🧠 Neural Network From Scratch using Numpy

This project demonstrates the complete implementation of a **Neural Network from scratch using only NumPy**, without relying on high-level frameworks like TensorFlow or PyTorch.

The goal is to deeply understand the internal mechanics of **Forward Propagation, Backpropagation, Gradient Descent, Activation Functions, and Loss Computation**.

Objective:

- Build a Neural Network from scratch
- Understand gradient computation using the chain rule
- Implement forward & backward propagation manually
- Train the model using Gradient Descent

Outline:

🌍 Real-World Applications

📜 History of Neural Networks

⚙ Working of Neural Network

🔥 Activation Functions

📉 Loss Functions

💻 Code Walkthrough

---

1 Real World Applications

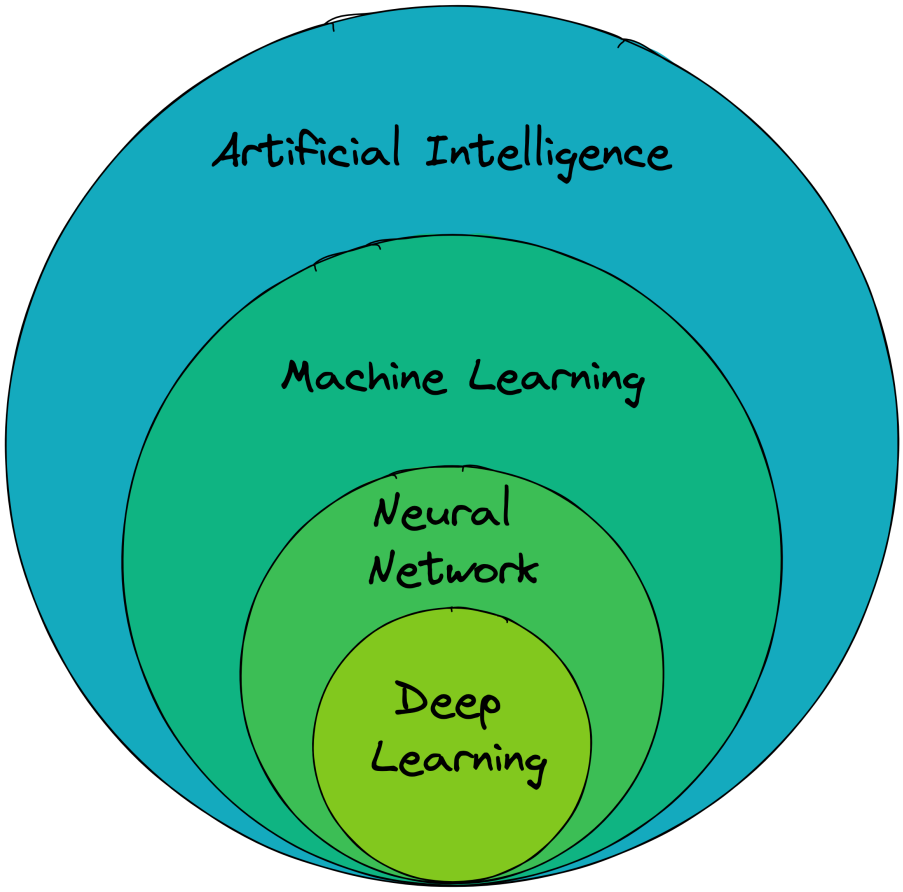
Before we dive into what Deep Learning is - let's review some applications of Deep Learning in our everyday life:

1. **Self Driving Cars:** Vehicles like Teslas have networked cameras and various other sensors fitted around the car which provide vast amount of data that is fed to an AI model that learns to identify all sourounding objects and predict what the best next action for the vehicle is. The AI model simulates human perception and the descision making process using Deep Learning under the hood.

2. **Netflix Recommendation Engine:** A major contribution to Netflix's business model is its recommendation engine that enables customer rentention. Netflix gathers a vast amount of data of their user base including viewer ratings, viewing histories, wathing preferences etc - all of these data points are fed into machine learning models to generate models that can provide you a personalised recommendation based on your own preferences.

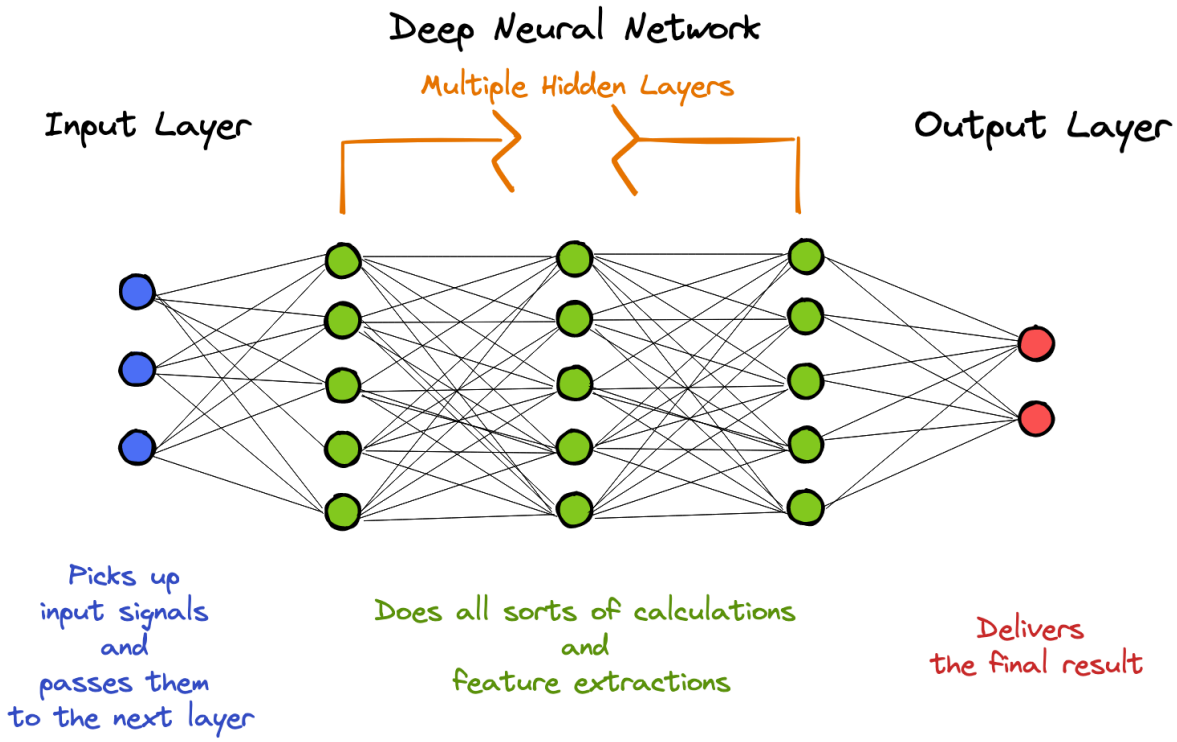
3. **Face Recognition:** A multitude of applications are able to identify and tag people in pictures and videos. A task which is very easily performed by humans, even if faces change or are abstructed with accessories. However, until recently this has remained a challenging computer vision problem for many decades. These days, models are able to leverage Deep Learning and utilise big datasets of faces to learn similarities and develop face recognition capabiilties that even outperform humans.

All of the above examples fall under the field of Artificial Intelligene which is essentially the theory and development of computer systems that perform tasks that would normally require human intelligence. Machine Learning is the application of AI based around the idea that computer systems should not be explicitely programmed but learn by experience and acquire skils without human intervention.



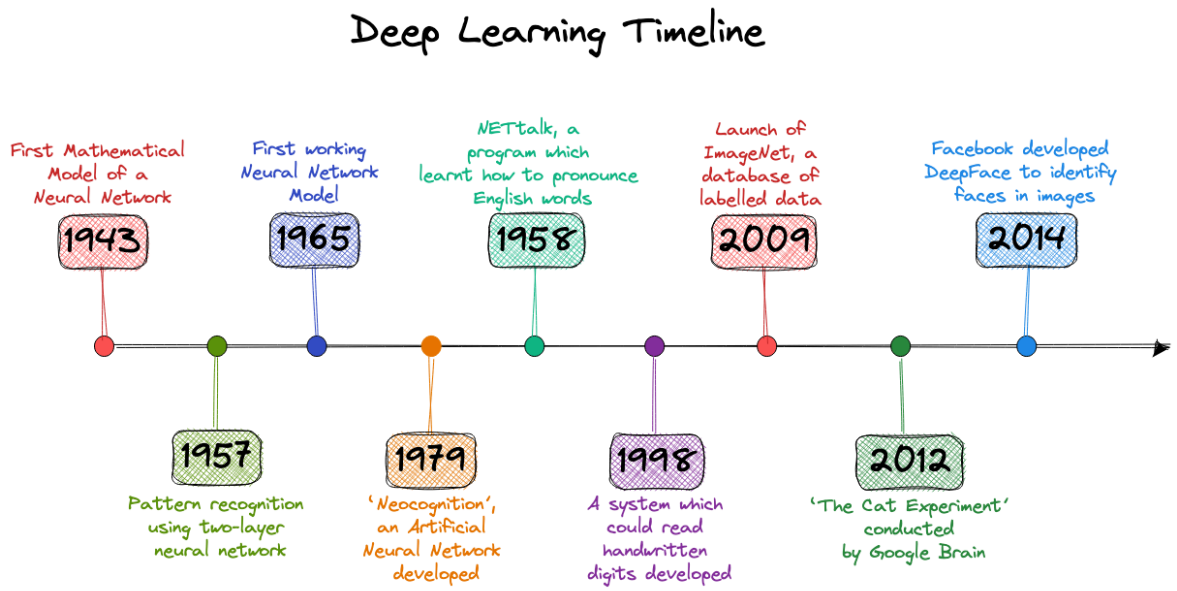
Deep Learning is a subfield of machine learning that uses **Artificial Neural Networks (ANN)** at its backbone. ANNs are often described as mimicing the human brain through a set of algorthims. A major difference between machine learning models and ANNs is that the former lacks the mechanism to identify errors and requires human intervention to tune the model for more accurate decision. Whereas ANNs can identify the inaccurate decision and self-correct.

At a very high level, a neural network comprises of **Input, Hidden and Output Layers** that are made up of interconnected Nodes which resemble the way that biological neurons in the human brain signal to another.



Deep Learning and ANNs are often referred to interchangeably in conversation. The 'deep' in Deep Learning simply refers to the depth of the hidden layers and it is generally accepted that a neural network with more than three layers is considered a deep learning model.

