuron in the layer.

2) Normalize the activations of each layer in the mini-batch using the following formula:

- $\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ Here, ϵ is a small constant added for numerical stability.

3) Scale and shift the normalized activations using the learned parameters γ and β, respectively:

- $y_i = \gamma \hat{x_i} + \beta$
- The parameters γ and β are learned during training using backpropagation.

4) During inference, the running mean and variance of each layer are used for normalization instead of the mini-batch statistics. These running statistics are updated using a moving average of the mini-batch statistics during training.

The benefits of batch normalization include:

- Improved training performance: Batch normalization reduces the internal covariate shift, which is the change in the distribution of the activations of each layer due to changes in the distribution of the inputs. This allows the network to converge faster and with more stable gradients.
- Regularization: Batch normalization acts as a form of regularization by adding noise to the activations of each layer, which can help prevent overfitting.
- **Increased robustness: Batch normalization makes the network more robust to changes in the input distribution, which can help improve its generalization performance.**

  Code example for batch normalization

```
import tensorflow as tf

# Define a fully connected layer
fc_layer = tf.keras.layers.Dense(units=128, activation='relu')

# Add batch normalization to the layer
bn_layer = tf.keras.layers.BatchNormalization()

# Define the model with the layer and batch normalization
model = tf.keras.models.Sequential([fc_layer, bn_layer])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy')

# Train the model
model.fit(x_train, y_train, batch_size=32, epochs=10)
```

- In the above code,the tf.keras.layers.BatchNormalization() layer is added after the fully connected layer to normalize the output before passing it to the activation function. The model.fit() function is then used to train the model using batch normalization.

Practical discussion of Callback functions:

A callback is a powerful tool to customize the behavior of a Keras model during training, evaluation, or inference.TensorBoard to visualize training progress and results with TensorBoard, or tf.keras.callbacks. ModelCheckpoint to periodically save your model during training. This callback reduces the learning rate when a metric you've mentioned during training eg. accuracy or loss has stopped improving. Models often benefit from reducing the learning rate.There are many callback functions,

- Early Stopping callback
- Model checkpointing callback
- Tensorboard callback Functions

We will also see how to save a model & load as well.

In [1]:
```
# Importing Libraries
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import os
```

In [2]:
```
# Checking version of Tensorflow and Keras
print(f"Tensorflow Version {tf.__version__}")
print(f"Keras Version {tf.keras.__version__}")
```

```
Tensorflow Version 2.10.0
Keras Version 2.10.0
```

GPU / CPU Check

In [3]:
```
tf.config.list_physical_devices("GPU")
```

Out[3]:
```
[]
```

In [4]:
```
tf.config.list_physical_devices("CPU")
```

Out[4]:
```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU')]
```

In [5]:
```
check_list = ['GPU','CPU']

for device in check_list:
  out = tf.config.list_physical_devices(device)
  if len(out) > 0:
    print(f"{device} is available!")
    print(f"Details >> {out}")
  else:
    print(f"{device} isn't available!")
```

```
GPU isn't available!
CPU is available!
Details >> [PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU')]
```

Creating a simple classifier using keras on MNIST data

In [6]:
```
mnist = tf.keras.datasets.mnist
(X_train_full, y_train_full), (X_test, y_test) = mnist.load_data()
```

In [7]:
```
print(f"data type of X_train_full: {X_train_full.dtype},\n shape of X_train_full: {X_tra
```

```
data type of X_train_full: uint8,
 shape of X_train_full: (60000, 28, 28)
```

In [8]:
```
X_test.shape
```

Out[8]:
```
(10000, 28, 28)
```

In [9]:
```
len(X_test[1][0])
```

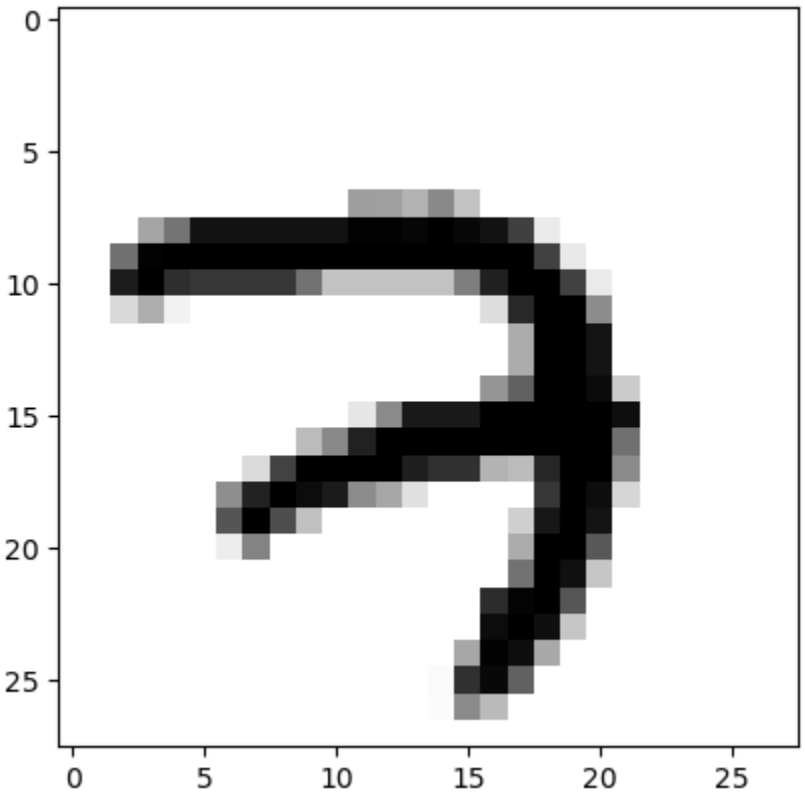
Out[9]:
```
28
```

In [10]:
```
# create a validation data set from the full training data
# Scale the data between 0 to 1 by dividing it by 255. as its an unsigned data between 0
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

# scale the test set as well
X_test = X_test / 255.
```

In [11]:
```
len(X_train_full[5000:] )
```

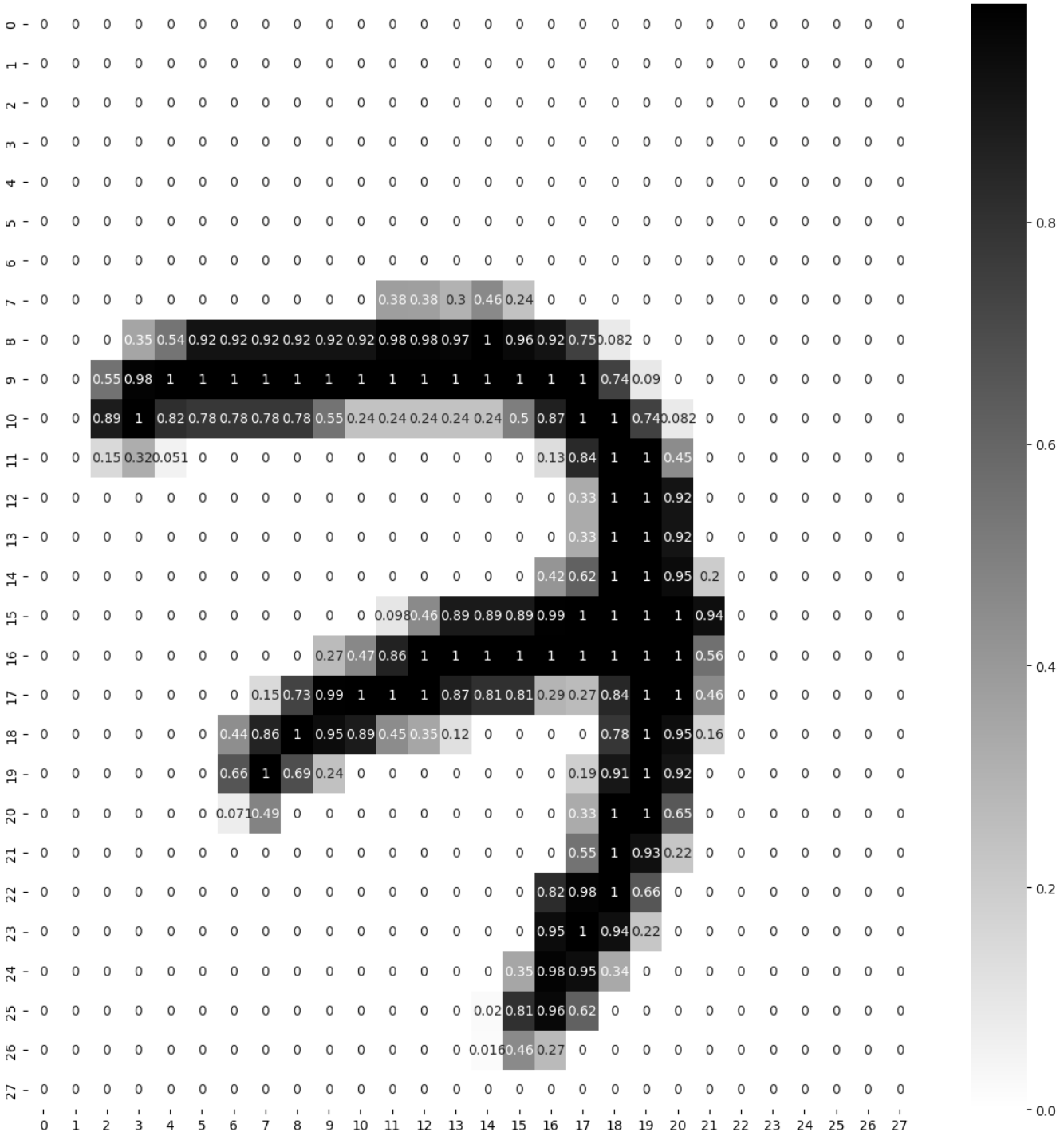
Out[11]:
```
55000
```

In [12]:
```
# Lets view some data
plt.imshow(X_train[0], cmap="binary")
plt.show()
```



In [13]:
```
plt.figure(figsize=(15,15))
sns.heatmap(X_train[0], annot=True, cmap="binary")
```

Out[13]: <Axes: >



In [14]:
```
LAYERS = [tf.keras.layers.Flatten(input_shape=[28, 28], name="inputLayer"),
          tf.keras.layers.Dense(300, activation="relu", name="hiddenLayer1"),
          tf.keras.layers.Dense(100, activation="relu", name="hiddenLayer2"),
          tf.keras.layers.Dense(10, activation="softmax", name="outputLayer")]

model_clf = tf.keras.models.Sequential(LAYERS)
```

In [15]:
```
model_clf.layers
```

Out[15]:
```
[<keras.layers.reshaping.flatten.Flatten at 0x19e30cc3460>,
 <keras.layers.core.dense.Dense at 0x19e30cc2bf0>,
 <keras.layers.core.dense.Dense at 0x19e30cc2c80>,
 <keras.layers.core.dense.Dense at 0x19e30cc2cb0>]
```

In [16]:
```
model_clf.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inputLayer (Flatten)        (None, 784)               0

 hiddenLayer1 (Dense)        (None, 300)               235500

 hiddenLayer2 (Dense)        (None, 100)               30100

 outputLayer (Dense)         (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

In [17]:
```
# first Layer * second Layer + bias
784*300 + 300, 300*100+100, 100*10+10
```

Out[17]:
```
(235500, 30100, 1010)
```

In [18]:
```
# Total parameters to be trained -
sum((235500, 30100, 1010))
```

Out[18]:
```
266610
```

In [19]:
```
hidden1 = model_clf.layers[1]
hidden1.name
```

Out[19]:
```
'hiddenLayer1'
```

In [20]:
```
model_clf.get_layer(hidden1.name) is hidden1
```

Out[20]:
```
True
```

In [21]:
```
len(hidden1.get_weights()[1])
```

Out[21]:
```
300
```

In [22]:
```python
hidden1.get_weights()
```

Out[22]:
```
[array([[-0.00863542,  0.07263291, -0.03879336, ...,  0.00046017,
          0.04086754, -0.00439219],
        [ 0.00946854, -0.05513434, -0.01372449, ...,  0.01021195,
         -0.04177602,  0.05911638],
        [-0.02636152, -0.06728062, -0.04932404, ..., -0.017939  ,
          0.01761802, -0.04399509],
        ...,
        [ 0.01695059, -0.00667475,  0.06896693, ..., -0.02042392,
         -0.02046927,  0.01085705],
        [-0.05604234, -0.04010153,  0.00630041, ..., -0.00471536,
         -0.06079594, -0.07128973],
        [-0.02956304,  0.06041798, -0.05449025, ..., -0.06903378,
          0.0262394 , -0.02772664]], dtype=float32),
 array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)]
```

In [23]:
```python
weights, biases = hidden1.get_weights()
```

In [24]:
```python
print("shape\n",weights.shape, "\n")
weights
```

```
shape
 (784, 300)
```

Out[24]:
```
array([[-0.00863542,  0.07263291, -0.03879336, ...,  0.00046017,
         0.04086754, -0.00439219],
       [ 0.00946854, -0.05513434, -0.01372449, ...,  0.01021195,
        -0.04177602,  0.05911638],
       [-0.02636152, -0.06728062, -0.04932404, ..., -0.017939  ,
         0.01761802, -0.04399509],
       ...,
       [ 0.01695059, -0.00667475,  0.06896693, ..., -0.02042392,
        -0.02046927,  0.01085705],
       [-0.05604234, -0.04010153,  0.00630041, ..., -0.00471536,
        -0.06079594, -0.07128973],
       [-0.02956304,  0.06041798, -0.05449025, ..., -0.06903378,
         0.0262394 , -0.02772664]], dtype=float32)
```

In [25]:
```
print("shape\n", biases.shape)
```

```
shape
 (300,)
```

In [26]:
```
LOSS_FUNCTION = "sparse_categorical_crossentropy" # use => tf.losses.sparse_categorical_
OPTIMIZER = "SGD" # or use with custom learning rate=> tf.keras.optimizers.SGD(0.02)
METRICS = ["accuracy"]

model_clf.compile(loss=LOSS_FUNCTION,
              optimizer=OPTIMIZER,
              metrics=METRICS)
```

Tensorboard callback Functions

In [27]:
```
# Logging

import time

def get_log_path(log_dir="logs/fit"):
  fileName = time.strftime("log_%Y_%m_%d_%H_%M_%S")
  logs_path = os.path.join(log_dir, fileName)
  print(f"Saving logs at {logs_path}")
  return logs_path

log_dir = get_log_path()
tb_cb = tf.keras.callbacks.TensorBoard(log_dir=log_dir)
```

```
Saving logs at logs/fit\log_2023_07_26_00_44_20
```

Early Stopping callback

In [28]:
```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patCKPT_path = "Model_ckpt.h5"
checkpointing_cb = tf.keras.callbacks.ModelCheckpoint(CKPT_path, save_best_only=True)ien
```

Model checkpointing callback

In [29]:
```
CKPT_path = "Model_ckpt.h5"
checkpointing_cb = tf.keras.callbacks.ModelCheckpoint(CKPT_path, save_best_only=True)
```

In [30]:
```
# Orginal train

EPOCHS = 30
VALIDATION_SET = (X_valid, y_valid)

history = model_clf.fit(X_train, y_train, epochs=EPOCHS,
                    validation_data=VALIDATION_SET, batch_size=32, callbacks=[tb_cb, ear
```

```
Epoch 1/30
1719/1719 [==============================] - 9s 4ms/step - loss: 0.5981 - accuracy: 0.8
476 - val_loss: 0.3059 - val_accuracy: 0.9170
Epoch 2/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.2917 - accuracy: 0.9
172 - val_loss: 0.2423 - val_accuracy: 0.9336
Epoch 3/30
1719/1719 [==============================] - 8s 5ms/step - loss: 0.2407 - accuracy: 0.9
321 - val_loss: 0.2069 - val_accuracy: 0.9420
Epoch 4/30
1719/1719 [==============================] - 8s 5ms/step - loss: 0.2053 - accuracy: 0.9
412 - val_loss: 0.1821 - val_accuracy: 0.9506
Epoch 5/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.1792 - accuracy: 0.
9499 - val_loss: 0.1622 - val_accuracy: 0.9564
Epoch 6/30
1719/1719 [==============================] - 14s 8ms/step - loss: 0.1592 - accuracy: 0.
9545 - val_loss: 0.1480 - val_accuracy: 0.9598
Epoch 7/30
1719/1719 [==============================] - 13s 7ms/step - loss: 0.1424 - accuracy: 0.
9602 - val_loss: 0.1381 - val_accuracy: 0.9626
Epoch 8/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.1285 - accuracy: 0.
9631 - val_loss: 0.1270 - val_accuracy: 0.9644
Epoch 9/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.1169 - accuracy: 0.9
667 - val_loss: 0.1200 - val_accuracy: 0.9672
Epoch 10/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.1070 - accuracy: 0.
9699 - val_loss: 0.1096 - val_accuracy: 0.9714
Epoch 11/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0982 - accuracy: 0.9
720 - val_loss: 0.1061 - val_accuracy: 0.9718
Epoch 12/30
1719/1719 [==============================] - 6s 4ms/step - loss: 0.0906 - accuracy: 0.9
746 - val_loss: 0.1002 - val_accuracy: 0.9704
Epoch 13/30
1719/1719 [==============================] - 6s 4ms/step - loss: 0.0838 - accuracy: 0.9
763 - val_loss: 0.0960 - val_accuracy: 0.9742
Epoch 14/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.0776 - accuracy: 0.9
783 - val_loss: 0.0928 - val_accuracy: 0.9732
Epoch 15/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.0721 - accuracy: 0.9
799 - val_loss: 0.0907 - val_accuracy: 0.9738
Epoch 16/30
1719/1719 [==============================] - 8s 4ms/step - loss: 0.0674 - accuracy: 0.9
820 - val_loss: 0.0865 - val_accuracy: 0.9754
Epoch 17/30
1719/1719 [==============================] - 8s 5ms/step - loss: 0.0628 - accuracy: 0.9
828 - val_loss: 0.0818 - val_accuracy: 0.9760
Epoch 18/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0587 - accuracy: 0.9
841 - val_loss: 0.0797 - val_accuracy: 0.9774
Epoch 19/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0550 - accuracy: 0.9
850 - val_loss: 0.0782 - val_accuracy: 0.9778
Epoch 20/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.0516 - accuracy: 0.
```

```
9865 - val_loss: 0.0761 - val_accuracy: 0.9776
Epoch 21/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0480 - accuracy: 0.9
874 - val_loss: 0.0754 - val_accuracy: 0.9764
Epoch 22/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0453 - accuracy: 0.9
883 - val_loss: 0.0746 - val_accuracy: 0.9782
Epoch 23/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0425 - accuracy: 0.9
889 - val_loss: 0.0727 - val_accuracy: 0.9790
Epoch 24/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0400 - accuracy: 0.9
899 - val_loss: 0.0730 - val_accuracy: 0.9784
Epoch 25/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0377 - accuracy: 0.9
910 - val_loss: 0.0728 - val_accuracy: 0.9774
Epoch 26/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0355 - accuracy: 0.9
913 - val_loss: 0.0709 - val_accuracy: 0.9798
Epoch 27/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.0333 - accuracy: 0.
9920 - val_loss: 0.0702 - val_accuracy: 0.9786
Epoch 28/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0315 - accuracy: 0.9
928 - val_loss: 0.0688 - val_accuracy: 0.9786
Epoch 29/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0294 - accuracy: 0.9
933 - val_loss: 0.0675 - val_accuracy: 0.9794
Epoch 30/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0279 - accuracy: 0.9
940 - val_loss: 0.0689 - val_accuracy: 0.9806
```

In [31]:
```
# Checkpoint training

#Loading Checkpoint model
ckpt_model = tf.keras.models.load_model(CKPT_path)

history = ckpt_model.fit(X_train, y_train, epochs=EPOCHS,
                    validation_data=VALIDATION_SET, batch_size=32, callbacks=[tb_cb, ear
```

```
Epoch 1/30
1719/1719 [==============================] - 8s 4ms/step - loss: 0.0278 - accuracy: 0.9
939 - val_loss: 0.0670 - val_accuracy: 0.9794
Epoch 2/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.0264 - accuracy: 0.9
944 - val_loss: 0.0673 - val_accuracy: 0.9794
Epoch 3/30
1719/1719 [==============================] - 6s 4ms/step - loss: 0.0248 - accuracy: 0.9
947 - val_loss: 0.0669 - val_accuracy: 0.9804
Epoch 4/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.0233 - accuracy: 0.9
952 - val_loss: 0.0694 - val_accuracy: 0.9798
Epoch 5/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.0221 - accuracy: 0.9
960 - val_loss: 0.0681 - val_accuracy: 0.9798
Epoch 6/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0208 - accuracy: 0.9
962 - val_loss: 0.0675 - val_accuracy: 0.9808
Epoch 7/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.0198 - accuracy: 0.9
967 - val_loss: 0.0664 - val_accuracy: 0.9802
Epoch 8/30
1719/1719 [==============================] - 11s 6ms/step - loss: 0.0186 - accuracy: 0.
9972 - val_loss: 0.0662 - val_accuracy: 0.9802
Epoch 9/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.0177 - accuracy: 0.
9972 - val_loss: 0.0667 - val_accuracy: 0.9798
Epoch 10/30
1719/1719 [==============================] - 11s 7ms/step - loss: 0.0169 - accuracy: 0.
9973 - val_loss: 0.0698 - val_accuracy: 0.9798
Epoch 11/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.0160 - accuracy: 0.
9977 - val_loss: 0.0660 - val_accuracy: 0.9800
Epoch 12/30
1719/1719 [==============================] - 13s 7ms/step - loss: 0.0150 - accuracy: 0.
9981 - val_loss: 0.0661 - val_accuracy: 0.9806
Epoch 13/30
1719/1719 [==============================] - 12s 7ms/step - loss: 0.0143 - accuracy: 0.
9982 - val_loss: 0.0676 - val_accuracy: 0.9804
Epoch 14/30
1719/1719 [==============================] - 11s 6ms/step - loss: 0.0136 - accuracy: 0.
9983 - val_loss: 0.0666 - val_accuracy: 0.9810
Epoch 15/30
1719/1719 [==============================] - 11s 6ms/step - loss: 0.0130 - accuracy: 0.
9985 - val_loss: 0.0660 - val_accuracy: 0.9812
Epoch 16/30
1719/1719 [==============================] - 11s 6ms/step - loss: 0.0123 - accuracy: 0.
9987 - val_loss: 0.0666 - val_accuracy: 0.9810
```

Saving the Model

In [32]:
```
import time
import os

def save_model_path(MODEL_dir = "TRAINED_MODEL"):
  os.makedirs(MODEL_dir, exist_ok= True)
  fileName = time.strftime("Model_%Y_%m_%d_%H_%M_%S_.h5")
  model_path = os.path.join(MODEL_dir, fileName)
  print(f"Model {fileName} will be saved at {model_path}")
  return model_path
```

In [33]:
```
UNIQUE_PATH = save_model_path()
UNIQUE_PATH
```

```
Model Model_2023_07_26_00_53_52_.h5 will be saved at TRAINED_MODEL\Model_2023_07_26_00_
53_52_.h5
```

Out[33]:
```
'TRAINED_MODEL\\Model_2023_07_26_00_53_52_.h5'
```

In [34]:
```
tf.keras.models.save_model(model_clf, UNIQUE_PATH)
```

In [35]:
```
history.params
```

Out[35]:
```
{'verbose': 1, 'epochs': 30, 'steps': 1719}
```

In [36]:
```
# history.history
```

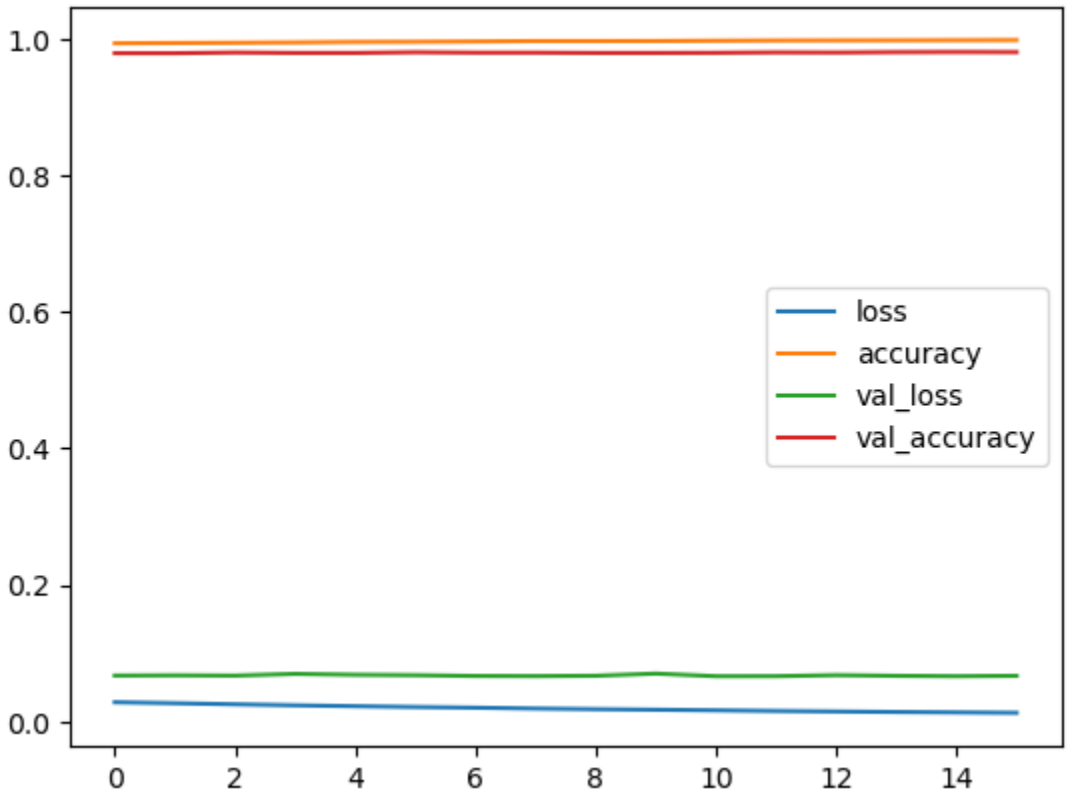
In [37]:
```
pd.DataFrame(history.history)
```

Out[37]:

	loss	accuracy	val_loss	val_accuracy
0	0.027838	0.993945	0.066974	0.9794
1	0.026399	0.994364	0.067307	0.9794
2	0.024844	0.994709	0.066889	0.9804
3	0.023314	0.995218	0.069436	0.9798
4	0.022058	0.995982	0.068107	0.9798
5	0.020838	0.996182	0.067498	0.9808
6	0.019833	0.996655	0.066364	0.9802
7	0.018600	0.997200	0.066153	0.9802
8	0.017650	0.997182	0.066678	0.9798
9	0.016868	0.997273	0.069837	0.9798
10	0.015999	0.997709	0.066010	0.9800
11	0.015030	0.998073	0.066082	0.9806
12	0.014267	0.998218	0.067568	0.9804
13	0.013561	0.998345	0.066585	0.9810
14	0.012970	0.998509	0.066026	0.9812
15	0.012334	0.998727	0.066566	0.9810

In [38]:
```
pd.DataFrame(history.history).plot()
```

Out[38]: <Axes: >



In [39]:
```
model_clf.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.0693 - accuracy: 0.977
7
```

Out[39]:
```
[0.06929262727499008, 0.9776999950408936]
```

In [40]:
```
x_new = X_test[:3]
# x_new
```

In [41]:
```
actual = y_test[:3]
actual
```

Out[41]:
```
array([7, 2, 1], dtype=uint8)
```

In [42]:
```
y_prob = model_clf.predict(x_new)
y_prob.round(3)
```

