SUMMARY

Structure of Neural Networks

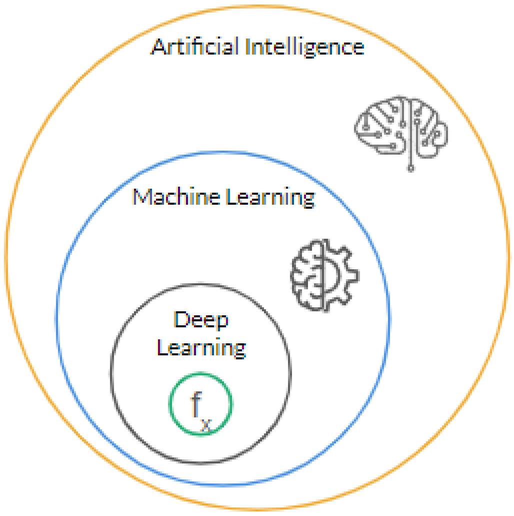
In this session, you learnt about certain limitations of previously learnt machine learning (ML) models and how to overcome some of them by using artificial neural networks (ANNs). Next, you were introduced to the concept of ANNs and how it is inspired from the human brain.

You learnt the working of the very first neural network model, i.e., perceptrons, and how they can be used for classifying problems. Next, you will see how the ANNs evolved from the basic perceptrons. You also learnt about the basic building blocks of ANNs, i.e., artificial neurons, and how they together build a neural network.

Introduction to Deep Learning

Deep learning is a branch of machine learning (ML) and usually deals with unsupervised data such as images, videos, text, and speech data. Some of the most common day-to-day deep learning-based applications are face recognition to unlock your devices or voice assistants such as Siri, Google Voice Assistant, and Alexa.

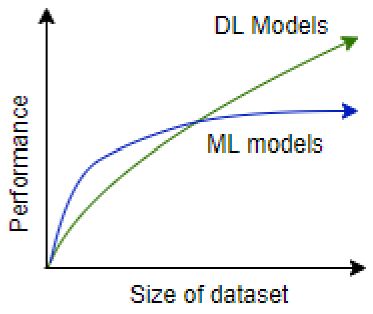
Deep learning is a subset of machine learning, which is, in turn, a subset of artificial intelligence. The image given below gives more clarity on this structure.



Similar to machine learning, deep learning has multiple algorithms to deal with a variety of data and problems. The word ‘deep’ in deep learning signifies the depth of the neural network that enables the network to approximate the feature representations by passing through multiple layers and learning from its errors.

Deep learning makes remarkably complex things possible, for example, sentiment analysis and self-driving cars. Most of these complex applications cannot be built using simple machine learning algorithms, so let us understand the reason for this from the points given below:

1. The performance of deep learning models increases with the increase in data available; however, with preliminary machine learning algorithms, this is not true as shown in this image.

   As the volume of data increases, the deep learning models tend to outperform the preliminary machine learning models.

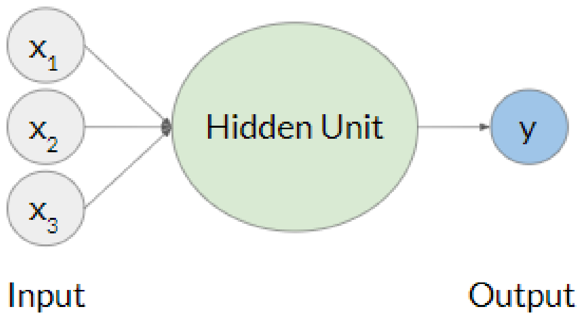
2. Machine learning models such as regression require a large number of assumptions to build models as compared with deep learning models.

3. Machine learning algorithms like regression usually require structured data, whereas deep learning networks can be applied to both structured and unstructured data.

4. Additionally, preliminary machine learning algorithms require substantial human intervention in terms of the time required in feature engineering, whereas in deep learning, the nested layers allow the data to learn from their errors.

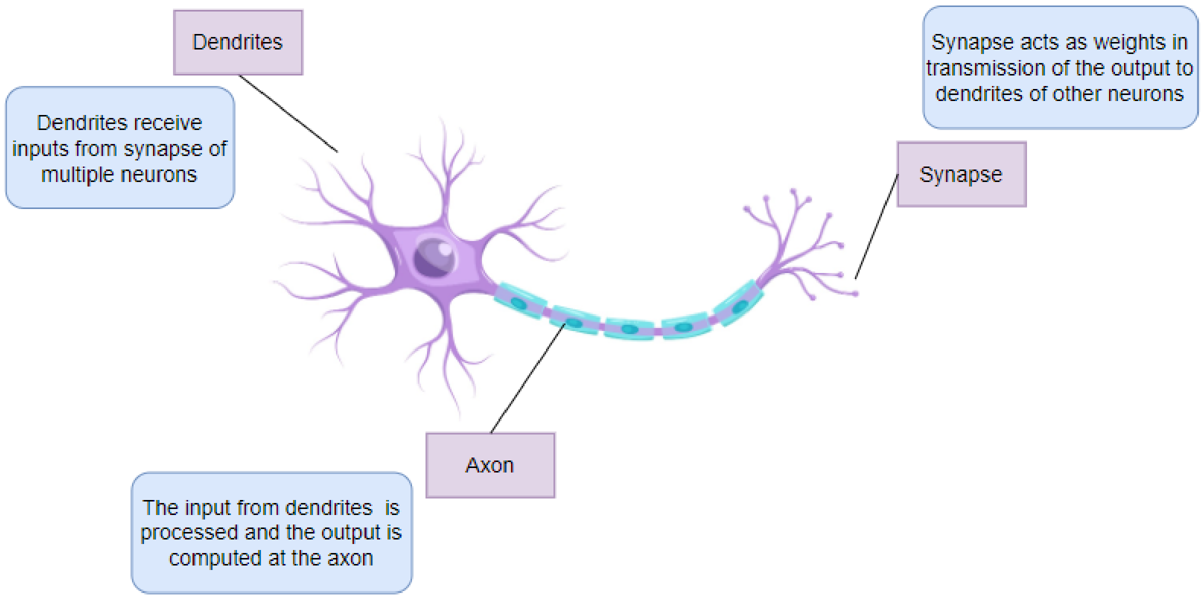
Human Brain and Artificial Neural Networks

As the name suggests, the design of ANNs is inspired by the human brain. Although not as powerful as the brain yet, artificial neural networks are the most powerful learning models in the field of artificial intelligence and machine learning. ANNs are a collection of a large number of simple devices called artificial neurons as shown in the image below.



A large number of these artificial neurons working together form the ANN. This network ‘learns’ to conduct certain tasks, such as recognising a cat, by training the neurons to ‘fire’ in a certain way when given a particular input, such as a cat.

In other words, the network inhibits or amplifies the input signals to implement a certain task, such as recognising a cat, speaking a word, or identifying a tree. The image given below provides an understanding of how different parts of an artificial neuron correspond to those of a biological neuron.


Dendrites
Dendrites receive inputs from synapse of multiple neurons
Synapse acts as weights in transmission of the output to dendrites of other neurons
Synapse
Axon
The input from dendrites is processed and the output is computed at the axon

A biological neuron receives information (inputs) from multiple neurons. These inputs are received in the dendrites of the neuron, processed in the axon (cell body) and then the output is computed. This output is then transmitted to the other neurons through the synapse.

Similarly, an artificial neuron receives information from multiple neurons with weights associated with them; the weighted inputs are then inhibited or amplified as they pass through the body of the artificial neuron. The output thus computed is passed on to the next neuron.

Let’s take a look at the basic differences between the biological and artificial neurons as mentioned below:

1. While the brain consists of billions of biological neurons, ANNs can consist of neurons ranging from 10 to 1000s of them. There is no restriction on the number of neurons that an ANN can have, but as the number of neurons increases, the complexity of the model also increases. Therefore, they are generally limited to thousands in a network.

2. The connections between the biological neurons are asynchronous and not arranged in any specific pattern, whereas artificial neurons are organised in the form of layers.

3. The speed of functioning in the human brain depends on how fast impulses travel from one neuron to another, which cannot be controlled manually. However, in the case of ANNs, you can control the speed of functioning as the neurons do not get tired.

Perceptron

In this segment, you learnt about a simple device called the perceptron, which was the first step towards creating the large neural networks you see today.

A perceptron acts like a tool that enables you to make a decision based on multiple factors. They take the different factors as input signals, attach a weight based on their importance, and perform basic operations to make a decision. In other words, the perceptron takes a weighted sum of multiple inputs (along with a bias) as the cumulative input and applies an output function on the cumulative input and returns the output.

$$\text{Cumulative Input} = w_1x_1 + w_2x_2 + w_3x_3 + b$$

Where, x_i's represent the inputs, w_i's represent the weights associated with inputs, and b is the bias.