2.0 History of Deep Learning



1943: The first mathematical model of a neural network was created by Walter Pitts and Warren McCulloch in 1943 which demonstrated the thought process of the

human brain. From here began the journey of deep neural network and deep learning.

1957: Frank Rosenblatt submitted a paper titled ‘The Perceptron: A Perceiving and Recognizing Automaton’, which consisted of an algorithm or a method for pattern recognition using a two-layer neural network. His work was mostly based on hardware rather than software.

1965: Alexey Ivakhnenko and V.G. Lapa developed the first working neural network. Afterwards, in 1971, Alexey Ivakhnenko created an 8-layer deep neural network which was demonstrated in the computer identification system, Alpha. This was the actual introduction to deep learning using statistical methods.

1979: Kunihiko Fukushima developed the ‘Neocognitron’, a multilayered ANN that will recognize visual patterns. His work served as the inspiraton of Convolution Neural Network a few decades later.

1985: Terry Sejnowski created NETtalk, a program which learnt how to pronounce English words. Around the same timeframe, David Rumelhart, Geoffrey Hinton, and Ronald J. Williams, released a paper entitle "Learning Representations by Back-propagating Errors,” which described in greater detail the process of backpropagation.

1998: Yann LeCun published a paper entitled “Gradient-Based Learning Applied to Document Recognition” which in conjunction with the backpropation alogrithm revolutionised an increasinly successfull approach to deep learning. A very popular dataset MNIST (a large database of handwritten digits) was used to show how deep learning models were able to learn abstract information by demonstrating that individual layers in the model identified different features, e.g. one layer finds circles and the next one would find edges etc.

2009: As deep learning models require a tremendous amount of labelled data to train themselves in supervised learning, Fei-Fei Li launched ImageNet, which is a large database of labelled images (14 million) available to researches, educators and students.

2012: The results of ‘The Cat Experiment’ conducted by Google Brain were released. This experiment was based on unsupervised learning in which the deep neural network worked with unlabelled data to recognize patterns and features in the images of cats. However, it could only recognize 15% of images correctly.

2014: Facebook developed, DeepFace, a deep learning system to identify and tag faces of users in the photographs.

The past decade has seen great leaps in the evolution of DL and its ability to learn more abstract information due to a combination of increased data, improved hardware and sofware. Today, Deep Learning is ever-present in our everyday lives (as shown in the previous examples) and as the field continues to evolve, humans become closer in their pursuit of AI.

3 Working of a Neural Network

So far we have looked at where *Deep Learning (DL)* fits into the broader field of AI, and taken a brief look at its history.

Before learning about the core of DL, the neural network, we will look at why and when we might want to use DL rather than other, simpler, *Machine Learning (ML)* algorithms.

First of all, DL has additional abilities compared to basic ML algorithms. It automatically learns representations from data without introducing any hand-coded rules or human domain knowledge. For example, to detect apples on a production line using machine learning, you might need to manually code the extraction of particular features like the roundness of an apple, its weight, or color, to help the machine to learn. DL needs no such instructions and can perform this *feature extraction* by itself.

Said that, the choice between DL or ML can be based on the data we are analysing:

- When the data is small, deep learning algorithms don't perform that well. This is because deep learning algorithms need a large amount of data to extract and *understand* meaning patterns. In this case, a regular machine learning algorithm may deliver more accurate results.
- When dealing with linear functions or less complex data, it can be easier and more efficient to consider alternative machine learning algorithms. If data is non-linear and complex, DL may be more suitable.

We are now going to see how DL components work, starting with the fundamental building block of deep learning - the *perceptron*.

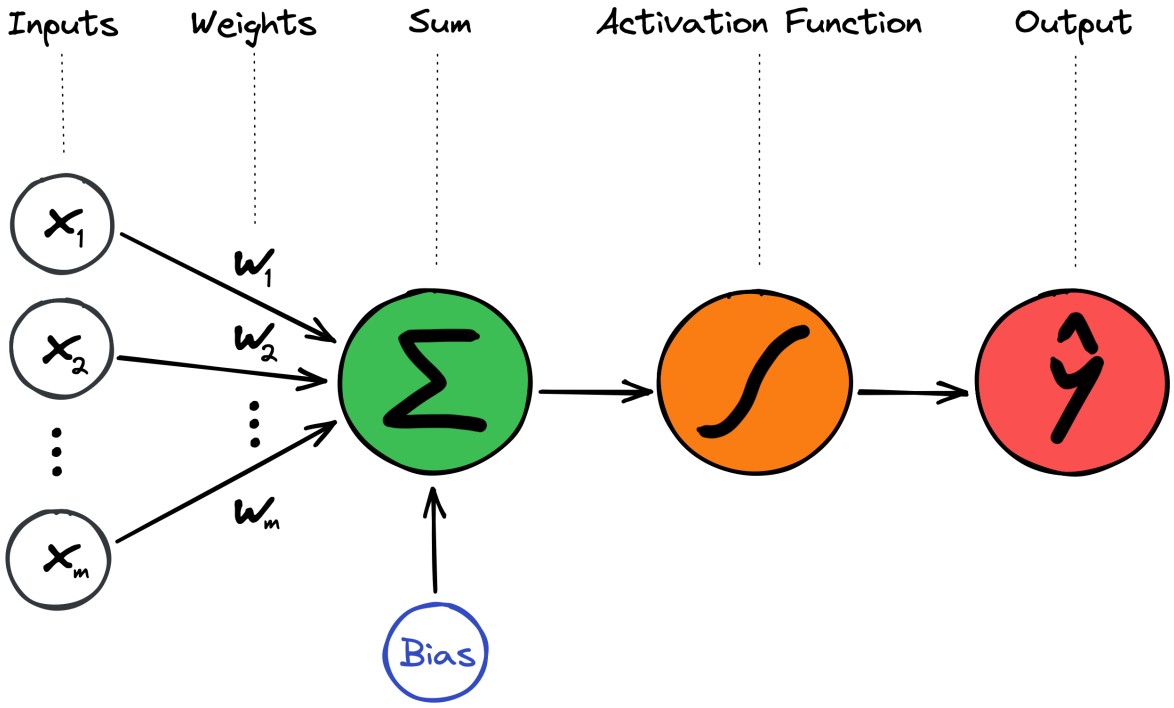
3.1 The Perceptron

A network may have three types of layers:

- **input layers** that take raw input from the domain,

- **hidden layers** that take input from another layer and pass output to another layer,
- and **output layers** that make a prediction.

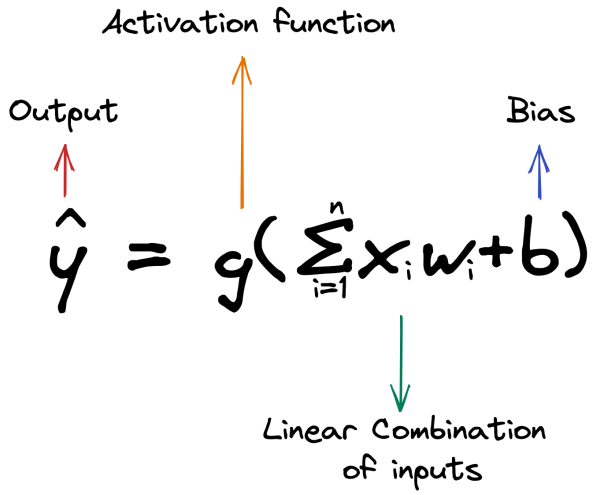
The *percepton* is a single-layer neural network with four main parameters, i.e., input values, weights and bias, net sum, and activation function.



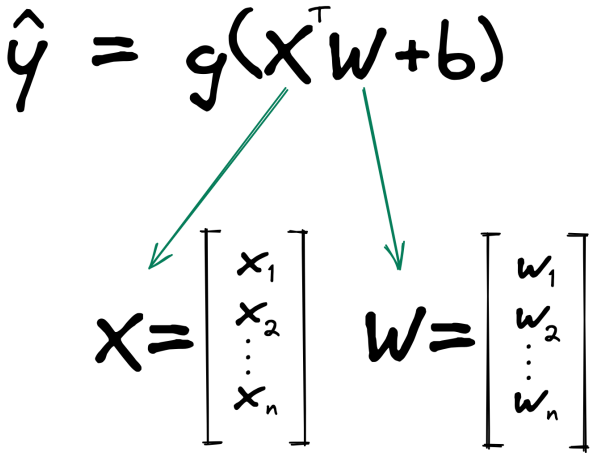
The above diagram shows the forward propagation of information through a single neuron. Here the information is defined by its inputs, x_{1}, x_{2}..x_{m} and their corresponding weights, w_{m}, w_{2}...w_{m}. We can then take the sum of the multiplication of the inputs and their respective weights (i.e., linear combination of inputs). A bias input is added to the summation, $Bias$ or b.

This calculation results in a single number, which is fed to the activation function, where it generates the output. The bias value allows to shift the activation function regardless of the input, similar to the role of a constant c in a linear function $y = mx+c$.

The above forward propagation can be represented mathematically with the below function.



Furthermore, we can use linear algebra to represent the linear combination of the inputs and its weight in vector format, and simply take the dot product of $X^{T}W$ to get the same result.



We now dive into the activation function and its use.

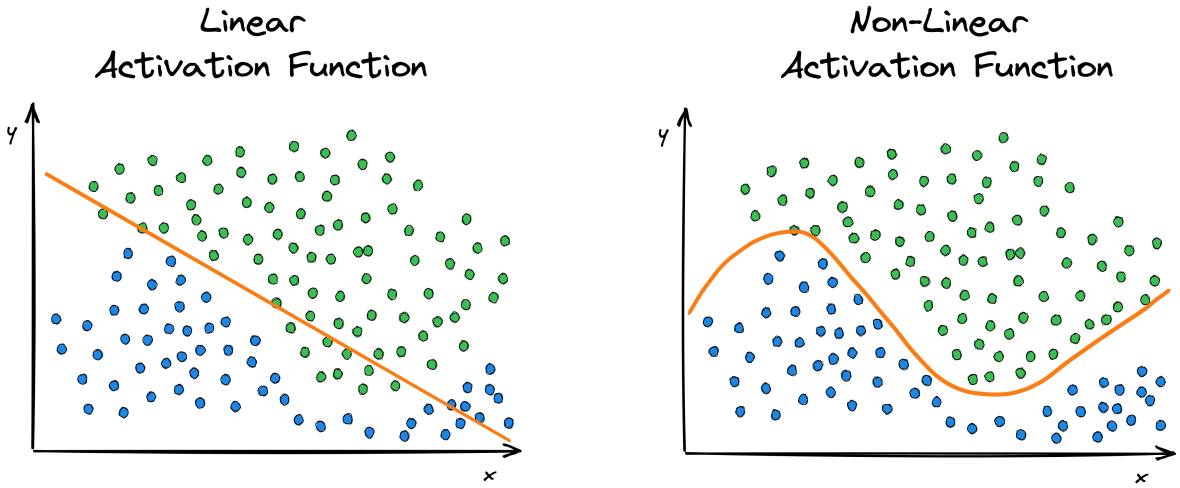
3. 2 Activation Function

An *Activation Function* in a neural network decides the output value of a neuron. The choice of activation function has a large impact on the capability and performance of the neural network: the activation function in the *hidden layer* controls how well the network model learns the training dataset, while that on the *output layer* defines the type of predictions the model can make.