```
1/1 [==============================] - 0s 302ms/step
```

Out[42]:
```
array([[0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 1.   , 0.   ,
        0.   ],
       [0.   , 0.   , 0.999, 0.001, 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   ],
       [0.   , 0.998, 0.   , 0.   , 0.   , 0.   , 0.   , 0.001, 0.001,
        0.   ]], dtype=float32)
```

In [43]:
```
y_prob
```

Out[43]:
```
array([[1.5794069e-06, 7.6853473e-07, 1.0801427e-04, 1.3993130e-04,
        5.5980118e-09, 2.3529131e-07, 2.5731752e-12, 9.9971408e-01,
        9.0249375e-07, 3.4513163e-05],
       [2.5457732e-06, 1.9585187e-04, 9.9901319e-01, 7.7849202e-04,
        1.9284004e-12, 2.7663469e-07, 1.9941704e-06, 1.1564657e-09,
        7.6111187e-06, 3.1541395e-13],
       [2.6645846e-06, 9.9758554e-01, 6.5624081e-05, 3.2423293e-05,
        2.2758909e-04, 8.4465260e-05, 8.3897270e-05, 8.2147971e-04,
        1.0715602e-03, 2.4860985e-05]], dtype=float32)
```

In [44]:
```
y_pred = np.argmax(y_prob, axis = -1)
```

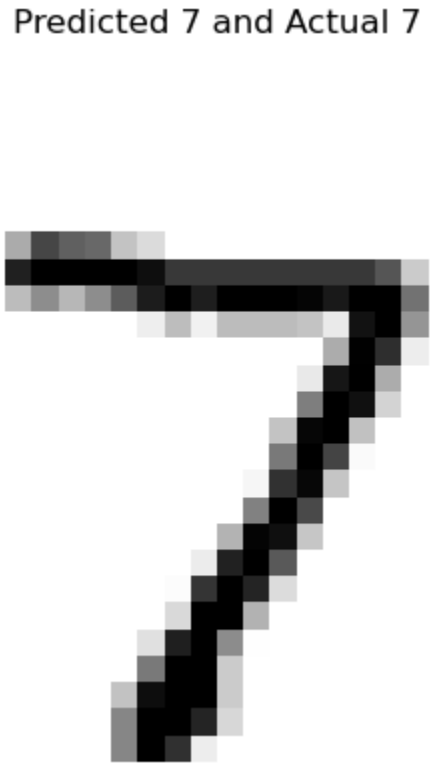
In [45]:
```
y_pred
```

Out[45]:
```
array([7, 2, 1], dtype=int64)
```

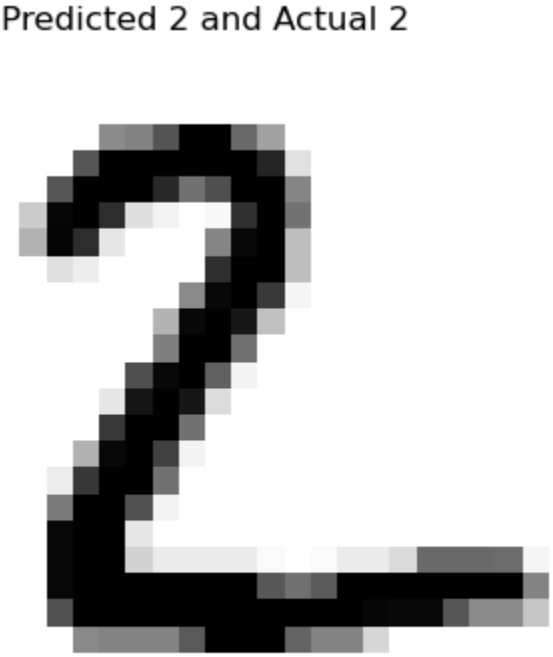
In [46]:
```
actual
```

Out[46]:
```
array([7, 2, 1], dtype=uint8)
```

In [47]:
```
# plot
for data, pred, actual_data in zip(x_new, y_pred, actual):
  plt.imshow(data, cmap="binary")
  plt.title(f"Predicted {pred} and Actual {actual_data}")
  plt.axis("off")
  plt.show()
  print("######################")
```

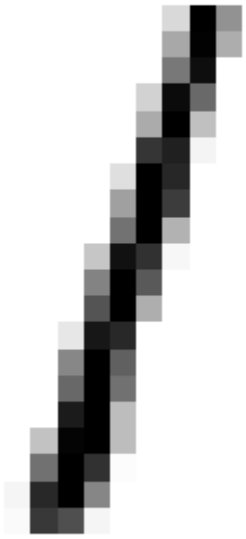


######################



######################

Predicted 1 and Actual 1



######################

In [ ]:

Assignment Questions

Q1. Explain the concept of batch normalization in the context of Artificial Neural Networks.

Batch normalization is a technique used in Artificial Neural Networks to normalize the inputs of each layer during training. It aims to stabilize and speed up the training process by reducing internal covariate shift. Internal covariate shift refers to the change in the distribution of each layer's inputs during training, which can slow down learning and require more careful tuning of hyperparameters.

Batch normalization addresses this issue by normalizing the inputs of each layer to have zero mean and unit variance. It does this by computing the mean and variance of the inputs within a mini-batch (a small subset of the training data) and then applying a normalization transformation. Additionally, batch normalization introduces learnable parameters, scale, and shift, which allow the network to learn the optimal mean and variance for each layer. These parameters give the model more flexibility to adapt to the data distribution.

Q2. Describe the benefits of using batch normalization during training.

Using batch normalization during training offers several benefits:

- Improved training stability: Batch normalization helps to mitigate the vanishing and exploding gradient problems, making it easier for deep neural networks to converge during training.
- Faster convergence: By reducing internal covariate shift, batch normalization accelerates the training process, leading to faster convergence and fewer training iterations required.
- Reduced sensitivity to initialization: Batch normalization makes neural networks less sensitive to the choice of weight initialization, allowing for a wider range of initialization strategies.
- Regularization effect: Batch normalization acts as a form of regularization, reducing the need for other regularization techniques like dropout.
- Allows for larger learning rates: The improved stability provided by batch normalization allows for the use of larger learning rates, which can speed up training further.

Q3. Discuss the working principle of batch normalization, including the normalization step and the learnable parameters.

The working principle of batch normalization involves two main steps: normalization and learnable parameters.

Normalization Step: For each mini-batch of input data during training, batch normalization computes the mean and variance of the data within that mini-batch. It then normalizes the data by subtracting the mean and dividing by the standard deviation, resulting in inputs with zero mean and unit variance:

x_normalized = (x - mean) / sqrt(variance + epsilon)

where x is the input data, mean and variance are the batch-wise statistics, and epsilon is a small constant added to avoid division by zero.

Learnable Parameters: Batch normalization introduces two learnable parameters for each layer: scale (gamma) and shift (beta). These parameters are applied after normalization and allow the model to learn the optimal scaling and shifting of the normalized inputs. The transformed output is given by:

y = gamma * x_normalized + beta

Impementation

In [1]:
```
# Importing Libraries
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import os
```

Creating a simple classifier using keras on MNIST data

In [2]:
```
mnist = tf.keras.datasets.mnist
(X_train_full, y_train_full), (X_test, y_test) = mnist.load_data()
```

In [3]:
```
print(f"data type of X_train_full: {X_train_full.dtype},\n shape of X_train_full: {X_tra
```

```
data type of X_train_full: uint8,
 shape of X_train_full: (60000, 28, 28)
```

In [4]:
```
X_test.shape
```

Out[4]:
```
(10000, 28, 28)
```

In [5]:
```
len(X_test[1][0])
```

Out[5]:
```
28
```

In [6]:
```
X_train_full[0]
```

```
Out[6]: array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,
                 18,  18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170,
                253, 253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253,
                253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253,
                253, 198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253,
                205,  11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,  14,   1, 154, 253,
                 90,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 139, 253,
                190,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 190,
                253,  70,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  35,
                241, 225, 160, 108,   1,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                 81, 240, 253, 253, 119,  25,   0,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,  45, 186, 253, 253, 150,  27,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0,  16,  93, 252, 253, 187,   0,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0,   0,   0, 249, 253, 249,  64,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
                  0,  46, 130, 183, 253, 253, 207,   2,   0,   0,   0,   0,   0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  39,
                148, 229, 253, 253, 253, 250, 182,   0,   0,   0,   0,   0,   0,
```

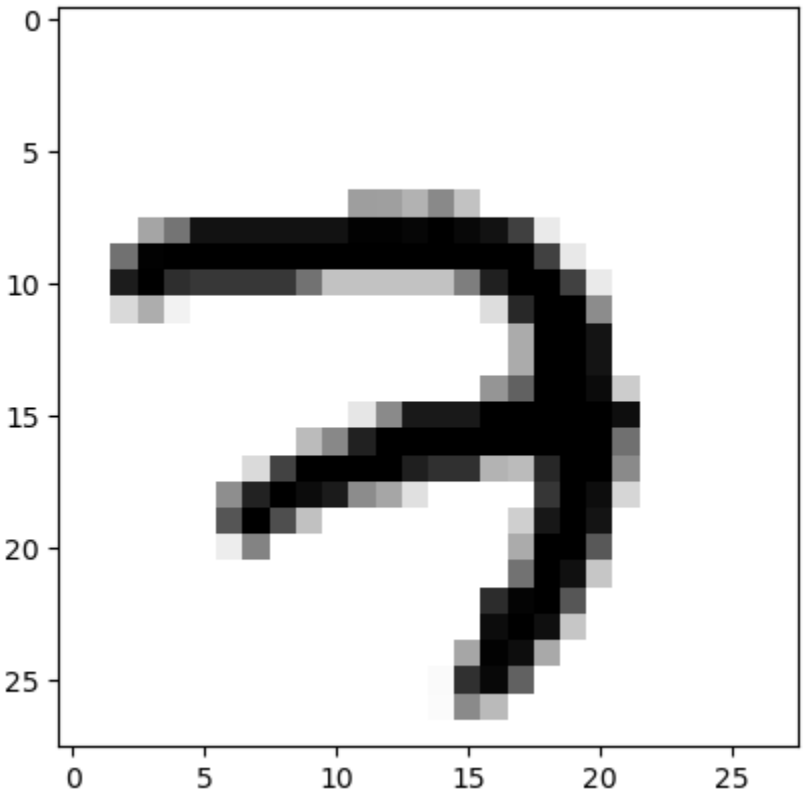
```
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 114, 221,
 253, 253, 253, 253, 201,  78,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,  23,  66, 213, 253, 253,
 253, 253, 198,  81,   2,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,  18, 171, 219, 253, 253, 253, 253,
 195,  80,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,  55, 172, 226, 253, 253, 253, 253, 244, 133,
  11,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0, 136, 253, 253, 253, 212, 135, 132,  16,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0]], dtype=uint8)
```

In [7]:
```
# create a validation data set from the full training data
# Scale the data between 0 to 1 by dividing it by 255. as its an unsigned data between 0
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

# scale the test set as well
X_test = X_test / 255.
```

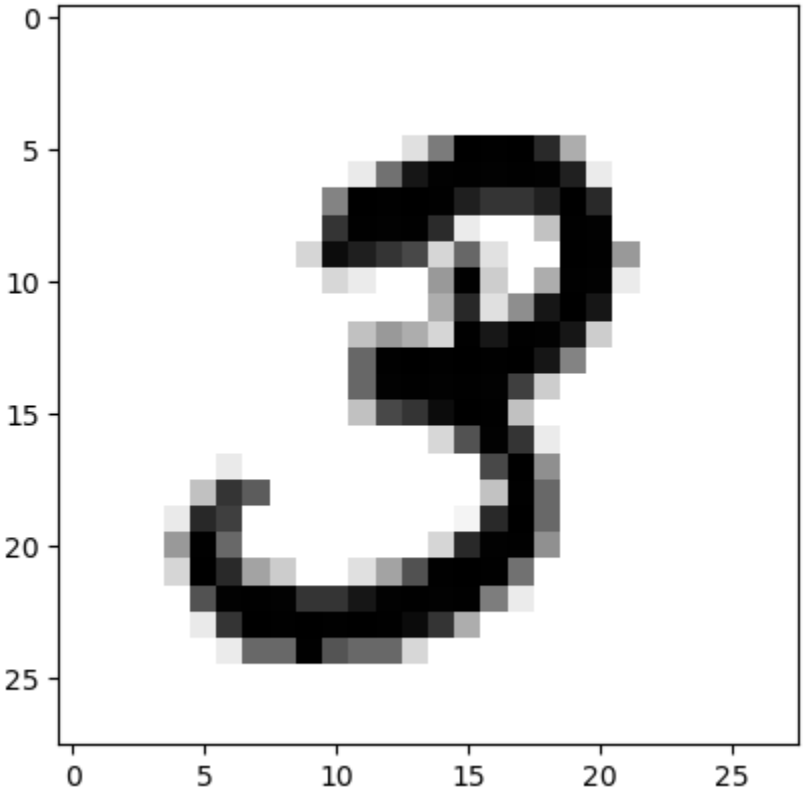
In [8]:
```
len(X_train_full[5000:] )
```

Out[8]: 55000

In [9]:
```
# Lets view some data
plt.imshow(X_train[0], cmap="binary")
plt.show()
```

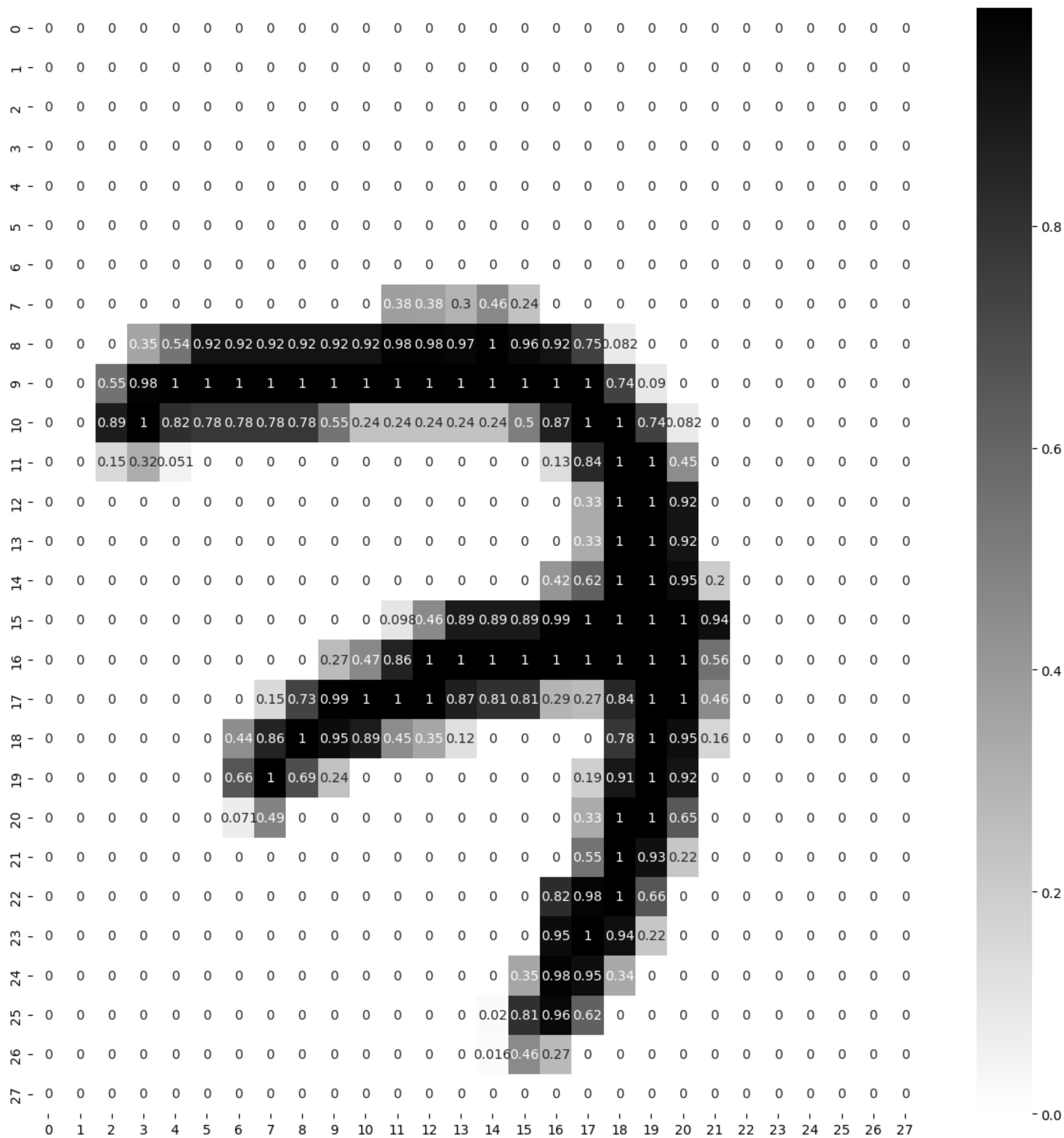


In [10]:
```
# Lets view some data
plt.imshow(X_train[1], cmap="binary")
plt.show()
```



In [11]:
```
plt.figure(figsize=(15,15))
sns.heatmap(X_train[0], annot=True, cmap="binary")
```

Out[11]: <Axes: >



In [ ]:
```
SO Input is : 28 * 28 :784 : flattern the input layeers

And we have 10 Output:[0,1,2,3,4,5,6,7,8,9]  ## Softmax is used their because it tell pr
```

Without Batch Normalization

In [12]:
```
# Creating Layers of ANN
LAYERS = [tf.keras.layers.Flatten(input_shape=[28, 28], name="inputLayer"),
          tf.keras.layers.Dense(300, activation="relu", name="hiddenLayer1"),
          tf.keras.layers.Dense(100, activation="relu", name="hiddenLayer2"),
          tf.keras.layers.Dense(10, activation="softmax", name="outputLayer")]

model_clf_without_bn = tf.keras.models.Sequential(LAYERS)
```

In [13]:
```
model_clf_without_bn.layers
```

Out[13]:
```
[<keras.layers.reshaping.flatten.Flatten at 0x21939ca6da0>,
 <keras.layers.core.dense.Dense at 0x21939ca6740>,
 <keras.layers.core.dense.Dense at 0x21939ca6500>,
 <keras.layers.core.dense.Dense at 0x21939ca43a0>]
```

In [14]:
```
model_clf_without_bn.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inputLayer (Flatten)        (None, 784)               0

 hiddenLayer1 (Dense)        (None, 300)               235500

 hiddenLayer2 (Dense)        (None, 100)               30100

 outputLayer (Dense)         (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

In [15]:
```
# first Layer * second Layer + bias
784*300 + 300, 300*100+100, 100*10+100
```

Out[15]:
```
(235500, 30100, 1100)
```

In [16]:
```
# Total parameters to be trained
sum((235500, 30100, 1010))
```

Out[16]:
```
266610
```

In [17]:
```
LOSS_FUNCTION = "sparse_categorical_crossentropy" # use => tf.losses.sparse_categorical_
OPTIMIZER = "ADAM" # or use with custom learning rate=> tf.keras.optimizers.SGD(0.02)
METRICS = ["accuracy"]


# Record starting time
start_time = time.time()

model_clf_without_bn.compile(loss=LOSS_FUNCTION,
              optimizer=OPTIMIZER,
              metrics=METRICS)
```

In [18]:
```
# training

EPOCHS = 10
VALIDATION_SET = (X_valid, y_valid)

# Record starting time
start_time = time.time()

history = model_clf_without_bn.fit(X_train, y_train, epochs=EPOCHS,
                    validation_data=VALIDATION_SET, batch_size=64)

# Record ending time
end_time = time.time()
 # Calculate training time
training_time = end_time - start_time
print(f"Total training time: {training_time:.2f} seconds")
```

```
Epoch 1/10
860/860 [==============================] - 6s 5ms/step - loss: 0.2387 - accuracy: 0.930
3 - val_loss: 0.1005 - val_accuracy: 0.9700
Epoch 2/10
860/860 [==============================] - 4s 4ms/step - loss: 0.0893 - accuracy: 0.972
8 - val_loss: 0.0895 - val_accuracy: 0.9714
Epoch 3/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0595 - accuracy: 0.981
7 - val_loss: 0.0751 - val_accuracy: 0.9762
Epoch 4/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0426 - accuracy: 0.986
1 - val_loss: 0.0703 - val_accuracy: 0.9814
Epoch 5/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0313 - accuracy: 0.989
2 - val_loss: 0.0843 - val_accuracy: 0.9780
Epoch 6/10
860/860 [==============================] - 5s 6ms/step - loss: 0.0278 - accuracy: 0.990
8 - val_loss: 0.0726 - val_accuracy: 0.9818
Epoch 7/10
860/860 [==============================] - 5s 6ms/step - loss: 0.0204 - accuracy: 0.993
3 - val_loss: 0.0752 - val_accuracy: 0.9812
Epoch 8/10
860/860 [==============================] - 5s 5ms/step - loss: 0.0159 - accuracy: 0.994
8 - val_loss: 0.0866 - val_accuracy: 0.9808
Epoch 9/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0155 - accuracy: 0.994
6 - val_loss: 0.0835 - val_accuracy: 0.9776
Epoch 10/10
860/860 [==============================] - 4s 4ms/step - loss: 0.0135 - accuracy: 0.995
4 - val_loss: 0.0837 - val_accuracy: 0.9814
```

In [19]:
```
history.params
```

Out[19]:
```
{'verbose': 1, 'epochs': 10, 'steps': 860}
```

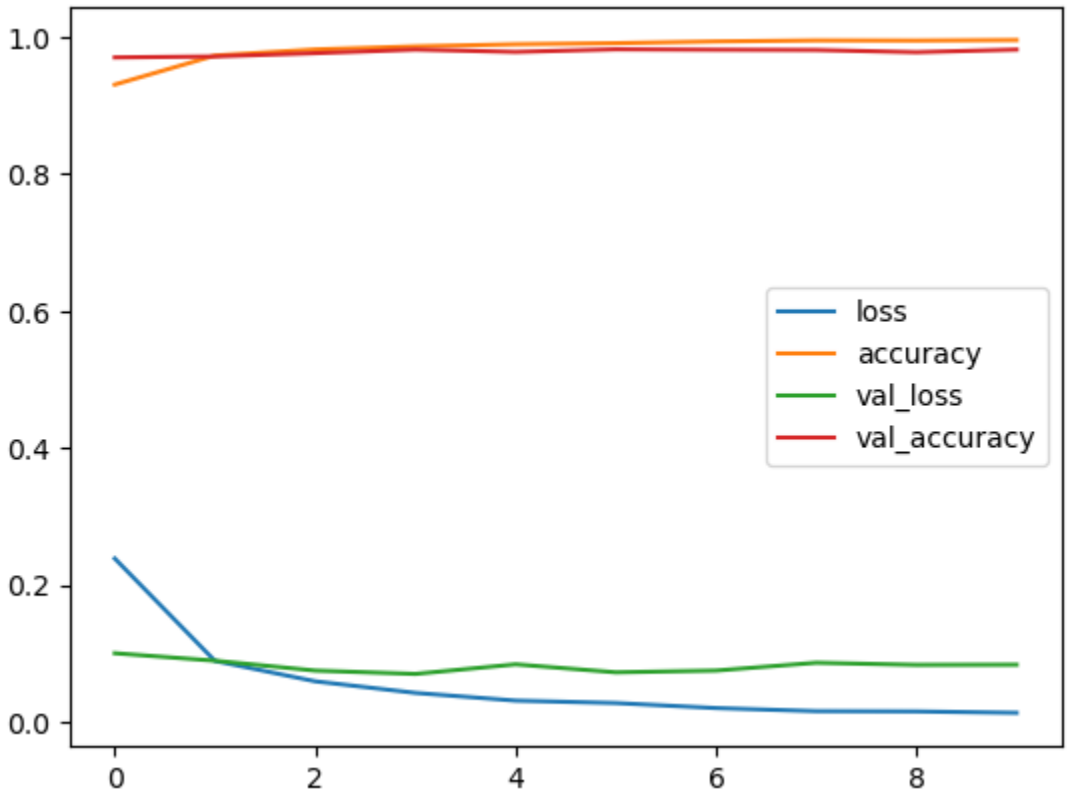
In [20]:
```
pd.DataFrame(history.history)
```

Out[20]:

	loss	accuracy	val_loss	val_accuracy
0	0.238738	0.930345	0.100475	0.9700
1	0.089264	0.972782	0.089536	0.9714
2	0.059478	0.981673	0.075137	0.9762
3	0.042577	0.986127	0.070268	0.9814
4	0.031306	0.989236	0.084280	0.9780
5	0.027780	0.990764	0.072608	0.9818
6	0.020448	0.993327	0.075167	0.9812
7	0.015920	0.994818	0.086569	0.9808
8	0.015546	0.994618	0.083470	0.9776
9	0.013514	0.995400	0.083705	0.9814

In [21]:
```
pd.DataFrame(history.history).plot()
```

Out[21]:
```
<Axes: >
```



In [23]:
```
# Evaluate both models on the test dataset
loss_without_bn, accuracy_without_bn = model_clf_without_bn.evaluate(X_test, y_test, ver

print("Model without Batch Normalization:")
print(f"Accuracy: {accuracy_without_bn*100:.2f}%, Loss: {loss_without_bn:.4f}")
```

```
Model without Batch Normalization:
Accuracy: 97.92%, Loss: 0.0877
```

With Batch Normalization

In [24]:
```
# Define the layers of the model with batch normalization
LAYERS1 = [tf.keras.layers.Flatten(input_shape=[28, 28], name="inputLayer"),
           tf.keras.layers.Dense(300, activation="relu", name="hiddenLayer1"),
           tf.keras.layers.BatchNormalization(name="batchNorm1"),
           tf.keras.layers.Dense(100, activation="relu", name="hiddenLayer2"),
           tf.keras.layers.BatchNormalization(name="batchNorm2"),
           tf.keras.layers.Dense(10, activation="softmax", name="outputLayer")]

model_clf_with_bn = tf.keras.models.Sequential(LAYERS1)
```

In [25]:
```
model_clf_with_bn.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inputLayer (Flatten)        (None, 784)               0

 hiddenLayer1 (Dense)        (None, 300)               235500

 batchNorm1 (BatchNormalizat  (None, 300)              1200
 ion)

 hiddenLayer2 (Dense)        (None, 100)               30100

 batchNorm2 (BatchNormalizat  (None, 100)              400
 ion)

 outputLayer (Dense)         (None, 10)                1010

=================================================================
Total params: 268,210
Trainable params: 267,410
Non-trainable params: 800
_________________________________________________________________
```

In [26]:
```
LOSS_FUNCTION = "sparse_categorical_crossentropy" # use => tf.losses.sparse_categorical_
OPTIMIZER = "ADAM" # or use with custom learning rate=> tf.keras.optimizers.SGD(0.02)
METRICS = ["accuracy"]



model_clf_with_bn.compile(loss=LOSS_FUNCTION,
              optimizer=OPTIMIZER,
              metrics=METRICS)
```

In [28]:
```
# training
import time

EPOCHS = 10
VALIDATION_SET = (X_valid, y_valid)

# Record starting time
start_time = time.time()

history = model_clf_with_bn.fit(X_train, y_train, epochs=EPOCHS,
                    validation_data=VALIDATION_SET, batch_size=64)

# Record ending time
end_time = time.time()
# Calculate training time
training_time = end_time - start_time
print(f"Total training time: {training_time:.2f} seconds")
```

```
Epoch 1/10
860/860 [==============================] - 8s 7ms/step - loss: 0.2105 - accuracy: 0.937
2 - val_loss: 0.0989 - val_accuracy: 0.9702
Epoch 2/10
860/860 [==============================] - 6s 7ms/step - loss: 0.0910 - accuracy: 0.972
2 - val_loss: 0.0835 - val_accuracy: 0.9756
Epoch 3/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0654 - accuracy: 0.979
1 - val_loss: 0.0852 - val_accuracy: 0.9726
Epoch 4/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0488 - accuracy: 0.984
3 - val_loss: 0.0738 - val_accuracy: 0.9774
Epoch 5/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0426 - accuracy: 0.985
8 - val_loss: 0.0756 - val_accuracy: 0.9786
Epoch 6/10
860/860 [==============================] - 5s 6ms/step - loss: 0.0369 - accuracy: 0.988
0 - val_loss: 0.0636 - val_accuracy: 0.9800
Epoch 7/10
860/860 [==============================] - 5s 5ms/step - loss: 0.0288 - accuracy: 0.990
2 - val_loss: 0.0733 - val_accuracy: 0.9802
Epoch 8/10
860/860 [==============================] - 5s 5ms/step - loss: 0.0281 - accuracy: 0.990
2 - val_loss: 0.0687 - val_accuracy: 0.9802
Epoch 9/10
860/860 [==============================] - 5s 5ms/step - loss: 0.0224 - accuracy: 0.992
2 - val_loss: 0.0764 - val_accuracy: 0.9810
Epoch 10/10
860/860 [==============================] - 5s 5ms/step - loss: 0.0203 - accuracy: 0.993
0 - val_loss: 0.0688 - val_accuracy: 0.9802
Total training time: 49.95 seconds
```

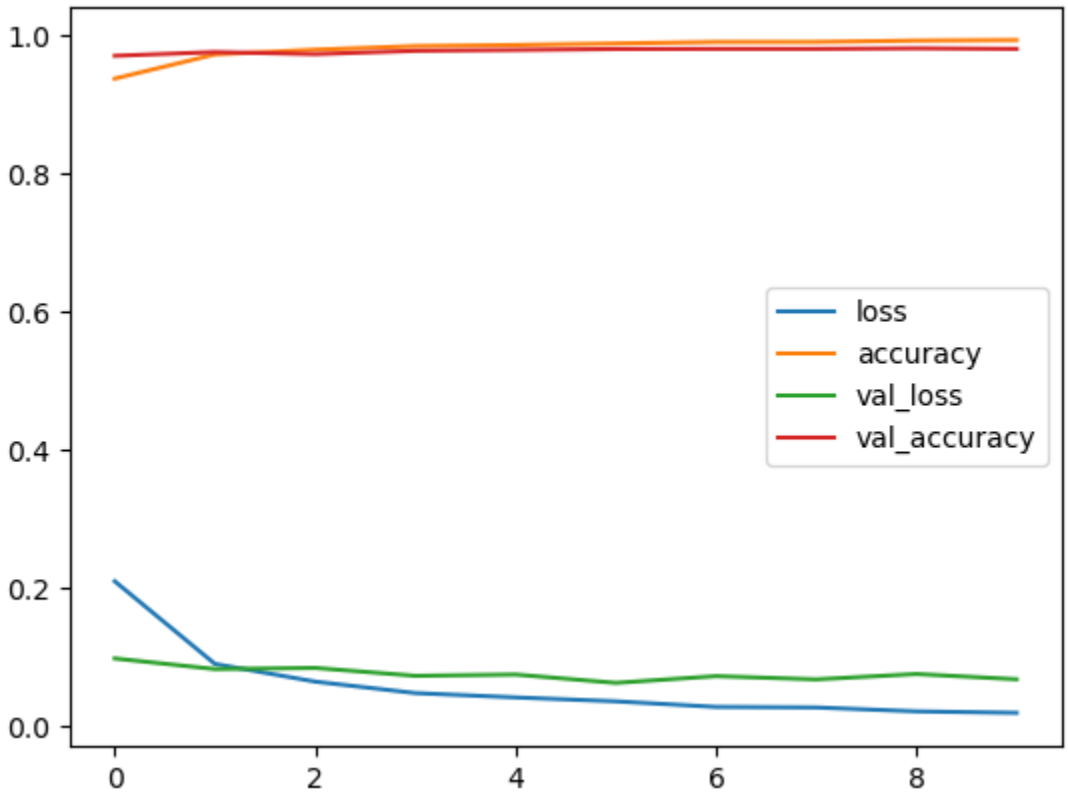
In [31]:
```
pd.DataFrame(history.history)
```

Out[31]:

	loss	accuracy	val_loss	val_accuracy
0	0.210480	0.937218	0.098913	0.9702
1	0.090966	0.972236	0.083534	0.9756
2	0.065417	0.979055	0.085216	0.9726
3	0.048829	0.984273	0.073797	0.9774
4	0.042590	0.985800	0.075644	0.9786
5	0.036925	0.988018	0.063586	0.9800
6	0.028844	0.990164	0.073334	0.9802
7	0.028057	0.990236	0.068650	0.9802
8	0.022416	0.992218	0.076421	0.9810
9	0.020310	0.993018	0.068799	0.9802

In [32]:
```
pd.DataFrame(history.history).plot()
```

Out[32]: <Axes: >



In [30]:
```
loss_with_bn, accuracy_with_bn = model_clf_with_bn.evaluate(X_test, y_test, verbose=0)

print("Model with Batch Normalization:")
print(f"Accuracy: {accuracy_with_bn*100:.2f}%, Loss: {loss_with_bn:.4f}")
```

```
Model with Batch Normalization:
Accuracy: 97.96%, Loss: 0.0748
```

SO the Difference is :

In [37]:
```
# Evaluate both models on the test dataset
loss_without_bn, accuracy_without_bn = model_clf_without_bn.evaluate(X_test, y_test, ver
loss_with_bn, accuracy_with_bn = model_clf_with_bn.evaluate(X_test, y_test, verbose=0)

print("Model without Batch Normalization:")
print(f"Accuracy: {accuracy_without_bn*100:.2f}%, Loss: {loss_without_bn:.4f}")

print("Model with Batch Normalization:")
print(f"Accuracy: {accuracy_with_bn*100:.2f}%, Loss: {loss_with_bn:.4f}")
```

```
Model without Batch Normalization:
Accuracy: 97.92%, Loss: 0.0877
Model with Batch Normalization:
Accuracy: 97.96%, Loss: 0.0748
```

The provided experiment results show the comparison of two models trained on the same dataset using different batch sizes. The models were trained with and without batch normalization. Let's analyze the effects of different batch sizes on training dynamics and model performance.