The above equation can be written in the vector form as well. A neat and concise way to represent the weighted sum of w and x is using the dot product of w^T and x. Let us understand this concept by taking the following weight and input vector.

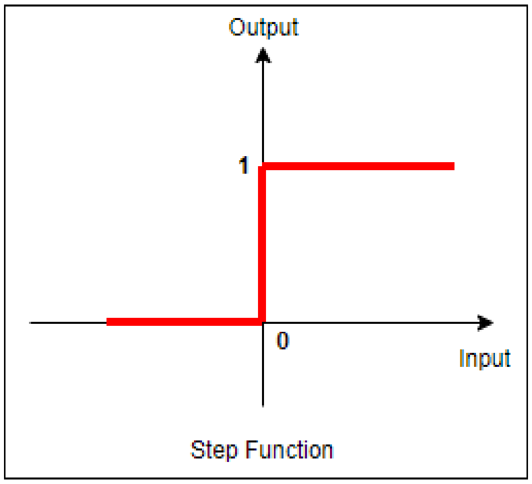
$$w = \begin{bmatrix} w_1 \\ w_2 \\ . \\ . \\ w_k \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_k \end{bmatrix}$$

The transpose of w is $w^T = [w_1 \ w_2 \ .. \ w_k]$, which is a row vector of size 1 x k. Taking the dot product of w^T with x, you will get the following result:

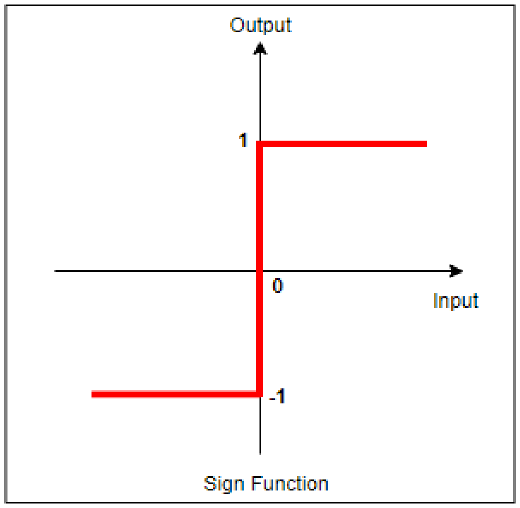
$$w^T.x = \begin{bmatrix} w_1 & w_2 & . & . & w_k \end{bmatrix} . \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_k \end{bmatrix} = w_1x_1 + w_2x_2 + ... + w_kx_k$$

Now, to obtain the output, an output function is applied to the cumulative input. There are two most commonly used output functions:

1. Step function: Y = 1 if x > 0 and Y = 0 if x <= 0



2. Sign function: Y = 1 if x > 0 and Y = -1 if x <= 0



As you can see above, the output of a perceptron in the case of a step function is either 1 or 0 and that in the case of a sign function is either 1 or -1. In both cases, the output is classified into two different classes. Perceptron, thus, forms a powerful classification algorithm for supervised learning.

Perceptron as a Classifier

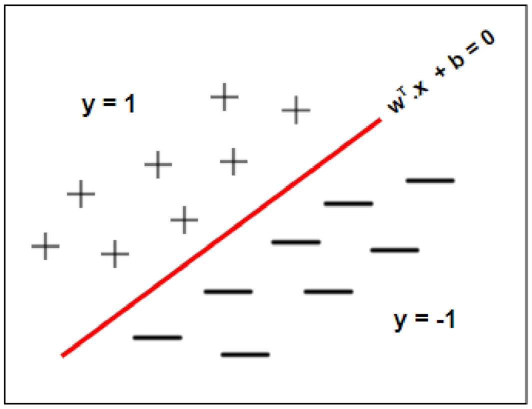
In this segment, you saw how a perceptron acts as a classifier. When Frank Rosenblatt proposed the model of a perceptron, it was designed to solve only binary classification problems. Given the input and output labels, one must be able to make the decision based on the weights of different inputs and the bias of the perceptron.

If you are given the following inputs and their corresponding outputs - (x_1,y_1), (x_2,y_2), ……, (x_n,y_n), and the output function is the sign function. According to the perceptron function, the output will be:

$$y = \text{sign}(w^T.x + b)$$

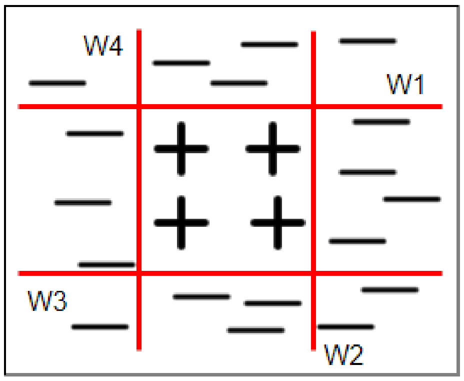
Now, given the inputs and their corresponding outputs, you are required to predict the weight matrix and bias such that you can correctly predict the output label from the input label. How can you do this?

Take a look at the equation of the perceptron given above; it is a linear expression and can be represented as a line with the equation, $w^T.x + b = 0$, such that all the points on one side of this line are positive values of y and those on the other side are negative values of y as shown below.

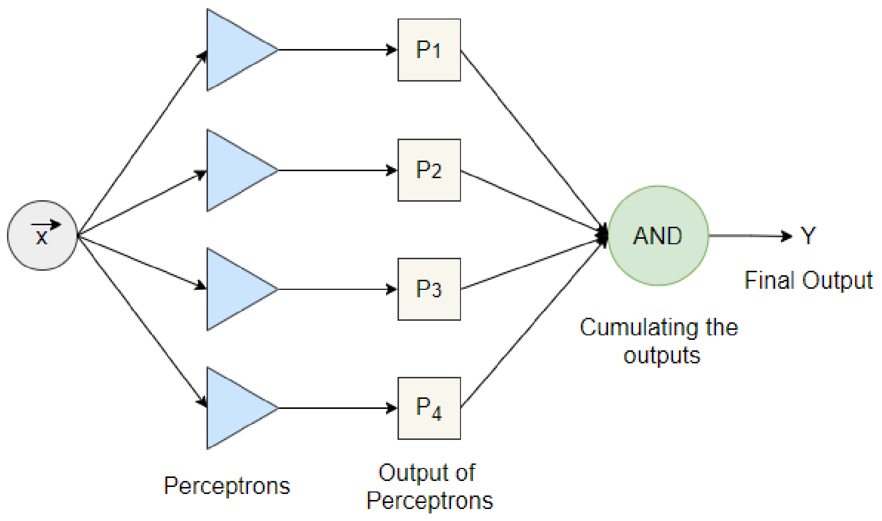


The line $w^T.x + b = 0$ is known as a **separator**. This separator is valid only if $w^T.x + b < 0$ when $y = -1$ and $w^T.x + b > 0$ when $y = 1$. Therefore, you can say that $y(w^T.x + b) > 0$ should stand true for all data points.

It is not possible for the decision boundary/separator to always be linear. There might be certain cases where you will come across complex data sets where you will not be able to split the data points into two planes. A single perceptron is not capable of performing nonlinear classification. But you can obtain the desired result by using multiple perceptrons (separators) to classify your data as shown in the image below.



As you can see, this can be done using four different perceptrons and then computing the resulting output from the individual output of each perceptron as shown in the image below.

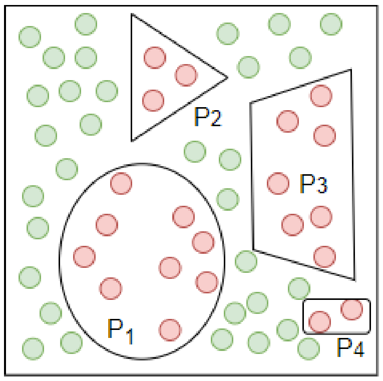


If sum of outputs = $P_1 + P_2 + P_3 + P_4 >_1 4$, then Y = 1
Else, Y = -1

Where, $P_i = W_i * X_i + b$

This is how a perceptron can act as an AND gate.

Perceptrons can also be used as an OR gate. Take a look at the example in the image below.