The same activation function is usually used on all hidden layers, with a different activation function on the output layer; the choice on the latter depends on the type of prediction required by the model.

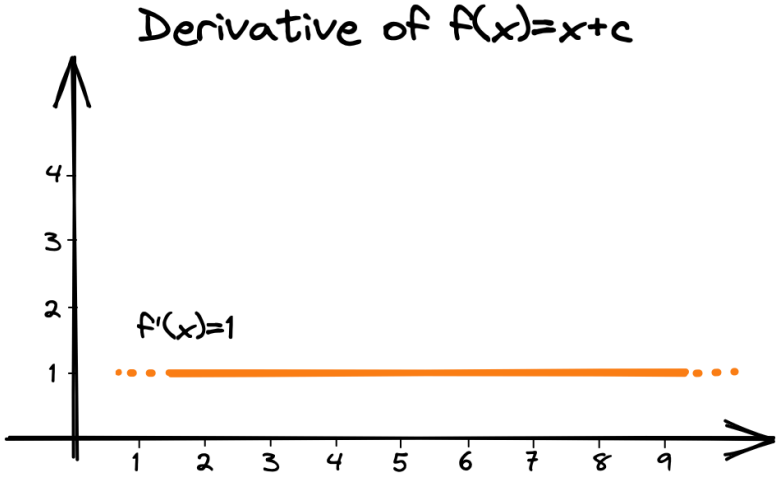
Activation functions can be linear or non-linear. By using a non-linear activation

function in the hidden layers of a neural network, the network can learn more complex patterns within the data.



In addition, neural networks are typically trained using the backpropagation of error algorithm that requires the derivative of a '*prediction error*' in order to update the weights of the model. Because of that, non-linear activation function is preferred. Let's have a look at an example.

If we consider the linear function $y = x+c$, the calculated gradient for any neuron output would be the same, i.e., 1. Therefore, we cannot apply backpropagation to find how the neuron weights should change based on the calculated error. Additionally, using linear activations would result in our model becomes a linear model because multiple linear functions chained together is still only considered to be a linear function. This would suggest that a linear activation function is not ideal, particularly when working with non-linear data.



Depending on the use case, we can use various non-linear activation functions. Among the most used:

- Rectified Linear Activation Function
- Sigmoid (or Logistic) Activation Function
- Tanh Hidden Layer Activation Function

Let's now have a quick look at them.

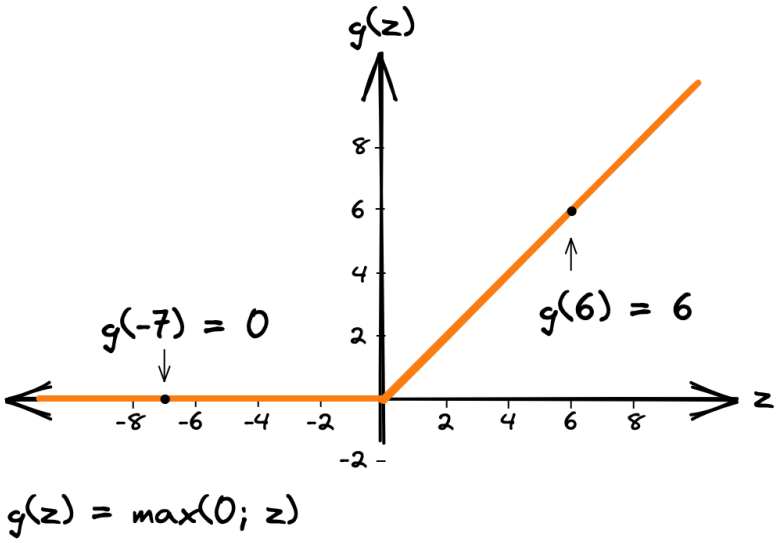
3.2.1. Rectified Linear Activation Function

The rectified linear activation function is a piecewise (or hinge) linear function, meaning that it is linear for half of the input domain and non-linear for the other half. More specifically, it is linear for values greater than 0 where it returns the value provided as input directly, and it is non-linear for values equal-to or less than 0, for which it returns 0.

Below is the formula for the ReLu function:

$$g(z) = max(z;0)$$

and its graphical representation.



A node or unit that implements this activation function is referred to as a rectified linear activation unit, or ReLU for short. Often, networks that use the rectifier function for the hidden layers are referred to as rectified networks.

3.2.2. Sigmoid (or Logistic) Activation Function

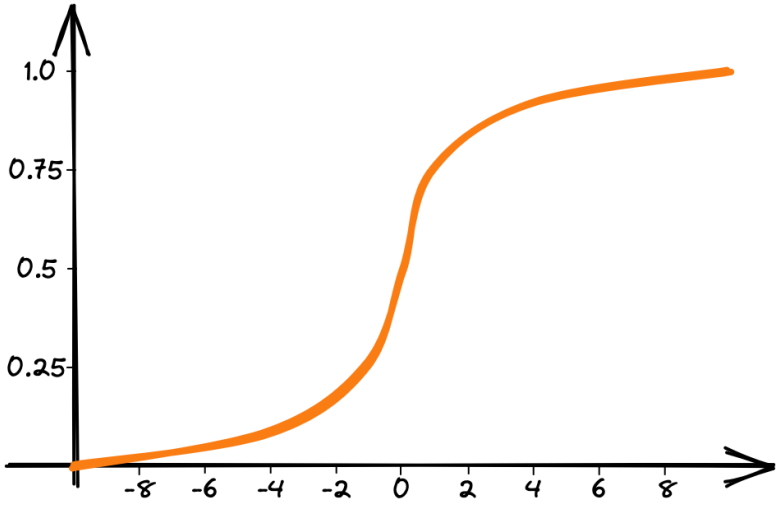
The *Sigmoid Activation Function* takes any real value as input and outputs values in the range $[0; 1]$. The larger the input (in the positive direction), the closer the

output value is to 1. And the smaller the input (in the negative direction), the closer the output to 0.

Below is the formula of the sigmoid activation:

$$σ(x) = \frac{1}{(1 + e^{-x})}$$

and its graphical representation.



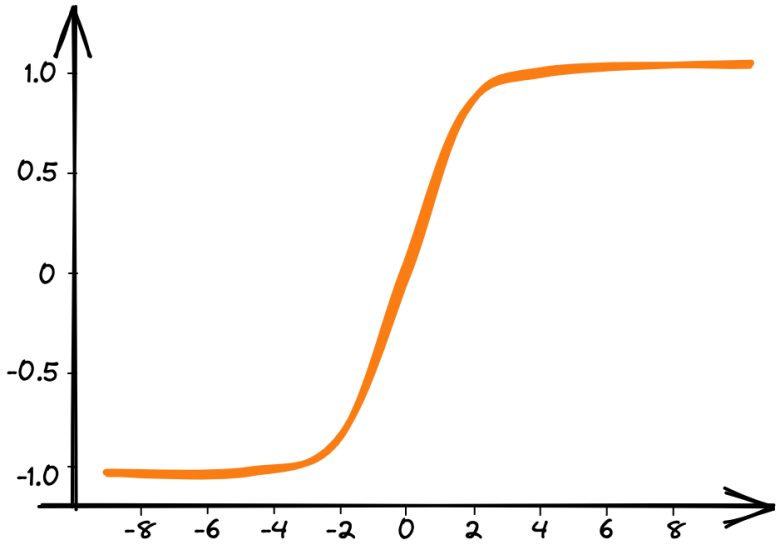
3.2.3. Tanh Hidden Layer Activation Function

The *Tanh Hidden Layer Activation Function* takes any real value as input and outputs values in the range $[-1; 1]$. The larger the input (in the positive direction), the closer the output value to 1. The smaller the input (in the negative direction), the closer the output is to -1.

Below is the formula of the Tanh activation function:

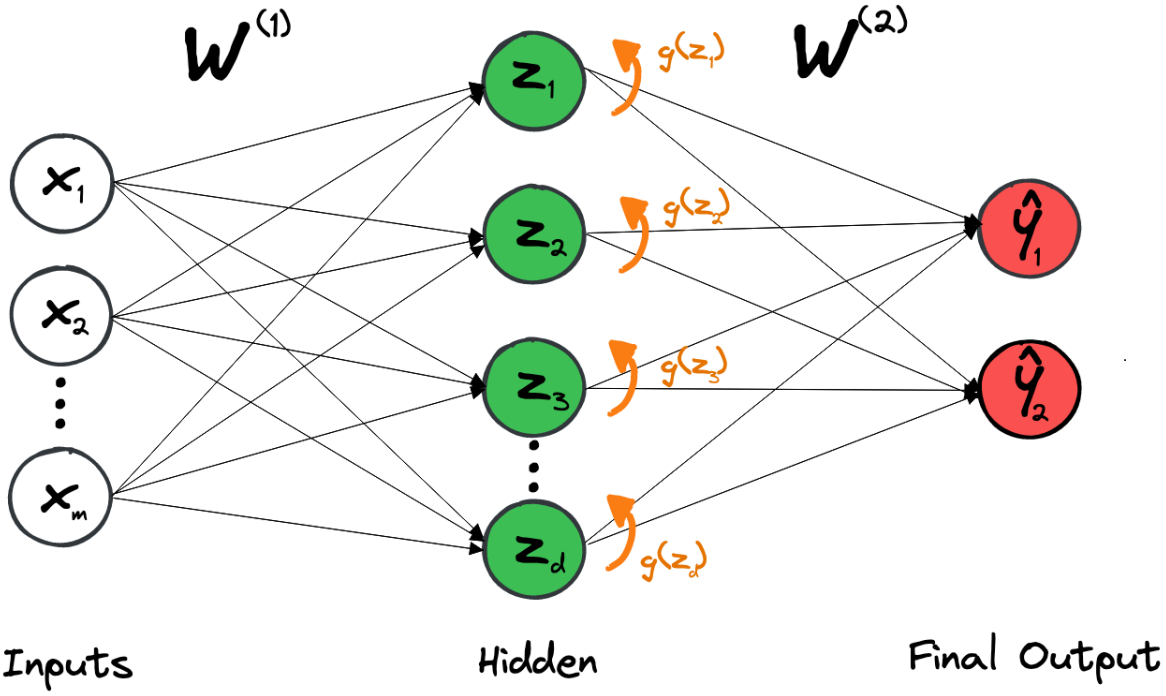
$$tanh(x) = \frac{(e^{x} – e^{-x})}{(e^{x} + e^{-x})}$$

and its graphical representation.