Experiment Results:

- Model without Batch Normalization:
- Accuracy: 97.92%
- Loss: 0.0877
- Model with Batch Normalization:
- Accuracy: 97.96%
- Loss: 0.0748

Observations:

Both models achieve high accuracy, indicating that they are able to effectively learn from the data and make accurate predictions on unseen samples. The model with batch normalization slightly outperforms the model without batch normalization in terms of accuracy and loss. This suggests that batch normalization has provided some improvement in the model's performance.

Testing The Model

In [39]:
```
x_new = X_test[:3]
x_new
```

Out[39]:
```
array([[[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]],

       [[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]],

       [[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]]])
```

In [40]:
```
actual = y_test[:3]
actual
```

Out[40]:
```
array([7, 2, 1], dtype=uint8)
```

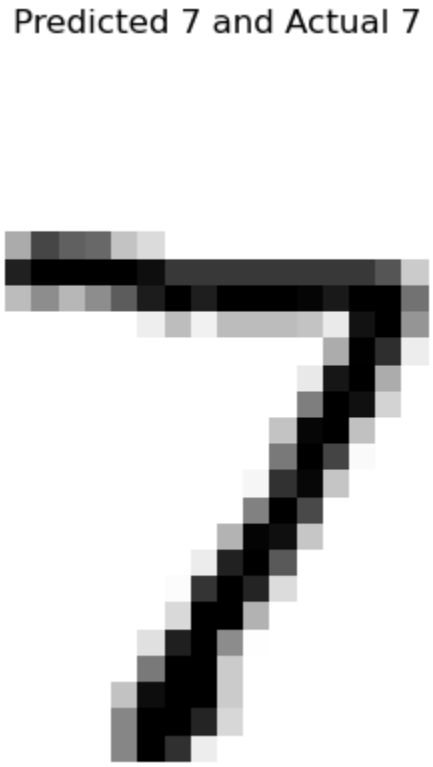
In [41]:
```
y_prob = model_clf_with_bn.predict(x_new)
y_prob.round(3)
```

```
1/1 [==============================] - 0s 307ms/step
```

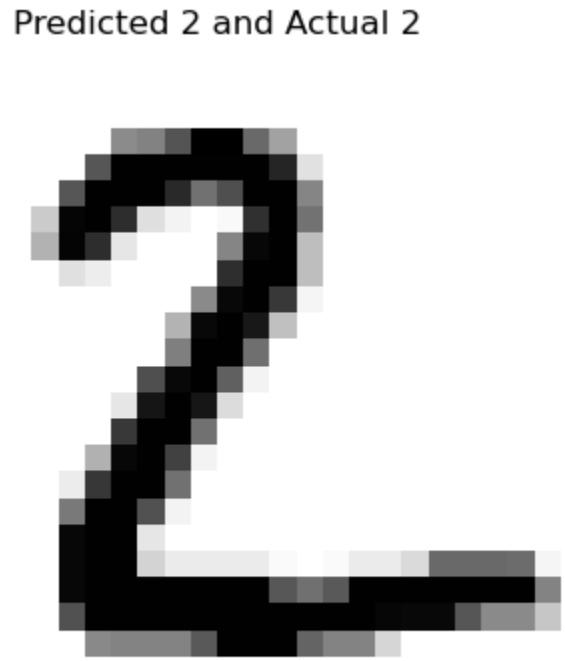
Out[41]:
```
array([[0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 1.   , 0.   ,
        0.   ],
       [0.   , 0.   , 1.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   ],
       [0.   , 0.998, 0.   , 0.   , 0.   , 0.001, 0.   , 0.   , 0.   ,
        0.   ]], dtype=float32)
```

In [43]:
```
y_pred = np.argmax(y_prob, axis = -1)
```

In [44]:
```
# plot
for data, pred, actual_data in zip(x_new, y_pred, actual):
  plt.imshow(data, cmap="binary")
  plt.title(f"Predicted {pred} and Actual {actual_data}")
  plt.axis("off")
  plt.show()
  print("######################")
```

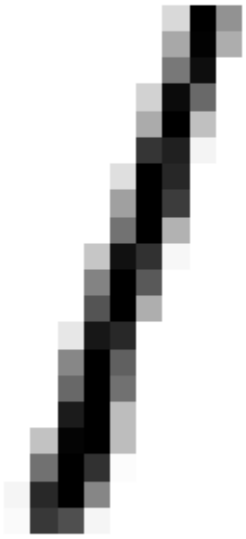



```
######################
```



######################

Predicted 1 and Actual 1



######################

----------Done----------

Deep Learning Applications Using Python:
https://t.me/AIMLDeepThaught/675

Assignment Questions

In [34]:
```
# Importing Libraries
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import os
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

In [35]:
```
# Checking version of Tensorflow and Keras
print(f"Tensorflow Version {tf.__version__}")
print(f"Keras Version {tf.keras.__version__}")
```

```
Tensorflow Version 2.10.0
Keras Version 2.10.0
```

In [36]:
```
df=pd.read_csv("wine.csv")
```

In [37]:
```
df.head()
```

Out[37]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	bad
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	bad
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	bad
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	good
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	bad

In [38]:
```
df.shape
```

Out[38]:
```
(1599, 12)
```

In [39]:
```
df.columns
```

Out[39]:
```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'quality'],
      dtype='object')
```

In [40]:
```
# Calculate class distribution
class_distribution = df['quality'].value_counts()

# Display the class distribution
print(class_distribution)

# Check if the target variable is imbalanced
if len(class_distribution) == 2:
    majority_class_count = max(class_distribution)
    minority_class_count = min(class_distribution)
    class_ratio = majority_class_count / minority_class_count

    if class_ratio > 2:
        print("The target variable is imbalanced.")
    else:
        print("The target variable is not imbalanced.")
else:
    print("The target variable is not binary.")
```

```
good    855
bad     744
Name: quality, dtype: int64
The target variable is not imbalanced.
```

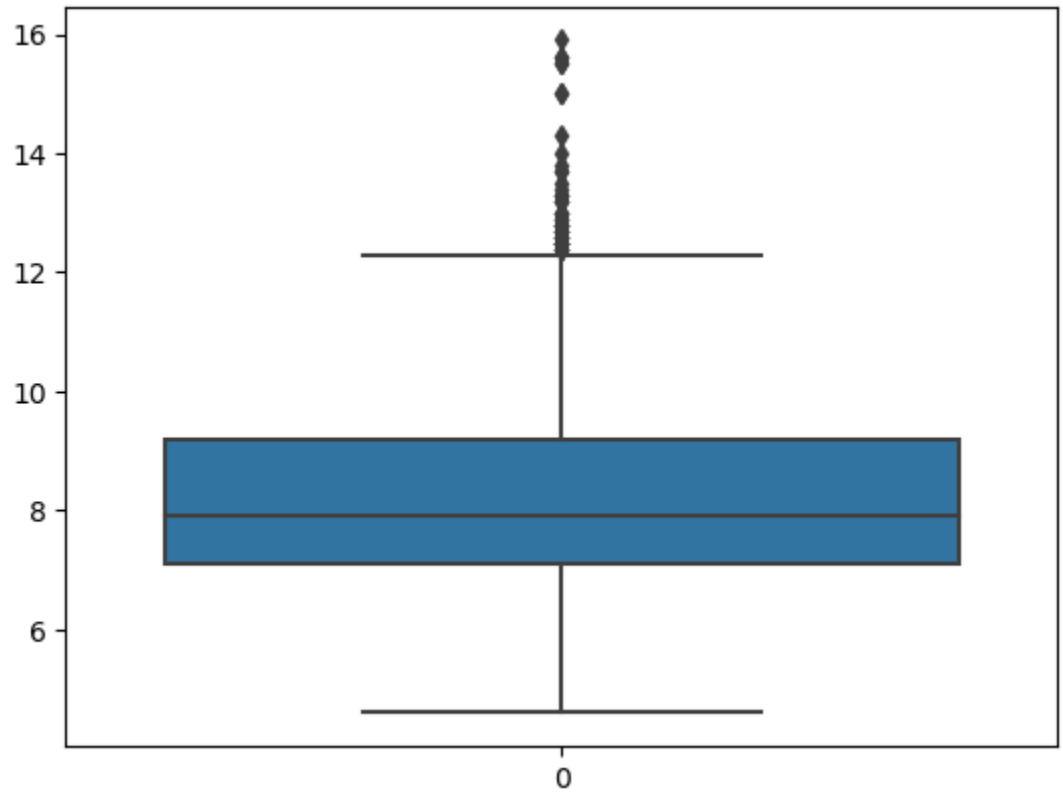
As Our target vaiable is not imbalanced

In [41]:
```
print('Checking for Null values in the dataframe:','\n',df.isnull().sum(),'\n')
```

```
Checking for Null values in the dataframe:
 fixed acidity          0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

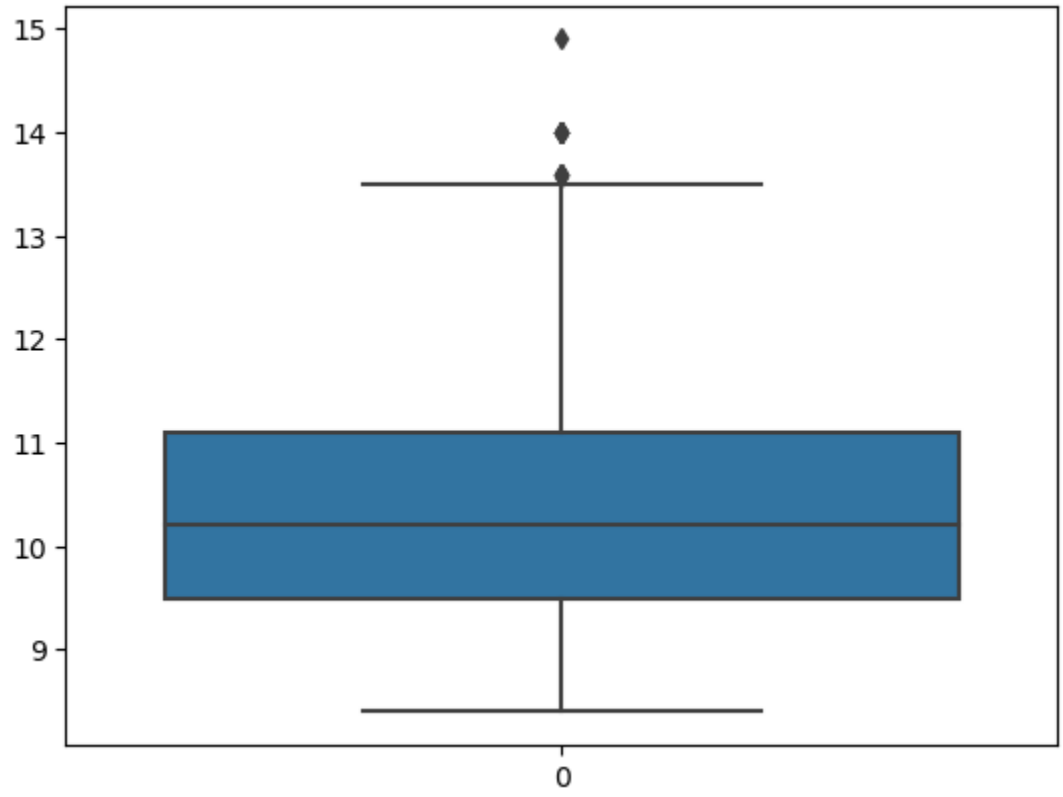
In [42]:
```
sns.boxplot(df['fixed acidity'])
```

Out[42]: <Axes: >



In [43]:
```
sns.boxplot(df['alcohol'])
```

Out[43]: <Axes: >



As we have some Outliers in data , But we are using Deep Learning Model so dont worry

In [44]:
```
y = df.quality
X = df.drop(columns = ['quality'])
```

In [45]:
```
X.shape
```

Out[45]:
```
(1599, 11)
```

In [46]:
```
X.head()
```

Out[46]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4

In [47]:
```
y.head()
```

Out[47]:
```
0     bad
1     bad
2     bad
3    good
4     bad
Name: quality, dtype: object
```

As we have our Target Feature in "Good" or "Bad" so we use Label encoder

In [48]:
```
# Let's perform categorical features encoding:
from sklearn.preprocessing import LabelEncoder
LE = LabelEncoder()
```

In [49]:
```
# Target_feature Encoding:
y = LE.fit_transform(y)
```

In [50]:
```
y  # 0= bad 1=Good
```

Out[50]:
```
array([0, 0, 0, ..., 1, 0, 1])
```

In [51]:
```
X_train_full, X_test, y_train_full, y_test = train_test_split(X,y, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full,y_train_full, random_
```

In [52]:
```
print(X_train_full.shape)
print(X_test.shape)
print(X_train.shape)
print(X_valid.shape)
```

```
(1199, 11)
(400, 11)
(899, 11)
(300, 11)
```

In [53]:
```
X_train.shape[1:]
```

Out[53]: (11,)

In [54]:
```
X_train.shape[1:]
```

Out[54]: (11,)

In [55]:
```
y_train.shape[:1]
```

Out[55]: (899,)

In [56]:
```
# Creating layers of ANN
LAYERS = [
    tf.keras.layers.Dense(30, activation="relu", name="HiddenLayer1", input_shape=X_trai
    tf.keras.layers.Dense(10, activation="relu", name="HiddenLayer2"),
    tf.keras.layers.Dense(5, activation='relu', name="HiddenLayer3"),
    tf.keras.layers.Dense(1, activation="sigmoid", name="OutputLayer")
]

# Create the model using Sequential API
model = tf.keras.models.Sequential(LAYERS)
```

In [57]:
```
# Logging

import time

def get_log_path(log_dir="logs/fit"):
  fileName = time.strftime("log_%Y_%m_%d_%H_%M_%S")
  logs_path = os.path.join(log_dir, fileName)
  print(f"Saving logs at {logs_path}")
  return logs_path

log_dir = get_log_path()
tb_cb = tf.keras.callbacks.TensorBoard(log_dir=log_dir)
```

```
Saving logs at logs/fit\log_2023_07_31_21_39_54
```

In [58]:
```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=5, restore_best_weights=Tr
```

In [59]:
```
CKPT_path = "Model_ckpt.h5"
checkpointing_cb = tf.keras.callbacks.ModelCheckpoint(CKPT_path, save_best_only=True)
```

In [60]:
```
# Q13. Use binary cross-entropy as the loss function, Adam optimizer, and ['accuracy'] a
loss_function = 'binary_crossentropy'
optimizer = 'adam'
metrics = ['accuracy']

# Q14. Compile the model
model.compile(optimizer=optimizer, loss=loss_function, metrics=metrics)
```

In [61]:
```
# Q12. Print the model summary
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 HiddenLayer1 (Dense)        (None, 30)                360

 HiddenLayer2 (Dense)        (None, 10)                310

 HiddenLayer3 (Dense)        (None, 5)                 55

 OutputLayer (Dense)         (None, 1)                 6

=================================================================
Total params: 731
Trainable params: 731
Non-trainable params: 0
_________________________________________________________________
```

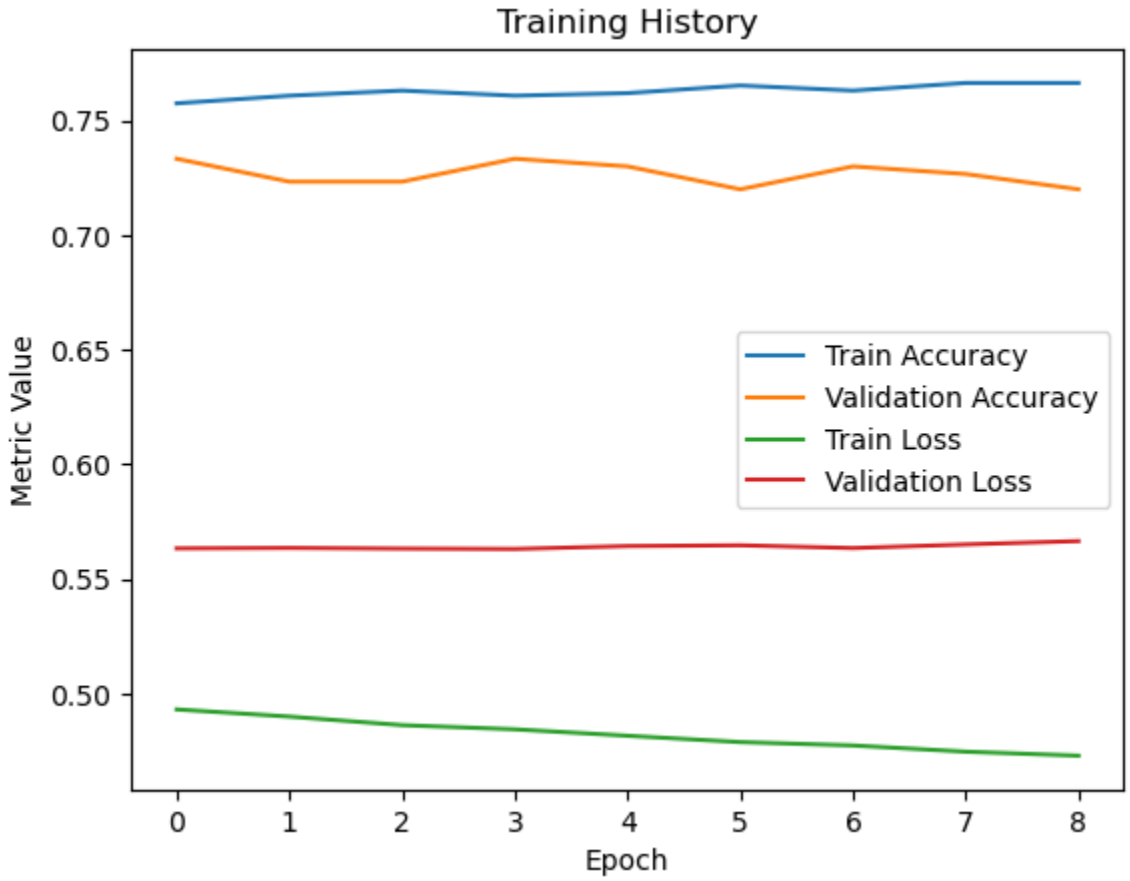
In [62]:
```
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

In [68]:
```
# Orginal train

EPOCHS = 40
VALIDATION_SET = (X_valid, y_valid)

history = model.fit(X_train, y_train, epochs=EPOCHS,
                    validation_data=VALIDATION_SET, batch_size=64, callbacks=[tb_cb, ear
```

```
Epoch 1/40
15/15 [==============================] - 1s 20ms/step - loss: 0.4931 - accuracy: 0.7575
- val_loss: 0.5633 - val_accuracy: 0.7333
Epoch 2/40
15/15 [==============================] - 0s 11ms/step - loss: 0.4900 - accuracy: 0.7608
- val_loss: 0.5635 - val_accuracy: 0.7233
Epoch 3/40
15/15 [==============================] - 0s 12ms/step - loss: 0.4862 - accuracy: 0.7631
- val_loss: 0.5632 - val_accuracy: 0.7233
Epoch 4/40
15/15 [==============================] - 0s 17ms/step - loss: 0.4844 - accuracy: 0.7608
- val_loss: 0.5631 - val_accuracy: 0.7333
Epoch 5/40
15/15 [==============================] - 0s 11ms/step - loss: 0.4816 - accuracy: 0.7620
- val_loss: 0.5643 - val_accuracy: 0.7300
Epoch 6/40
15/15 [==============================] - 0s 12ms/step - loss: 0.4788 - accuracy: 0.7653
- val_loss: 0.5647 - val_accuracy: 0.7200
Epoch 7/40
15/15 [==============================] - 0s 12ms/step - loss: 0.4773 - accuracy: 0.7631
- val_loss: 0.5635 - val_accuracy: 0.7300
Epoch 8/40
15/15 [==============================] - 0s 12ms/step - loss: 0.4746 - accuracy: 0.7664
- val_loss: 0.5651 - val_accuracy: 0.7267
Epoch 9/40
15/15 [==============================] - 0s 14ms/step - loss: 0.4729 - accuracy: 0.7664
- val_loss: 0.5665 - val_accuracy: 0.7200
```

In [69]:
```
# Q16. Get the model's parameters
model_params = model.get_weights()

# Q17. Store the model's training history as a Pandas DataFrame
history_df = pd.DataFrame(history.history)
```

In [70]:
```
# Q18. Plot the model's training history
plt.plot(history_df['accuracy'], label='Train Accuracy')
plt.plot(history_df['val_accuracy'], label='Validation Accuracy')
plt.plot(history_df['loss'], label='Train Loss')
plt.plot(history_df['val_loss'], label='Validation Loss')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Metric Value')
plt.title('Training History')
plt.show()

# Q19. Evaluate the model's performance using the test data
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}')
```




```
13/13 [==============================] - 0s 4ms/step - loss: 0.5381 - accuracy: 0.7400
Test Loss: 0.5381, Test Accuracy: 0.7400
```

In [71]:
```
X_test.shape
```

Out[71]:
```
(400, 11)
```

In [72]:
```
new = X_test[0]
```

In [74]:
```
new.reshape((1,11))
```

Out[74]:
```
array([[-0.34408707,  0.15940242, -0.98437473, -0.02598979,  0.46615703,
        -0.18979812, -0.0378264 ,  0.21710754, -0.46498919, -0.02247738,
        -0.78903789]])
```

In [81]:
```
model.predict(new.reshape((1,11)))
```

```
1/1 [==============================] - 0s 64ms/step
```

Out[81]:
```
array([[0.39237672]], dtype=float32)
```

In [83]:
```
# Assuming you have already trained the model and loaded the test data (X_test, y_test)

# Make predictions on the test data using the trained model
y_pred_probs = model.predict(X_test)

# Convert the predicted probabilities to class labels (0 or 1)
y_pred_labels = (y_pred_probs > 0.5).astype(int)

# If you have used the sigmoid activation function in the output layer
# and want to predict the class with the highest probability directly:
# y_pred_labels = np.argmax(y_pred_probs, axis=1)

# Evaluate the model's performance
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred_labels)

# Create a confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred_labels)

# Print the classification report
class_report = classification_report(y_test, y_pred_labels)

print("Accuracy:", accuracy)
print("Confusion Matrix:")
print(conf_matrix)
print("Classification Report:")
print(class_report)
```

```
13/13 [==============================] - 0s 4ms/step
Accuracy: 0.74
Confusion Matrix:
[[137  41]
 [ 63 159]]
Classification Report:
              precision    recall  f1-score   support

           0       0.69      0.77      0.72       178
           1       0.80      0.72      0.75       222

    accuracy                           0.74       400
   macro avg       0.74      0.74      0.74       400
weighted avg       0.75      0.74      0.74       400
```

In [87]:
```
# Assuming you have already trained the model and loaded the test data (X_test)

# Make predictions on the test data using the trained model
y_pred_probs = model.predict(X_test)

# Verify the shape of the y_pred_probs array
print("Shape of y_pred_probs:", y_pred_probs.shape)

# Assuming '0' represents 'Bad' and '1' represents 'Good',
# you can access the probability of 'Good' wine for the first sample (index 0) as follow
prob_good = y_pred_probs[0][0]

print("Probability of Good Wine:", prob_good)
```

```
13/13 [==============================] - 0s 4ms/step
Shape of y_pred_probs: (400, 1)
Probability of Good Wine: 0.39237672
```

In [ ]:

Assignment Question

Q1. What is the purpose of forward propagation in a neural network?

Answer: The purpose of forward propagation in a neural network is to compute the output of the network based on given input data. It involves passing the input through the network's layers, applying weights and biases to the data, and activating neurons using specific functions until the final output is generated.

Q2. How is forward propagation implemented mathematically in a single-layer feedforward neural network?

Answer: In a single-layer feedforward neural network, the forward propagation process can be mathematically represented as follows:

Input: The input features are denoted as a vector x = [x_1, x_2, ..., xn]. Weighted Sum: Each input feature is multiplied by its corresponding weight, and the biases are added to the weighted sum. Activation Function: The weighted sum is then passed through an activation function, producing the output of the neuron.

Q3. How are activation functions used during forward propagation?

Answer: Activation functions are applied during forward propagation to introduce non-linearity in the neural network, allowing it to learn complex patterns and make predictions for more diverse datasets. The activation function takes the output of a neuron and transforms it into a new value, which becomes the input for the next layer in the network.

Q4. What is the role of weights and biases in forward propagation?

Answer: The weights and biases are crucial parameters in forward propagation. The weights determine the strength of connections between neurons, controlling how much influence each input has on the neuron's output. Biases, on the other hand, shift the output of the activation function and allow the network to learn from different parts of the data distribution.

Q5. What is the purpose of applying a softmax function in the output layer during forward propagation?

Answer: The softmax function is typically applied in the output layer of a neural network when dealing with multi-class classification problems. It converts the raw output scores (logits) of the network into probabilities. The softmax function ensures that the output probabilities sum up to 1, making it easier to interpret the model's certainty about each class.

Q6. What is the purpose of backward propagation in a neural network?

Answer: The purpose of backward propagation, also known as backpropagation, is to adjust the network's weights and biases based on the computed error during forward propagation. By propagating the error backward through the network, it allows the model to learn and improve its performance through the process of gradient descent.

Q7. How is backward propagation mathematically calculated in a single-layer feedforward neural network?

Answer: In a single-layer feedforward neural network, backward propagation involves calculating the gradients of the loss function with respect to the weights and biases. These gradients indicate how the weights and biases should be adjusted to minimize the error. The chain rule is used to compute these gradients layer-by-layer, starting from the output layer and moving backward towards the input layer.

Q8. Can you explain the concept of the chain rule and its application in backward propagation?

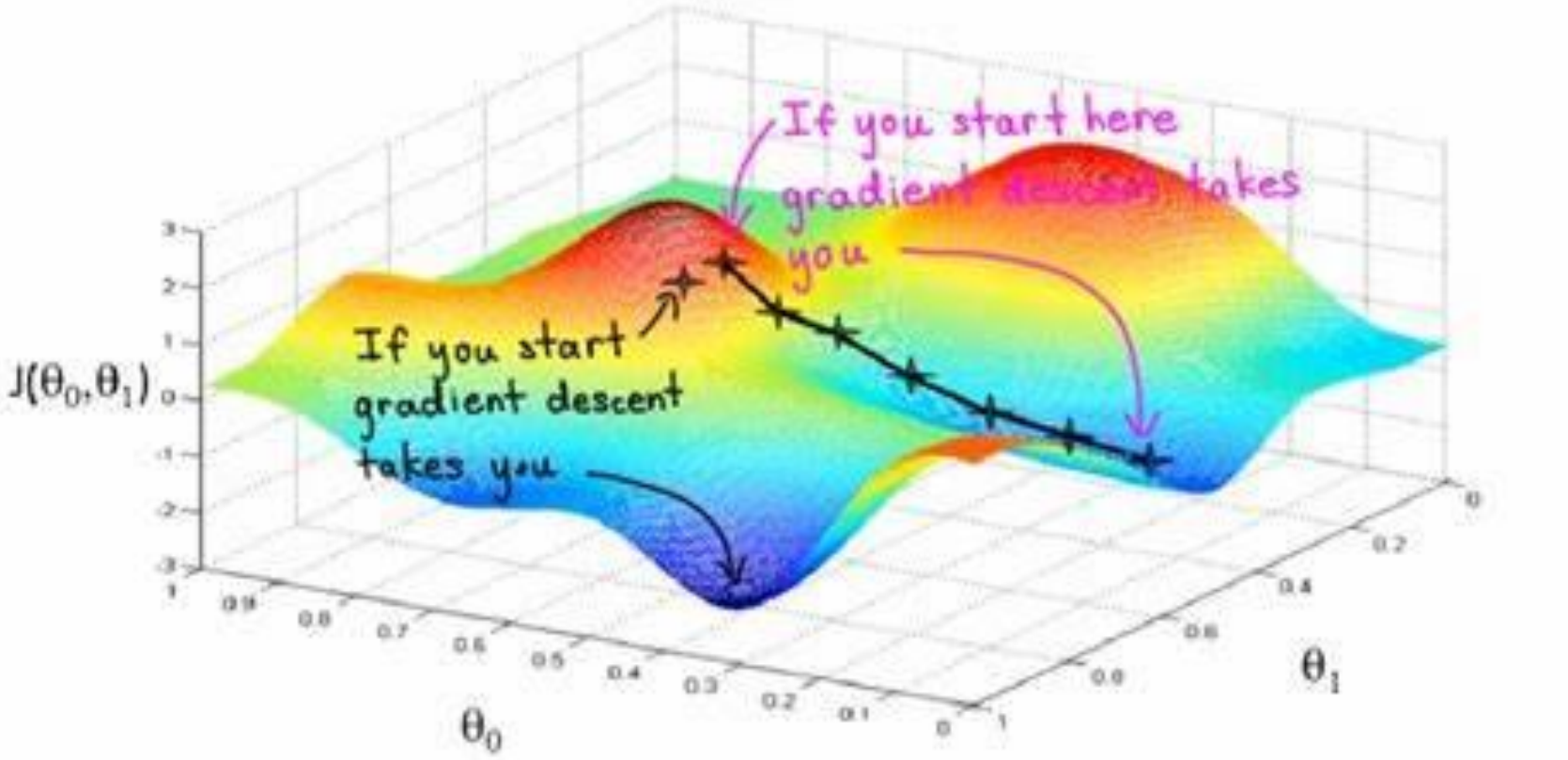
Answer: The chain rule is a fundamental concept in calculus that allows us to find the derivative of a composite function. In the context of neural networks, it enables us to compute the gradients of the loss function with respect to the weights and biases of each layer. During backward propagation, the chain rule is applied to calculate how the changes in the output of a layer affect the error, and these gradients are used to update the parameters of the network through gradient descent.

Q9. What are some common challenges or issues that can occur during backward propagation, and how can they be addressed?

Answer: Some common challenges or issues during backward propagation are:

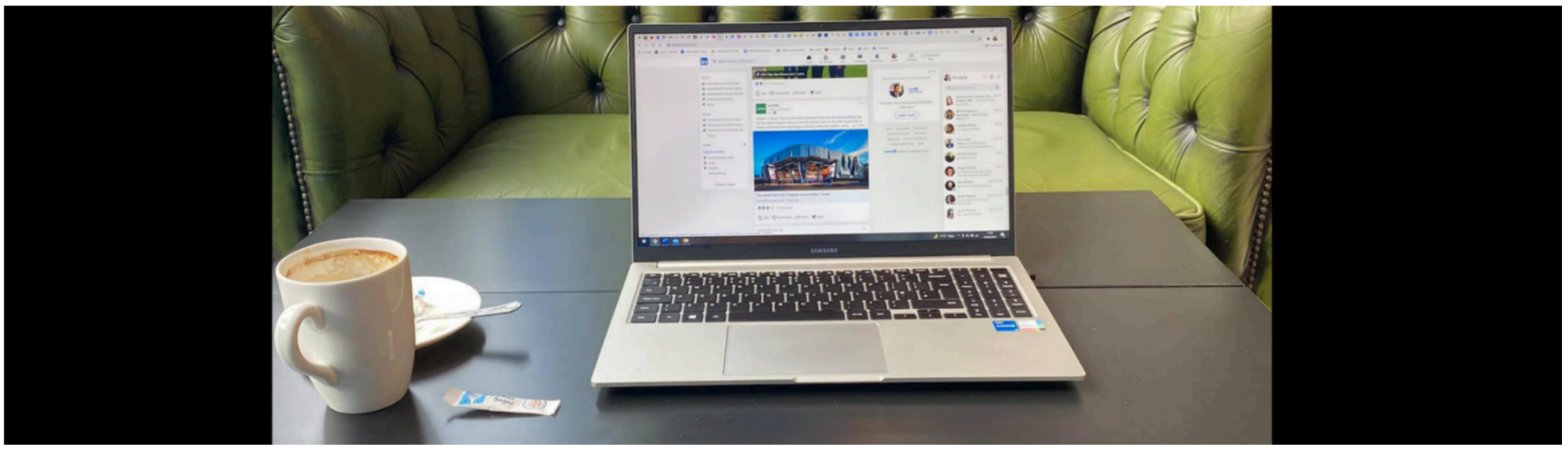
- Vanishing gradients: When gradients become very small, hindering the learning process. This can be mitigated using activation functions that preserve gradients, like ReLU.
- Exploding gradients: When gradients become extremely large, causing instability during learning. Techniques like gradient clipping can be applied to control the magnitude of gradients.
- Overfitting: When the model becomes too complex and performs well on training data but poorly on unseen data. Regularization techniques, such as L1 or L2 regularization, can be used to address this problem.
- Learning rate selection: Choosing an appropriate learning rate is essential for stable and efficient learning. Techniques like learning rate scheduling or adaptive learning rate methods (e.g., Adam) can be used.
- Local minima: The optimization process may get stuck in local minima, leading to suboptimal solutions

GRADIENT DESCENT



GRADIENT DESCENT

MACHINE LEARNING & DEEP LEARNING - GRADIENT DESCENT



HEMANT THAPA

In [88]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.animation as ani
from matplotlib.animation import FuncAnimation
import statistics as st
import yfinance as yf
import tensorflow as tf
import PIL
import math
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization, Dropout
from keras.callbacks import EarlyStopping
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam
```