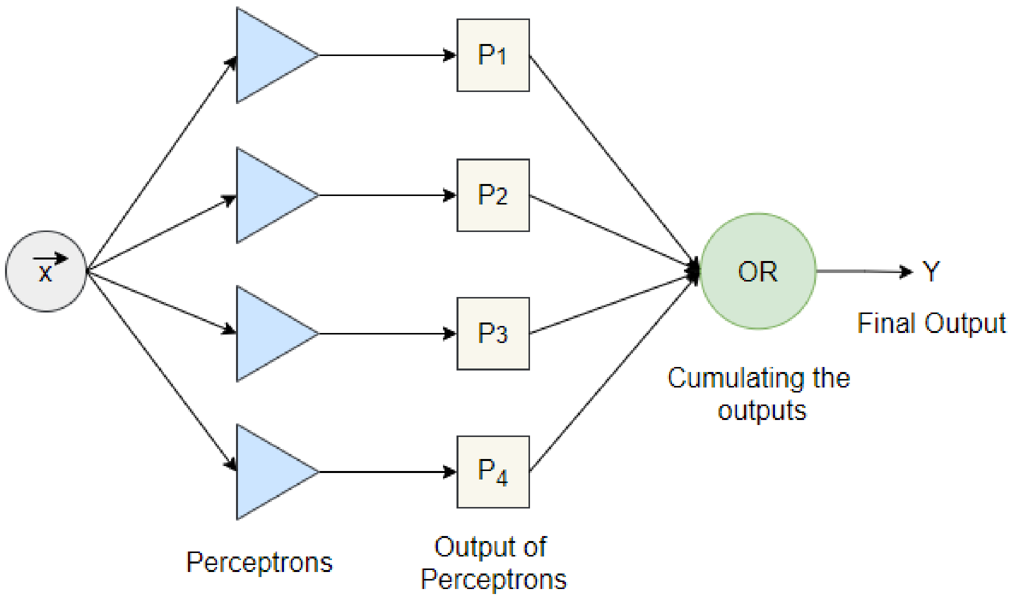


Now, for this example, if the data points lie in a polygon (non-linear separator), the output of the perceptron P_i will be 1; else, it will be -1, i.e., P_1 = 1, if data points lie in the separator; else, it will be -1. If neither of these data points lies in any of the separators, the overall sum of outputs of each perceptron will be $P_1 + P_2 + P_3 + P_4 = -4$ and, in such a case, the overall output will be -1.

Hence, you can conclude that:

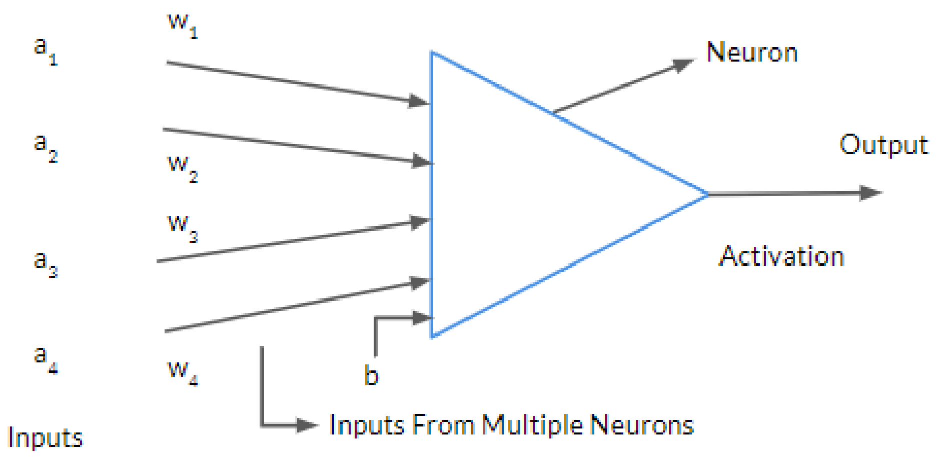
If the sum of outputs = $P_1 + P_2 + P_3 + P_4 > -4$, then Y = 1
Else, Y = -1

Take a look at the following image to understand how a perceptron works as an OR gate.



Artificial Neuron

Neural networks are a collection of artificial neurons arranged in a particular structure. Take a look at the structure of an artificial neuron in the image below.



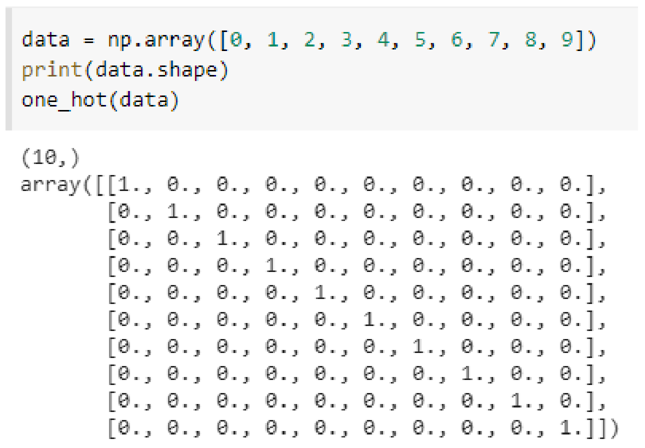
Where 'a' represents the inputs, 'w' represents the weights associated with the inputs and 'b' represents the bias of the neuron.

The first layer is known as the **input layer**, and the last layer is called the **output layer**. The layers in between these two are the **hidden layers**. The number of neurons in the input layer is equal to the number of attributes in the data set and that in the output layer is equal to several classes of the target variable (for a classification problem), and for a regression problem, the number of neurons in the output layer would be 1 (a numeric variable).

Inputs and Outputs of Neural Networks

In this segment, you learnt about the common input and outputs of a neural network. There are three common types of inputs:

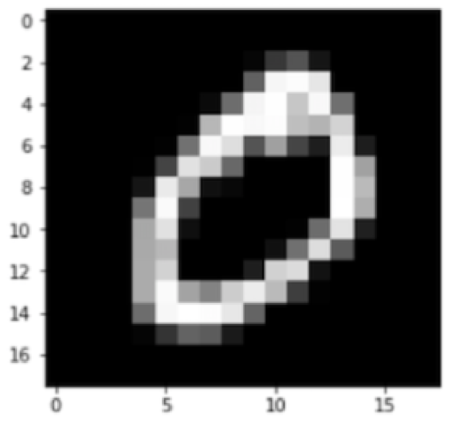
1. **Text data:** For text data, you can use a one-hot vector or word embeddings corresponding to a certain word. The one-hot encoded array of the digits zero (0) to 9 will look like the image below.



```
data = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(data.shape)
one_hot(data)
```

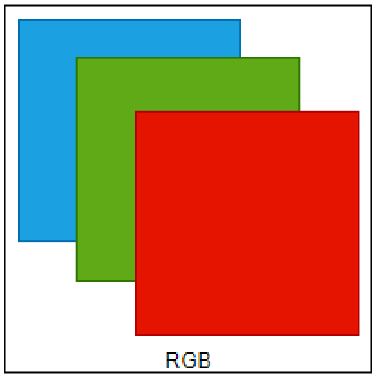
```
(10,)
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

2. **Images:** Images are naturally represented as arrays of numbers and can thus be fed to the network directly. These numbers are the raw pixels of the image. '**Pixel**' is the short form of a picture element. In images, pixels are arranged in rows and columns (an array of pixel elements). The figure given below shows an image of a handwritten 'zero' in the MNIST data set (black and white) and its corresponding representation in NumPy as an array of numbers. The pixel values are high where the intensity is high, i.e., the colour is bright, while the values are low in the black regions.



```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   6  55  84  22   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  96 244 250 228   0   0   0   0   0]
 [  0   0   0   0   0   0   0   9 108 243 252 196 247 110   0   0   0   0]
 [  0   0   0   0   0   0   2 181 252 247 251 189 178 210   0   0   0   0]
 [  0   0   0   0   0   2 112 247 220  84 159  69  30 234  29   0   0   0]
 [  0   0   0   0   1  68 223 201 103   0   0   0   0 252 160   0   0   0]
 [  0   0   0   0  21 232 166  17   7   0   0   0   0 252 184   0   0   0]
 [  0   0   0   0 116 248  65   0   0   0   0   0   0 253 172   0   0   0]
 [  0   0   0   0 167 223  15   0   0   0   0   2 107 225  33   0   0   0]
 [  0   0   0   0 168 182   0   0   0   0  16 111 219  90   0   0   0   0]
 [  0   0   0   0 169 208   0   0   0  30 207 217  18   0   0   0   0   0]
 [  0   0   0   0 169 248 162 130 202 234 184  62   2   0   0   0   0   0]
 [  0   0   0   0 108 245 253 251 229  99   0   0   0   0   0   0   0   0]
 [  0   0   0   0   5  52  98  91  26   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   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 a coloured image (called an RGB image - Red, Green, Blue), each pixel would have three channels, one each for red, blue and green as shown below. Hence, the number of neurons in the input layer would be 18 x 18 x 3 = 972. The three channels of an RGB image are shown here.

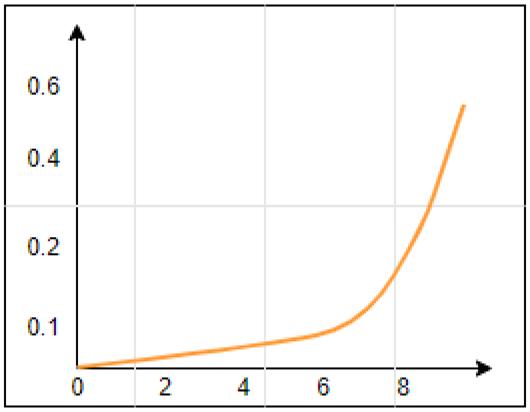


3. **Speech:** In the case of speech/voice input, the basic input unit is in the form of **phonemes**. These are the distinct units of speech in any particular language. The speech signal is in the form of waves, and to convert these waves into numeric inputs, you can use Fourier transforms. Take a look at the formula of the discrete Fourier transform.