3.3 Shallow Neural Network

We can now use our understanding of the *perceptron* and create a *Shallow Neural Network*, characterised by a single hidden layer with multiple neurons (bias has been ignored for simplicity), that feed into an output layer. This differs from a *Deep Neural Network*, which characterised by several hidden layers (often of various types).

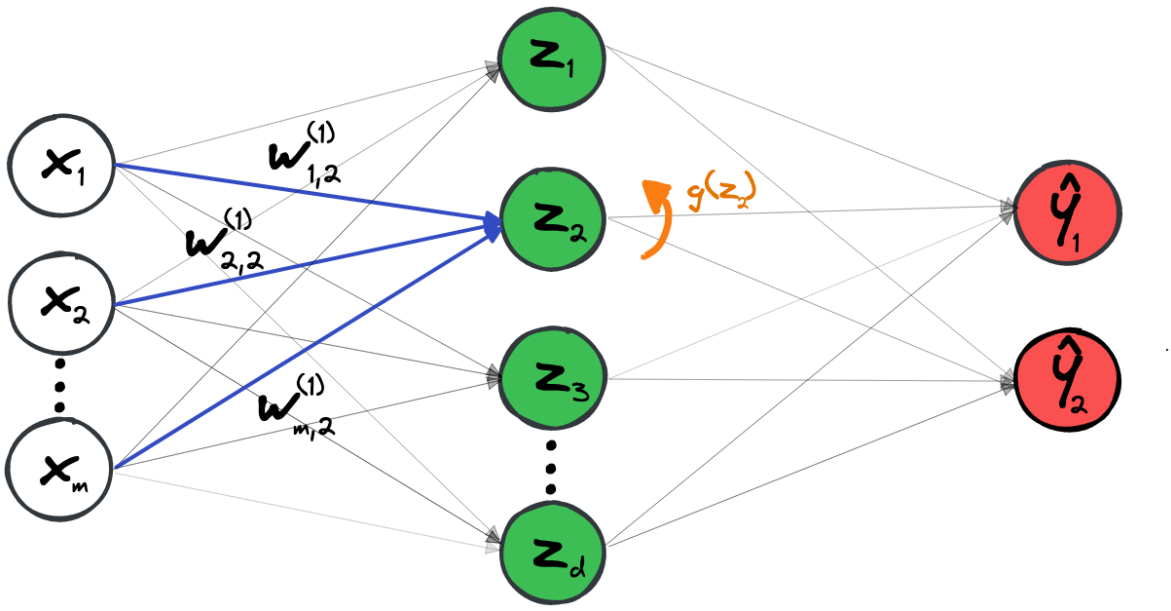


The layer is '*hidden*' as its state is not directly observable, whereas the input and the output layers are directly observable. We can probe to find the state, but they are learned rather than enforced (we will discuss this later).

There are two transformation processes with their respective weight matrices denoted as $W^{\left(1\right) }$ (between the input and hidden layers) and $W^{\left(2\right) }$ (between the hidden and output layers).

We can then perform a single perceptron computation in the hidden layer with $z_{i}=w_{0,j}^{\left(1\right) }+\sum ^{m}_{j=1}x_{j}w_{j,i}^{\left(1\right) }$. This computation is the weighted sum of all of its inputs, transformed by a non-linearity function($g\left(z_{i}\right)$). The result is then passed on to an output layer node which is calculated in a similar manner: $\widehat{y}_{i}=g\left(w_{0,i}^{\left(2\right) }+\sum ^{d}_{j=1}z_{j}w_{j,i}^{\left(2\right) }\ \right)$.

For demonstration, z_2:

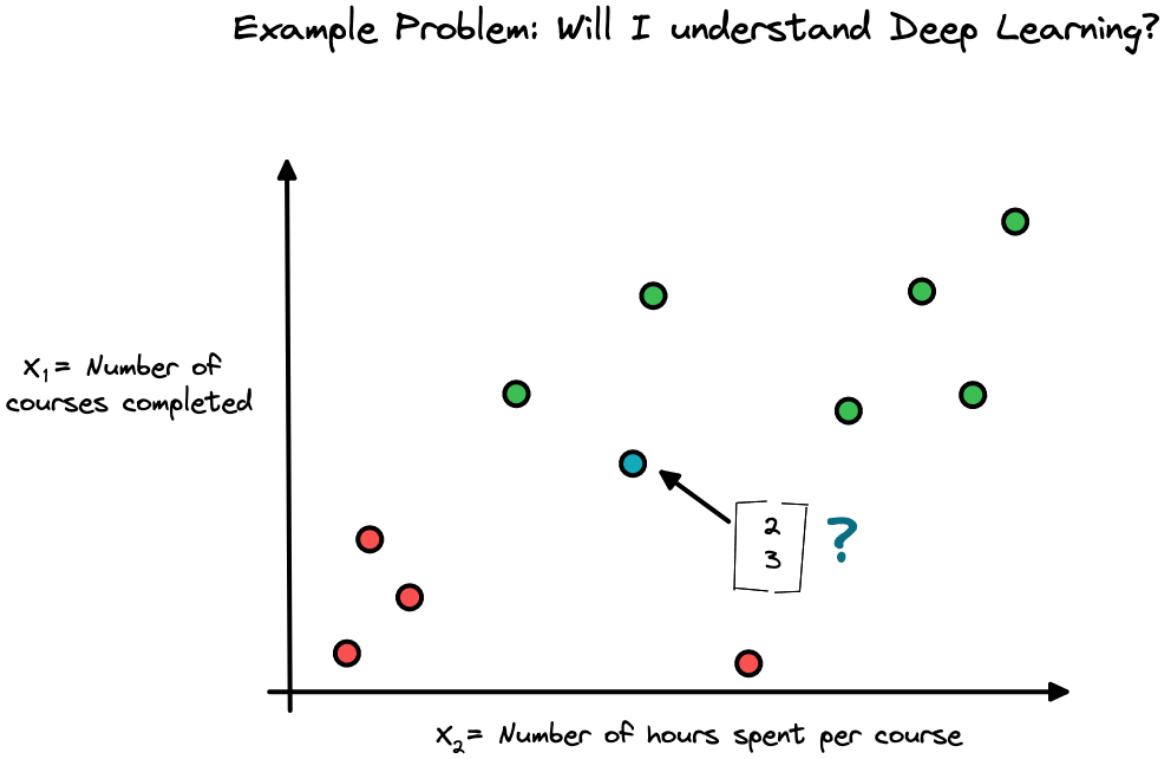


$$z_{2}=w_{0,2}^{\left(1\right) }+\sum ^{m}_{j=1}x_{j}w_{j,2}^{\left(1\right) }$$

$$z_{2}=w_{0,2}^{\left(1\right)}+ x_1w_{1,2}^{\left(1\right)} + x_2w_{2,2}^{\left(1\right)} + x_mw_{m,2}^{\left(1\right)} $$

The same process is repeated for every neuron in the hidden layer and their results are forward propagated as an input to the nodes in the next layer (in this case the output layer). We will going to demonstrate this with an example in the next section.

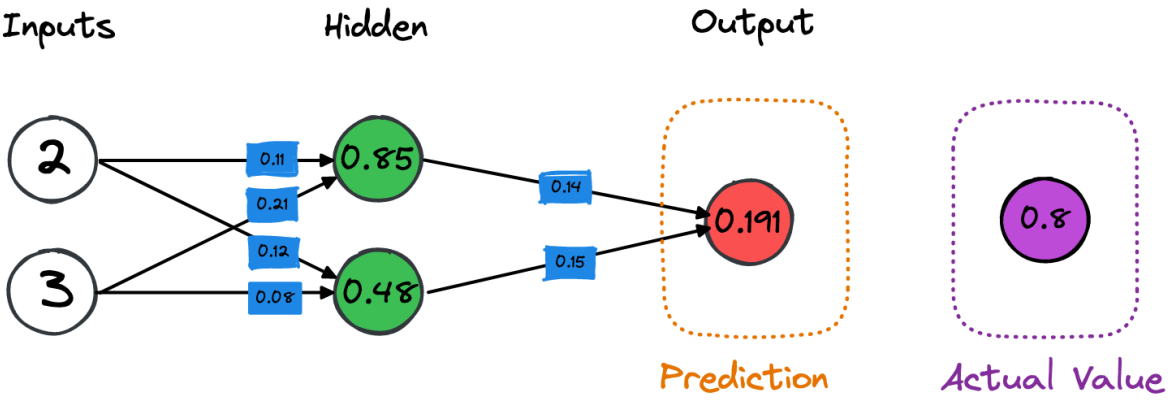
3.4 Shallow Neural Network Example



In this example we are trying to predict the propability of someone understanding deep learning based on two features:

1. The number of courses a person has completed.
2. The average number of hours spent on each course.

Our example input is *2* courses and an average of *3* hours spent on each course.



$$\text{Error} = 0.5 (\text{Prediction} - \text{Actual})^2$$

$$\text{Error} = 0.5 (0.191 - 0.8)^2 = 0.37$$

You can see that our network is not very good at predicting the correct probability. That is due to the fact that our network weights are completely randomised as the network has not yet been trained. To improve the prediction, we need to find weights so that the $Error$ is reduced to nearly *0*.

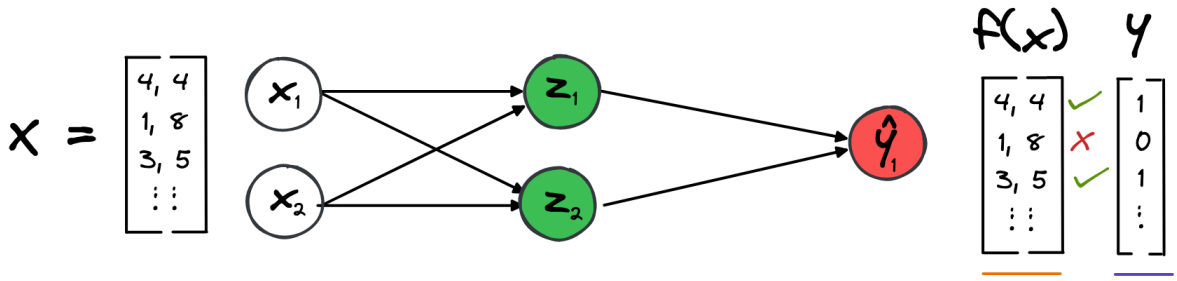
3.5 Loss Function

In order for the model to correct itself, it will need to find a way to measure its error. This is defined as the loss of the network, which measures the difference between predictions and actual values. Note that $f(x)$ below denotes the final model output for input values, x.

$$L\Big(\underbrace{f\Big(x^{(i)}; W\Big)}_{\text{Prediction}}, \underbrace{y^{(i)}}_{\text{Actual}}\Big)$$

The input to the loss function are the $Prediction$ and the $Actual$ values. The closer the difference between those values, the better the model - it is therefore necessary to try and minimise the loss L.