In [3]:
```
class stock:
    def __init__(self, ticker, time):
        self.ticker = ticker
        self.time = time

    def chart(self):
        return yf.download(self.ticker, period=self.time)
```

DATASET

In [4]:
```
df = pd.read_csv('advertising.csv')
```

In [5]:
```
df.isnull().sum()
```

Out[5]:
```
TV           0
Radio        0
Newspaper    0
Sales        0
dtype: int64
```

In [6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   TV         200 non-null    float64
 1   Radio      200 non-null    float64
 2   Newspaper  200 non-null    float64
 3   Sales      200 non-null    float64
dtypes: float64(4)
memory usage: 6.4 KB
```

STANDARD SCALE

In [7]:
```
class StandardScale:
    def __init__ (self, data):
        self.data = data

    def scale_fit(self):
        return (self.data - self.data.mean())/self.data.std()
```

In [8]:
```
data = StandardScale(df).scale_fit()
```

In [9]:
```
data[:10]
```

Out[9]:

	TV	Radio	Newspaper	Sales
0	0.967425	0.979066	1.774493	1.548168
1	-1.194379	1.080097	0.667903	-0.694304
2	-1.512360	1.524637	1.779084	-0.905135
3	0.051919	1.214806	1.283185	0.858177
4	0.393196	-0.839507	1.278593	-0.215143
5	-1.611365	1.726701	2.040809	-1.307629
6	-1.042960	0.642293	-0.323896	-0.425974
7	-0.312652	-0.246787	-0.870303	-0.157644
8	-1.612530	-1.425491	-1.357019	-1.767624
9	0.614501	-1.391814	-0.429504	-0.655971

In [10]:
```
model = LinearRegression()
```

In [11]:
```
model
```

Out[11]:
```
▾ LinearRegression
LinearRegression()
```

In [12]:
```
X = data['TV'].values.reshape(-1,1)
y = data['Sales'].values
```

In [13]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3)
```

In [14]:
```
model.fit(X_train, y_train)
```

Out[14]:
```
▾ LinearRegression
LinearRegression()
```

INTERCEPTION

In [15]:
```
model.intercept_
```

Out[15]:
```
-0.009917747255374178
```

COEFFICIENT

In [16]:
```
model.coef_
```

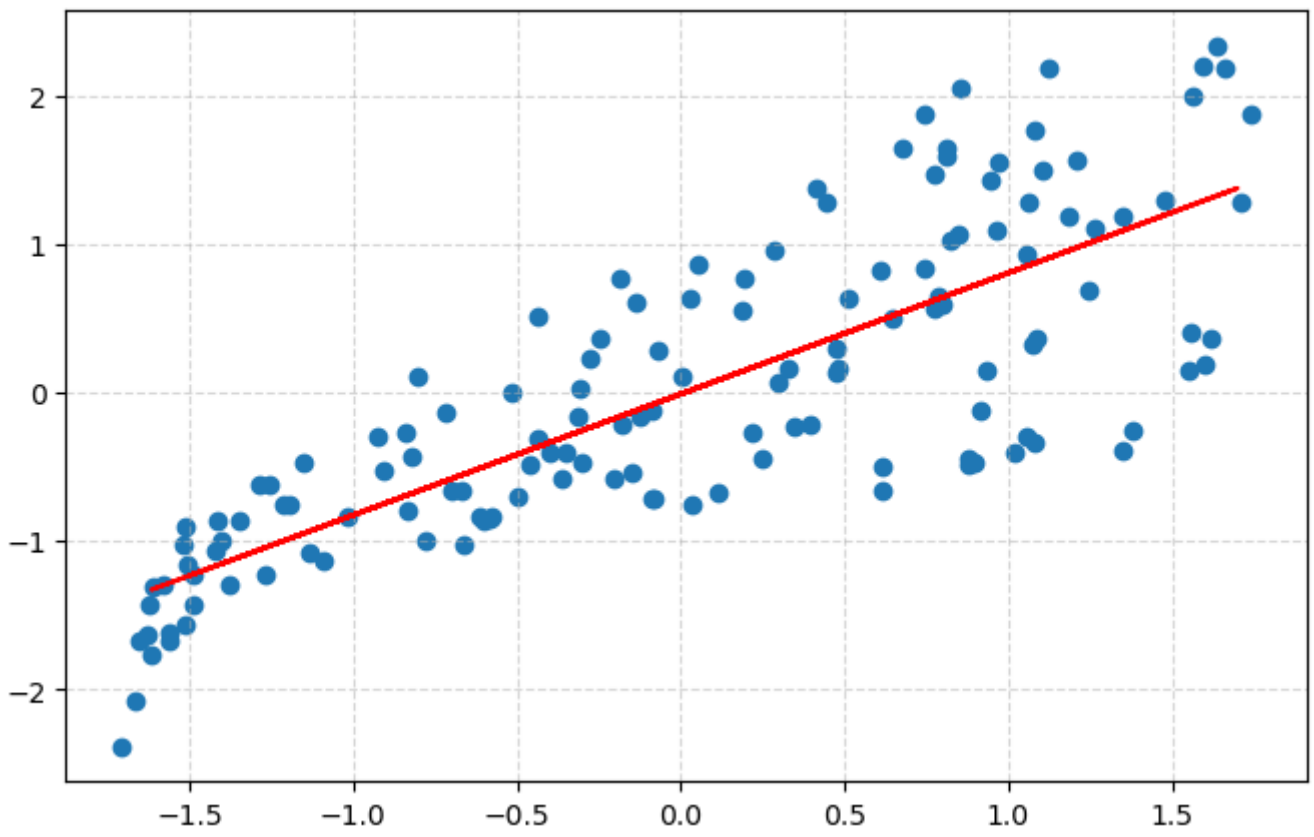
Out[16]:
```
array([0.81519156])
```

In [17]:
```
y_pred = model.predict(X_test)
```

In [18]:
```
y_pred
```

Out[18]:
```
array([ 0.7616275 , -0.04925105,  1.35411956,  0.15584235, -0.86012961,
       -0.98356546,  0.95627634,  1.37500871, -0.69681449, -0.8145533 ,
       -0.8724732 , -1.1801133 , -1.04528338,  0.687566  ,  0.34954168,
        1.22118865, -1.14498156,  0.47012901,  0.27453066, -0.70915808,
       -0.56768161, -1.32633731, -1.16871923, -0.7566334 ,  1.07496465,
        0.66762636, -0.75093636, -0.73574426,  0.43309825, -0.770876  ,
        1.12813701,  0.42835072, -0.99780806, -0.68067288,  0.5574836 ,
        0.47012901,  0.58406979, -0.51450924, -0.0843828 , -1.22853814,
       -0.09672638,  0.26218708,  0.20996422,  0.79770875, -0.45279132,
        0.66857587,  0.44923986, -1.22758863,  0.87366927,  0.48247259,
        0.177681  , -0.69111745,  1.11579343,  0.53184693, -0.35499215,
        1.22308767, -0.49267059,  0.55843311, -0.68162239, -0.77847205])
```

In [19]:
```
plt.figure(figsize=(8,5))
plt.scatter(X_train,y_train)
plt.plot(X_test, y_pred, color="red")
plt.grid(True, linestyle="--", alpha=0.5)
plt.show()
```



In [20]:
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
```

In [21]:
```
r_square = r2_score(y_test, y_pred)
r_square
```

Out[21]:
```
0.526344665266148
```

In [22]:
```
mean_absolute_error(y_test, y_pred)
```

Out[22]:
```
0.48123630257314403
```


In [23]:
```
mean_squared_error(y_test, y_pred)
```

Out[23]:
```
0.40920214509628555
```

In [24]:
```
root_mean_square_error = math.sqrt(mean_squared_error(y_test, y_pred))
root_mean_square_error
```

Out[24]:
```
0.6396891003419439
```

In [25]:
```
data = StandardScale(df).scale_fit()
```

In [26]:
```
data[:5]
```

Out[26]:

	TV	Radio	Newspaper	Sales
0	0.967425	0.979066	1.774493	1.548168
1	-1.194379	1.080097	0.667903	-0.694304
2	-1.512360	1.524637	1.779084	-0.905135
3	0.051919	1.214806	1.283185	0.858177
4	0.393196	-0.839507	1.278593	-0.215143

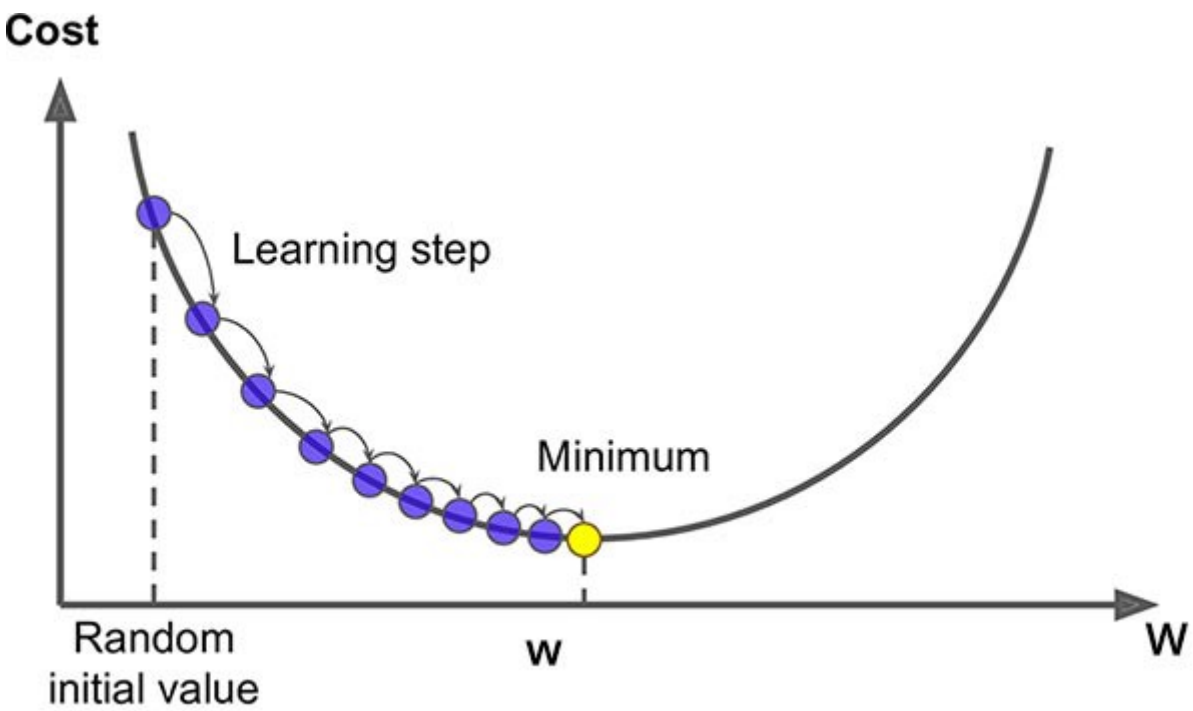
In [27]:
```
X = data['TV']
y = data['Sales']
```

GRADIENT DESCENT

In [28]:
```python
class GradientDescent:
    def __init__(self, x, y, m_curr=0, c_curr=0, iteration=100, rate=0.01):
        self.x = x
        self.y = y
        self.predicted_y = (m_curr * x) + c_curr  # Initialize predicted_y using initial slope and intercept
        self.m_curr = m_curr
        self.c_curr = c_curr
        self.iteration = iteration
        self.rate = rate

    def cost_function(self):
        N = len(self.y)
         #mean squared error
        return sum((self.y - self.predicted_y) ** 2) / N

    def calculation(self):
        N = float(len(self.y))
        gradient_descent = pd.DataFrame(columns=['m_curr', 'c_curr', 'cost'])
        # Perform gradient descent iterations
        for i in range(self.iteration):
            # Calculate the predicted y values using current slope and intercept
            self.predicted_y = (self.m_curr * self.x) + self.c_curr
            cost = self.cost_function()
             # Calculate gradients for slope (m_grad) and intercept (c_grad)
            m_gradient = -(2/N) * np.sum(self.x * (self.y - self.predicted_y))
            c_gradient = -(2/N) * np.sum(self.y - self.predicted_y)
            # Update the slope and intercept using gradient and learning rate
            self.m_curr -= self.rate * m_gradient
            self.c_curr -= self.rate * c_gradient
            gradient_descent.loc[i] = [self.m_curr, self.c_curr, cost]

        return gradient_descent
```

In [29]:
```python
gd = GradientDescent(X, y)
gradient_descent_result = gd.calculation()
```

GRADIENT DESCENT



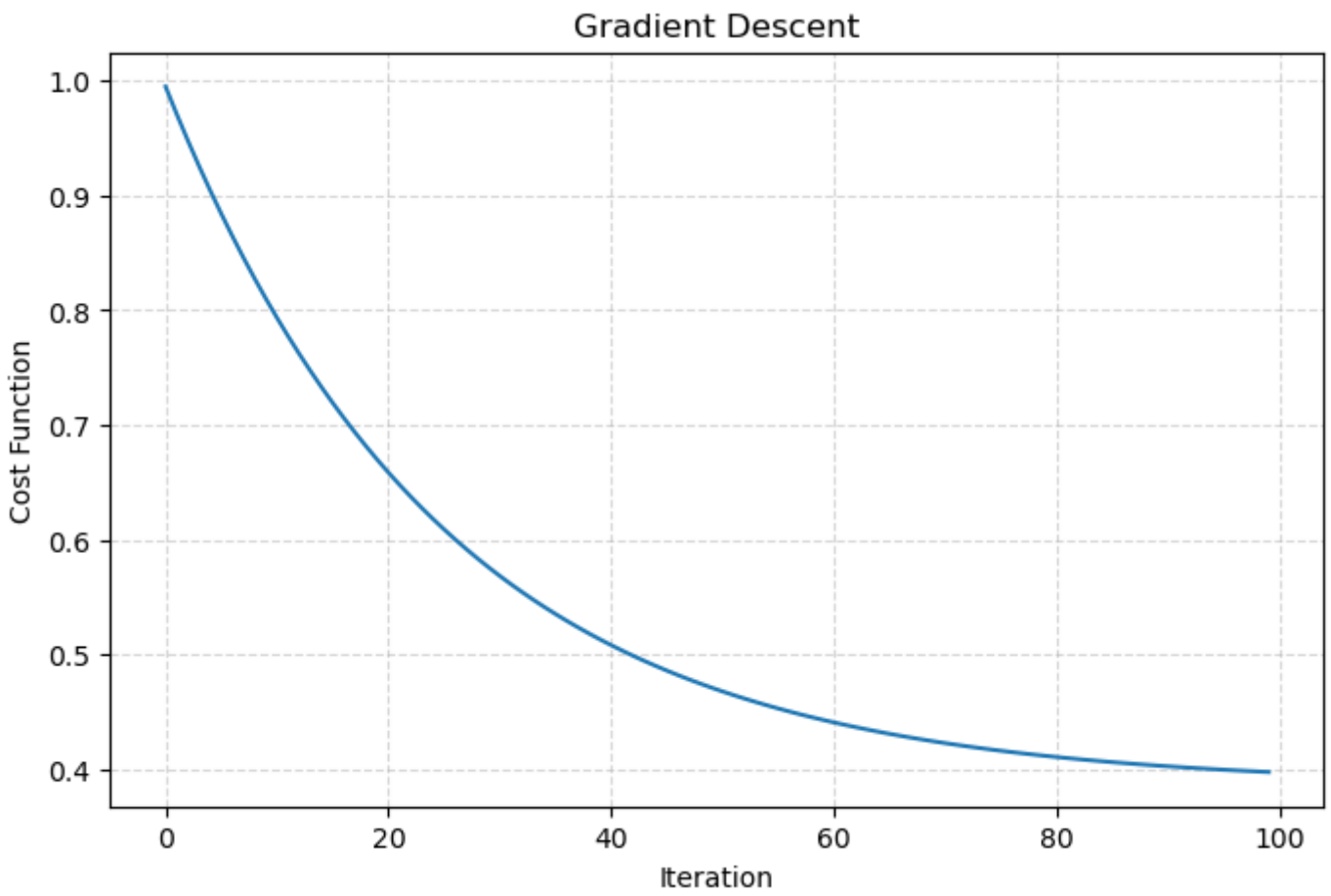
In [30]:
```python
gradient_descent_result[:10]
```

Out[30]:

	m_curr	c_curr	cost
0	0.015566	-3.641532e-18	0.995000
1	0.030823	-7.283063e-18	0.971010
2	0.045776	-1.021405e-17	0.947966
3	0.060431	-1.412204e-17	0.925829
4	0.074795	-1.749711e-17	0.904565
5	0.088873	-2.096101e-17	0.884139
6	0.102670	-2.380318e-17	0.864517
7	0.116193	-2.708944e-17	0.845669
8	0.129447	-3.081979e-17	0.827564
9	0.142438	-3.428369e-17	0.810172

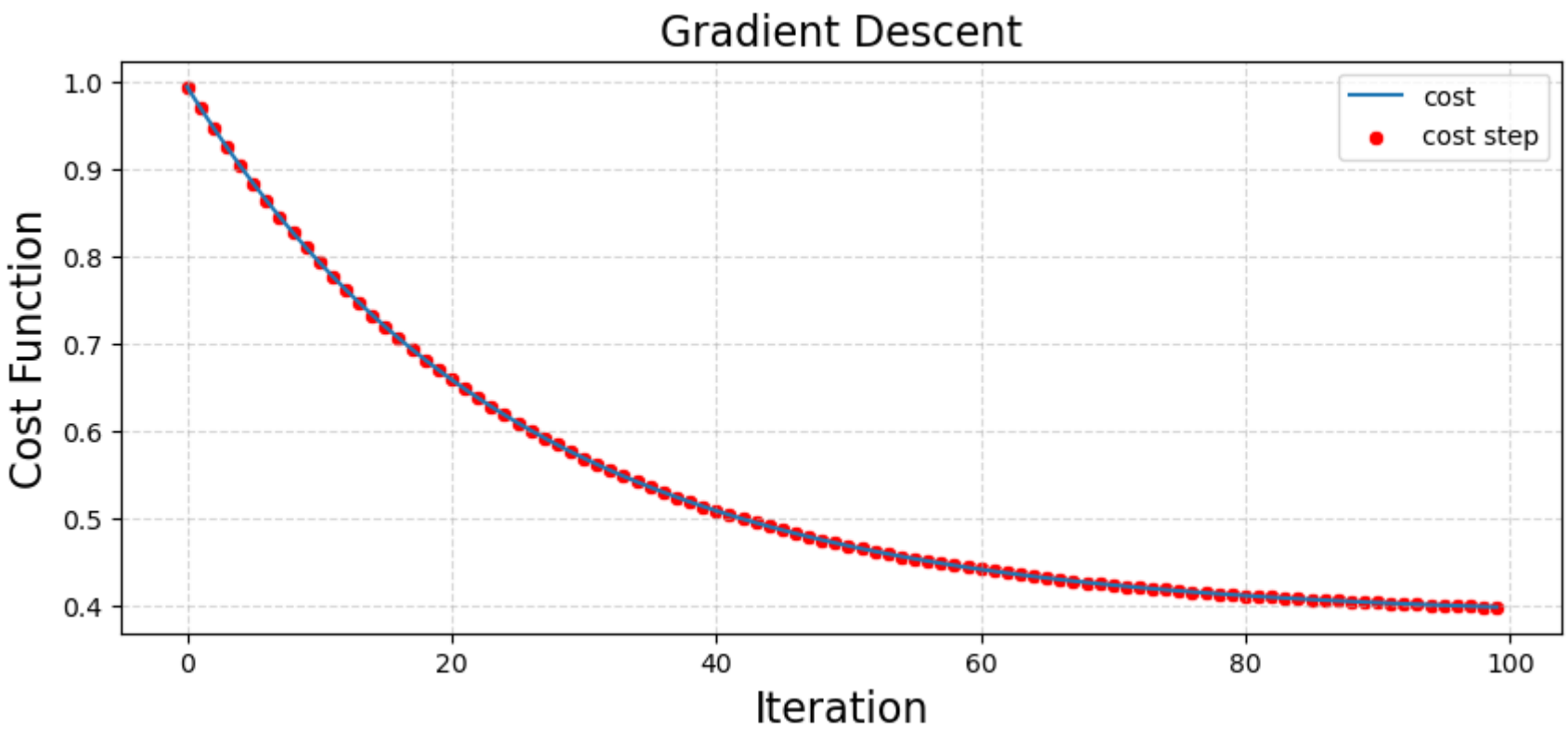
In [31]:
```python
plt.figure(figsize=(8,5))
gradient_descent_result.cost.plot()
plt.grid(True, linestyle="--", alpha=0.5)
plt.xlabel('Iteration')
```

```
plt.ylabel('Cost Function')
plt.title('Gradient Descent')
plt.show()
```



In [32]:
```
gradient_descent_result.reset_index(inplace=True)
```

In [33]:
```
plt.figure(figsize=(10, 4))
sns.lineplot(data=gradient_descent_result, x='index', y='cost', label="cost")
sns.scatterplot(data=gradient_descent_result, x='index', y='cost', color="red", label="cost step")
plt.grid(True, linestyle='--', alpha=0.5)
plt.xlabel('Iteration', fontsize=16)
plt.ylabel('Cost Function', fontsize=16)
plt.title('Gradient Descent', fontsize=16)
plt.show()
```



MACHINE LEARNING

In [34]:
```
jp = stock("JPM", "2y").chart()
gs = stock("GS", "2y").chart()
```

```
[**********100%**********]  1 of 1 completed
[**********100%**********]  1 of 1 completed
```

In [35]:
```
X = jp.Close.values.reshape(-1,1)
y = gs.Close.values
```

In [36]:
```
print(X.shape)
```