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi kn}{N}}$$

Now that you have looked at how to feed input vectors to neural networks, let's look into the outputs. Depending on the nature of the task, the outputs of neural networks can either be in the form of classes (if it is a classification problem) or numeric (if it is a regression problem).

One of the commonly used output functions is the **softmax function.** Take a look at the graphical representation of the softmax function shown below.



A softmax output is a multiclass logistic function commonly used to compute the probability of an input belonging to one of the multiple classes. It is given by the following formula:

$$p_i = \frac{e^{w_i . x'}}{\sum_{t=0}^{c-1} e^{w_t . x'}}$$

Where, c is the number of neurons in the output layer, x' is the input to the network, and w_i's are the weights associated with the inputs.

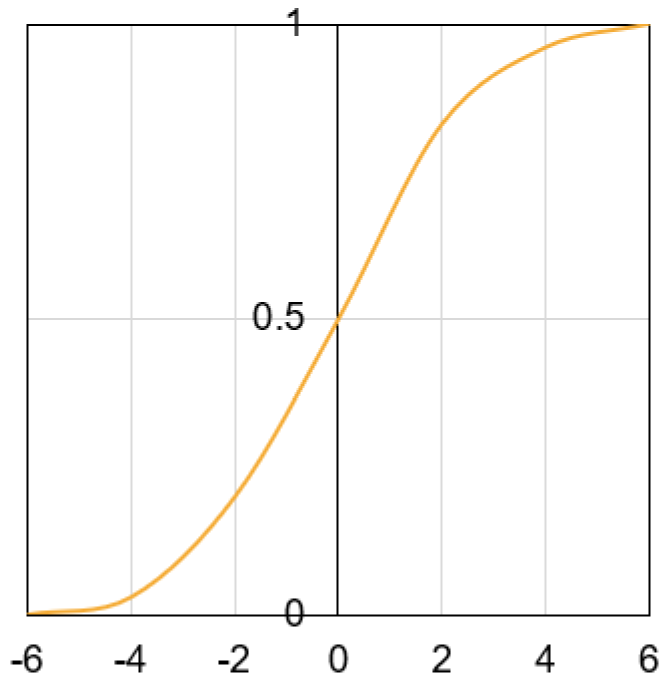
Activation Functions

The main conditions that you must keep in mind while choosing activation functions are as follows:

1. Non-linearity
2. Continuous
3. Monotonically increasing

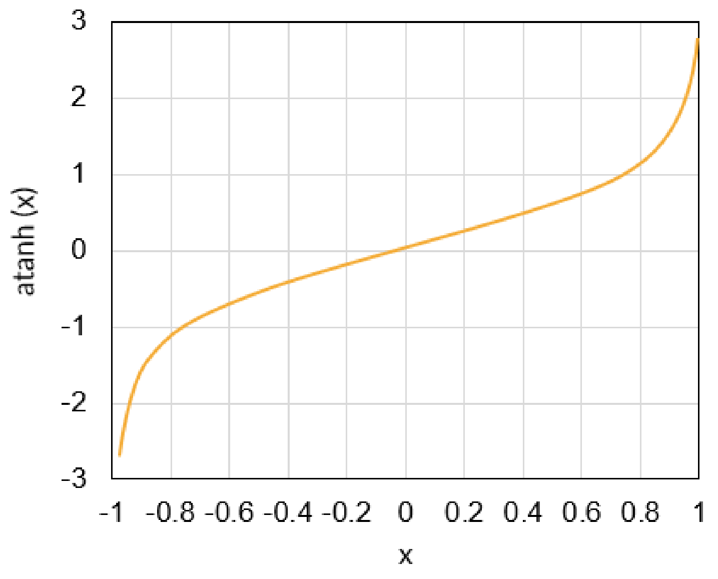
Following are the most popular activation functions used for neural networks:

- Logistic/Sigmoid function



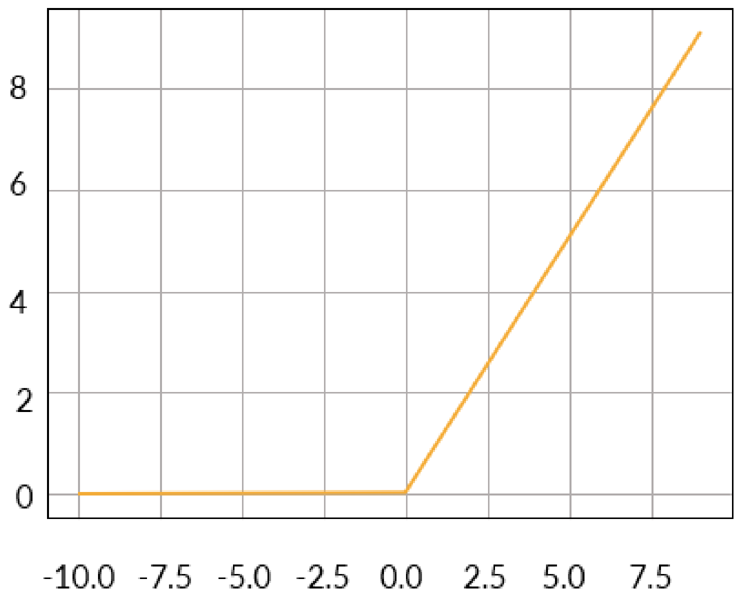
$$Output = f(x) = \frac{1}{1 + e^{-x}}$$

- Hyperbolic tangent function



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

- Rectified linear unit (ReLU)

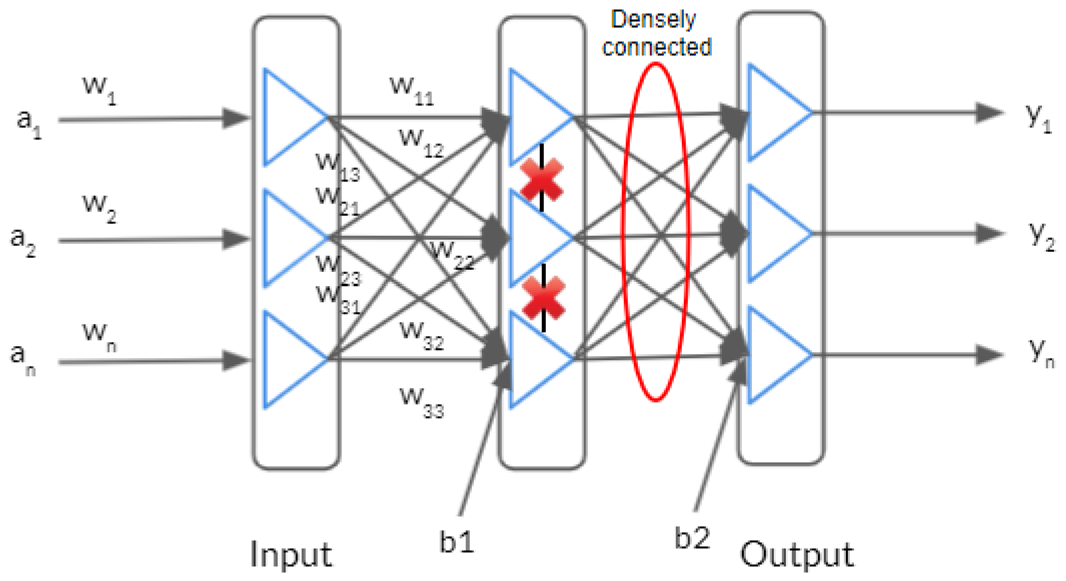


$$Output = x \; for \; x >= 0 \; and \; 0 \; otherwise$$

- There is another activation function called Leaky Relu, which is defined as follows: Output = x for x ≥ 0 and Output = α∗x otherwise

Assumptions for Simplifying Neural Networks

In order to simplify the complex structure of neural networks, you studied some assumptions. Some of these assumptions are shown in the following image.



To summarise, commonly used neural network architectures make the following simplifying assumptions:

1. The neurons in an ANN are arranged in layers, and these layers are arranged sequentially.
2. The neurons within the same layer do not interact with each other.
3. The inputs are fed to the network through the input layer, and the outputs are sent out from the output layer.
4. Neurons in consecutive layers are densely connected, i.e., all neurons in layer l are connected to all neurons in layer l+1.
5. Every neuron in the neural network has a bias associated with it, and each interconnection has a weight associated with it.
6. All neurons in all layers use the same activation function.