The empirical loss $J(W)$ (also known as the cost function) measures the total average loss across all of our data points, not just one sample. It is computed by finding the mean of the loss of all our data points and their respective predictions.



$$J(W) = \frac{1}{n}\sum_{i=1}^{n} L\Big(\underbrace{f\Big(x^{(i)}; W\Big)}_{\text{Prediction}}, \underbrace{y^{(i)}}_{\text{Actual}}\Big)$$

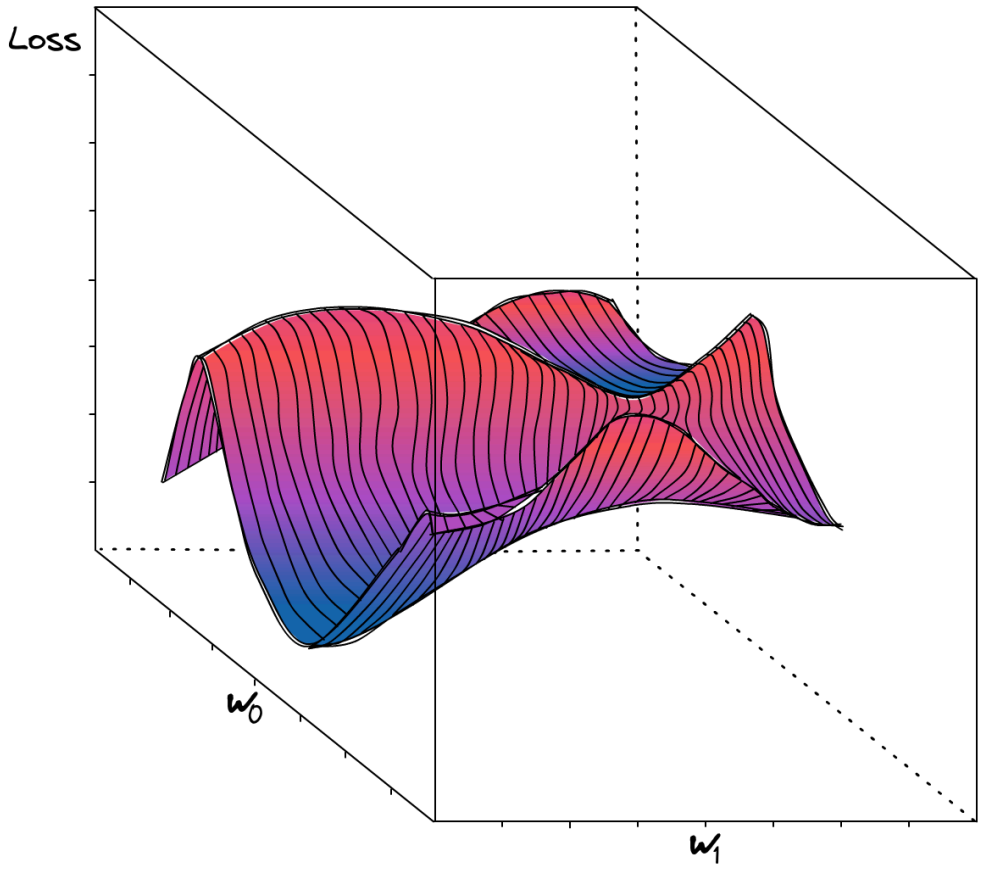
We now need to train the network using the loss function to find the weights W^* that achieve the lowest loss. This can be mathematically represented as

such:

$$W^{\ast }=\argmin_{W}\dfrac{1}{n}\sum ^{n}_{i=1}L\left(f\left(x^{\left(i\right) };W\right) ,y^{\left(i\right) }\right)$$ $$W^{\ast }=\argmin_{W}J\left(W\right)$$ $$W=\left\{ W^{\left(0\right) },W^{\left(1\right) },\ldots \right\}$$

W is the collection of all the weights across all layers.

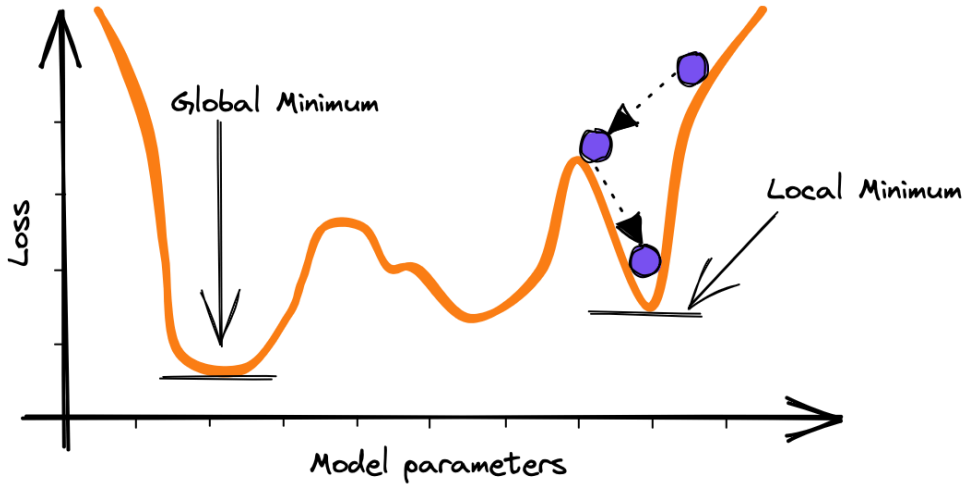
The loss can be described as a function of our weights, and for the purposes of visualisation, the loss landscape created across two weights w_{0} and w_{1} would look like this:



This landscape shows us the loss that is expected for all varitions of w_{0} and w_{1}. The whole process of training the model is to establish the optimal w_{0} and w_{1}, i.e., $ W^{\ast }$ that would provide the lowest loss.

The best way to do that is by randomly picking a point (w_{0},w_{1}) in the landscape. At that point, we need to determine the slope of the angle of our $J(W)$ to understand if we are headed in the right direction. A negative slope means that we are headed downward, while a posive slope means that we moved beyond the minimum. To determine the slope, we compute the gradient of the function on that point, $\dfrac{\partial J\left(W\right) }{\partial W}$. Given the gradient can be

interpreted as the direction and rate of fastest increase, we will need to take a small step in the opposite direction to find a lower loss. We repeat this process until convergence, i.e., we hit a local minimum. This process is anologous of letting a ball loose at a random point on a hill and it coming to a stop at the bottom (global minimum) or in a small indent higher up the hill (local minima).



Note that it is ulikely to find the global minimum by selecting a single random point. To increase the chance of finding it, we could choose different points of the loss function and follow the same steps for all of them. Such an approach is called *random restart*.

The above algorithm is called **Gradient Descent** and can be summarised as follows:

1. Initialize weights randomly.
2. Loop until convergence:

   2.a. Compute gradient, $\dfrac{\partial J\left(W\right) }{\partial W}$.

   2.b. Update weights, $W\leftarrow W-\alpha\dfrac{\partial J\left(W\right) }{\partial W}$.

   2.c. Return weights.

Note that α is the *learning rate*, a factor used to control the speed at which the model learns.

The choice of the learning rate is relevant:

- Lower learning rates result in smaller changes. Smaller changes may allow the model to learn a more optimal set of weights, but may take

longer to train.

- Too low learning rates may lead the model to never converge or get stuck on a suboptimal set of weights.
- Higher learning rates result in bigger changes. Bigger changes may allow the model to learn faster, at the cost of never converging or arriving on a suboptimal final set of weights
- Too large learning rates may result in weights that will be too large and the model will generate divergent oscillations. In this case, gradient descent can inadvertently increase rather than decrease the training error.

Therefore, we should be careful to choose a learning rate that is not too small or large. We need to choose the one that allows us to find, on average, a good-enough set of weights.

We have now seen how a simple *Shallow Neural Network* is assembled and trained to optimise a model that will yield a prediction with the lowest error.

In the next section, we are going to use Python to create a model to predict house prices.

4 Code Walkthrough

In this notebook we will work through a code example of a simple and shallow neural net. Before doing so, we should consider the *business use-case*. Our scenario will look at house price prediction, is there a business use case here?

Of course, the ability to use AI-powered prediction is a real advantage. If done well, an estate agent that uses accurate predictions for the properties they deal with has a big advantage over an estate agent that uses human experts to make these predicitons, the AI-powered company can operate much more efficiently, consider:

A new property is brought to the estate agent, a form describing the property is filled out and entered into a database.

- The traditional company must then pay a human expert to make an initial price estimate, this can take time and estimates can be affected by human error or even the mood of that person on that day. If the expert is unfamiliar with the area the house is located in, or typically deals with smaller houses, their prediction take longer (costing the

estate agent more money), and be inaccurate.

- The AI-powered company has a program that automatically feeds this new data into the house prediction neural net, returning an estimate almost instantaneously. The neural net can do this for millions of new houses almost every second, the costs of running this neural net are minimal.

Already we can see the potential of something like this, however, it must be done carefully and a human may still need provide a final estimate. There is the case of *Zillow*, a real-estate marketplace that used AI to power a rapid home buying and reselling tool called "*Zillow Offers*". Zillow Offers was their AI-powered house price estimation tool, that provided start-to-end house pricing with little human oversight, unfortunately this turned out very poorly for Zillow who quickly shut down Zillow Offers, wrote off $569M worth of homes, and laid off 25% of its staff [source].

On the other hand, other companies such as Opendoor and Offerpad seem to have implemented AI powered price estimates with much more success [source], showing that this approach can be implemented successfully, if done right.