```
(504, 1)
```

In [37]:
```
print(y.shape)
```

```
(504,)
```

In [38]:
```
y = y[:2519]
print(y.shape)
```

```
(504,)
```

In [39]:
```
X_train, X_test_sklearn, y_train, y_test_sklearn = train_test_split(X,y, test_size=0.2)
```

In [40]:
```
model = LinearRegression()
```

In [41]:
```
model.fit(X_train, y_train)
```

Out[41]:
```
▾ LinearRegression
LinearRegression()
```

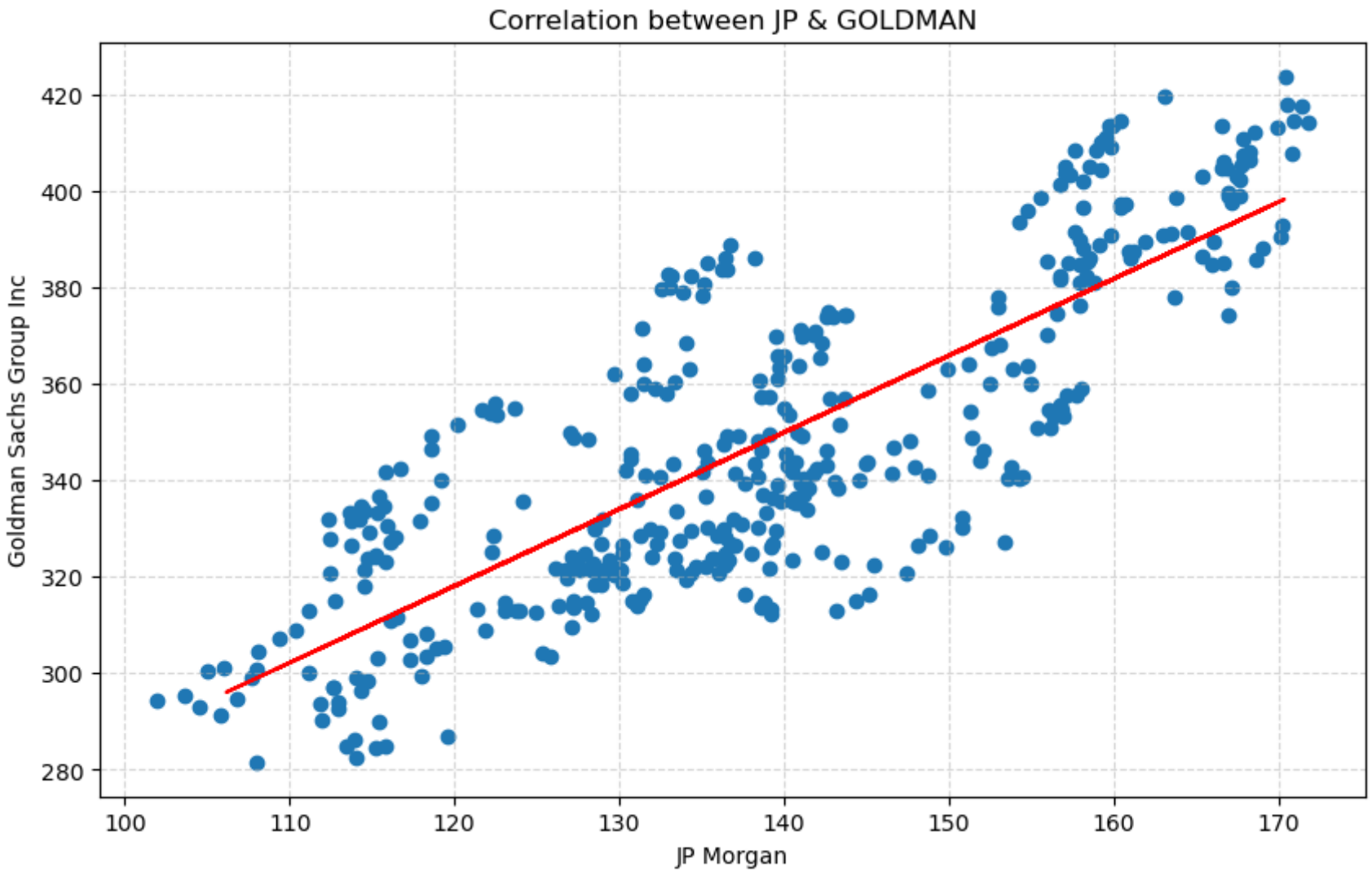
In [42]:
```
y_pred_sklearn = model.predict(X_test_sklearn)
```

In [43]:
```
y_pred_sklearn
```

Out[43]:
```
array([360.50447976, 322.41572255, 354.00501614, 341.61143858,
       311.51957366, 375.38314135, 353.03328207, 384.54292054,
       340.27332015, 379.66832889, 334.2039747 , 318.25797833,
       315.15162329, 311.455852  , 342.55130575, 355.07232662,
       373.50340702, 337.48556127, 319.03855957, 310.65934946,
       351.26504546, 336.49788159, 342.07339936, 375.49466337,
       383.79420621, 351.58364161, 346.47010893, 397.44627138,
       312.34793095, 385.92885148, 309.62389373, 386.69348711,
       331.68702376, 307.01136638, 379.11076739, 347.25066586,
       382.08970341, 372.18120989, 327.48147918, 306.74055843,
       384.62257565, 325.44243463, 310.97794562, 340.60783761,
       383.30037852, 323.43524483, 371.73517042, 315.54987456,
       358.4335683 , 316.66497325, 343.45933031, 336.08370902,
       350.64377445, 327.80008749, 383.71455109, 311.72665995,
       363.67456862, 318.83146113, 300.8145776 , 316.26672198,
       315.1834902 , 334.92082212, 321.58736526, 396.96836499,
       336.57753671, 340.57599501, 398.33835033, 375.60616108,
       349.38531114, 305.03604349, 308.12647722, 375.27164363,
       362.67096764, 348.28613374, 390.6282116 , 325.44243463,
       334.93674342, 329.0107626 , 346.69312867, 396.26743887,
       382.53574289, 345.78510411, 357.36625782, 349.30565602,
       394.94524174, 334.53849216, 336.81650205, 395.2797835 ,
       336.22707365, 393.4956256 , 338.8236797 , 296.06742928,
       334.14024089, 307.63263739, 332.91362017, 349.70390729,
       364.2002389 , 373.79013627, 378.58509711, 313.4630418 ,
       388.76439857])
```

CORRELATION ANALYSIS BETWEEN JP MORGAN AND GOLDMAN SACHS

In [44]:
```
plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train)
plt.plot(X_test_sklearn, y_pred_sklearn, color="red")
plt.grid(True, linestyle="--", alpha=0.5)
plt.xlabel("JP Morgan")
plt.ylabel("Goldman Sachs Group Inc")
plt.title("Correlation between JP & GOLDMAN")
plt.show()
```



In [45]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

In [46]:
```
mae = mean_absolute_error(y_test_sklearn, y_pred_sklearn)
mse = mean_squared_error(y_test_sklearn, y_pred_sklearn)
rmse = np.sqrt(mse)
r_square = r2_score(y_test_sklearn, y_pred_sklearn)

print("Mean Absolute Error:", mae)
print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse)
print("R-squared:", r_square)
```

```
Mean Absolute Error: 14.87905864860798
Mean Squared Error: 317.44505798284086
Root Mean Squared Error: 17.816987904324368
R-squared: 0.7107722694961836
```

DEEP LEARNING (NEURAL NETWORK)

In [47]:
```
jp = stock("JPM", "2y").chart()
gs = stock("GS", "2y").chart()
```

```
[**********100%**********]  1 of 1 completed
[**********100%**********]  1 of 1 completed
```

In [48]:
```
x = jp.Close.values.reshape(-1,1)
y = gs.Close.values
```

In [49]:
```
X_train_temp, X_test_tf, y_train_temp, y_test_tf = train_test_split(x, y, test_size=0.2, random_state=42)
X_train_tf, X_val, y_train_tf, y_val = train_test_split(X_train_temp, y_train_temp, test_size=0.2, random_state=42)
```

In [50]:
```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_tf.reshape(-1, 1))
X_val_scaled = scaler.transform(X_val.reshape(-1, 1))
X_test_scaled = scaler.transform(X_test_tf.reshape(-1, 1))
```

In [51]:
```
model = Sequential([
    Dense(units=1, input_shape=(1,), activation='linear', use_bias=True)
])
```

In [52]:
```
model.compile(optimizer='sgd', loss='mean_squared_error')
```

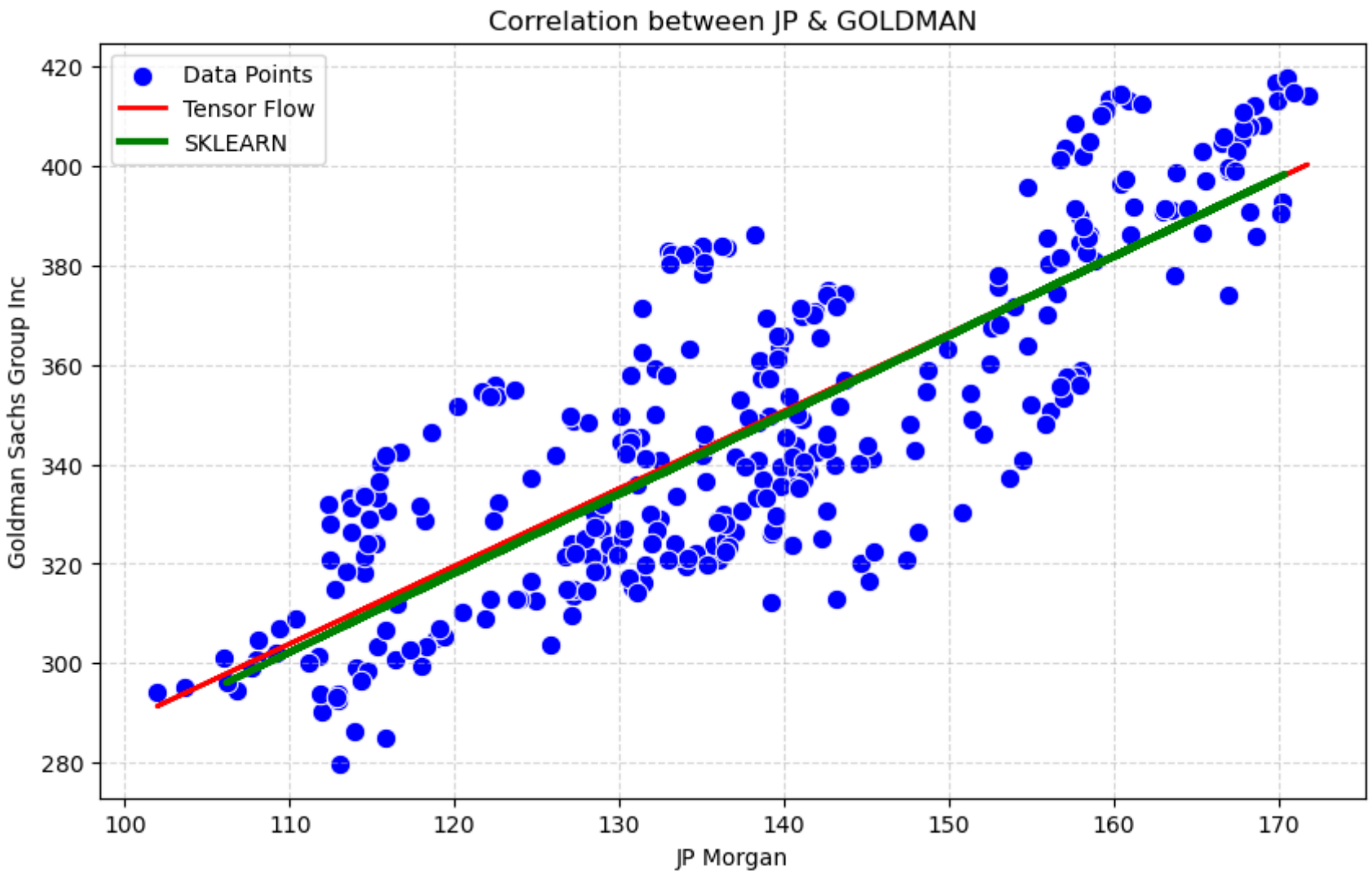
In [53]:
```
history = model.fit(X_train_scaled, y_train_tf, validation_data=(X_val_scaled, y_val), epochs=100, verbose=0)
```

In [54]:
```
y_pred_tf = model.predict(X_train_scaled)
y_pred_tf[:5]
```

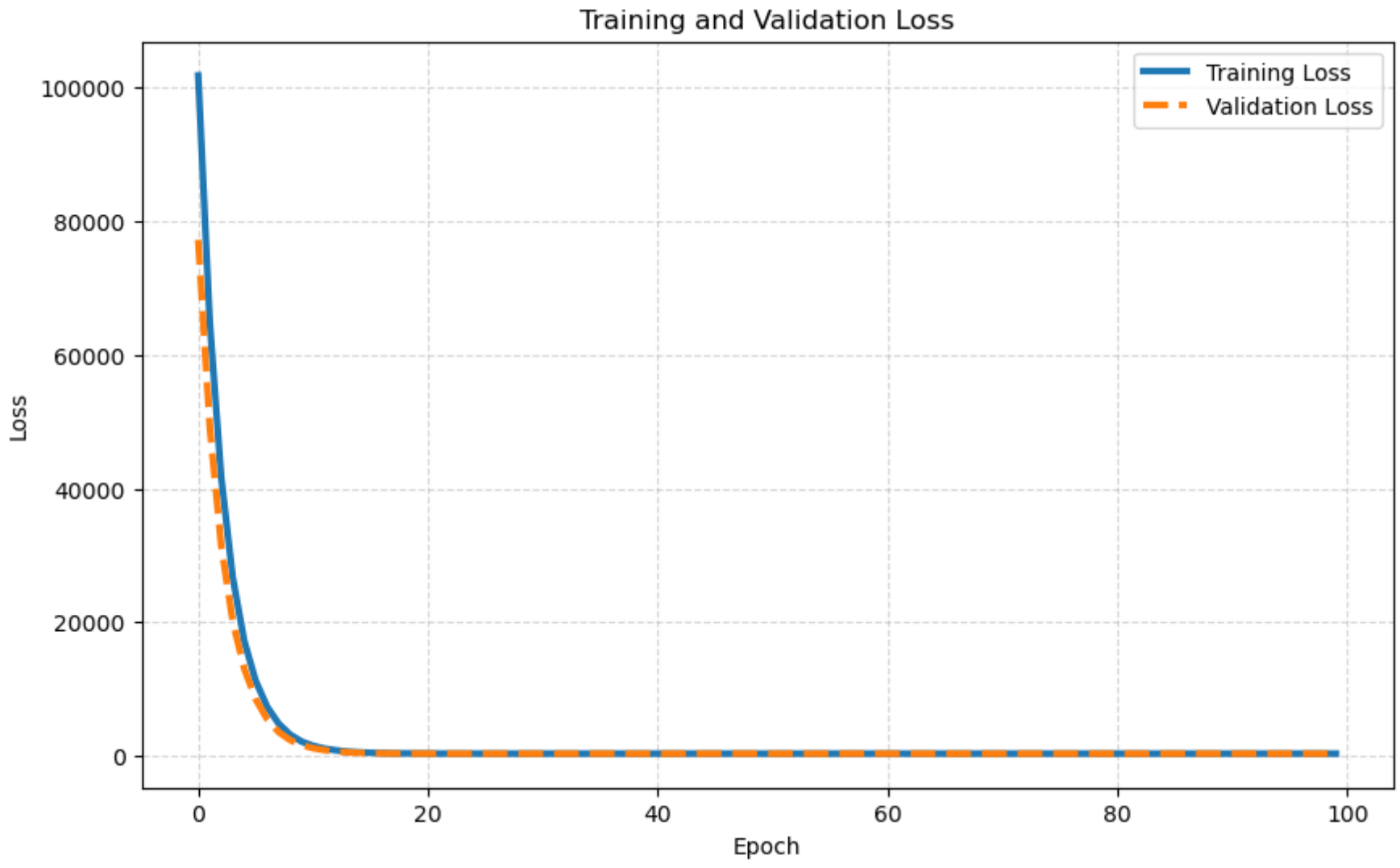
```
11/11 [==============================] - 0s 1ms/step
```

Out[54]:
```
array([[309.51276],
       [395.4769 ],
       [345.35068],
       [312.0581 ],
       [396.10153]], dtype=float32)
```

In [55]:
```
plt.figure(figsize=(10, 6))
sns.scatterplot(x=X_train_tf[:, 0], y=y_train_tf, color='blue', label='Data Points', s=80)
plt.plot(X_train_tf, y_pred_tf, color='red', label='Tensor Flow', linewidth=2)
plt.plot(X_test_sklearn, y_pred_sklearn, color="green", label="SKLEARN", linewidth=3)
plt.grid(True, linestyle="--", alpha=0.5)
plt.xlabel("JP Morgan")
plt.ylabel("Goldman Sachs Group Inc")
plt.title("Correlation between JP & GOLDMAN")
plt.legend()
plt.show()
```



In [56]:
```
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss', linewidth=3)
plt.plot(history.history['val_loss'], label='Validation Loss', linewidth=3, linestyle="--")
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True, linestyle="--", alpha=0.5)
plt.show()
```



In [57]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

In [58]:
```
mae = mean_absolute_error(y_train_tf, y_pred_tf)
mse = mean_squared_error(y_train_tf, y_pred_tf)
rmse = np.sqrt(mse)
r_square = r2_score(y_train_tf, y_pred_tf)

print("Mean Absolute Error:", mae)
print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse)
print("R-squared:", r_square)
```

```
Mean Absolute Error: 15.514148972789693
Mean Squared Error: 339.1804931076903
Root Mean Squared Error: 18.416853507254988
R-squared: 0.6750332917778075
```

SOLVING MANUAL (LINEAR PROGRAMMING)

In [59]:
```
X = stock("JPM", "2y").chart()
Y = stock("GS", "2y").chart()
```

```
[**********************100%**********************]  1 of 1 completed
[**********************100%**********************]  1 of 1 completed
```

In [60]:
```
x = X.Close.values
y = Y.Close.values
```

In [61]:
```
print(x.shape)
print(y.shape)
```

```
(504,)
(504,)
```

In [62]:
```
dataset = { "x":x,
            "y":y
}
```

In [63]:
```
df = pd.DataFrame(dataset)
df[:5]
```

Out[63]:

	x	y
0	158.929993	408.350006
1	157.009995	404.970001
2	155.580002	398.799988
3	154.279999	393.570007
4	154.720001	395.869995

In [64]:
```
print(f"X Mean: {st.mean(df['x'])}")
print(f"y Mean: {st.mean(df['y'])}")
```

```
X Mean: 138.566210413736
y Mean: 347.76824412270196
```

TOTAL SUM OF SQUARE (SST)

In [65]:
```
y_mean = sum(df['y'])/len(df['y'])
x_mean = sum(df['x'])/len(df['x'])
sst = sum([(y - y_mean)**2 for y in df['y']])
print(f'Total Sum of Square: {sst}')
```

```
Total Sum of Square: 557665.0178584817
```

In [66]:
```
df['xi - x mean'] = df['x'] - x_mean
df['yi - y mean'] = df['y'] - y_mean
df['(xi - x mean)(yi - y mean)'] = df['xi - x mean'] * df['yi - y mean']
df['(xi - x mean)**2'] = df['xi - x mean'] * df['xi - x mean']
```

In [67]:
```
df[:5]
```

Out[67]:

	x	y	xi - x mean	yi - y mean	(xi - x mean)(yi - y mean)	(xi - x mean)**2
0	158.929993	408.350006	20.363782	60.581762	1233.673810	414.683628
1	157.009995	404.970001	18.443784	57.201757	1055.016858	340.173172
2	155.580002	398.799988	17.013791	51.031744	868.243442	289.469098
3	154.279999	393.570007	15.713788	45.801763	719.719214	246.923145
4	154.720001	395.869995	16.153791	48.101751	777.025623	260.944957

In [68]:
```
b1 = sum(df['(xi - x mean)(yi - y mean)'])/sum(df['(xi - x mean)**2'])
b1
```

Out[68]:
```
1.5913570686889646
```

In [69]:
```
bo = y_mean - b1 * x_mean
bo
```

Out[69]:
```
127.25992569936074
```

In [70]:
```
x_value = df['x']
x_value = x_value.tolist()
x_value[:5]
```

Out[70]:
```
[158.92999267578125,
 157.00999450683594,
 155.5800018310547,
 154.27999877929688,
 154.72000122070312]
```

In [71]:
```
predicted_values = []

for i in x_value:
    y = bo + b1 * i
    predicted_values.append(y)
```

SUM OF SQUARE OF RESIDUAL (SSE)

In [72]:
```
df['y predicted'] = predicted_values
df['Residual'] = df['y'] - df['y predicted']
```

In [73]:
```
sse = sum((df['Residual'])* df['Residual'])
print(f'Sum of Square Residual Error: {sse}')
```

```
Sum of Square Residual Error: 180951.54047437938
```

In [74]:
```
df[:5]
```

Out[74]:

	x	y	xi - x mean	yi - y mean	(xi - x mean)(yi - y mean)	(xi - x mean)**2	y predicted	Residual
0	158.929993	408.350006	20.363782	60.581762	1233.673810	414.683628	380.174293	28.175713
1	157.009995	404.970001	18.443784	57.201757	1055.016858	340.173172	377.118890	27.851111
2	155.580002	398.799988	17.013791	51.031744	868.243442	289.469098	374.843261	23.956726
3	154.279999	393.570007	15.713788	45.801763	719.719214	246.923145	372.774492	20.795515
4	154.720001	395.869995	16.153791	48.101751	777.025623	260.944957	373.474693	22.395302

SUM OF SQUARE REGRESSION (SSR)

In [75]:
```
df['(y pred - y mean)**2'] = (df['y predicted'] - y_mean)**2
ssr = sum(df['(y pred - y mean)**2'])
print(f'Total Sum of Square Regression: {ssr}')
```

```
Total Sum of Square Regression: 376713.47738410207
```

R SQUARE

In [76]:
```
r_square = ssr/sst
f"R square: {r_square}"
```

Out[76]:
```
'R square: 0.6755192908293566'
```

In [77]:
```
r_square = 1 - sse/sst
f"R square: {r_square}"
```

Out[77]:
```
'R square: 0.6755192908293571'
```

MEAN SQUARE ERROR

In [78]:
```
n = len(df['x'])
mse = 1/n * sum((df['y'] - df['y predicted']) * (df['y'] - df['y predicted']))
print(f"Mean Square Error: {mse}")
```

```
Mean Square Error: 359.0308342745622
```

ROOT MEAN SQUARE ERROR

In [79]:
```
rmse = math.sqrt(mse)
print(f"Root Mean Square Error: {rmse}")
```

```
Root Mean Square Error: 18.948108989410056
```

PEARSON CORRELATION

In [80]:
```
df['x square'] = df['x'] * df['x']
df['y square'] = df['y'] * df['y']
df['xy'] = df['x'] * df['y']
```

In [81]:
```
df[:5]
```

Out[81]:

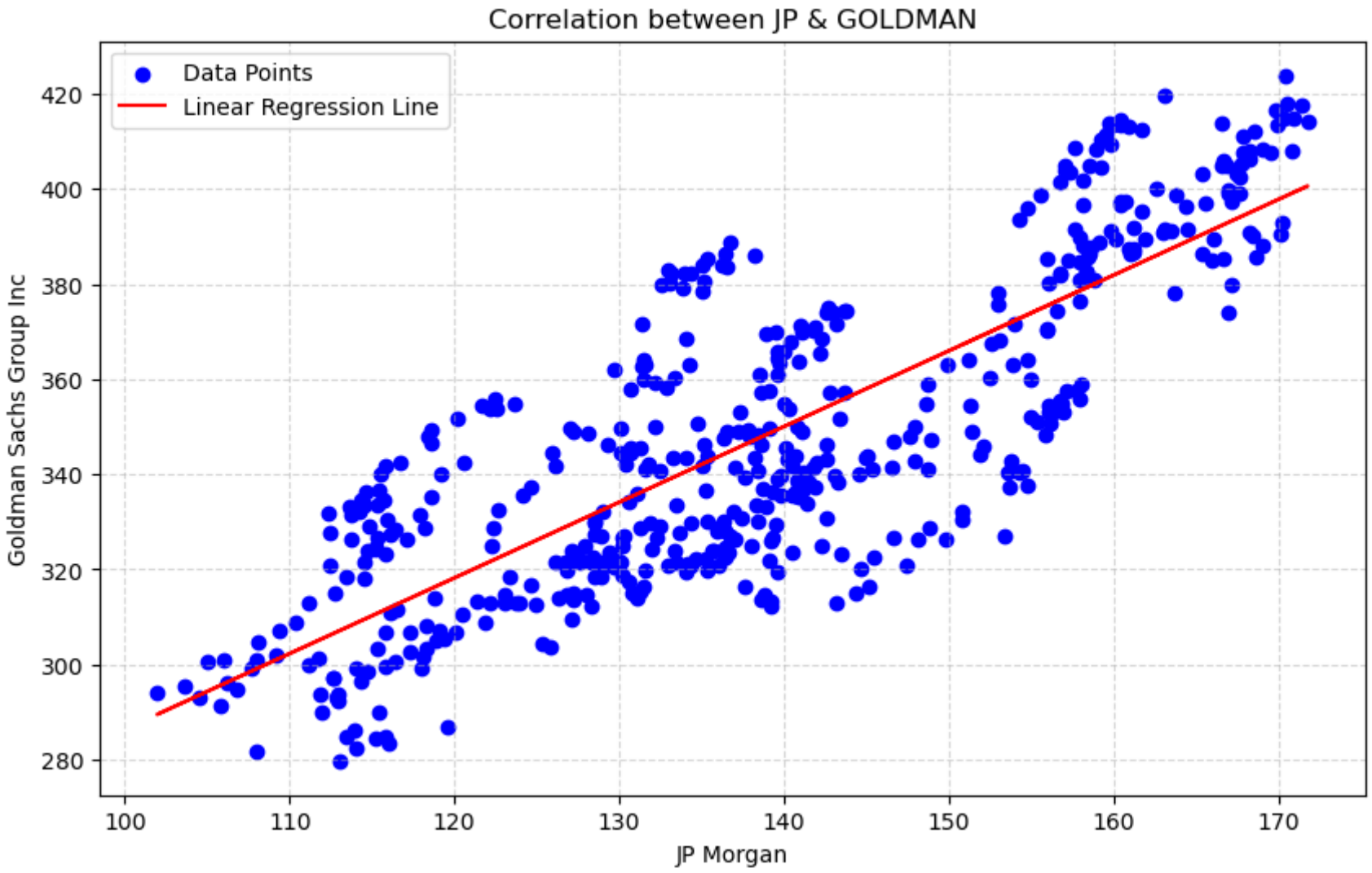
	x	y	xi - x mean	yi - y mean	(xi - x mean)(yi - y mean)	(xi - x mean)**2	y predicted	Residual	(y pred - y mean)**2	x square	y square
0	158.929993	408.350006	20.363782	60.581762	1233.673810	414.683628	380.174293	28.175713	1050.152002	25258.742572	166749.727485
1	157.009995	404.970001	18.443784	57.201757	1055.016858	340.173172	377.118890	27.851111	861.460432	24652.138375	164000.701889
2	155.580002	398.799988	17.013791	51.031744	868.243442	289.469098	374.843261	23.956726	733.056558	24205.136970	159041.430264
3	154.279999	393.570007	15.713788	45.801763	719.719214	246.923145	372.774492	20.795515	625.312449	23802.318023	154897.350665
4	154.720001	395.869995	16.153791	48.101751	777.025623	260.944957	373.474693	22.395302	660.821530	23938.278778	156713.053034

In [82]:
```
sum_of_product = sum(df['xy']) - (sum(df['x']) * sum(df['y'])) / len(df['x'])
sum_of_square_of_first_variable = sum(df['x square']) - (sum(df['x'])**2) / len(df['x'])
sum_of_square_of_second_variable = sum(df['y square']) - (sum(df['y'])**2) / len(df['y'])
```

In [83]:
```
pearson = sum_of_product / (math.sqrt(sum_of_square_of_first_variable * sum_of_square_of_second_variable))
print(f"Pearson Correlation: {pearson}")
```

```
Pearson Correlation: 0.8218998058338737
```

In [84]:
```
plt.figure(figsize=(10, 6))
plt.scatter(df['x'], df['y'], color='blue', label='Data Points')
plt.plot(df['x'], df['y predicted'], color='red', label='Linear Regression Line')
plt.xlabel("JP Morgan")
plt.ylabel("Goldman Sachs Group Inc")
plt.title("Correlation between JP & GOLDMAN")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```



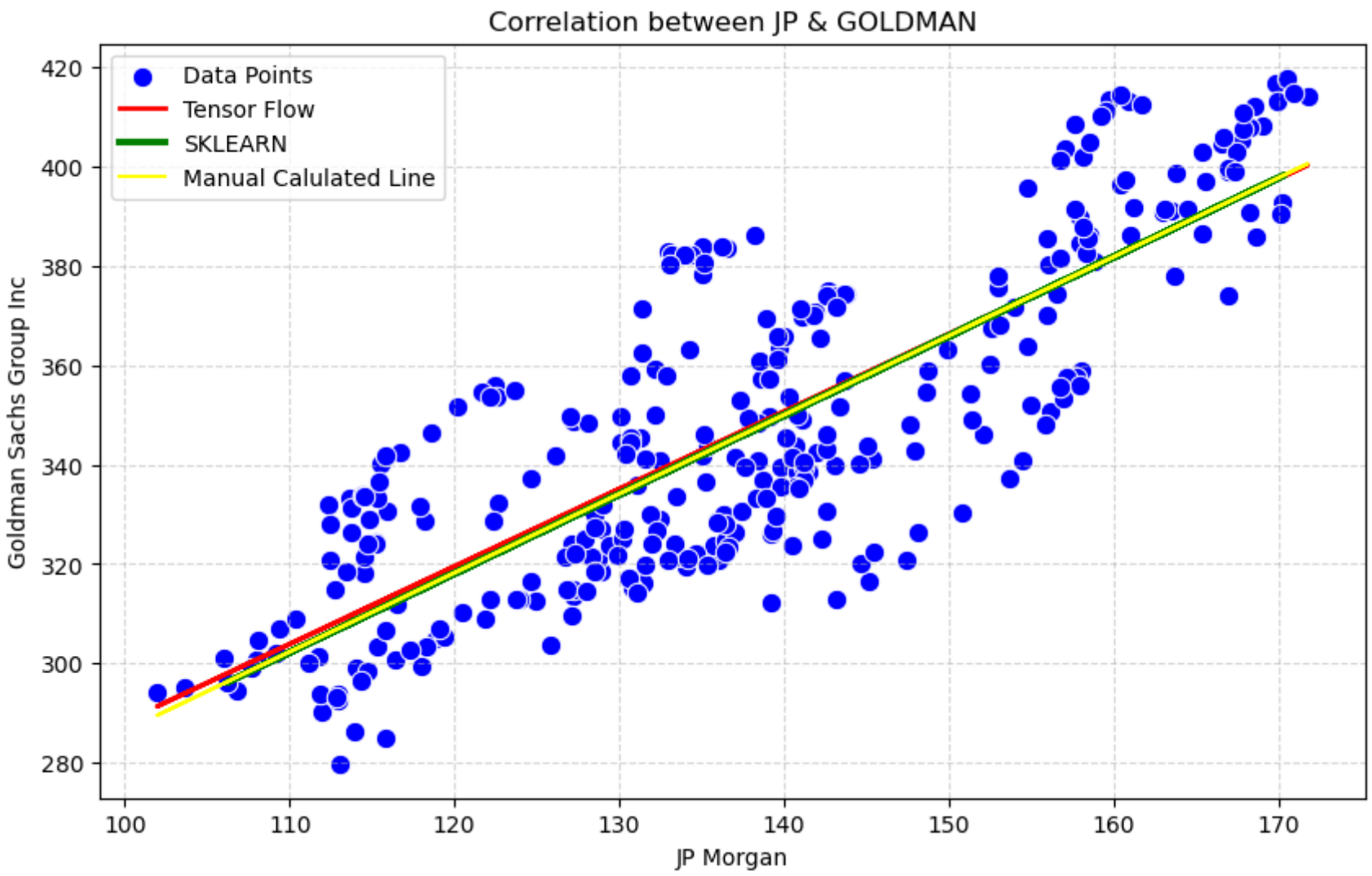
In [85]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

In [86]:
```
mae = mean_absolute_error(df['y'], df['y predicted'])
mse = mean_squared_error(df['y'], df['y predicted'])
rmse_manual = np.sqrt(mse)
r_square = r2_score(df['y'], df['y predicted'])

print("Mean Absolute Error:", mae)
print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse_manual)
print("R-squared:", r_square)
```

```
Mean Absolute Error: 15.901469786486903
Mean Squared Error: 359.03083427456215
Root Mean Squared Error: 18.948108989410056
R-squared: 0.6755192908293575
```

In [87]:
```
plt.figure(figsize=(10, 6))
sns.scatterplot(x=X_train_tf[:, 0], y=y_train_tf, color='blue', label='Data Points', s=80)
plt.plot(X_train_tf, y_pred_tf, color='red', label='Tensor Flow', linewidth=2)
plt.plot(X_test_sklearn, y_pred_sklearn, color="green", label="SKLEARN", linewidth=3)
plt.plot(df['x'], df['y predicted'], color='yellow', label='Manual Calulated Line')
plt.grid(True, linestyle="--", alpha=0.5)
plt.xlabel("JP Morgan")
```

```
plt.ylabel("Goldman Sachs Group Inc")
plt.title("Correlation between JP & GOLDMAN")
plt.legend()
plt.show()
```