Parameters and Hyperparameters of Neural Networks

Neural networks are trained on weights and biases. The purpose of the training is to obtain optimum **weights and the biases,** which form the parameters of the network.

During training, the neural network learning algorithm fits various models to the training data and selects the best prediction model. The learning algorithm is trained with a fixed set of hyperparameters associated with the network structure:

- Number of layers

- Number of neurons in the input, hidden and output layers
- Learning rate
- Number of epochs, etc.

The notations used throughout the module are as follows:

1. *W* is for weight matrix
2. *b* stands for the bias
3. *x* stands for input
4. *y* is the ground truth label
5. *p* is the probability vector of the predicted output
6. *h* is the output of the hidden layers
7. The superscript stands for layer number. The weight matrix connecting the first layer to the second layer will be denoted as w^1
8. The subscript stands for the index of the individual neuron. The weight connecting the first neuron of the first layer to the third neuron of the second layer will be denoted as w^1_{13}

Next, you learnt when you should use deep learning algorithms:

1. First, if you have very large data sets, the performance of deep learning models will be better than that of preliminary machine learning models. Therefore, it is preferable to use deep learning.

2. In cases where the problems are too complex for machine learning algorithms, one can use deep learning to solve them. Deep learning algorithms can efficiently perform complex feature engineering tasks. For example, in the case of self-driving cars and image recognition.

3. Deep learning requires a large number of computational resources and expenses to drive hardware and software for training. If you have these resources available, you should go ahead and train your models using deep learning algorithms.

4. In the case of deep learning algorithms, accuracy is the priority. So, if small performance gains are also extremely critical for your model, you should consider using deep learning algorithms.

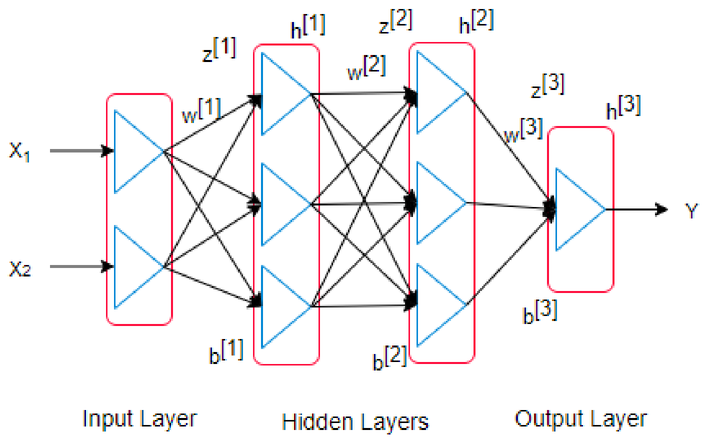
SUMMARY

Feedforward Neural Networks

In this session you learnt how the information is fed forward in a neural network. You also saw how the predicted output of the model is computed.

Flow of Information in Neural Networks

In this session you were introduced to a simple neural network with two hidden layers and an input and output layer, with each layer having a different number of neurons, weights and biases associated with them as shown in the image given below.



In artificial neural networks, the output from one layer is used as an input to the next layer, which is referred to as a **feedforward neural network**. This means the network contains no loops, that is, information is always fed forward and never fed back. The input, weights and biases of each layer are given below.

$$X = \begin{bmatrix} 1 \\ 1 \end{bmatrix},\ w^{[1]} = \begin{bmatrix} 1 & 0 \\ 3 & 2 \\ 1 & 2 \end{bmatrix},\ w^{[2]} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 3 \\ 1 & 2 & 1 \end{bmatrix},\ w^{[3]} = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix},\ b = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$$

Note: You can have different values of biases for each neuron in a layer but here you will be taking all the bias values to be the **same** throughout the **layer**.

The dimensions of weights and bias for any given neural network as follows:

1. The dimensions of weight matrices can be given by $[n^{[l]}, n^{[l-1]}]$.
2. The bias vector dimensions can be given by $[n^{[l]}, 1]$.

Here, 'l' denotes the layer number, and 'n' denotes the number of neurons in that layer.

Next, you also went through the feedforward equations, i.e., the equations for the z and h vectors. The cumulative input $z^{[l]}$ and the output of the hidden layer $h^{[l]}$ are calculated from the values of inputs, weights and biases using the formulas given below.

$$\text{Cumulative Input, } z^{[l]} = w^{[l]}.h^{[l-1]} + b^{[l]}$$

$$\text{Output of hidden layer(sigmoid), } h^{[l]} = 1/(1 + e^{-z[l]})$$

You also computed the output of this neural network; the steps involved in this are given below.

1. $z_1^{[1]} = w_{11}^{[1]}.h_1^{[0]} + w_{21}^{[1]}.h_2^{[0]} + b^{[1]} = 1*1 + 0*1 + 2 = 3$
   Where $h^{[0]}$ is the input vector x

2. $z_2^{[1]} = w_{12}^{[1]}.h_1^{[0]} + w_{22}^{[1]}.h_2^{[0]} + b^{[1]} = 3*1 + 2*1 + 2 = 7$

3. $z_3^{[1]} = w_{13}^{[1]}.h_1^{[0]} + w_{23}^{[1]}.h_2^{[0]} + b^{[1]} = 1*1 + 2*1 + 2 = 5$

Therefore, $z^{[1]} = \begin{bmatrix} 3 \\ 7 \\ 5 \end{bmatrix}$

1. $h_1^{[1]} = 1/(1 + e^{-z_1^{[1]}}) = 1/(1 + e^{-3}) = 0.95257$

2. $h_2^{[1]} = 1/(1 + e^{-z_2^{[1]}}) = 1/(1 + e^{-7}) = 0.99909$

3. $h_3^{[1]} = 1/(1 + e^{-z_3^{[1]}}) = 1/(1 + e^{-5}) = 0.99331$

Therefore, $h^{[1]} = \begin{bmatrix} 0.95257 \\ 0.99909 \\ 0.99331 \end{bmatrix}$

Similarly, you computed $z^{[2]}$, $h^{[2]}$, $z^{[3]}$ and $h^{[3]}$ as given below.

$$z^{[2]} = w^{[2]}.h^{[1]} + b^{[2]} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 3 \\ 1 & 2 & 1 \end{bmatrix} . \begin{bmatrix} 0.95257 \\ 0.99909 \\ 0.99331 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2.94588 \\ 4.97902 \\ 4.94406 \end{bmatrix}$$

$$h^{[2]} = \begin{bmatrix} \frac{1}{\left(1+e^{-2.94588}\right)} \\ \frac{1}{\left(1+e^{-4.97902}\right)} \\ \frac{1}{\left(1+e^{-4.94406}\right)} \end{bmatrix} = \begin{bmatrix} 0.95007 \\ 0.99317 \\ 0.99292 \end{bmatrix}$$

$$z^{[3]} = w^{[3]}.h^{[2]} + b^{[3]} = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 0.95007 \\ 0.99317 \\ 0.99292 \end{bmatrix} + \begin{bmatrix} 1 \end{bmatrix} = 2.94299$$

$$y = h^{[3]} = \frac{1}{1+e^{-2.94299}} = 0.94993$$

As calculated above, the **feedforward output** or the **predicted output** of the model is 0.94993.

FeedForward Algorithm

In this segment, you understood and wrote the pseudocode for the feedforward algorithm. For doing so you considered a large neural network with ‘n’ inputs, ‘k’ outputs, ‘L’ hidden layers, and the sigmoid activation function for all the hidden layers, except the output layer. The output layer activation function will be the **softmax function**.

Previously, you have used the sigmoid activation function for your neural network, as it was a binary classification problem. Now, you will consider a multiclass classification problem, and the softmax function is the best choice for such a problem because the output is in terms of probabilities of the occurrence of each class; this allows you to compare the results of all the classes with each other.

The pseudocode of the feedforward algorithm can be written as follows:

```
1. $h^{[0]} = x$
2. for l in [0,1,2...L]:
       a. $h^{[l]} = \sigma(w^{[l]}.h^{[l-1]} + b^{[l]})$
3. $p = f(h^{[l]})$
```

Since you will be building your neural network using the MNIST data set, which is a multiclass classification problem, the output activation function will be the softmax function. The softmax output $p = f(h^{[l]})$, as you learnt in the previous session, looks like this.

$$p_{ij} = \frac{e^{w_j h^L}}{\sum_{t=1}^{k} e^{w_t h^L}}$$

Where, k is the number of classes and j = [1,2…,k]. Here, note that the calculation of p_{ij} is known as normalising the p_i vector.

Therefore, your pseudocode can be written as follows:

```
1. $h^{[0]} = x$
2. for l in [0,1,2...L]:
     a. $h^{[l]} = \sigma(w^{[l]}.h^{[l-1]} + b^{[l]})$
3. $p_i = e^{w[o]h[l]}$
4. $p_i$ = normalize($p_i$)
```

Where, w^o (refers to the weights of the output layer) can also be written as w^{L+1}.