In [11]:
```
# !pip install numpy==1.21.5
# !pip install pandas==1.3.5
# !pip install scikit_learn==1.0.2
# !pip install openpyxl==3.0.9
# !pip install matplotlib==3.3.4
```

In [12]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

# to show all the generated plots inline in the notebook
%matplotlib inline
```

4.1 The Dataset

The dataset consists of house price data for properties in Pune, India. We have 200 individual data points across 17 different features. Our aim is to see if we can build a simple *Deep Learning Model* using *Numpy* only. Note that we won't be using *PyTorch* at this stage - this will be covered in a future course.

We will start by loading and visualising the data, analysing the variable types, and selecting the variables needed.

See the *DataDescription.pdf* for each variable's description.

In [13]:
```
# load the data
df = pd.read_excel('Real_Estate_Data.xlsx')
```

In [14]:
```
# view the top rows
df.head()
```

Out[14]:

	Sr. No.	Location	Sub-Area	Propert Type	Property Area in Sq. Ft.	Price in lakhs	Price in Millions	Company Name	To
0	1	Pune, Maharashtra, India	Bavdhan	1 BHK	492.0	39	3.9	Shapoorji Paloonji	
1	2	Pune, Maharashtra, India	Bavdhan	2 BHK	774.0	65	6.5	Shapoorji Paloonji	
2	3	Pune, Maharashtra, India	Bavdhan	3 BHK	889.0	74	7.4	Shapoorji Paloonji	
3	4	Pune, Maharashtra, India	Bavdhan	3 BHK Grand	1018.0	89	8.9	Shapoorji Paloonji	
4	5	Pune, Maharashtra, India	Mahalunge	2BHK	743.0	74	7.4	Godrej Properties	

In [15]:
```
# check the data types in our dataframe:
df.dtypes
```

Out[15]:

	0
Sr. No.	int64
Location	object
Sub-Area	object
Propert Type	object
Property Area in Sq. Ft.	float64
Price in lakhs	object
Price in Millions	float64
Company Name	object
TownShip Name/ Society Name	object
Total TownShip Area in Acres	float64
ClubHouse	object
School / University in Township	object
Hospital in TownShip	object
Mall in TownShip	object
Park / Jogging track	object
Swimming Pool	object
Gym	object

dtype: object

Analysing the dataset, we can see that we have a mixture of data types, some are quantitative (i.e., '*int64*', '*float64*'), but the majority qualitative in nature ('*object*'). To run our *Neural Network Model (Multilayer Perceptrons)*, we will need to have floating variables only ('*float64*') as these are the only variable type the neural network can perform computation on; it can also work with integer numbers, but it will convert them into floating.

Therefore, in the next section, we are going to clean our main dataset to improve our data quality and be able to run the model.

4.2 Dataset Cleaning

As discussed in the previous section, we are not going to clean our dataset. Specifically, we are going to follow the below steps:

1. Extract the relevant variables.
2. Remove rows containing '*NaN*' values, if any.
3. Convert qualitative variables into quantitative, i.e., from '*object*' to '*float64*'.
4. Scale variables, if needed.

Let's start!

4.2.1 Extract the Relevant Variables

For our model, we are going to select 5 variables that are probably the most relevant:

1. Propert Type
2. Property Area in Sq. Ft.
3. Sub-Area
4. Swimming Pool
5. Price in Millions

Important to note, '*Price in Millions*' is the variable we want to predict, therefore, we will split it from the dataset afterwards.

Let's start by selecting those variables.

In [16]:
```
# variables selection
df_selection = df[['Propert Type','Property Area in Sq. Ft.','Sub-Area','Swimm
```

In [17]:
```
df_selection.head()
```

Out[17]:

	Propert Type	Property Area in Sq. Ft.	Sub-Area	Swimming Pool	Price in Millions
0	1 BHK	492.0	Bavdhan	Yes	3.9
1	2 BHK	774.0	Bavdhan	Yes	6.5
2	3 BHK	889.0	Bavdhan	Yes	7.4
3	3 BHK Grand	1018.0	Bavdhan	Yes	8.9
4	2BHK	743.0	Mahalunge	Yes	7.4

4.2.2 Remove Missing Data

Missing data can be handle in different ways, e.g., replacing, interpolating, or

removing data, depending on the size of the dataset, and the usefulness of that data point. In this case, we are going to remove each row containing *missing data*, or *NaN*, to avoid errors when running the model.

Let's check if we have any missing data in our dataset.

In [18]:
```
# let's check if we have any 'NaN' in the dataset
nan_rows = df_selection[df_selection.isnull().any(axis = 1)]
nan_rows
```

Out[18]:

	Propert Type	Property Area in Sq. Ft.	Sub-Area	Swimming Pool	Price in Millions
41	3BHK	1705.0	Keshav Nagar	Yes	NaN

We can see that row 41 includes missing values, therefore, we are going to drop them using the `drop` function.

In [19]:
```
# given we do have 'NaN' values, let's store them in a new variable ('nan_rows
nan_rows = nan_rows.index.values
```

In [20]:
```
# drop the respective rows
df_selection = df_selection.drop(nan_rows)
```

In [21]:
```
# let's now check that all rows containing NaN values have been removed
df_selection[df_selection.isnull().any(axis = 1)]
```

Out[21]:

	Propert Type	Property Area in Sq. Ft.	Sub-Area	Swimming Pool	Price in Millions

4.2.3 Variables Conversion from Qualitative to Quantitative

All the variables extracted from the original database, except for `'Price in Millions'`, are qualitative variables. Therefore, we will need to convert them into quantitative variables.

Let's start with the first variable, `'Propert Type'`, which, as the header suggests, represents the property type (i.e., 1 BHK is 1 bedroom, hall, and kitchen, 2 BHK is 2 bedroom, hall, and kitchen and so on). To convert the variable into *float*, we are going to extract the number included in the variable's value.

In [22]:
```
# extract the numerical value for the 'Propert Type'
df_selection['Propert Type'] = df_selection['Propert Type'].str.extract('(\d+)
```

```
<>:2: SyntaxWarning: invalid escape sequence '\d'
<>:2: SyntaxWarning: invalid escape sequence '\d'
/tmp/ipython-input-2787260432.py:2: SyntaxWarning: invalid escape sequence '\d'
  df_selection['Propert Type'] = df_selection['Propert Type'].str.extrac
t('(\d+)')
```

We must be aware that, this operation could have generated additional missing values. In the case one variable's value didn't include a number, but text only, the resulting value would be *NaN*. Therefore, we need to check once again, if missing values have been generated.

In [23]:
```
# let's check if we have any 'NaN' in the dataset
nan_rows = df_selection[df_selection.isnull().any(axis = 1)]
nan_rows
```

Out[23]:

	Propert Type	Property Area in Sq. Ft.	Sub-Area	Swimming Pool	Price in Millions
51	NaN	163.0	pimpri pune	no	5.4

Oops! As expected, row 51 is now *NaN*. Let's drop it and go on with the data cleaning.

In [24]:
```
# given we do have 'NaN' values, let's store them in a new variable ('nan_rows
nan_rows = nan_rows.index.values
# drop the respective rows
df_selection = df_selection.drop(nan_rows)
```

Let's now move to the next variable, `'Sub-Area`. In this case, we want to convert the variable, which is a categorical variable, into a format that can be readily used by our neural network. To do so, we are using '*one-hot encoding*', which helps us creating new binary variables (i.e.,including only True or False) to represent the original categorical values.

In [25]:
```
# converting our categorical variables using 'one-hot encoding'
area_encoded = pd.get_dummies(df_selection['Sub-Area'].str.lower().str.strip()

# check the categorical variables
area_encoded.head()
```

Out[25]:

	area_akurdi	area_balewadi	area_baner	area_bavdhan	area_bavdhan budruk	area_ kawad
0	0.0	0.0	0.0	1.0	0.0	0
1	0.0	0.0	0.0	1.0	0.0	0
2	0.0	0.0	0.0	1.0	0.0	0
3	0.0	0.0	0.0	1.0	0.0	0
4	0.0	0.0	0.0	0.0	0.0	0

5 rows × 33 columns

We can then concatenate the data base to our new binary variables using *concat*.

In [26]:
```
# concatenate df_selection with encoded variables
df_selection = pd.concat([df_selection,area_encoded], axis = 1)

# check the updated dataset
df_selection.head()
```

Out[26]:

	Propert Type	Property Area in Sq. Ft.	Sub-Area	Swimming Pool	Price in Millions	area_akurdi	area_balewad
0	1	492.0	Bavdhan	Yes	3.9	0.0	0.
1	2	774.0	Bavdhan	Yes	6.5	0.0	0.
2	3	889.0	Bavdhan	Yes	7.4	0.0	0.
3	3	1018.0	Bavdhan	Yes	8.9	0.0	0.
4	2	743.0	Mahalunge	Yes	7.4	0.0	0.

5 rows × 38 columns

Happy with the result, we can now remove the `'Sub-Area'` variable, using again the *drop* function.

In [27]:
```
# drop the 'Sub-Area' variable
df_selection = df_selection.drop('Sub-Area', axis = 1)
```

Let's finally have a look at the last variable, `'Swimming Pool'`. In particular, we want to analyse the its values.

In [28]:
```
df_selection['Swimming Pool'].unique()
```

Out[28]:
```
array(['Yes', 'No', 'no', 'yes', 'no '], dtype=object)
```

We can see that the array contains different values (some in lowercase, some in uppercase, and others with unnecessary space). Therefore, we need to (1) convert the values in lowercase using `lower()`, (2) remove any space using `strip()`, and (3) convert from string to binary using `map()`.

In [29]:
```
# convert the 'Swimming Pool' variable from string to binary
df_selection['Swimming Pool'] = df_selection['Swimming Pool'].str.lower().str.
    {'yes': True , 'no': False}
)
```

Let's check if we have converted correctly the variable `'Swimming Pool'` to binary, so that its values are True or False.

In [30]:
```
# check if the conversion happened correctly
df_selection['Swimming Pool'].unique()
```

Out[30]:
```
array([ True, False])
```

It looks like we achieved what we wanted! We can then move to the next step.

4.2.4 Variables Scaling

The final step consists of scaling (i.e., normalising between 0 and 1) the `'Property Area in Sq.Ft.'` as its values are too large compared to the other variables' values; this would overpower the other inputs' activations resulting in an unfair representation in the network. To do so, we are going to use the `MinMaxScaler()`.

In [31]:
```
# scale the 'Property Area in Sq.Ft.' variable
scaler = MinMaxScaler()
df_selection['Property Area in Sq. Ft.'] = scaler.fit_transform(df_selection[[
```

Let's check the resulting scaled variable:

In [32]:
```
df_selection['Property Area in Sq. Ft.'].head()
```

Out[32]:

	Property Area in Sq. Ft.
0	0.162382
1	0.283205
2	0.332476
3	0.387746
4	0.269923

dtype: float64

4.3 The Clean Dataset

At this point, we should have cleaned our dataset, which should contain only float variables. Let's ensure that this is true by converting all values to float.

In [33]:
```
# ensuring our variables are float
df_selection['Propert Type'] = df_selection['Propert Type'].astype(float, erro
df_selection['Property Area in Sq. Ft.'] = df_selection['Property Area in Sq.
df_selection['Swimming Pool'] = df_selection['Swimming Pool'].astype(float, er
df_selection['Price in Millions'] = df_selection['Price in Millions'].astype(f
```

In [34]:
```
df_selection.head()
```

Out[34]:

	Propert Type	Property Area in Sq. Ft.	Swimming Pool	Price in Millions	area_akurdi	area_balewadi	area_ban
0	1.0	0.162382	1.0	3.9	0.0	0.0	0
1	2.0	0.283205	1.0	6.5	0.0	0.0	0
2	3.0	0.332476	1.0	7.4	0.0	0.0	0
3	3.0	0.387746	1.0	8.9	0.0	0.0	0
4	2.0	0.269923	1.0	7.4	0.0	0.0	0

5 rows × 37 columns

Happy with the result, we can split our dataset in 2: one including all variables except for the `'Price in Millions'`, and one including only `'Price in Millions'`. This step will allow us to compare the variable predicted with the model to the actual variable, which is `'Price in Millions'`.

In [35]:
```
# dataset including all variables, but 'Price in Millions'
X = df_selection.drop(['Price in Millions'], axis = 1)
```