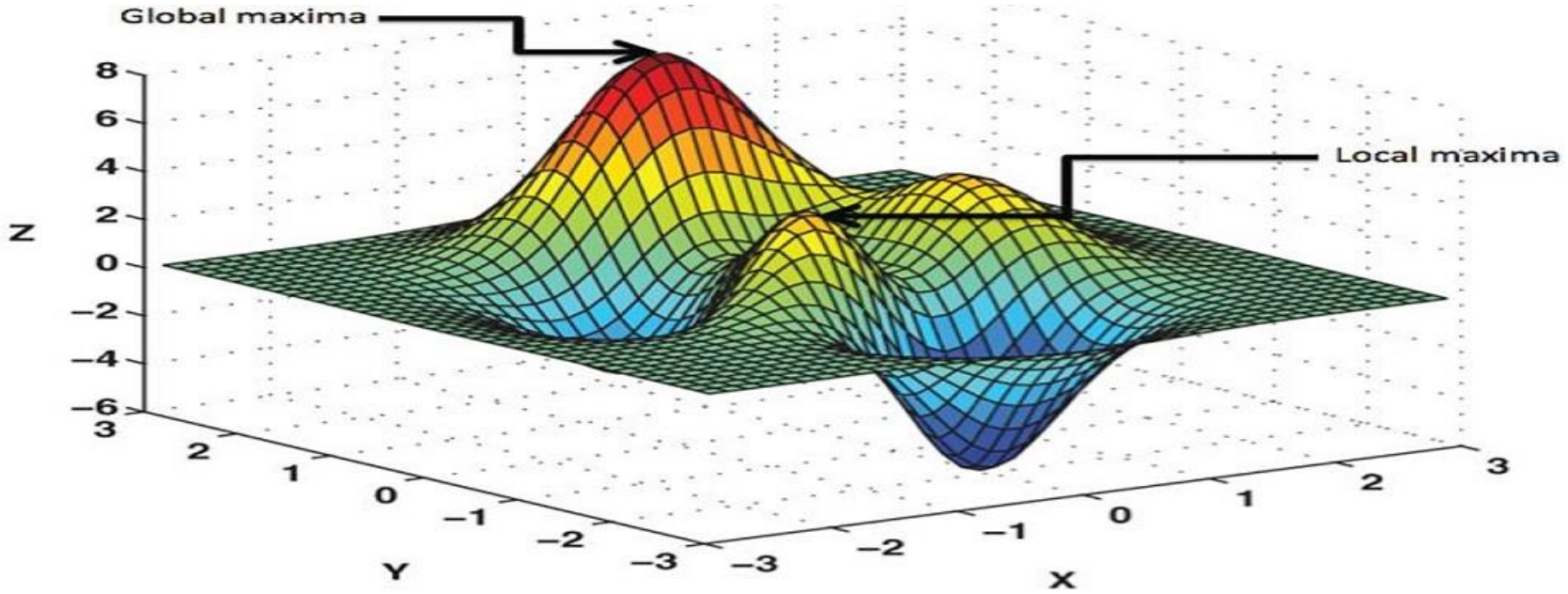
REFRENCES:

D. Kass. (2021). Gradient Descent with Linear Regression from Scratch. Retrieved from: https://dmitrijskass.netlify.app/2021/04/03/gradient-descent-with-linear-regression-from-scratch/

S. Sayad. Machine Learning Recipes. Retrieved from: http://saedsayad.com/mlr.htm

Loss Function

1	Loss functions in Regression based problem	a. Mean Square Error Loss
		b. Mean Absolute Error Loss
		c. Huber Loss
2	Loss functions in Binary classification-based problem	a. Binary Cross Entropy Loss
		b. Hinge Loss
3	Loss functions in Multiclass classification-based problem	a. Multiclass Cross Entropy Loss
		b. Spare Multiclass Cross Entropy Loss
		c. Kullback Leibler Divergence Loss



In [2]:
```
# Import Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
plt.style.use("fivethirtyeight")
```

Common loss functions

For Regression:-

1. Mean Absolute Error (MAE) | l_1 Loss function | L1-norm Loss Function (Least Absolute Error LAE):

$$MAE = \frac{1}{m}\sum_{i=1}^{m}|y^{(i)} - \hat{y}^{(i)}|$$

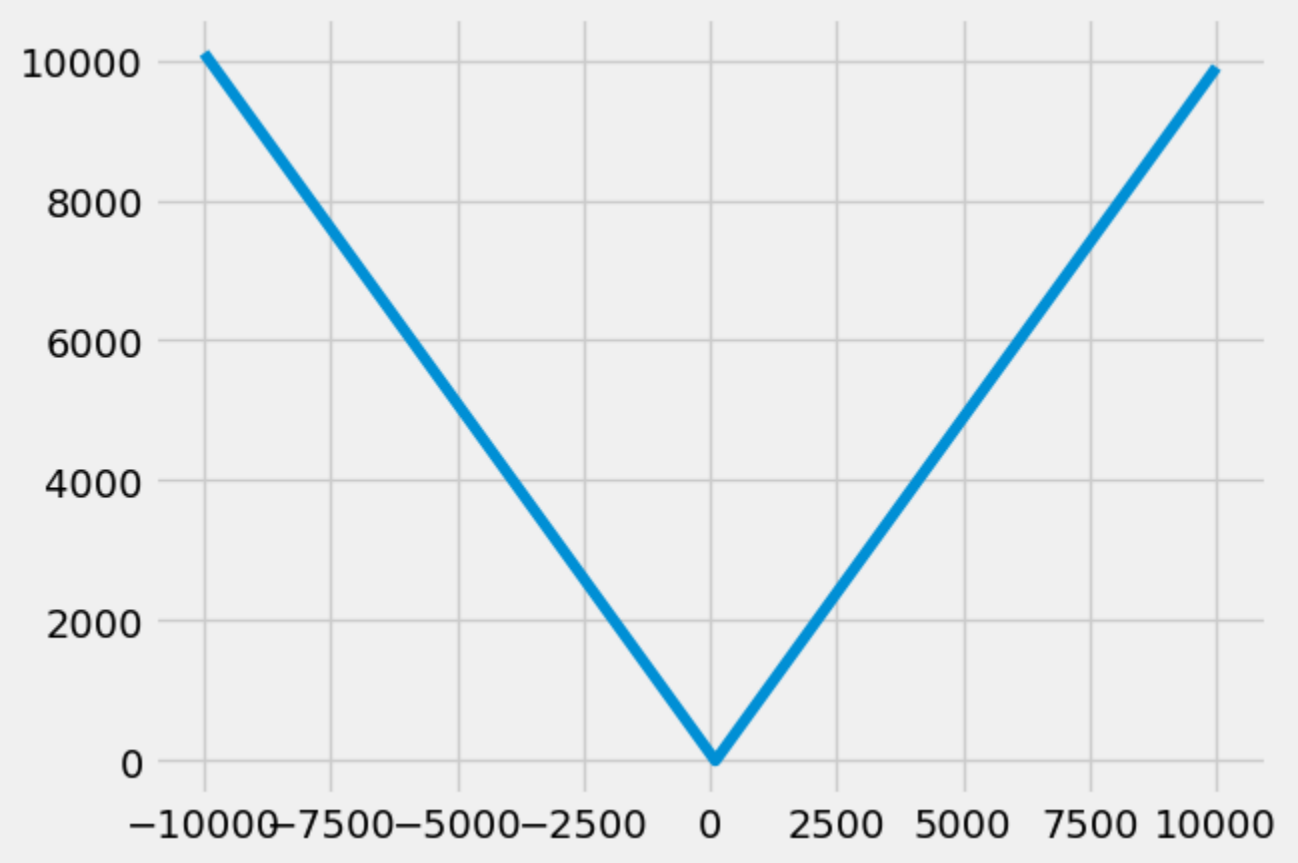
L1 loss function are also known as **Least Absolute Deviations** in short **LAD**.

It is used to minimize the error which is the mean of sum of all the absolute differences in between the true value and the predicted value.

In [3]:
```
def MAE(actual, pred):

  return np.abs(actual - pred)
```

In [4]:
```
actual_arr = np.repeat(100, 10000) ## actual [100, 100, 100]
pred_arr = np.arange(-10000, 10000, 2) ## predict [-10000. - 9998,......, 9998, 10000]

loss_mae = [MAE(actual, pred) for actual, pred in zip(actual_arr, pred_arr)]

plt.plot(pred_arr, loss_mae)
```

Out[4]:
```
[<matplotlib.lines.Line2D at 0x25b224d9880>]
```



In [5]:
```
total_loss = np.mean(np.sum(loss_mae))
total_loss
```

Out[5]:
```
50005100.0
```

2. Mean Squared Error (MSE) | l_2 Loss function | L2-norm Loss Function (Least Squares Error LSE):

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

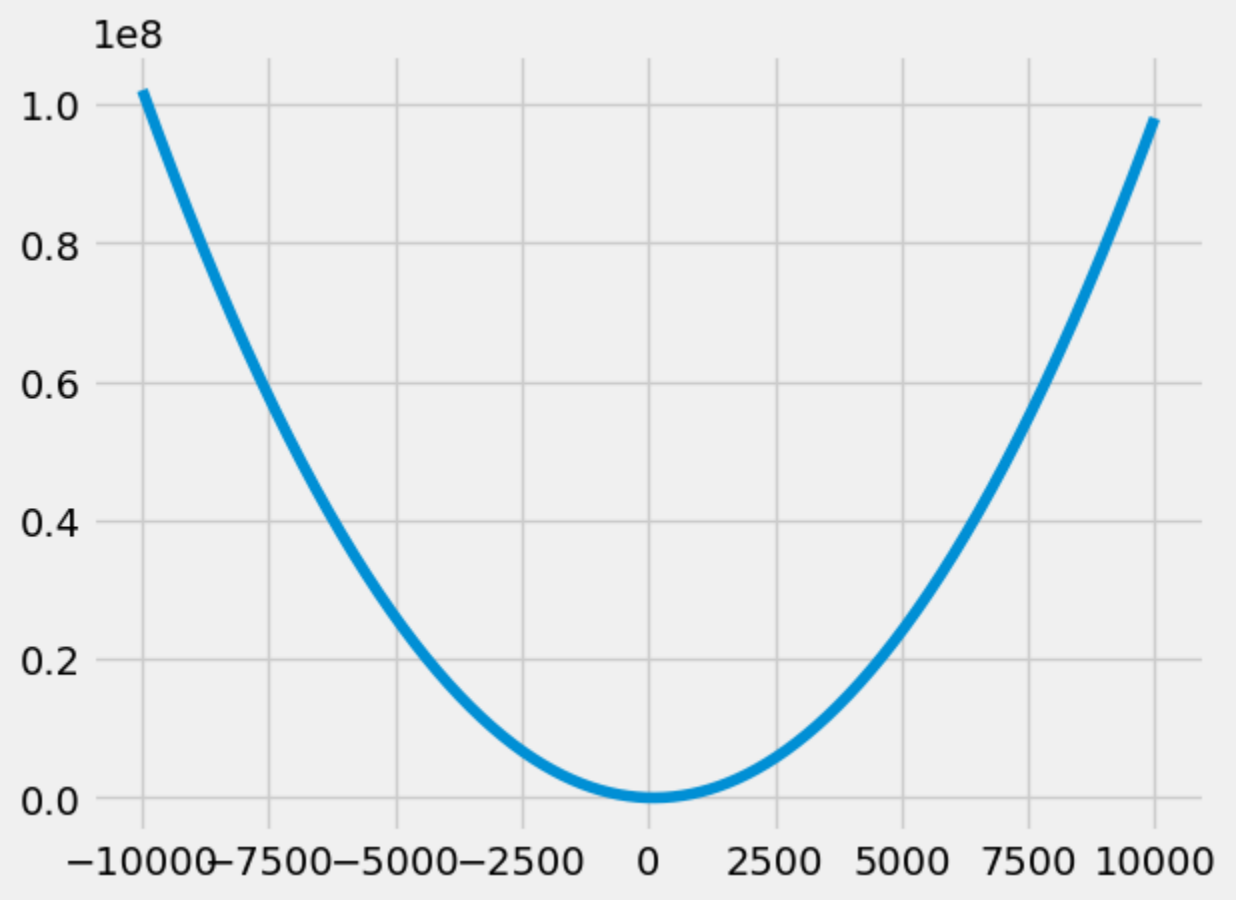
It is used to minimize the error which is the mean of sum of all the squared differences in between the true value and the predicted value.

The disadvantage of the **L2 norm** is that when there are outliers, these points will account for the main component of the loss.

In [6]:
```
def MSE(actual, pred):

  return np.square(actual - pred)
```

In [7]:
```
actual_arr = np.repeat(100, 10000)
pred_arr = np.arange(-10000, 10000, 2)

loss_mSe = [MSE(actual, pred) for actual, pred in zip(actual_arr, pred_arr)]

plt.plot(pred_arr, loss_mSe)
```

Out[7]:
```
[<matplotlib.lines.Line2D at 0x25b22c1f310>]
```



In [8]:
```
total_loss = np.mean(np.sum(loss_mSe))
total_loss
```

Out[8]:
```
-1572109088.0
```

3. Huber Loss

Huber Loss is often used in regression problems. Compared with L2 loss, Huber Loss is less sensitive to outliers(because if the residual is too large, it is a piecewise function, loss is a linear function of the residual).

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2, & for\ |y - \hat{y}| \le \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta), & otherwise \end{cases}$$

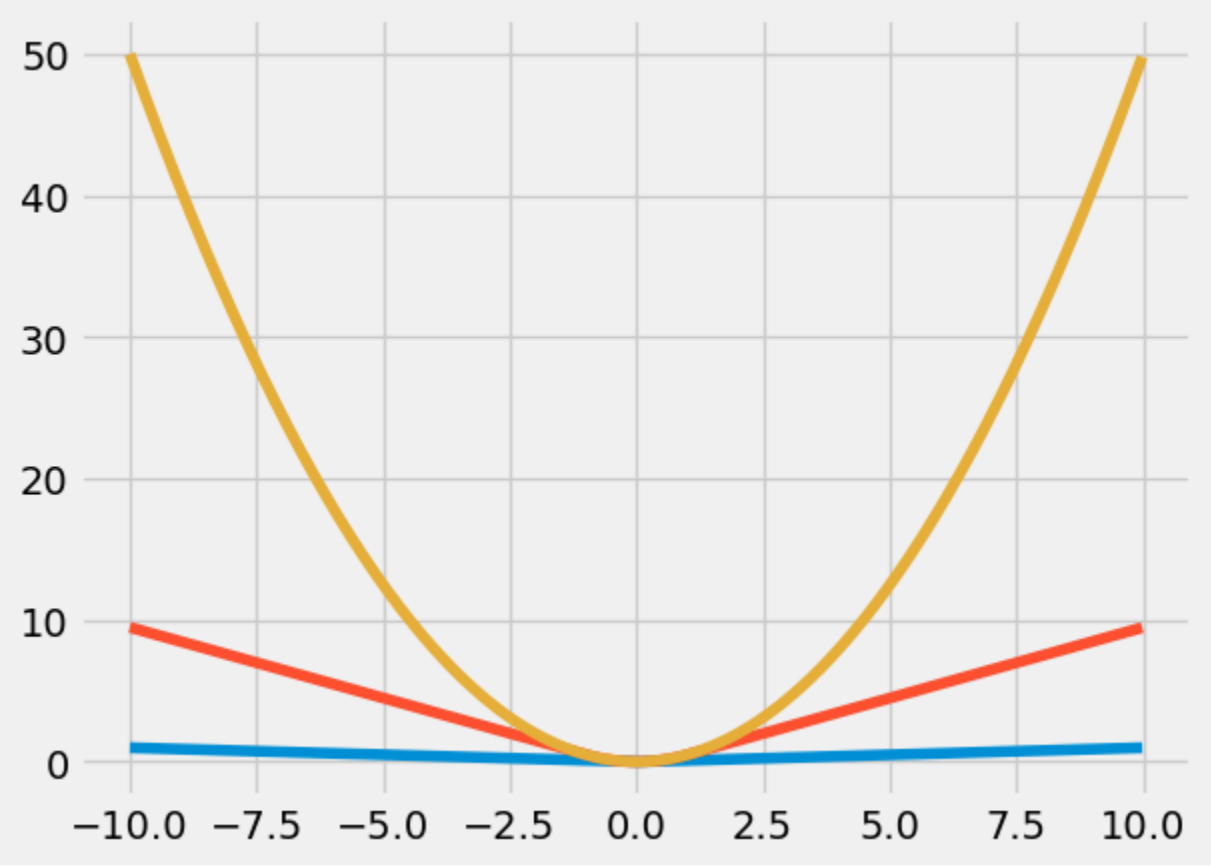
Among them, δ is a set parameter, y represents the real value, and $f(x)$ represents the predicted value.

The advantage of this is that when the residual is small, the loss function is L2 norm, and when the residual is large, it is a linear function of L1 norm

Wiki (https://en.wikipedia.org/wiki/Huber_loss)

In [9]:
```
def huber(true, pred, delta):
    """
    true: array of true values
    pred: array of predicted values

    returns: smoothed mean absolute error loss
    """
    loss = np.where(np.abs(true-pred) < delta , 0.5*((true-pred)**2), delta*np.abs(true
    return loss
```

In [10]:
```
actual_arr = np.repeat(0, 1000)
pred_arr = np.arange(-10, 10, 0.02)

deltas = [0.1, 1, 10]

loss_huber = [[huber(actual, pred, delta) for actual, pred in zip(actual_arr, pred_arr)]

for idx in range(len(deltas)):
  plt.plot(pred_arr, loss_huber[idx])
```



4. Pseudo Huber Loss

The Pseudo-Huber loss function can be used as a smooth approximation of the Huber loss function.

It combines the best properties of $L2$ squared loss and $L1$ absolute loss by being strongly convex when close to the target/minimum and less steep for extreme values.

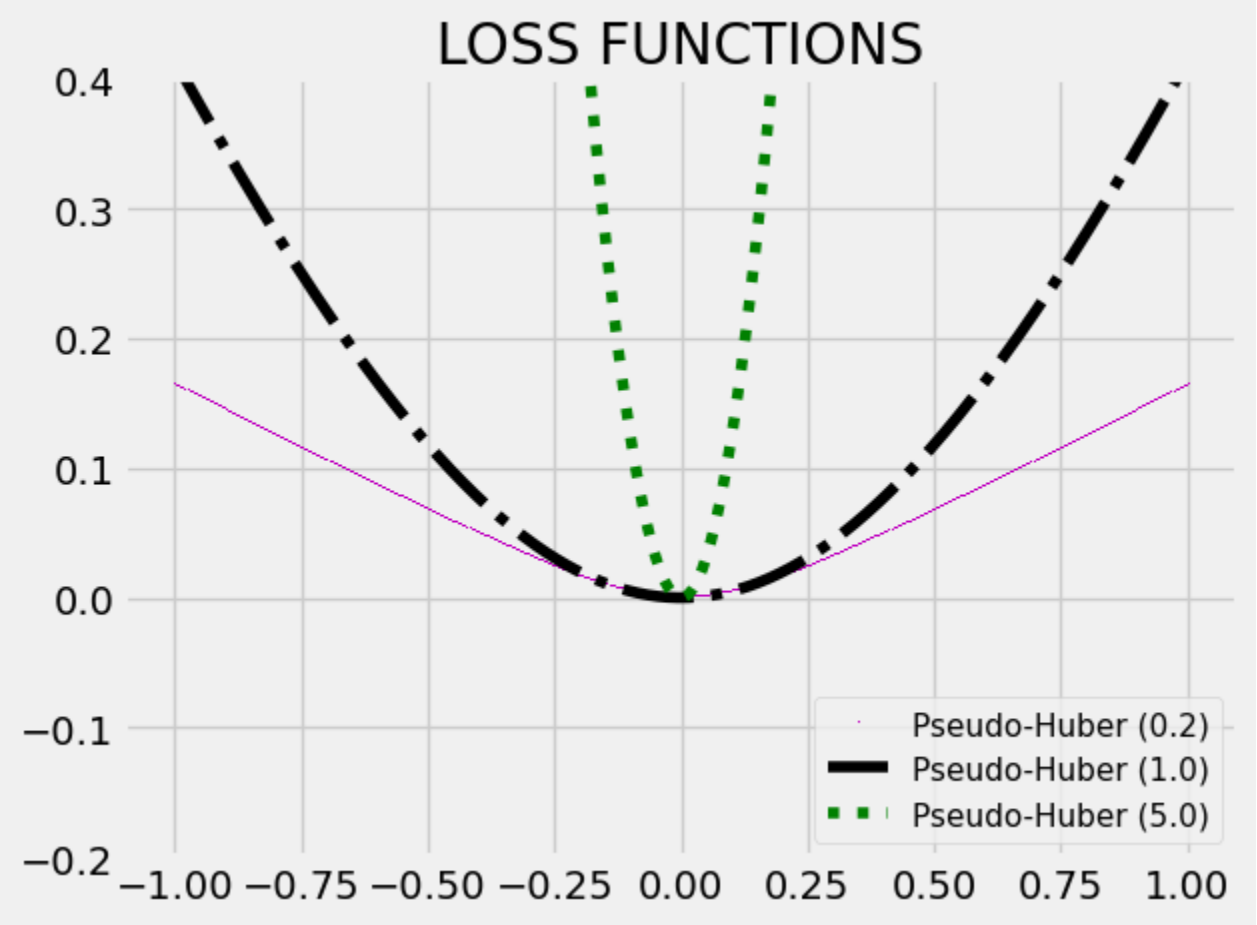
The scale at which the Pseudo-Huber loss function transitions from $L2$ loss for values close to the minimum to $L1$ loss for extreme values and the steepness at extreme values can be controlled by the δ value.

The Pseudo-Huber loss function ensures that derivatives are continuous for all degrees

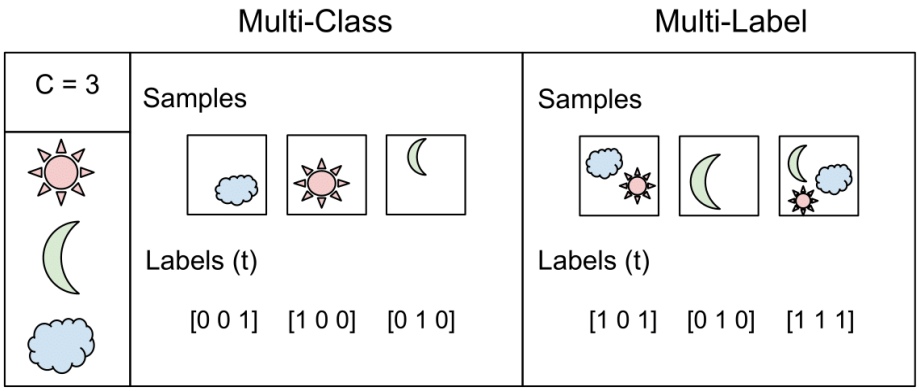
$$L_\delta(y, \hat{y}) = \delta^2(\sqrt{1 + ((y - \hat{y})/\delta)^2} - 1)$$

In [11]:
```
x_function = tf.linspace(-1., 1., 500)
target = tf.constant(0.)

delta1 = tf.constant(0.2)
pseudo_huber1_output = tf.multiply(tf.square(delta1), tf.sqrt(1. + tf.square((target - x
# pseudo_huber1_output = sess.run(pseudo_huber1)

delta2 = tf.constant(1.)
pseudo_huber2_output = tf.multiply(tf.square(delta2), tf.sqrt(1. + tf.square((target - x
# pseudo_huber2_output = sess.run(pseudo_huber2)

delta3 = tf.constant(5.)
pseudo_huber3_output = tf.multiply(tf.square(delta3), tf.sqrt(1. + tf.square((target - x
# pseudo_huber3_output = sess.run(pseudo_huber3)
```

In [12]:
```
x_array = x_function
plt.plot(x_array, pseudo_huber1_output, 'm,', label='Pseudo-Huber (0.2)')
plt.plot(x_array, pseudo_huber2_output, 'k-.', label='Pseudo-Huber (1.0)')
plt.plot(x_array, pseudo_huber3_output, 'g:', label='Pseudo-Huber (5.0)')
plt.ylim(-0.2, 0.4)
plt.legend(loc='lower right', prop={'size': 11})
plt.title('LOSS FUNCTIONS')
plt.show()
```



For Classification



5.Hinge Loss

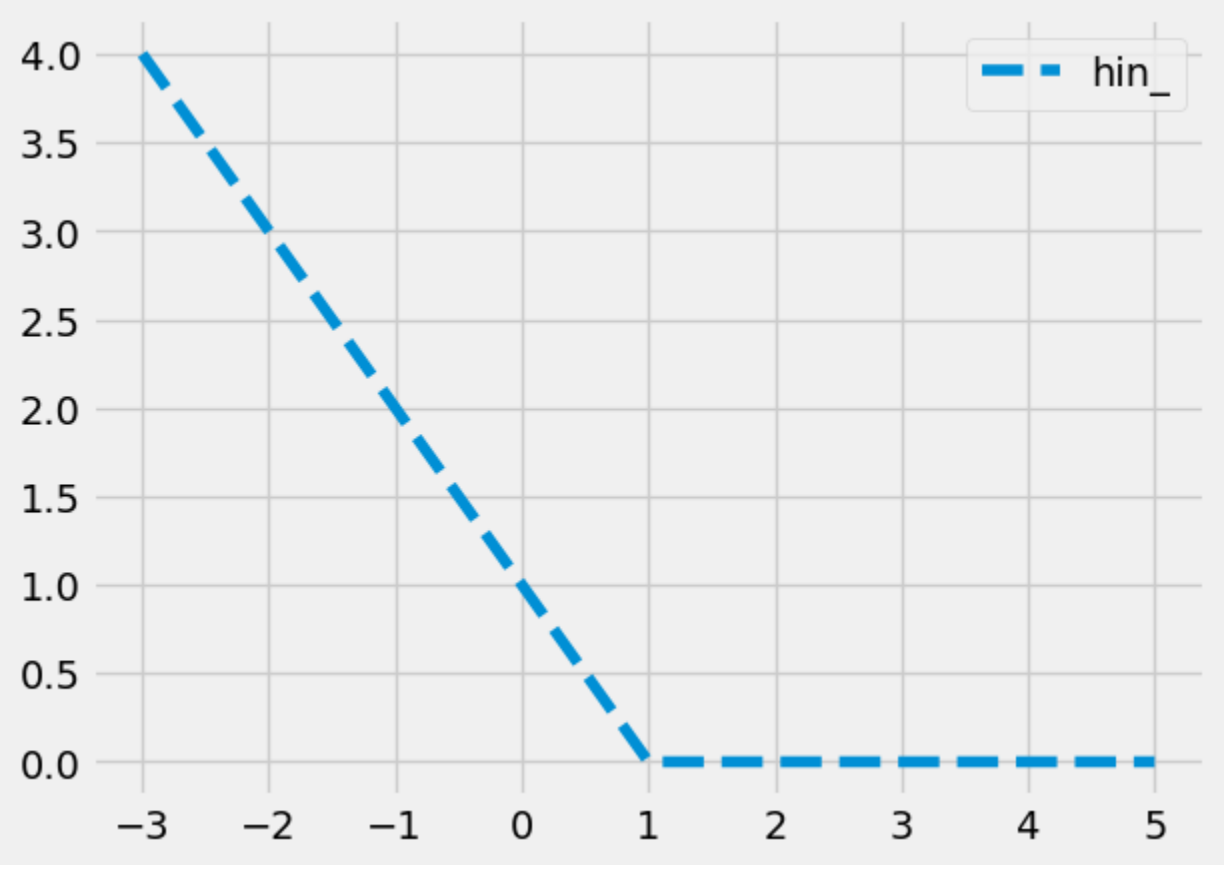
Hinge loss is often used for binary classification problems, such as ground true: t = 1 or -1, predicted value y = wx + b

In the svm classifier, the definition of hinge loss is

$$l(y) = max(0, 1 - t.y)$$

In other words, the closer the y is to t, the smaller the loss will be.

In [13]:
```
x_guess2 = tf.linspace(-3.,5.,500)
x_actual2 = tf.convert_to_tensor([1.]*500)

#Hinge loss
#hinge_loss = tf.losses.hinge_loss(labels=x_actual2, logits=x_guess2)
hinge_loss = tf.maximum(0.,1.-(x_guess2*x_actual2))
# with tf.Session() as sess:
x_,hin_ = [x_guess2, hinge_loss]
plt.plot(x_,hin_,'--', label='hin_')
plt.legend()
plt.show()
```



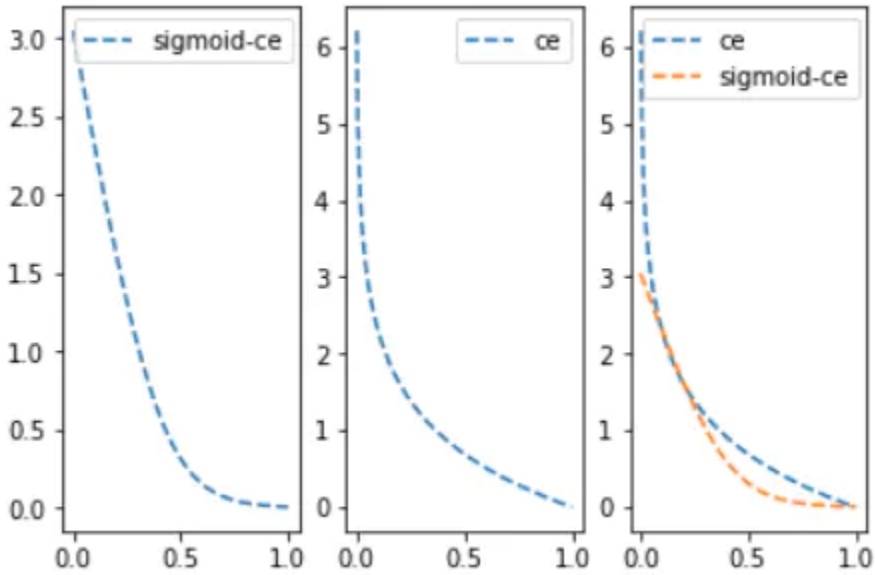
6.Cross-entropy loss

$$J(w) = -y.log(\hat{y}) - (1 - y).log(1 - \hat{y}) = -\sum_i p_i.log(q_i)$$

Cross-entropy loss is mainly applied to binary classification problems. The predicted value is a probability value and the loss is defined according to the cross entropy. Note the value range of the above value: the predicted value of y should be a probability and the value range is [0,1]

5.Sigmoid-Cross-entropy loss

The above cross-entropy loss requires that the predicted value is a probability. Generally, we calculate $scores = x * w + b$. Entering this value into the sigmoid function can compress the value range to (0,1).