Vectorised FeedForward Algorithm

In this segment, you saw the working of feedforward in batches as the training data includes multiple data points, and you need to perform feedforward computation for all of them. For a very large data set, doing one point at a time using a loop might be very time-consuming. Therefore, it requires a vectorised approach to perform feedforward for multiple points at once.

Vectorised implementation means performing the computation (here, feedforward) for multiple data points using matrices. This will be much quicker than working with one data point at a time. A way to achieve this would be to write a 'for loop' iterating through all the data points.

For this, you reconsidered the neural network with ‘n’ inputs, ‘k’ outputs, ‘L’ hidden layers, and the sigmoid activation function for all the hidden layers, except the output layer. The output layer activation function will be the softmax function. The feedforward pseudocode for a set of m data points using a 'for loop' can be written as follows.

```
1. for i in [1,2,...m]:
     a. $h^{[0]} = x$
     b. for l in [0,1,2...L]:
          i. $h^{[l]} = \sigma(w^{[l]}.h^{[l-1]} + b^{[l]})$
     c. $p_i = f(h^{[L]})$
```

You will notice that you require two nested 'for loops'. This will become computationally quite expensive if you have a large data set (which is often the case with neural networks). There is a much more efficient way to do so using matrices. Let us understand it in more depth by considering a batch B to show a vectorised implementation that consists of m data points stacked side by side, which is represented as follows.

$$B = \begin{bmatrix} | & | & | & | & | \\ x_i & x_{i+1} & . & . & x_{i+m-1} \\ | & | & | & | & | \end{bmatrix}$$

All data points, x_i, x_{i+1}, etc., are d-dimensional vectors, i.e., every input vector has d numerical features. Each data point is a **column vector** in matrix B. Thus, the feedforward algorithm for batch B is as follows.

```
1. $H^{[0]} = B$
2. for l in [0,1,2...L]:
     a. $H^{[l]} = \sigma(W^{[l]}.H^{[l-1]} + b^{[l]})$
3. $P = normalize(exp(W^{[o]}.H^{[L]} + b^{[o]}))$
```

SUMMARY

Backpropagation in Neural Networks

In this session, you learnt about loss functions and how to optimise the model by minimising the loss through backpropagation. You also learnt about regularisation techniques.

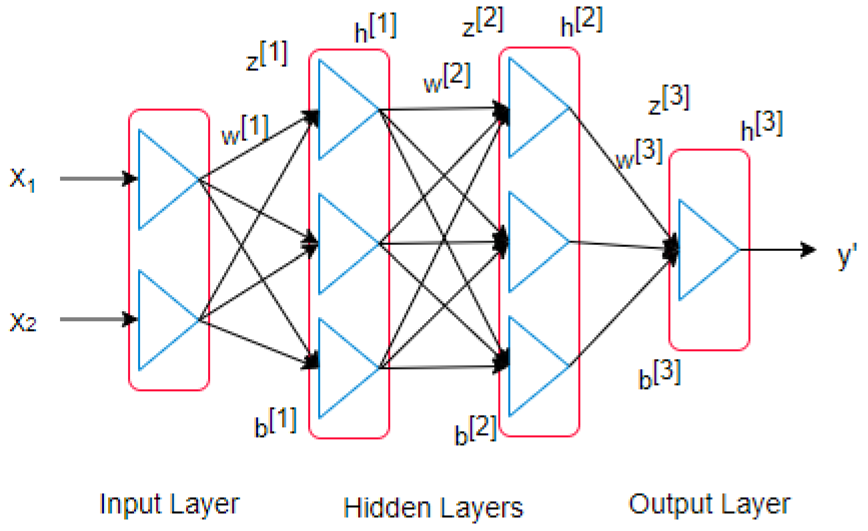
Training a Neural Network

In this segment, you learnt how to train a neural network. The aim of **training a model** is to **optimise the model parameters** in order to **minimise the error**. This error is a function of the predicted and actual outputs of the model. It is given by the following equation.

```
Error / Loss = f(Predicted output, Actual output)
```

In the case of neural networks, the training task involves computing the optimal **weights** and **biases** by minimising some cost function.

In order to compute loss, you used the same neural network that you saw in the previous session. This model consists of an input layer with two neurons, two hidden layers with three neurons each, and an output layer with a single neuron, as shown below.



You have already computed the predicted output of the model to be 0.94993. If the actual output of the model is 1, then you can compute the error in the regression model by using the following mean squared error formula.

$$MSE = 1/n\ \Sigma^{n}_{i=1}\ (y'-y)^{2}$$

Where,
n is the number of data points in the output,
y is the actual output, and
y’ is the predicted output.

For this example, you know that n = 1, y’ = 0.94993, and y = 1.
Therefore, after adding the values, you will get the following:

$$MSE = (0.94993-1)^{2} = 0.00251$$

You also went through some of the commonly used loss functions as given below:

- Classification problems:
  - Binary cross-entropy loss: $-y\ \log(y') - (1 - y)\ \log(1 - y')$
  - Multi-class cross-entropy Loss: $-\ \Sigma^{n}_{i=1}\ y\ \log(y')$
- Regression problems:
  - Mean squared error: $1/n\ \Sigma^{n}_{i=1}\ (y'-y)^{2}$
  - Mean squared error: $1/n\ \Sigma^{n}_{i=1}\ |y'-y|$

An important point to note here is that if the volume of data is large (which is often the case), the loss calculation can become quite complicated. For example, if you have a million data points, they will have to be fed into the network (in batch), the output will be calculated using feedforward, and then the loss L_i (for the ith data point) will be calculated. The total loss is the sum of the losses of all the individual data points and is also referred to as the cost function.

Hence, Total loss = L1 + L2 + L3+...................................+L1000000

The total loss L is a function of w's and b's. Once the total loss is computed, the weights and biases are updated (in the direction of decreasing loss). In other words, L is minimised with respect to the w's and b's. This can also be written in the form of an algorithm as follows:

1. Output, $y_i' = h^{l+1}\ (w,b,x_i)$
2. Loss $= L\ (h^{l+1}\ (w,b,x_i),\ y_i) = L(y_i',\ y_i)$
3. Total cost $= -\Sigma^{n}_{i=1} L(y_i',\ y_i)$
4. Training problem: Minimise the overall cost/loss of the model $= \min_{(w,b)}\ -\Sigma^{n}_{i=1} L(y_i',\ y_i)$

Gradient Descent Algorithm

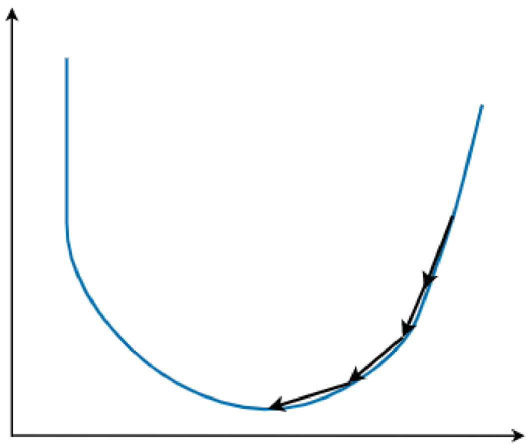
In this segment, you learnt how to minimise the loss function using the **gradient descent** algorithm. As you learnt in the previous segment, training refers to the task of finding the optimal combination of weights and biases to minimise the total loss (with a fixed set of hyperparameters).

Gradient descent is an optimisation algorithm used to find the minimum value of a function. The basic idea is to use the gradient of the function to find the direction of the steepest descent, i.e., the direction in which the value of the function decreases most rapidly, and move towards the minima iteratively according to the following rule:

$$w_{new} = w_{old} - \alpha . \partial L / \partial w$$

$$b_{new} = b_{old} - \alpha . \partial L / \partial b$$

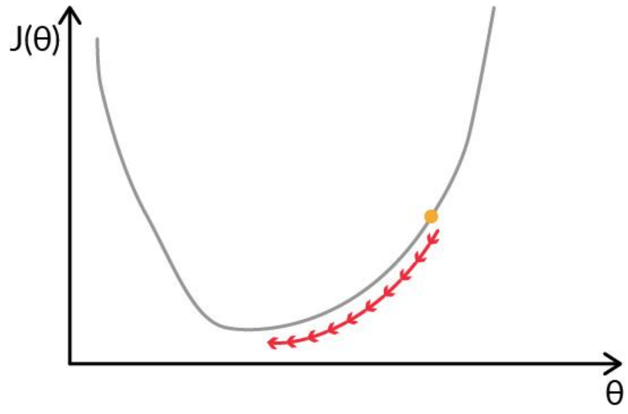
With the gradient descent algorithm, your model will keep on updating the weights and biases iteratively until the **point of convergence** is reached. The point of convergence is a point where your parameters will obtain optimal value. Another interesting thing to notice here is that while trying to optimise the parameters using the gradient descent algorithm, you are trying to reach the minima as shown in the image below.