```
print(X.shape)

# dataset including only 'Price in Millions'
Y = df_selection['Price in Millions']
print(Y.shape)
```

```
(198, 36)
(198,)
```

Later on, we will need to perform array operations; therefore, if we have arrays with different shapes, we need to reshape them to make them comparable and be able to perform element-wise operations. This is the case of the variable `'Y'`, which has currently the shape (198,), but should be (198,1). Let's reshape it using *`reshape`*.

In [36]:
```
# reshape the array
Y = Y.values.reshape((Y.shape[0], 1))
print(Y.shape)
```

```
(198, 1)
```

We can now create *test data* and *training data*. Test data excludes the first 150 rows, while test data includes the last 48 rows. Also in this case, we need to transpose the data to be able to perfrom array operations.

In [37]:
```
# test data and shape
X_test=X[150:].T
Y_test=Y[150:].T
print('X_test:', X_test.shape)
print('Y_test:', Y_test.shape)

#training data and shape
X_train = X[:150].T
Y_train = Y[:150].T
print('X_train:', X_train.shape)
print('Y_train:', Y_train.shape)
```

```
X_test: (36, 48)
Y_test: (1, 48)
X_train: (36, 150)
Y_train: (1, 150)
```

We can now move to the next section, where we will start looking at the model training!

4.4 Single-Layer Neural Network Computation

We are now going through the *Single-Layer* computation, the simplest form of neural network. Defined like that as there is only one layer of input nodes that send weighted inputs to a subsequent layer of receiving nodes.

We will start with the forward-propagation and finish with backward propagation. Let's begin!

4.4.1 Forward-Propagation

The process is based on the below two steps:

1. *Inputs initialization*. Weighted sum of inputs from the previous layer plus the bias.
2. *Activation*. The calculated weighted sum of inputs is passed to the activation function to add non-linearity to the network. Among the most popular activation functions, we are considering the *Rectified Linear Unit (ReLU) activation function*.

With the *inputs initialization*, we want to initialize the weights (`W`), the bias (`b`), and the resultant output (`Z`) from these.

The weight initialization usually depends on the activation function considered in the analysis. The standard approach when using a ReLU activation function is the *he* initialization method.

This method generates weights as a random number with a Gaussian probability distribution (G) with a mean of 0.0 and a standard deviation of sqrt(2/n), where n is the number of inputs to the node.

weight ~ G (0.0, sqrt(2/n))

In our example, n equals 36.

In [38]:
```
# the 'he' initialization
n_in = X_train
n_out = Y_train
n = n_in.shape[0] # this equals to 36
print(n)
# calculate the standard deviation
std = np.sqrt(2.0/n_out.shape[0])
# calculate the weights
W = np.random.randn(n_out.shape[0], n_in.shape[0]) * std
W
```

```
36
```

Out[38]:
```
array([[ 1.1528171 ,  0.00994152,  0.49172867, -0.65851438, -1.21481457,
         0.45385015, -2.04466581,  1.21255874,  0.53089738, -2.05636223,
         0.52442176,  1.39343178,  3.41098992,  2.17534524,  0.48407963,
         3.15902586,  0.27065191, -1.24109468,  2.49589863, -1.28956183,
         1.1982218 ,  2.62502814, -0.82922011, -0.19658357, -0.0191041 ,
        -1.59708965,  1.01908022,  0.76006963,  2.17664068, -0.62530853,
        -0.03110986,  0.56442724, -3.74679821,  0.59516615, -1.10438279,
        -0.02926421]])
```

Once we initialized the weights, we should do the same with the bias (`b`). In this case, we will set it to zero, for simplicity.

In [39]:
```
n_out.shape
```

Out[39]:
```
(1, 150)
```

In [40]:
```
# initialize bias 'b' as zero
b = np.zeros((n_out.shape[0], 1))
```

Now, we are ready to perform a *forward-propagation*, which refers to the calculation and storage of intermediate variable (`Z`) for a neural network from the input layer to the output layer.

$$Z = X^{T}W + b$$

In [41]:
```
# initialize the intermediate variable
Z = np.dot(W, n_in) + b
Z.shape
```

Out[41]:
```
(1, 150)
```

Next, we pass to the *activation*. The scope is to process the output Z through an *activation function*. As mentioned, we have chosen to use the *ReLU* function for our activation. This is a linear function that will output the input directly if it is positive, otherwise, it will output zero.

The output $\hat{Y}$ represents our *prediction*.

In [42]:
```
# ReLU function
Y_hat = np.maximum(0,Z)
```

In [43]:
```
Y_hat.shape[1]
```

Out[43]:
```
150
```

We are now going to compare the prediction $\hat{Y}$ to a *true* value Y to calculate the *Mean Squared Error (MSE)* between predicted and true values. This can be calculated using the Loss Function $L(\hat{Y},Y)$. $$MSE = L(\hat{Y},Y) =

\frac{1}{n}\ (\hat{Y} - Y)^{2}$$

In [44]:
```
# define the true value 'Y' and the sample size 'n'
Y = Y_train
n = Y_hat.shape[1]

# calculate the mean squared error or loss function
E = (1 / n) * np.sum(np.square(Y_hat - Y))
```

In [45]:
```
# this is the error before training our neural network
E, E.shape
```

Out[45]:
```
(np.float64(149.33398848980875), ())
```

4.4.2 Backward-Propagation

We are not stepping into the backward-propagation. This is a way of propagating the total loss back into the neural network to see how much of the loss every node is responsible for, and then updating the weights to minimize the loss by giving the nodes with higher error, lower weights and vice-versa.

First, we need to derive the slope of the function to understand the direction we need to go to. If the slope is positive, then we would need to move to the right; otherwise, we would need to move to the left.

To do that, we can calculate the derivative of the mean squared error (MSE) with respect to the output activation $\hat{Y}$.

$$\dfrac{\partial E }{\partial \hat{Y}}=\dfrac{2}{n}\left(\hat{Y}- Y\right)$$

In [46]:
```
# derivative of the loss function
dE = (2 / n) * (Y_hat - Y)
dE.shape
```

Out[46]:
```
(1, 150)
```

After, we need calculate the derivative of the ReLU function to update the weights of a node as part of the backpropagation of error.

The derivative of the ReLU function is the slope. The slope for negative values is 0.0 and the slope for positive values is 1.0.

In [47]:
```
# derivative of the ReLU function
dY_hat = (Z > 0)
dY_hat.shape
```

Out[47]:
```
(1, 150)
```

How to optimize the weights? We can use a backward propagation with gradient descent.

The idea is to calculate gradients of the loss function with respect to the model weights and propagate them back layer by layer. This way, the model knows the weights, responsible for creating a larger error, and tunes them accordingly.

What we are aiming is minimize the cost. Gradient indicates the direction of increase. As we want to find the minimum point, we need to go in the opposite direction of the gradient. We update parameters in the negative gradient direction to minimize the loss.

In [48]:
```
# input data
dZ = n_in
dZ.shape
```

Out[48]:
```
(36, 150)
```

In [49]:
```
# calculate gradients of the loss function with respect to the weights
dW = np.dot((dE * dY_hat),dZ.T)
dW, dW.shape
```

Out[49]:
```
(array([[-3.44552529e+01, -5.24316524e+00, -9.96079591e+00,
         -6.13677222e-01, -3.31335398e-01, -1.01840445e-01,
         -2.43017321e-01,  0.00000000e+00, -9.08244085e-01,
          0.00000000e+00, -4.38817274e-01, -1.01721811e-01,
         -2.89536048e-01, -5.75635650e-01, -8.11503169e-01,
          1.15388321e-02, -6.69892575e-01, -2.03571595e-02,
         -1.02859624e-01, -2.33654513e-01, -2.34655504e-01,
          0.00000000e+00, -1.79837781e-01, -5.24074933e-01,
         -1.12818104e-01,  0.00000000e+00, -9.68854374e-02,
         -1.16694760e+00, -2.30229487e+00, -9.03088670e-01,
         -8.01711310e-02,  0.00000000e+00,  0.00000000e+00,
         -4.33807513e-02, -8.38149955e-01, -2.06606545e-01]]),
 (1, 36))
```

In [50]:
```
W, W.shape
```

Out[50]:
```
(array([[ 1.1528171 ,  0.00994152,  0.49172867, -0.65851438, -1.21481457,
          0.45385015, -2.04466581,  1.21255874,  0.53089738, -2.05636223,
          0.52442176,  1.39343178,  3.41098992,  2.17534524,  0.48407963,
          3.15902586,  0.27065191, -1.24109468,  2.49589863, -1.28956183,
          1.1982218 ,  2.62502814, -0.82922011, -0.19658357, -0.0191041 ,
         -1.59708965,  1.01908022,  0.76006963,  2.17664068, -0.62530853,
         -0.03110986,  0.56442724, -3.74679821,  0.59516615, -1.10438279,
         -0.02926421]]),
 (1, 36))
```

How can we increase/decrease the size of weight updates? We can use the *learning rate* α.

Unfortunately, we cannot calculate the optimal learning rate, but we can derive a good-enough learning rate via trial and error to find, on average, a good-enough set of weights. Typically, the range of values to consider for the learning rate is less than 1 and greater than 10^{-6}.

After running the model several times, we chose $\alpha = 0.1$, and used it for the below weight optimization equation.

$$W = W - \alpha \dfrac{\partial L }{\partial w}$$

In [51]:
```
# define the 'learning rate'
alpha = 0.1