It can be seen that the sigmoid function smoothen the predicted value(such as directly inputting 0.1 and 0.01 and inputting 0.1, 0.01 sigmoid and then entering, the latter will obviously have a much smaller change value), which makes the predicted value of sigmoid-ce far from the label loss growth is not so steep.

6.Softmax cross-entropy loss

First, the softmax function can convert a set of fraction vectors into corresponding probability vectors. Here is the definition of softmax function

$$softmax(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

As above, softmax also implements a vector of 'squashes' k-dimensional real value to the [0,1] range of k-dimensional, while ensuring that the cumulative sum is 1.

According to the definition of cross entropy, probability is required as input.Sigmoid-cross-entropy-loss uses sigmoid to convert the score vector into a probability vector, and softmax-cross-entropy-loss uses a softmax function to convert the score vector into a probability vector.

According to the definition of cross entropy loss.

$$H(p,q) = -\sum_{x} p(x) log(q(x))$$

where $p(x)$ represents the probability that classification x is a correct classification, and the value of p can only be 0 or 1. This is the prior value

$q(x)$ is the prediction probability that the x category is a correct classification, and the value range is (0,1)

So specific to a classification problem with a total of C types, then $p(x_j), (0 \leq j \leq C)$ must be only 1 and C-1 is 0(because there can be only one correct classification, correct the probability of classification as correct classification is 1, and the probability of the remaining classification as correct classification is 0)

Then the definition of softmax-cross-entropy-loss can be derived naturally.

Here is the definition of softmax-cross-entropy-loss.

$$L_i = -log(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}})$$

Type *Markdown* and LaTeX: α^2

Optimization

Challenges

- Choosing a proper learning rate can be difficult.
- Same learning rate applies to all parameter updates.
- Getting trapped in suboptimal local minima and saddle points.

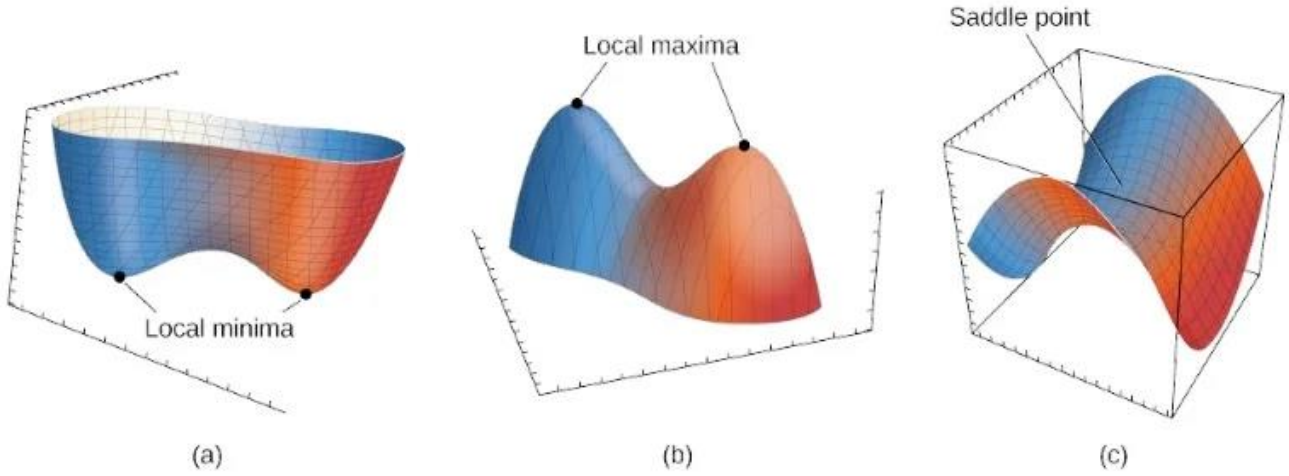


Image source: link

Optimizers

Batch gradient descent

Gradient update rule: BGD uses the data of the entire training set to calculate the gradient of the cost function to the parameters:

Disadvantages:

Because this method calculates **the gradient for the entire data set in one update, the calculation is very slow**, it will be very tricky to encounter a large number of data sets, and you cannot invest in new data to update the model in real time.

We will define an iteration number epoch in advance, first calculate the gradient vector params_grad, and then update the parameter params along the direction of the gradient. The learning rate determines how big we take each step.

Batch gradient descent can converge to a global minimum for convex functions and to a local minimum for non-convex functions.

SGD (Stochastic gradient descent)

Gradient update rule: Compared with BGD's calculation of gradients with all data at one time, SGD updates the gradient of each sample with each update.

```
x += - learning_rate * dx
```

where x is a parameter, dx is the gradient and learning rate is constant

For large data sets, there may be similar samples, so BGD calculates the gradient. **There will be redundancy, and SGD is updated only once, there is no redundancy, it is faster, and new samples can be added.**

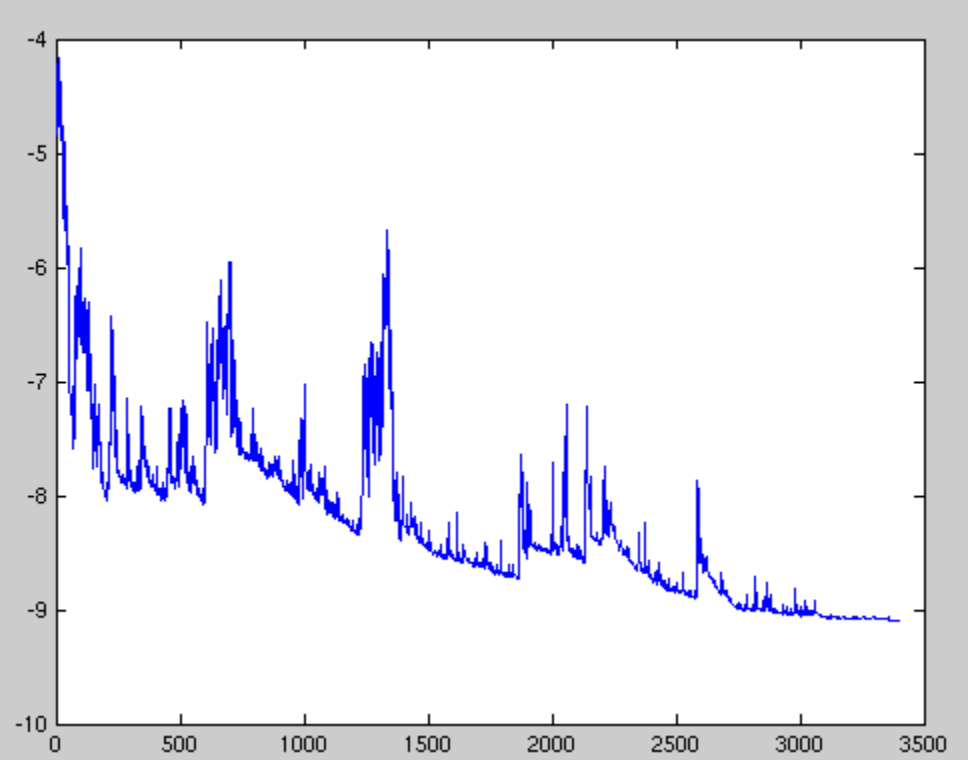


Figure :- Fluctuations in SGD

Disadvantages: However, because SGD is updated more frequently, the cost function will have severe oscillations. BGD can converge to a local minimum, of course, the oscillation of SGD may jump to a better local minimum.

When we decrease the learning rate slightly, the convergence of SGD and BGD is the same.

Mini-batch gradient descent

Gradient update rule:

MBGD uses a small batch of samples, that is, n samples to calculate each time. In this way, it can reduce the variance when the parameters are updated, and the convergence is more stable. It can make full use of the highly optimized matrix operations in the deep learning library for more efficient gradient calculations.

The difference from SGD is that each cycle does not act on each sample, but a batch with n samples.

> Setting value of hyper-parameters: n Generally value is 50 ~ 256

Cons:

- Mini-batch gradient descent does not guarantee good convergence,
- If the learning rate is too small, the convergence rate will be slow. If it is too large, the loss function will oscillate or even deviate at the minimum value. One measure is to set a **larger learning rate**. When the change between two iterations is lower than a certain threshold, the learning rate is reduced.

However, the setting of this threshold needs to be written in advance adapt to the characteristics of the data set.

In addition, this method is to apply the **same learning rate** to all parameter updates. If our data is sparse, we would prefer to update the features with lower frequency.

In addition, for **non-convex functions**, it is also necessary to avoid trapping at the local minimum or saddle point, because the error around the saddle point is the same, the gradients of all dimensions are close to 0, and SGD is easily trapped here.

Saddle points are the curves, surfaces, or hyper surfaces of a saddle point neighborhood of a smooth function are located on different sides of a tangent to this point. For example, this two-dimensional figure looks like a saddle: it curves up in the x-axis direction and down in the y-axis direction, and the saddle point is (0,0).

Momentum

One disadvantage of the SGD method is that its update direction depends entirely on the current batch, so its update is very unstable. A simple way to solve this problem is to introduce momentum.

Momentum is momentum, which simulates the inertia of an object when it is moving, that is, the direction of the previous update is retained to a certain extent during the update, while the current update gradient is

Adagrad

Adagrad is an algorithm for gradient-based optimization which adapts the learning rate to the parameters, using low learning rates for parameters associated with frequently occurring features, and using high learning rates for parameters associated with infrequent features.

So, it is well-suited for dealing with sparse data.

But the same update rate may not be suitable for all parameters. For example, some parameters may have reached the stage where only fine-tuning is needed, but some parameters need to be adjusted a lot due to the small number of corresponding samples.

Adagrad proposed this problem, an algorithm that adaptively assigns different learning rates to various parameters among them. The implication is that for each parameter, as its total distance updated increases, its learning rate also slows.

> **GloVe word embedding uses adagrad where infrequent words required a greater update and frequent words require smaller updates.**

> **Adagrad eliminates the need to manually tune the learning rate.**

Adadelta

There are three problems with the Adagrad algorithm

- The learning rate is monotonically decreasing.
- The learning rate in the late training period is very small.
- It requires manually setting a global initial learning rate.

> **Adadelta is an extension of Adagrad and it also tries to reduce Adagrad's aggressive, monotonically reducing the learning rate.**

> It does this by restricting the window of the past accumulated gradient to some fixed size of w. Running average at time t then depends on the previous average and the current gradient.

> In Adadelta we do not need to set the default learning rate as we take the ratio of the running average of the previous time steps to the current gradient.

RMSProp

The full name of RMSProp algorithm is called **Root Mean Square Prop**, which is an adaptive learning rate optimization algorithm proposed by Geoff Hinton.

> RMSProp tries to resolve Adagrad's radically diminishing learning rates by using a moving average of the squared gradient. It utilizes the magnitude of the recent gradient descents to normalize the gradient.

Adagrad will accumulate all previous gradient squares, and RMSprop just calculates the corresponding average value, so it can alleviate the problem that the learning rate of the Adagrad algorithm drops quickly.

The difference is that RMSProp calculates the **differential squared weighted average of the gradient** . This method is beneficial to eliminate the direction of large swing amplitude, and is used to correct the swing amplitude, so that the swing amplitude in each dimension is smaller. On the other hand, it also makes the network function converge faster.

> In RMSProp learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.

> RMSProp divides the learning rate by the average of the exponential decay of squared gradients

Adam

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop.

> Adam also keeps an exponentially decaying average of past gradients, similar to momentum.

Adam can be viewed as a combination of Adagrad and RMSprop,(Adagrad) which works well on sparse gradients and (RMSProp) which works well in online and non-stationary settings respectively.

Adam implements the **exponential moving average of the gradients** to scale the learning rate instead of a simple average as in Adagrad. It keeps an exponentially decaying average of past gradients.

Adam is computationally efficient and has very less memory requirement.

Adam optimizer is one of the most popular and famous gradient descent optimization

Comparisons

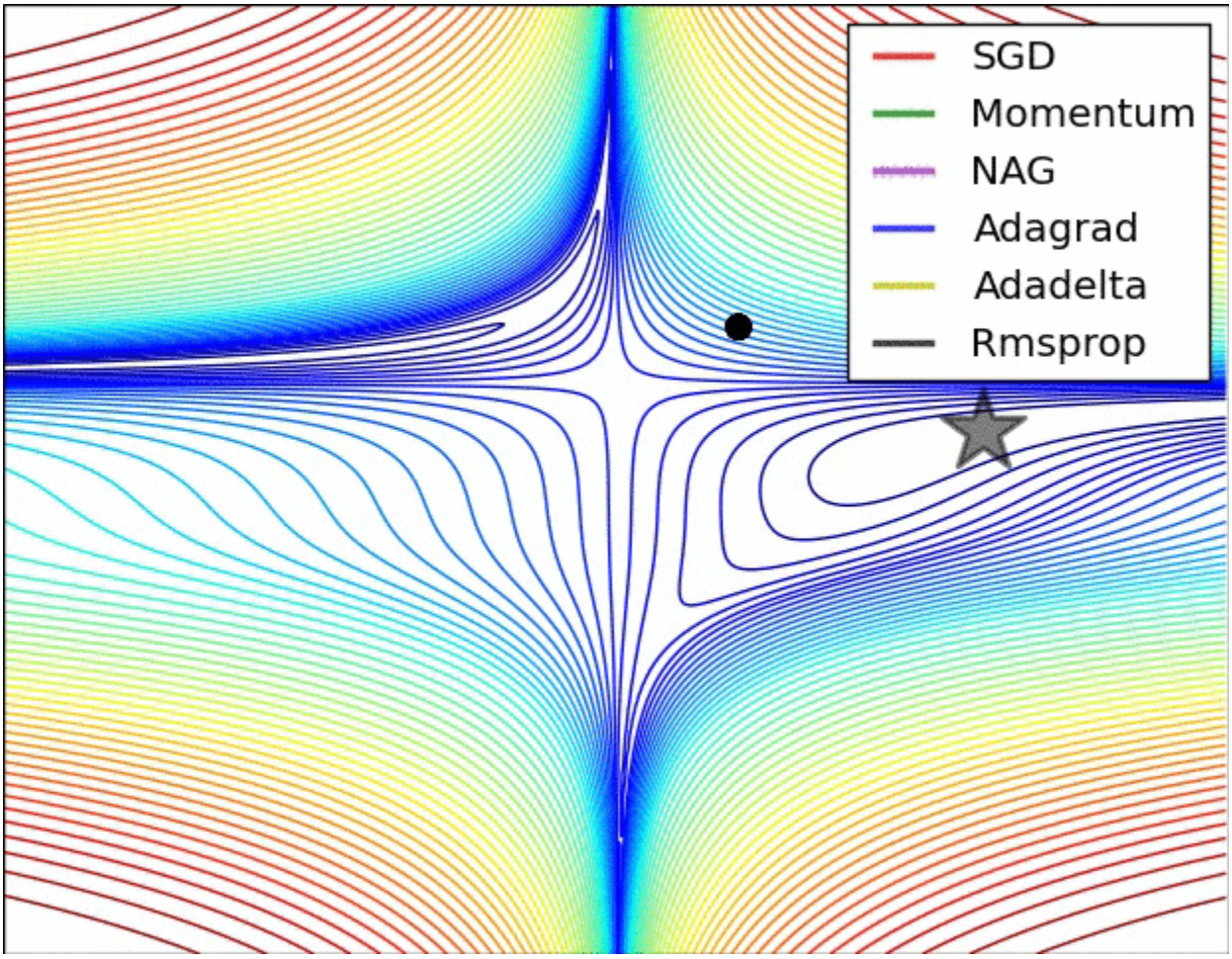


Figure :- SGD optimization on loss surface contours

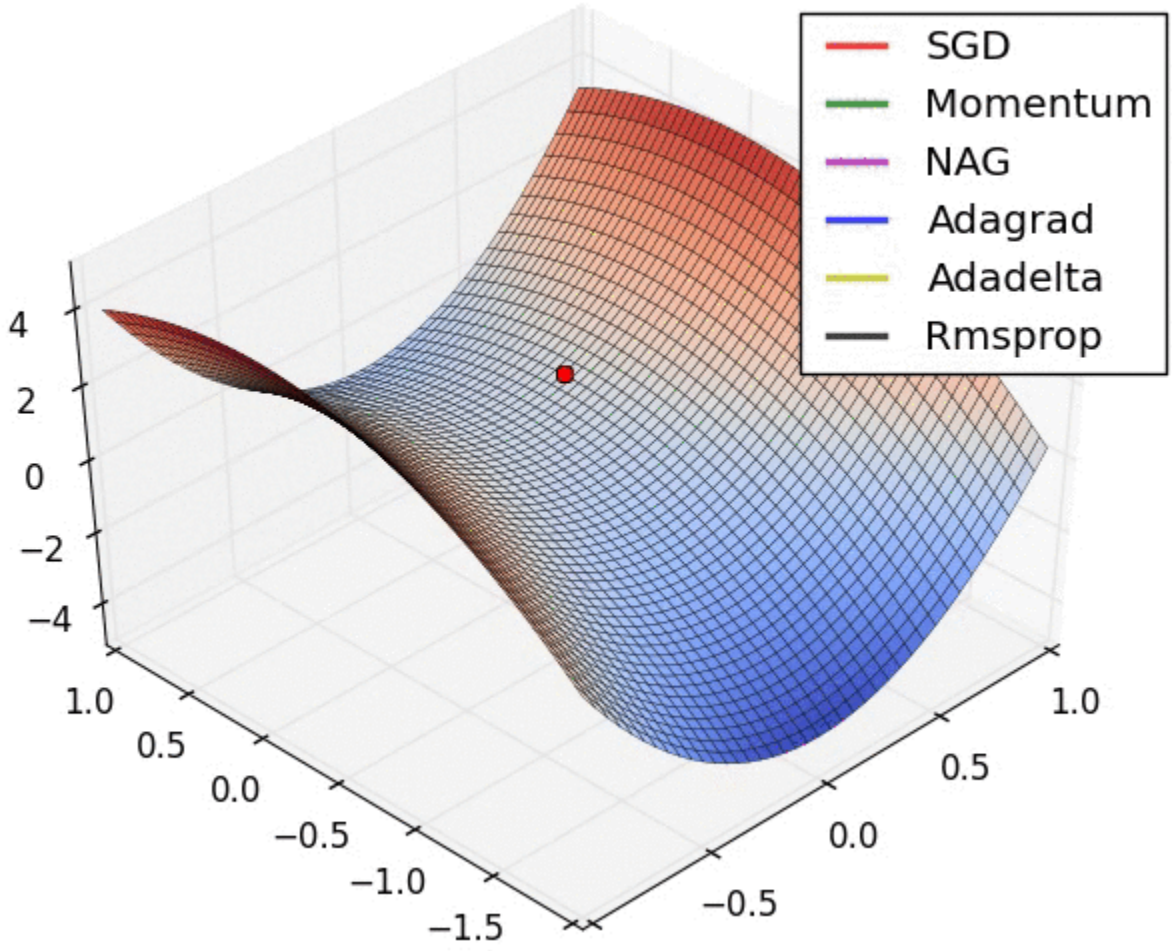


Figure :- SGD optimization on saddle point

How to choose optimizers?

- If the data is sparse, use the self-applicable methods, namely Adagrad, Adadelta, RMSprop, Adam.
- RMSprop, Adadelta, Adam have similar effects in many cases.
- Adam just added bias-correction and momentum on the basis of RMSprop,
- As the gradient becomes sparse, Adam will perform better than RMSprop.

In [ ]:

Assignment Question

Q1: What is the role of optimization algorithms in artificial neural networks? Why are they necessary?

A A1: Optimization algorithms play a crucial role in artificial neural networks as they are responsible for updating the network's parameters during the training process. The main objective of training a neural network is to minimize the loss function, which measures the difference between the predicted outputs and the actual targets. Optimization algorithms are necessary because they determine how the network should adjust its weights and biases to reach the optimal configuration that minimizes the loss and improves the model's performance.

Q2: Explain the concept of gradient descent and its variants. Discuss their differences and tradeoffs in terms of convergence speed and memory requirements.

A A2: Gradient descent is a popular optimization algorithm used in training neural networks. It works by calculating the gradient of the loss function with respect to the model parameters and then updating the parameters in the opposite direction of the gradient to minimize the loss. The basic variants of gradient descent include:

- Batch Gradient Descent: It computes the gradient using the entire training dataset, making it computationally expensive and memory-intensive. However, it provides a more stable convergence path, leading to better convergence.
- Stochastic Gradient Descent (SGD): This variant randomly selects one training sample at a time to compute the gradient, which reduces memory requirements but introduces more noise and oscillations during training. It converges faster in many cases but might not reach the global minimum.
- Mini-batch Gradient Descent: It strikes a balance between batch and stochastic gradient descent. It randomly samples a small subset (mini-batch) of the training data to compute the gradient, combining the advantages of both batch and SGD. It is widely used due to its efficiency and convergence properties.

Q3: Describe the challenges associated with traditional gradient descent optimization methods (e.g., slow convergence, local minima). How do modern optimizers address these challenges?

A A3: Traditional gradient descent methods, such as Batch Gradient Descent and Stochastic Gradient Descent, face several challenges during training:

- Slow Convergence: Traditional methods may take a long time to converge to the optimal solution, especially when dealing with complex and high-dimensional data.
- Local Minima: They can get trapped in local minima, preventing the model from finding the global minimum of the loss function.

Modern optimization algorithms address these challenges in the following ways:

- Momentum: Momentum is a concept in optimization that accelerates the convergence by adding a fraction of the previous update direction to the current update direction. This helps in overcoming slow convergence and escaping shallow local minima.
- Adaptive Learning Rates: Modern optimizers use adaptive learning rate techniques that adjust the learning rate during training to control the step size in parameter updates. This approach enables faster convergence by taking larger steps when the gradients are steep and smaller steps in flatter regions.

Q4: Discuss the concepts of momentum and learning rate in the context of optimization algorithms. How do they impact convergence and model performance?

A A4: Momentum and learning rate are essential concepts in optimization algorithms:

- Momentum: Momentum introduces inertia to the parameter updates, helping the optimizer to continue moving in the same direction as previous updates. This accelerates convergence and smoothes the optimization path, reducing oscillations. It can help escape local minima and speed up convergence, especially in areas with high curvature.
- Learning Rate: The learning rate controls the step size taken during parameter updates. A large learning rate may lead to overshooting and unstable updates, while a small learning rate may slow down convergence. It needs to be carefully tuned to find the right balance between fast convergence and avoiding divergence.
- Both momentum and learning rate significantly impact convergence and model performance. Properly tuned momentum can help the optimizer navigate complex loss surfaces, while a suitable learning rate is crucial for achieving fast and stable convergence without overshooting the optimal solution.

Part 2: Optimizer Techniques

Q1: Explain the concept of Stochastic Gradient Descent (SGD) and its advantages compared to traditional gradient descent. Discuss its limitations and scenarios where it is most suitable.

A A1: Stochastic Gradient Descent (SGD) is an optimization technique that computes the gradient and updates the model's parameters for each individual training sample. Its advantages over traditional gradient descent methods include:

Faster Updates: Since it processes one training sample at a time, it updates the parameters more frequently, leading to faster convergence, especially in large datasets.

Lower Memory Requirements: SGD uses less memory as it only needs to store information about a single sample at a time, making it suitable for training on large datasets that do not fit entirely in memory.

Escaping Local Minima: The noise introduced by the randomness of sample selection in SGD can help escape shallow local minima, leading to the possibility of finding better solutions.

However, SGD has some limitations:

Noisy Updates: The stochastic nature of SGD can cause noisy updates, leading to fluctuations in the optimization path, which may slow down convergence.

Learning Rate Sensitivity: It requires careful tuning of the learning rate, as a large learning rate can lead to divergence, while a small learning rate may slow down convergence.

SGD is most suitable when working with large datasets where memory constraints are an issue and when the optimization landscape has many local minima, as it increases the chances of finding better solutions.

Q2: Describe the concept of the Adam optimizer and how it combines momentum and adaptive learning rates. Discuss its benefits and potential drawbacks.

A A2: The Adam optimizer is an extension of the Stochastic Gradient Descent with momentum. It combines the advantages of both momentum and adaptive learning rates. The key features of Adam are:

Momentum: Adam uses momentum, just like traditional momentum-based optimization, to smooth the optimization path and speed up convergence.

Adaptive Learning Rates: It incorporates adaptive learning rates for each parameter based on the historical gradient information. It maintains separate learning rates for each parameter, allowing faster convergence for frequently updated parameters and more stability for less frequently updated ones.

Benefits of Adam:

Fast Convergence: Adam typically converges faster compared to standard stochastic gradient descent methods due to its adaptive learning rates.

Robustness: It performs well across a wide range of different architectures and datasets, requiring less manual tuning of hyperparameters.

Potential Drawbacks:

Memory Intensive: Adam needs to store the historical gradient information for each parameter, making it more memory-intensive compared to basic SGD.

Sensitivity to Learning Rate: While Adam adapts the learning rates, it can still be sensitive to the initial learning rate and may require tuning.

Q3: Explain the concept of RMSprop optimizer and how it addresses the challenges of adaptive learning rates. Compare it with Adam and discuss their relative strengths and weaknesses.

A A3: RMSprop (Root Mean Square Propagation) is an optimization algorithm that addresses the challenges of adaptive learning rates. It works by dividing the learning rate for each parameter by the root mean square of the historical gradients for that parameter. The formula for RMSprop update is similar to that of Adam but lacks the momentum term.

Comparison and Trade-offs:

Adaptive Learning Rates: Both Adam and RMSprop adapt learning rates based on historical gradients, allowing them to perform well in various situations without extensive manual tuning.

Momentum: Adam includes momentum, which helps it accumulate velocity and overcome potential noisy updates. RMSprop lacks momentum and, therefore, may exhibit more oscillations in the optimization path.

Memory Requirements: RMSprop requires less memory compared to Adam since it does not store the momentum information.

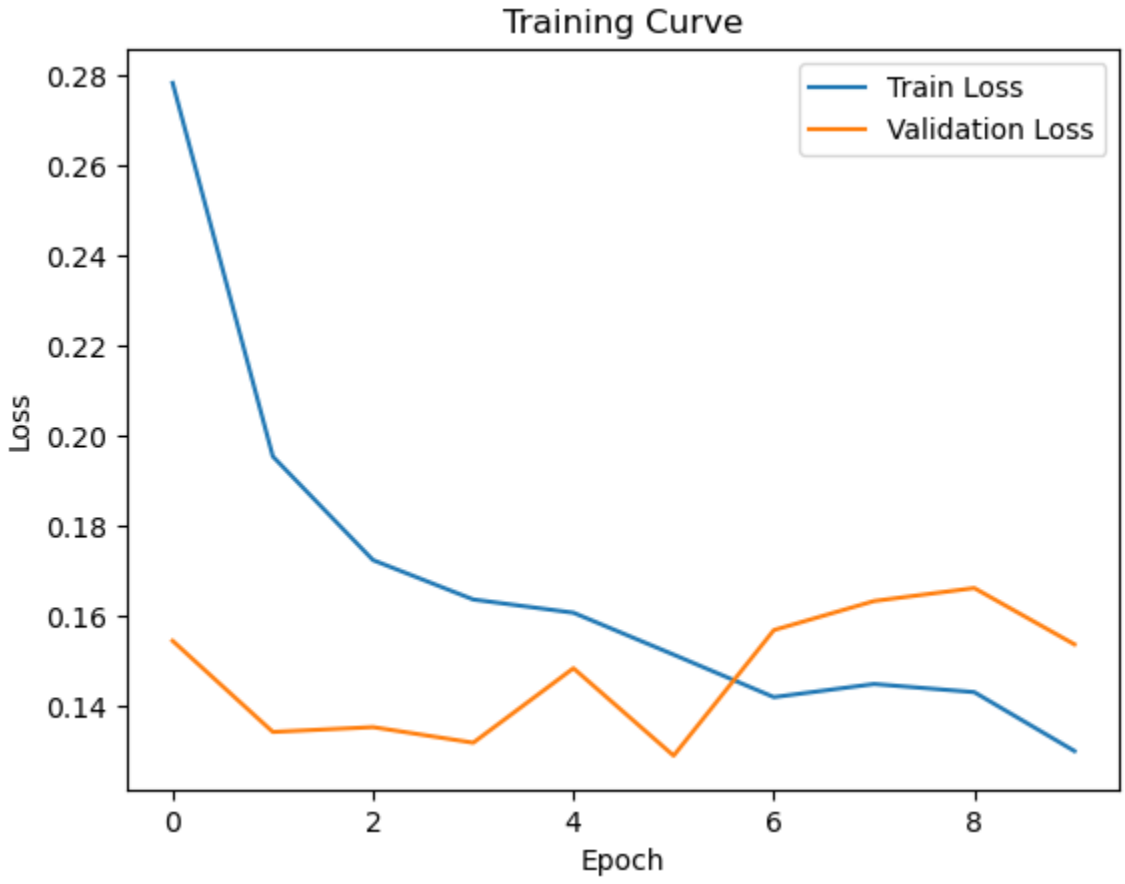
Performance: Adam often shows faster convergence than RMSprop, but this can vary depending on the dataset and architecture.