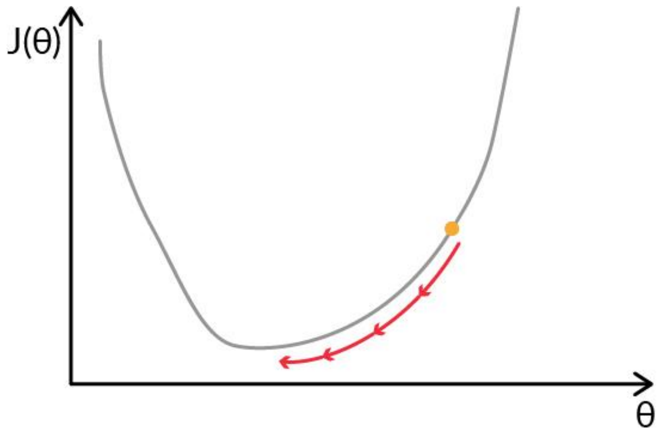
This is done because the aim here is to minimise the overall loss/cost of the model by updating the values of parameters (weights and biases). The $-\alpha$ term, which is being multiplied by the gradient or $\partial L/\partial b$, ensures that the iterations move towards the negative value until the point of convergence is reached.

The **learning rate** α is a hyperparameter that can take any value as set by the user while writing the code. This hyperparameter defines how fast the values move towards the minima of the curve. The values of the learning rate usually lie between 0 and 1. This is because very small or very large values might not work to your advantage. Let’s try to understand why this happens with the following points:

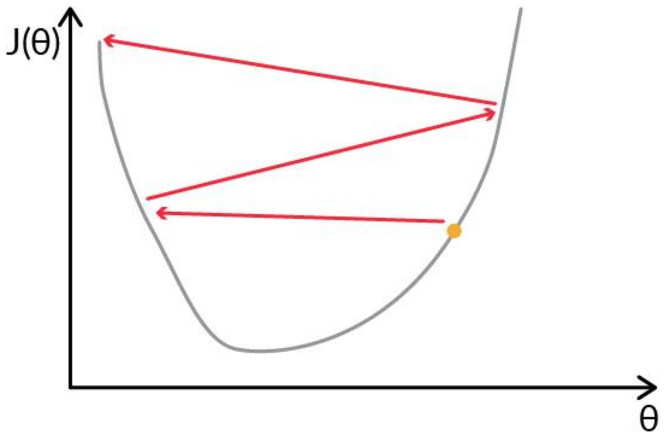
1. If the learning rate is too small, it will take a very long time to reach the minima, as each step size is very small as shown below.



2. If the learning rate is optimal, your function will take certain steps that are neither too small nor too large and reach the point of minima.



3. If the learning rate is very large, your function will take very large jumps and might never reach the point of minima.



Backpropagation Algorithm

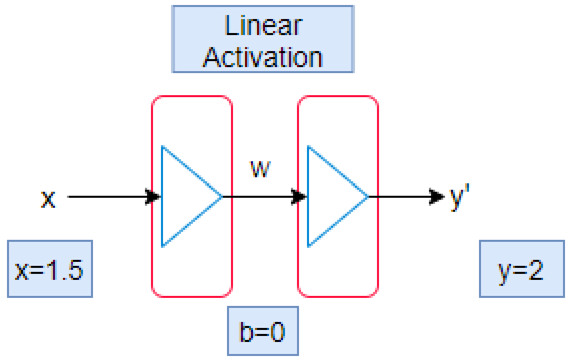
In this segment, you learnt about the **backpropagation algorithm** in **neural networks** and also solved the implementation of the algorithm using simple networks.

Backpropagation is one of the most important steps/algorithms involved in the process of building and training neural network models. It was initially derived by researchers in the early 1960s and was enhanced and familiarised as an important concept in neural networks in 1986.

Backpropagation, as the name suggests, refers to the **backwards propagation of errors** by **updating** the values of **model parameters** to **minimise the error** and in turn **optimise the model.**

The first network that you studied consisted of a network with an input layer and an output layer with one neuron each. The input and the actual output are given. For the sake of simplicity, you will take bias as zero and activation as linear activation for this example.

The neural network and the corresponding values are given below.



Let's start by calculating the value of the cumulative input as follows:

$$z = w.x+b = 1.5w + 0 = 1.5w$$

The output of the model: $y' = f(z) = 1.5w$ (as it is the linear activation function)

$$\text{Loss} = f(y', y)$$

$$\text{MSE Loss} = (y'-y)^2 = (1.5w-2)^2$$

According to the gradient descent algorithm, $w_{new} = w_{old} - \alpha.\partial L/\partial w$

Using the chain rule, $\partial L/\partial w = (\partial L/\partial y').(\partial y'/\partial w)$

1. $\partial L/\partial y' = \partial(y'-y)^2/\partial y' = 2(y'-y)$
2. $\partial y'/\partial w = \partial(1.5w)/\partial w = 1.5$

Therefore, $\partial L/\partial w = 2(y'-y)*1.5 = 3(y'-y) = 3(1.5w-2) = 4.5w-6$

In neural networks, the weights are initialised to random values, and during backpropagation, these values are updated to the optimised values. If the initial weight, w = 0.3 and α = 0.1, the predicted output of the model will be as follows:

$$z = w.x+b = 1.5w = 1.5*0.3 = 0.45$$
$$y' = f(z) = 0.45$$

The loss of this model will be as follows:

$$\text{MSE loss} = (y'-y)^2 = (1.5w-2)^2 = (0.45-2)^2 = 2.4025$$

The updated weight can be calculated as follows:

$$w_{new} = w_{old} - \alpha.\partial L/\partial w$$
$$w_{new} = w_{old} - \alpha(4.5w-6) = 0.3 - 0.1\ (4.5*0.3-6)$$
$$w_{new} = 0.3-0.1(-4.65) = 0.765$$

The predicted output of the model can be calculated as follows:

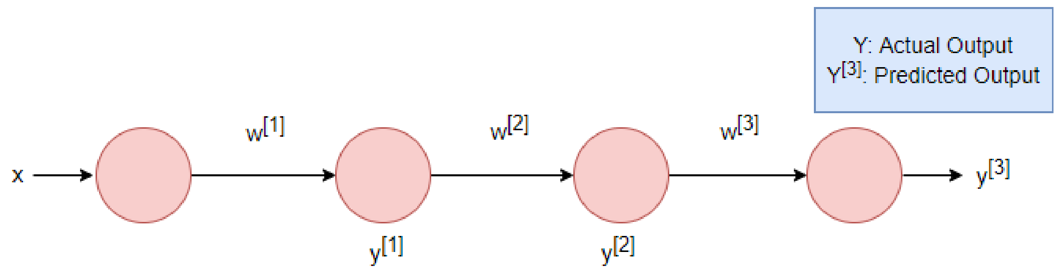
$$y' = 1.5w = 1.5*0.765 = 1.1475$$

The new loss of this model can be calculated as follows:

$$\text{MSE loss} = (y'-y)^2 = (1.5w-2)^2 = (1.1475-2)^2 = 0.7267$$

As you can see there is a significant decrease in the loss of the model. Similarly, you can perform more iterations until the point of convergence is reached.

You solved another example of a simple neural network. For this example, you will take a simple neural network with an input layer, an output layer and two hidden layers. Also, the network will have zero bias and linear activation for simplicity. Take a look at the network given below.