# update the weights using the learning rate
W = W - (alpha * dW)

# see updated weights
W, W.shape
```

Out[51]:
```
(array([[ 4.59834239,  0.53425805,  1.48780826, -0.59714666, -1.18168103,
          0.46403419, -2.02036408,  1.21255874,  0.62172179, -2.05636223,
          0.56830349,  1.40360396,  3.43994353,  2.2329088 ,  0.56522995,
          3.15787197,  0.33764117, -1.23905897,  2.50618459, -1.26619638,
          1.22168735,  2.62502814, -0.81123633, -0.14417608, -0.00782229,
         -1.59708965,  1.02876877,  0.87676439,  2.40687017, -0.53499966,
         -0.02309275,  0.56442724, -3.74679821,  0.59950422, -1.02056779,
         -0.00860356]]),
 (1, 36))
```

As we did for the weights, we need now to calculate the derivative of the loss function, but with respect to the bias (b). After, we need to update the bias using the learning rate (α).

In [52]:
```
# derivative of loss function with respect to bias
db = np.sum((dE * dY_hat), axis = 1, keepdims = True)

# update bias using the learning rate
b = b - (db * alpha)
```

The above calculations represent an *epoch*, i.e., an entire loop through the forward and back-propagation process using every sample in the training set; however, only one epoch will lead to *underfitting error*. That is why, to get to an optimal model, we need to increase the number of epochs. Note that using too many epochs might result in *overfitting error* instead. There is no correct number of epochs, it depends on the size and type of dataset and model. In this example, we are going to use 100 epochs, as it seems to give us the optimal curve.

Let's now iterate the process to *optimize* our weights and predict a value $\hat{Y}$

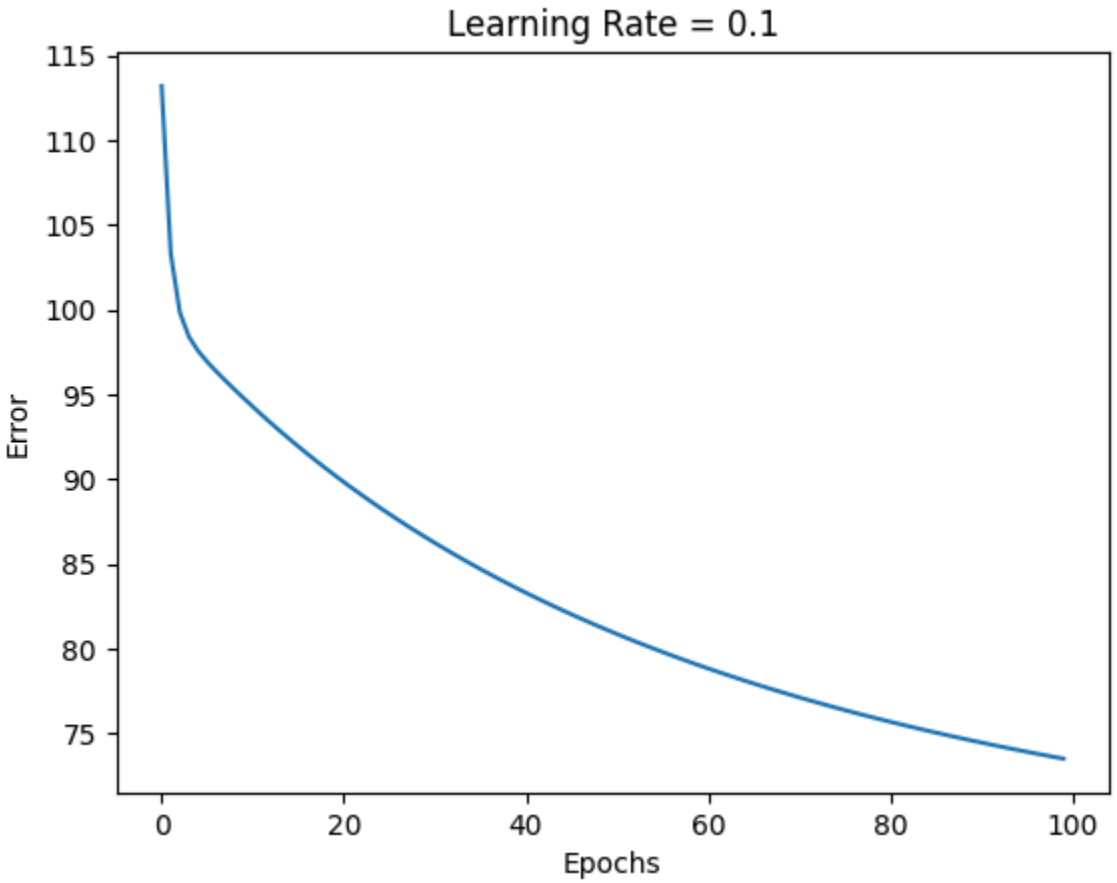
closer to the true value Y. To do so, we are going to use the `for` loop.

In [53]:
```
# define a variable to store the error calculated in each iteration
error = []

# set number of epochs
epochs = 100

# iteration
for i in range(epochs):

    # initialize the intermediate variable
    Z = np.dot(W, n_in) + b

    # forward-propagation
    # ReLU activation
    Y_hat = np.maximum(0,Z)

    # define the sample size 'n'
    # note that the true value 'Y' is already defined in the above calculation
    n = Y_hat.shape[1]

    # calculate the mean squared error or loss function
    E = (1 / n) * np.sum(np.square(Y_hat - Y))

    # backward-propagation
    # derivative of the loss function
    dE = (2 / n) * (Y_hat - Y)

    # derivative of Relu Activation function
    dY_hat = (Z > 0)

    # input data
    dZ = n_in

    # derivative of loss function with respect to weights
    dW = np.dot((dE * dY_hat),dZ.T)

    # derivative of loss function with respect to bias
    db = np.sum((dE * dY_hat), axis = 1, keepdims = True)

    # update weights
    W = W - (dW * alpha)

    # update bias
    b = b - (db * alpha)

    # error
    error.append(E)
```

We can now plot the error using `matplotlib`.

In [54]:
```
# plot the error
```

```
plt.ylabel('Error')
plt.xlabel('Epochs')
plt.title("Learning Rate = " + str(alpha))
plt.plot(np.squeeze(error))
plt.show()
```



Now we are going to apply the knowledge from the trained neural network model on the test data and use it to infer the result.

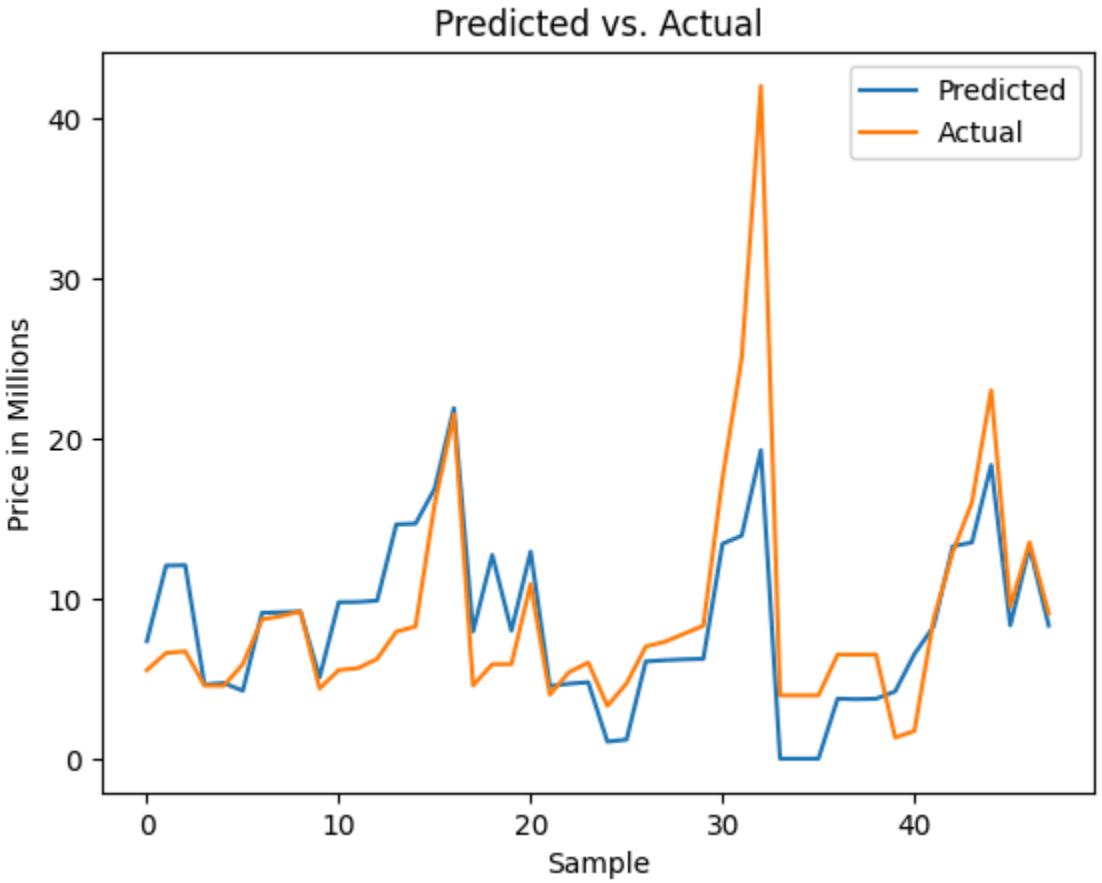
In [55]:
```
# let's now push our test set through the neural network with our learned weig
Z_predict = np.dot(W, X_test) + b

# activation function
Y_hat_predict=np.maximum(0,Z_predict)

# define the size
n = X_test.shape[1]

# error prediction
E_predict = (1 / n) * np.sum(np.square(Y_hat_predict - Y_test))
```

Let's look at the error after training.

In [56]:
```
# updated error
E_predict
```

Out[56]: np.float64(22.580249623715662)

In [57]:
```
# plot the predicted values vs. the actual values
plt.ylabel('Price in Millions')
plt.xlabel('Sample')
plt.title("Predicted vs. Actual")
plt.plot(np.squeeze(Y_hat_predict),label = "Predicted")
plt.plot(np.squeeze(Y_test), label = "Actual")
plt.legend()
plt.show()
```



You can see from even this very simple example that a neural net can be setup to train and update its own weights to improve upon its error/cost.

However, you can also see that using the current single hidden layer network, our predictions are generally correlated but not too accurate. We will cover this in more depth in future chapters, but the reason for this is that our network is unable to represent complex relationships with only a few weights. However, when we take the same concepts discussed above and apply them across multiple layers with more neurons and adjustable weights/biases more complex relationships can be represented.

The End:)