Choosing between RMSprop and Adam depends on the specific problem, and it is advisable to experiment with both optimizers to determine which one performs better in a given scenario. Generally, Adam is preferred when faster convergence is crucial, while RMSprop can be a good

In [1]:

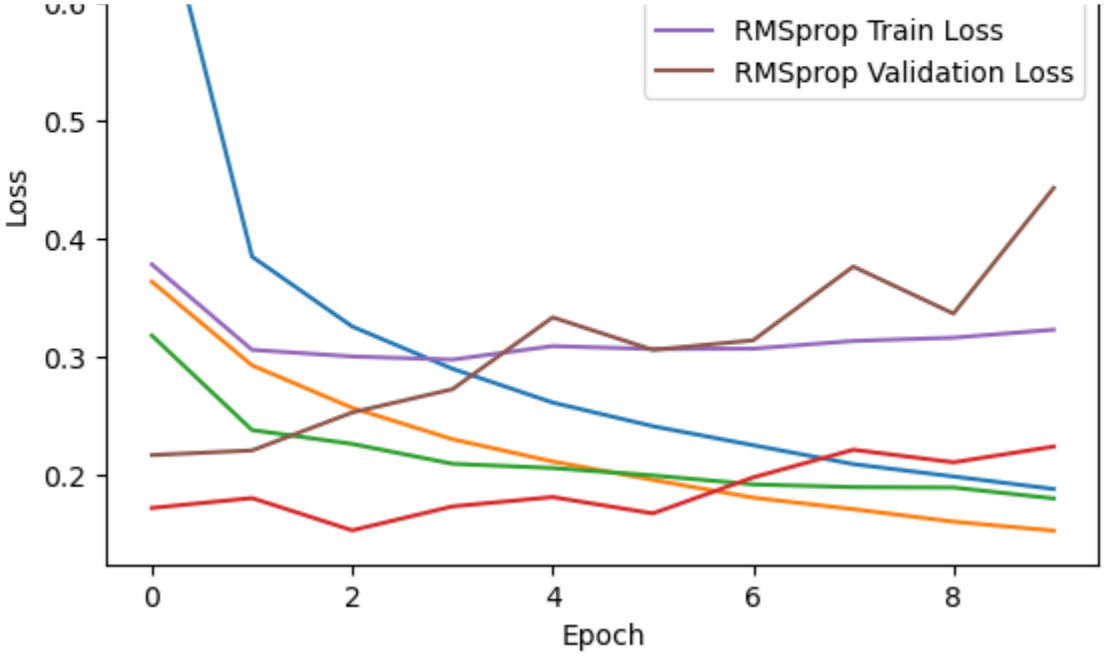
```python
import tensorflow as tf
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Define the deep learning model
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Function to get the chosen optimizer
def get_optimizer(optimizer_name, learning_rate):
    if optimizer_name == 'SGD':
        return tf.keras.optimizers.SGD(learning_rate=learning_rate)
    elif optimizer_name == 'Adam':
        return tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif optimizer_name == 'RMSprop':
        return tf.keras.optimizers.RMSprop(learning_rate=learning_rate)
    else:
        raise ValueError("Invalid optimizer name")

# Compile the model with the chosen optimizer
optimizer_name = 'Adam'  # Replace with 'Adam' or 'RMSprop' to use different optimizers
learning_rate = 0.01    # Experiment with different learning rates
optimizer = get_optimizer(optimizer_name, learning_rate)
model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['acc

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test,

# Compare model performance
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Curve')
plt.show()
```

```
Epoch 1/10
938/938 [==============================] - 9s 7ms/step - loss: 0.2784 - accuracy: 0.916
1 - val_loss: 0.1545 - val_accuracy: 0.9533
Epoch 2/10
938/938 [==============================] - 7s 8ms/step - loss: 0.1955 - accuracy: 0.943
4 - val_loss: 0.1343 - val_accuracy: 0.9592
Epoch 3/10
938/938 [==============================] - 6s 7ms/step - loss: 0.1725 - accuracy: 0.949
4 - val_loss: 0.1354 - val_accuracy: 0.9614
Epoch 4/10
938/938 [==============================] - 5s 6ms/step - loss: 0.1637 - accuracy: 0.952
8 - val_loss: 0.1319 - val_accuracy: 0.9652
Epoch 5/10
938/938 [==============================] - 6s 6ms/step - loss: 0.1608 - accuracy: 0.954
6 - val_loss: 0.1484 - val_accuracy: 0.9651
Epoch 6/10
938/938 [==============================] - 6s 7ms/step - loss: 0.1515 - accuracy: 0.957
6 - val_loss: 0.1290 - val_accuracy: 0.9675
Epoch 7/10
938/938 [==============================] - 5s 5ms/step - loss: 0.1420 - accuracy: 0.959
8 - val_loss: 0.1569 - val_accuracy: 0.9658
Epoch 8/10
938/938 [==============================] - 5s 5ms/step - loss: 0.1449 - accuracy: 0.961
1 - val_loss: 0.1634 - val_accuracy: 0.9631
Epoch 9/10
938/938 [==============================] - 5s 5ms/step - loss: 0.1431 - accuracy: 0.961
1 - val_loss: 0.1662 - val_accuracy: 0.9652
Epoch 10/10
938/938 [==============================] - 5s 5ms/step - loss: 0.1301 - accuracy: 0.964
5 - val_loss: 0.1537 - val_accuracy: 0.9666
```



In [2]:

```
# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Function to get the chosen optimizer
def get_optimizer(optimizer_name, learning_rate):
    if optimizer_name == 'SGD':
        return tf.keras.optimizers.SGD(learning_rate=learning_rate)
    elif optimizer_name == 'Adam':
        return tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif optimizer_name == 'RMSprop':
        return tf.keras.optimizers.RMSprop(learning_rate=learning_rate)
    else:
        raise ValueError("Invalid optimizer name")

# Function to create and train the model with the given optimizer
def train_model(optimizer_name, learning_rate):
    optimizer = get_optimizer(optimizer_name, learning_rate)
    model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=[
    history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_t
    return history

# List of optimizers to compare
optimizers = ['SGD', 'Adam', 'RMSprop']

# Train and evaluate models with different optimizers
for optimizer_name in optimizers:
    learning_rate = 0.01  # Experiment with different learning rates
    history = train_model(optimizer_name, learning_rate)

    # Plot training curves for each optimizer
    plt.plot(history.history['loss'], label=f'{optimizer_name} Train Loss')
    plt.plot(history.history['val_loss'], label=f'{optimizer_name} Validation Loss')

# Visualize the training curves for all optimizers
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Curves for Different Optimizers')
plt.show()
```



Let's analyze the results:

SGD:

Final Training Accuracy: 94.65% Final Validation Accuracy: 95.60% Final Training Loss: 0.1883 Final Validation Loss: 0.1530 Adam:

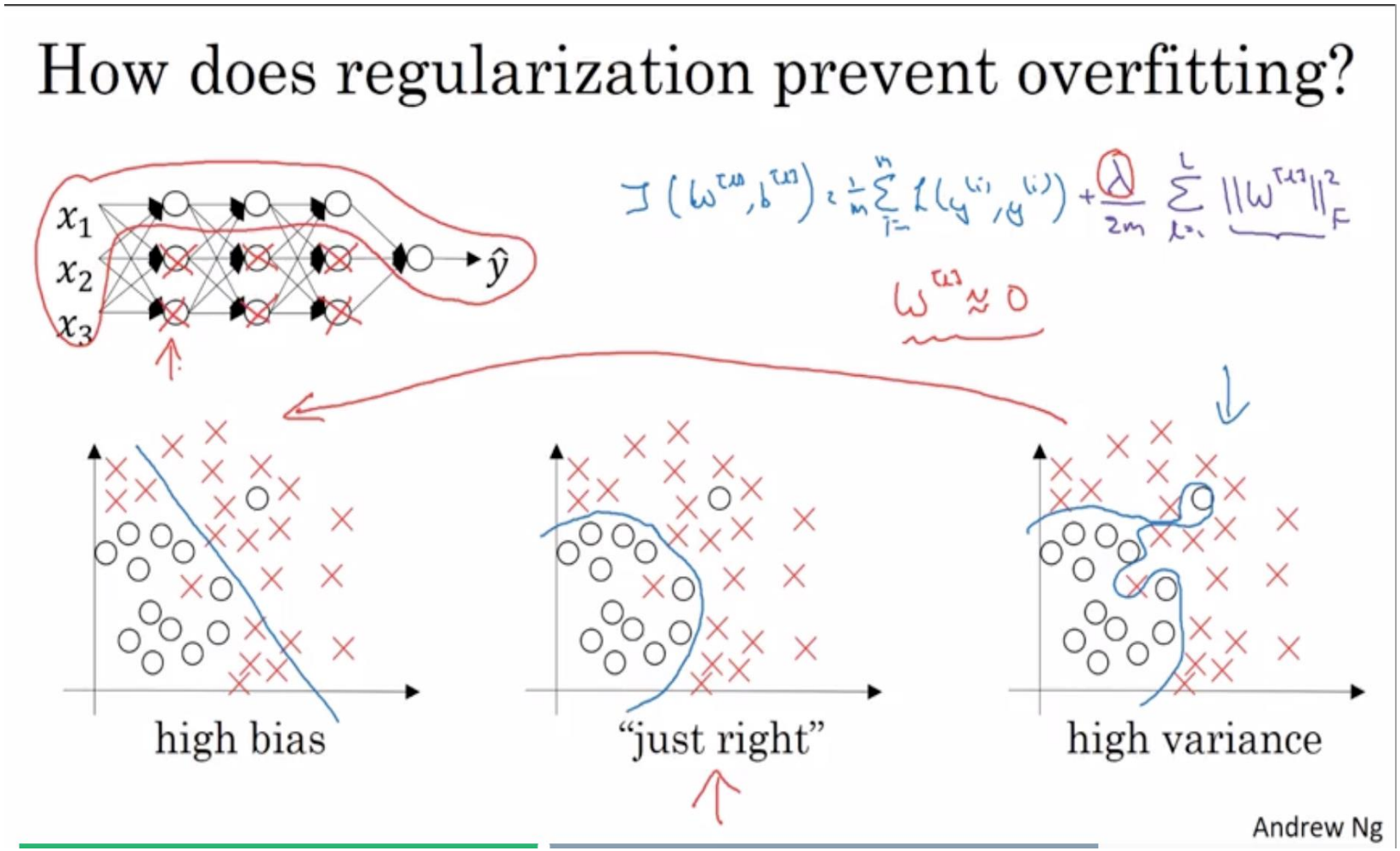
Final Training Accuracy: 95.49% Final Validation Accuracy: 95.91% Final Training Loss: 0.1803 Final Validation Loss: 0.2242 RMSprop:

Final Training Accuracy: 95.76% Final Validation Accuracy: 96.45% Final Training Loss: 0.3231 Final Validation Loss: 0.4432

Based on the provided results, RMSprop achieved the highest validation accuracy of 96.45% and the lowest validation loss of 0.4432. It appears that RMSprop performed the best among the three optimizers on the given neural network architecture and MNIST dataset.

In [ ]:

Regularization



Avoiding Overfitting Through Regularization

Regularization is the process of adding information in order to solve an ill-posed problem or to prevent overfitting. Regularization can be applied to objective functions in ill-posed optimization problems.

L1 Regularization | Lasso | Least Absolute:

$j_n(\theta) = j_0(\theta) + \alpha \sum_{i=1}^{m} |\theta_i|$

L2 Regularization | Ridge

$j_n(\theta) = j_0(\theta) + \frac{\alpha}{2} \sum_{i=1}^{m} (\theta_i)^2$

L1 - L2 Regularization

$j_n(\theta) = j_0(\theta) + r\alpha \sum_{i=1}^{m} |\theta_i| + \frac{(1-r)}{2} \alpha \sum_{i=1}^{m} (\theta_i)^2$

Dropout:

Refer the paper (https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf)

ℓ_1 and ℓ_2 regularization

In [ ]:
```
from tensorflow import keras
```

In [ ]:
```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
# or l1(0.1) for ℓ1 regularization with a factor or 0.1
# or l1_l2(0.1, 0.01) for both ℓ1 and ℓ2 regularization, with factors 0.1 and 0.01 respe
```

In [ ]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="elu",
                       kernel_initializer="he_normal",
                       kernel_regularizer=keras.regularizers.l2(0.01)),
    keras.layers.Dense(100, activation="elu",
                       kernel_initializer="he_normal",
                       kernel_regularizer=keras.regularizers.l2(0.01)),
    keras.layers.Dense(10, activation="softmax",
                       kernel_regularizer=keras.regularizers.l2(0.01))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accur
# n_epochs = 2
# history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
#                     validation_data=(X_valid_scaled, y_valid))
```

In [ ]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_4 (Dense)              (None, 300)               235500
_________________________________________________________________
dense_5 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_6 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accur
# n_epochs = 2
# history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
#                     validation_data=(X_valid_scaled, y_valid))
```

In [ ]:
```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_7 (Dense)              (None, 300)               235500
_________________________________________________________________
dense_8 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Max-Norm Regularization

In [ ]:
```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01),
                           kernel_constraint=keras.constraints.max_norm(1.))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accur
# n_epochs = 2
# history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
#                     validation_data=(X_valid_scaled, y_valid))
```

In [ ]:
```
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_6 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_13 (Dense)             (None, 300)               235500
_________________________________________________________________
dense_14 (Dense)             (None, 100)               30100
_________________________________________________________________
dense_15 (Dense)             (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Dropout

In [ ]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accur
# n_epochs = 2
# history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
#                     validation_data=(X_valid_scaled, y_valid))
```

In [ ]:
```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
dropout (Dropout)            (None, 784)               0
_________________________________________________________________
dense_10 (Dense)             (None, 300)               235500
_________________________________________________________________
dropout_1 (Dropout)          (None, 300)               0
_________________________________________________________________
dense_11 (Dense)             (None, 100)               30100
_________________________________________________________________
dropout_2 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_12 (Dense)             (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Solving a Regression Problem using ANN:

In [1]:
```
import pandas as pd
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

In [2]:
```
housing = fetch_california_housing()
housing
```

Out[2]:
```
{'data': array([[   8.3252    ,   41.        ,    6.98412698, ...,    2.55555556,
           37.88      , -122.23      ],
        [   8.3014    ,   21.        ,    6.23813708, ...,    2.10984183,
           37.86      , -122.22      ],
        [   7.2574    ,   52.        ,    8.28813559, ...,    2.80225989,
           37.85      , -122.24      ],
        ...,
        [   1.7       ,   17.        ,    5.20554273, ...,    2.3256351 ,
           39.43      , -121.22      ],
        [   1.8672    ,   18.        ,    5.32951289, ...,    2.12320917,
           39.43      , -121.32      ],
        [   2.3886    ,   16.        ,    5.25471698, ...,    2.61698113,
           39.37      , -121.24      ]]),
 'target': array([4.526, 3.585, 3.521, ..., 0.923, 0.847, 0.894]),
 'frame': None,
 'target_names': ['MedHouseVal'],
 'feature_names': ['MedInc',
  'HouseAge',
  'AveRooms',
  'AveBedrms',
  'Population',
  'AveOccup',
  'Latitude',
  'Longitude'],
 'DESCR': '.. _california_housing_dataset:\n\nCalifornia Housing dataset\n--------------
-------------\n\n**Data Set Characteristics:**\n\n    :Number of Instances: 20640\n\n
:Number of Attributes: 8 numeric, predictive attributes and the target\n\n    :Attribut
e Information:\n        - MedInc        median income in block group\n        - HouseAg
e      median house age in block group\n        - AveRooms      average number of rooms
per household\n        - AveBedrms     average number of bedrooms per household\n
- Population    block group population\n        - AveOccup      average number of house
hold members\n        - Latitude      block group latitude\n        - Longitude     blo
ck group longitude\n\n    :Missing Attribute Values: None\n\nThis dataset was obtained
from the StatLib repository.\nhttps://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.h
tml\n\nThe target variable is the median house value for California districts,\nexpress
ed in hundreds of thousands of dollars ($100,000).\n\nThis dataset was derived from the
1990 U.S. census, using one row per census\nblock group. A block group is the smallest
geographical unit for which the U.S.\nCensus Bureau publishes sample data (a block grou
p typically has a population\nof 600 to 3,000 people).\n\nAn household is a group of pe
ople residing within a home. Since the average\nnumber of rooms and bedrooms in this da
taset are provided per household, these\ncolumns may take surpinsingly large values for
block groups with few households\nand many empty houses, such as vacation resorts.\n\nI
t can be downloaded/loaded using the\n:func:`sklearn.datasets.fetch_california_housing`
function.\n\n.. topic:: References\n\n    - Pace, R. Kelley and Ronald Barry, Sparse Sp
atial Autoregressions,\n      Statistics and Probability Letters, 33 (1997) 291-297\n'}
```

In [3]:
```
housing.keys()
```

Out[3]:
```
dict_keys(['data', 'target', 'frame', 'target_names', 'feature_names', 'DESCR'])
```

In [4]:
```
X = pd.DataFrame(housing.data, columns= housing.feature_names)
X.head()
```

Out[4]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25

In [5]:
```
y = pd.DataFrame(housing.target, columns=['target'])
y.head()
```

Out[5]:

	target
0	4.526
1	3.585
2	3.521
3	3.413
4	3.422

In [6]:
```
X.shape
```

Out[6]: (20640, 8)

In [7]:
```
y.shape
```

Out[7]: (20640, 1)

In [8]:
```
X_train_full, X_test, y_train_full, y_test = train_test_split(X,y, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full,y_train_full, random_
```

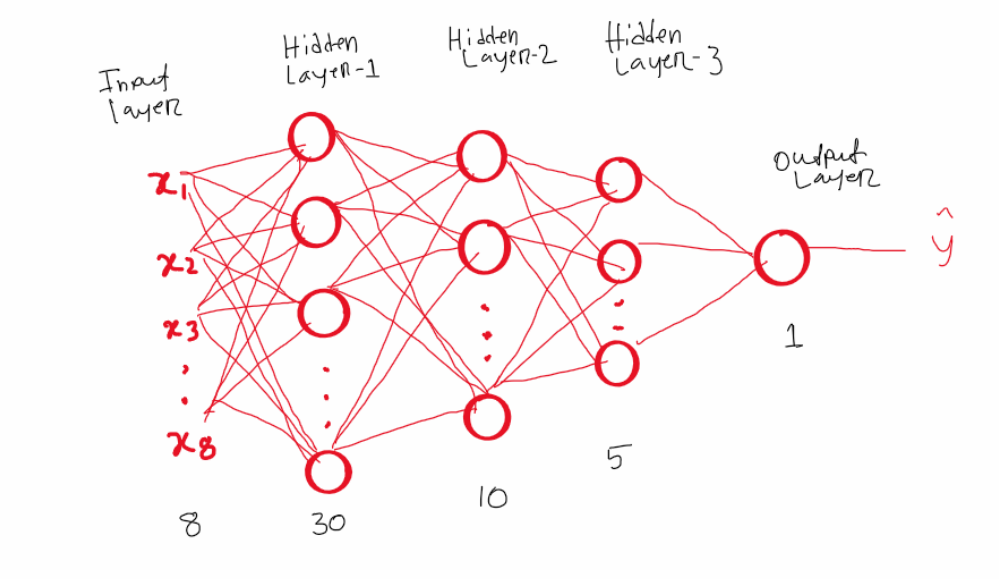
In [9]:
```
print(X_train_full.shape)
print(X_test.shape)
print(X_train.shape)
print(X_valid.shape)
```

```
(15480, 8)
(5160, 8)
(11610, 8)
(3870, 8)
```

In [10]:
```
X_train.shape[1:]
```

Out[10]: (8,)

Architecture used:



In [11]:
```
LAYERS = [
          tf.keras.layers.Dense(30, activation="relu", input_shape = X_train.shape[1:]),
          tf.keras.layers.Dense(10, activation="relu"),
          tf.keras.layers.Dense(5, activation='relu'),
          tf.keras.layers.Dense(1)

]
```

Q)while defining the layer in classification you didn't applied Activation function and used Flatten ,but here you directly started from dense and applied RELU in the very first layer ,why?

The choice of layer architecture and activation functions in a neural network can vary depending on the specific task and the desired behavior of the model. Better option is, add relu activation function in dense layers and in output layer if it is binary classification add sigmoid otherwise add softmax.

In [12]:
```
model = tf.keras.models.Sequential(LAYERS)
```

In [13]:
```
LOSS = "mse"
OPTIMIZER = "sgd"

model.compile(optimizer= OPTIMIZER, loss= LOSS)
```

In [14]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 30)                270

 dense_1 (Dense)             (None, 10)                310

 dense_2 (Dense)             (None, 5)                 55

 dense_3 (Dense)             (None, 1)                 6

=================================================================
Total params: 641
Trainable params: 641
Non-trainable params: 0
_________________________________________________________________
```

In [15]:
```
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

In [16]:
```
EPOCHS = 20

history = model.fit( X_train, y_train, epochs= EPOCHS, validation_data=(X_valid, y_valid
```

```
Epoch 1/20
363/363 [==============================] - 3s 5ms/step - loss: 0.7655 - val_loss: 0.603
9
Epoch 2/20
363/363 [==============================] - 1s 3ms/step - loss: 0.4610 - val_loss: 0.392
6
Epoch 3/20
363/363 [==============================] - 1s 3ms/step - loss: 0.4087 - val_loss: 0.434
6
Epoch 4/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3891 - val_loss: 0.379
9
Epoch 5/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3752 - val_loss: 0.356
9
Epoch 6/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3666 - val_loss: 0.355
4
Epoch 7/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3619 - val_loss: 0.377
0
Epoch 8/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3593 - val_loss: 0.386
4
Epoch 9/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3572 - val_loss: 0.349
5
Epoch 10/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3527 - val_loss: 0.348
6
Epoch 11/20
363/363 [==============================] - 1s 2ms/step - loss: 0.3498 - val_loss: 0.348
5
Epoch 12/20
363/363 [==============================] - 1s 2ms/step - loss: 0.3467 - val_loss: 0.337
2
Epoch 13/20
363/363 [==============================] - 1s 2ms/step - loss: 0.3429 - val_loss: 0.361
6
Epoch 14/20
363/363 [==============================] - 1s 2ms/step - loss: 0.3407 - val_loss: 0.328
2
Epoch 15/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3392 - val_loss: 0.338
2
Epoch 16/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3389 - val_loss: 0.338
5
Epoch 17/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3336 - val_loss: 0.353
1
Epoch 18/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3334 - val_loss: 0.388
9
Epoch 19/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3309 - val_loss: 0.366
5
Epoch 20/20
```

```
363/363 [==============================] - 1s 3ms/step - loss: 0.3313 - val_loss: 0.328
7
```

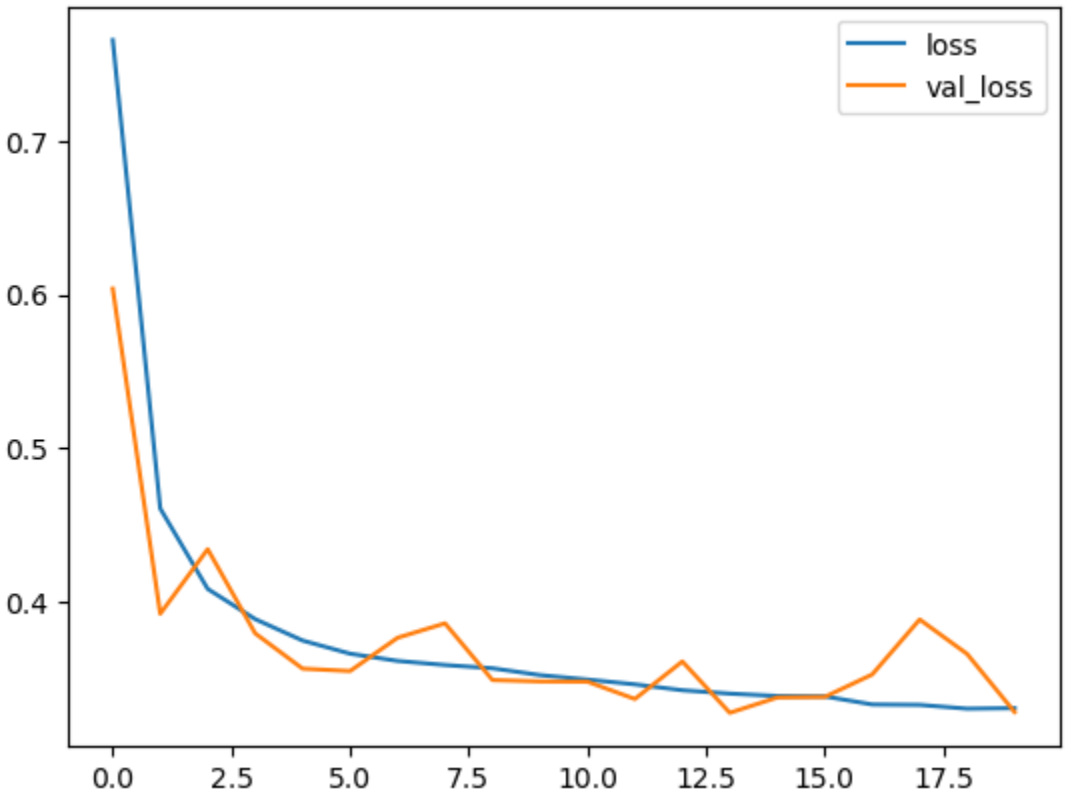
In [17]:
```
pd.DataFrame(history.history)
```

Out[17]:

	loss	val_loss
0	0.765455	0.603858
1	0.461002	0.392607
2	0.408732	0.434563
3	0.389118	0.379874
4	0.375180	0.356915
5	0.366648	0.355371
6	0.361943	0.376984
7	0.359288	0.386361
8	0.357164	0.349506
9	0.352737	0.348626
10	0.349832	0.348474
11	0.346712	0.337208
12	0.342892	0.361606
13	0.340700	0.328218
14	0.339156	0.338185
15	0.338889	0.338527
16	0.333643	0.353137
17	0.333382	0.388871
18	0.330911	0.366475
19	0.331260	0.328746

In [18]:
```
pd.DataFrame(history.history).plot()
```

Out[18]: <Axes: >



In [19]:
```
model.evaluate(X_test, y_test)
```

```
162/162 [==============================] - 0s 2ms/step - loss: 0.3218
```

Out[19]:
```
0.3217606842517853
```

In [20]:
```
X_test.shape
```

Out[20]:
```
(5160, 8)
```

In [21]:
```
new = X_test[0]
```

In [29]:
```
new2 = X_test[1]
```

In [30]:
```
new
```

Out[30]:
```
array([-1.15780104, -0.28673138, -0.49550877, -0.16618097, -0.02946012,
        0.38899735,  0.19374821,  0.2870474 ])
```

In [23]:
```
new.shape
```

Out[23]:
```
(8,)
```

```
In [24]: X_test[0]

Out[24]: array([-1.15780104, -0.28673138, -0.49550877, -0.16618097, -0.02946012,
                0.38899735,  0.19374821,  0.2870474 ])

In [25]: new.reshape((1,8))

Out[25]: array([[-1.15780104, -0.28673138, -0.49550877, -0.16618097, -0.02946012,
                 0.38899735,  0.19374821,  0.2870474 ]])

In [31]: new2.reshape((1,8))

Out[31]: array([[-0.7125531 ,  0.10880952, -0.16332973,  0.20164652,  0.12842117,
                -0.11818174, -0.23725261,  0.06215231]])

In [26]: model.predict(new.reshape((1,8)))

         1/1 [==============================] - 0s 206ms/step

Out[26]: array([[0.77056193]], dtype=float32)

In [32]: model.predict(new2.reshape((1,8)))

         1/1 [==============================] - 0s 41ms/step

Out[32]: array([[1.4972047]], dtype=float32)
```

Model with callback

In [33]:
```
model_2 = tf.keras.models.Sequential(LAYERS)

LOSS = "mse"
OPTIMIZER = tf.keras.optimizers.SGD(learning_rate=1e-3)

model_2.compile(loss=LOSS , optimizer=OPTIMIZER)

EPOCHS = 20

checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("my_keras_model.h5", save_best_only=T
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=5, restore_best_weights=Tr
tensorboard_cb = tf.keras.callbacks.TensorBoard(log_dir="logs")

CALLBACKS = [checkpoint_cb, early_stopping_cb, tensorboard_cb]


history = model_2.fit(X_train, y_train, epochs = EPOCHS, validation_data=(X_valid, y_val
```

```
Epoch 1/20
363/363 [==============================] - 2s 4ms/step - loss: 0.3214 - val_loss: 0.323
4
Epoch 2/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3196 - val_loss: 0.323
5
Epoch 3/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3190 - val_loss: 0.321
9
Epoch 4/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3188 - val_loss: 0.322
1
Epoch 5/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3183 - val_loss: 0.320
9
Epoch 6/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3179 - val_loss: 0.319
8
Epoch 7/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3177 - val_loss: 0.320
0
Epoch 8/20
363/363 [==============================] - 2s 5ms/step - loss: 0.3173 - val_loss: 0.321
6
Epoch 9/20
363/363 [==============================] - 1s 4ms/step - loss: 0.3169 - val_loss: 0.320
7
Epoch 10/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3170 - val_loss: 0.319
4
Epoch 11/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3167 - val_loss: 0.319
8
Epoch 12/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3164 - val_loss: 0.320
5
Epoch 13/20
363/363 [==============================] - 1s 4ms/step - loss: 0.3161 - val_loss: 0.319
6
Epoch 14/20
363/363 [==============================] - 1s 4ms/step - loss: 0.3157 - val_loss: 0.320
8
Epoch 15/20
363/363 [==============================] - 1s 4ms/step - loss: 0.3155 - val_loss: 0.322
5
```

In [34]:
```
%load_ext tensorboard
```