Let’s start solving the problem step by step as follows:

1. Feedforward propagation:
    a. $y^{[1]} = w^{[1]}.x$
    b. $y^{[2]} = w^{[2]}.y^{[1]}$
    c. $y^{[3]} = w^{[3]}.y^{[2]}$

2. Loss computation:
    a. MSE loss = $(y^{[3]} - y)^2$

3. Backpropagation:
    a. $\partial L/\partial w^{[3]} = (\partial L/\partial y^{[3]}).(\partial y^{[3]}/\partial w^{[3]})$
    b. $\partial L/\partial w^{[2]} = (\partial L/\partial y^{[3]}).(\partial y^{[3]}/\partial y^{[2]}).(\partial y^{[2]}/\partial w^{[2]})$
    c. $\partial L/\partial w^{[1]} = (\partial L/\partial y^{[3]}).(\partial y^{[3]}/\partial y^{[2]}).(\partial y^{[2]}/\partial y^{[1]}).(\partial y^{[1]}/\partial w^{[1]})$

4. These values of $\partial L/\partial w^{[1]}, \partial L/\partial w^{[2]}$ and $\partial L/\partial w^{[3]}$ can then be used to update the weights $w^{[1]}$, $w^{[2]}$ and $w^{[3]}$ by using the gradient descent algorithm:
    $w_{new} = w_{old} - \alpha.\partial L/\partial w$

Let’s update $w^{[1]}$ by substituting the values that you derived above as follows:

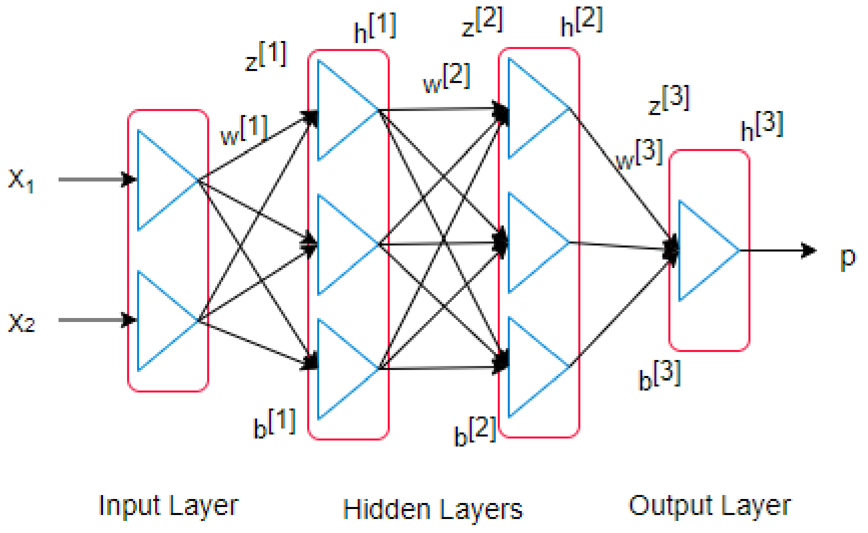
- $w^{[1]}_{new} = w^{[1]}_{old} - \alpha.\partial L/\partial w^{[1]}$
- $\partial L/\partial w^{[1]} = (\partial L/\partial y^{[3]}).(\partial y^{[3]}/\partial y^{[2]}).(\partial y^{[2]}/\partial y^{[1]}).(\partial y^{[1]}/\partial w^{[1]})$
    - $\partial L/\partial y^{[3]} = \partial(y^{[3]} - y)^2/\partial y^{[3]} = 2(y^{[3]} - y)$
    - $\partial y^{[3]}/\partial y^{[2]} = \partial(w^{[3]}.y^{[2]})/\partial y^{[2]} = w^{[3]}$
    - $\partial y^{[2]}/\partial y^{[1]} = \partial(w^{[2]}.y^{[1]})/\partial y^{[1]} = w^{[2]}$
    - $\partial y^{[1]}/\partial w^{[1]} = \partial(w^{[1]}.x)/\partial W^{[1]} = x$
- Therefore, $\partial L/\partial w^{[1]} = 2(y^{[3]} - y) * w^{[3]} * w^{[2]} * x$
- Hence, $w^{[1]}_{new} = w^{[1]}_{old} - \alpha.(2(y^{[3]} - y) * w^{[3]} * w^{[2]} * x)$

Similarly, you also solved for weight matrices for the other two layers as well.

Updating Weights and Biases

In this segment, you learnt how to backpropagate errors to update the model parameters, i.e., weights and biases in a neural network with multiple layers and multiple neurons in each layer.

For this, you reconsidered the same neural network that we have been using so far, as shown below.



This model consists of an input layer with 2 neurons, 2 hidden layers with 3 neurons each, and an output layer with a single neuron. For this derivation, you must consider the activation function to be Sigmoid for all the layers. The weights and biases for this derivation are mentioned below.

$$W^{[1]} = \begin{bmatrix} w_{11}^{[1]} & w_{21}^{[1]} \\ w_{12}^{[1]} & w_{22}^{[1]} \\ w_{13}^{[1]} & w_{23}^{[1]} \end{bmatrix} \quad W^{[2]} = \begin{bmatrix} w_{11}^{[2]} & w_{21}^{[2]} & w_{31}^{[2]} \\ w_{12}^{[2]} & w_{22}^{[2]} & w_{32}^{[2]} \\ w_{13}^{[2]} & w_{23}^{[2]} & w_{33}^{[2]} \end{bmatrix} \quad b = \begin{bmatrix} b^{[1]} \\ b^{[2]} \\ b^{[3]} \end{bmatrix}$$

$$W^{[3]} = \begin{bmatrix} w_{11}^{[3]} & w_{21}^{[3]} & w_{31}^{[3]} \end{bmatrix}$$

The feedforward equations for this example, as derived in the previous session, are as follows:

$$\text{Cumulative Input: } z^{[l]} = w^{[l]}.h^{[l-1]} + b^{[l]}$$
$$\text{Output of the hidden layer (sigmoid): } h^{[l]} = 1/(1 + e^{-z[l]})$$

In this derivation, you will solve a classification problem using binary cross-entropy loss, where this model will classify whether or not the input feature belongs to a certain class. The loss of this model can be written as follows:

$$L = -(y\log(p) + (1-y)\log(1-p))$$

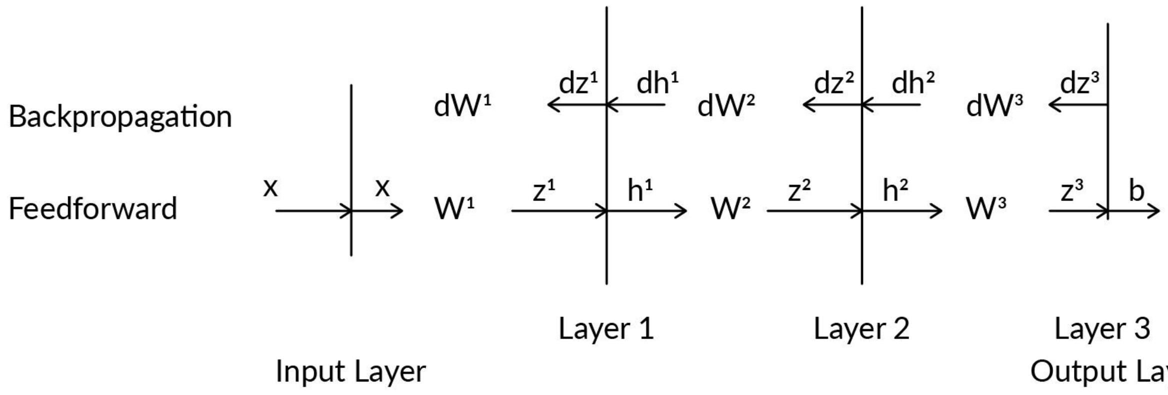
Now that you are well-equipped with the knowledge of the loss function, let's proceed further. The updated weight can be calculated as follows:

$$w_{new} = w_{old} - \alpha . \partial L/\partial w$$

Using the above matrices, vectors and formulas, you derived the equations for backpropagation and got the following results:

1. $dz^{[3]} = q - y = p - y$
2. $dW^{[3]} = dz^{[3]} . (h^{[2]})^T$
3. $dh^{[2]} = (W^{[3]})^T . dz^{[3]}$
4. $dz^{[2]} = dh^{[2]} \otimes \sigma'(z^{[2]})$
5. $dW^{[2]} = dz^{[2]} . (h^{[1]})^T$
6. $dh^{[1]} = (W^{[2]})^T . dz^{[2]}$
7. $dz^{[1]} = dh^{[1]} \otimes \sigma'(z^{[1]})$
8. $dW^{[1]} = dz^{[1]} . (x)^T$

The path followed in the derivation can be summarised using the following image.



Finally, you also wrote the pseudocode for the algorithm as given below:

```
1. $h^0 = x$

2. for l in [1,2,.......,L]:
    a. $h^l = \sigma(W^l.h^{l-1} + b^l)$

3. $p = normalize(exp(W^o.h^l + b^o))$

4. $L = -y^T.log(p)$

5. $dz^o = p-y$

6. $dW^o = dz^o.(h^2)^T$

7. for l in [L,L-1,........1]:
    a. $dh^l = (W^{l+1})^T.dz^{l+1}$
    b. $dz^l = dh^l \otimes.\sigma'(z^l)$
    c. $dW^l = dz^l.(h^{l-1})^T$
```

Batch in Backpropagation

In this segment, you learnt how to modify the feedforward and backpropagation algorithms such that you can work with **batches of multiple data points.** In a way, the batch acts as a proxy for the whole data set. Therefore, for a batch size of m, the **average loss** will be as follows:

$$1/m * \sum_{x \in B} L(N(x),y)$$

This is the average loss over m data points of the batch. N(x) is the network output (vector p). Let's denote a batch of input data points by matrix B. The backpropagation algorithm for a batch is as follows:

```
    1. $H^0 = B$
    2. For l in [1,2,........,L]:
        1. $H^l = \sigma(W^l.H^{l-1} + b^l)$
    3. $P = normalize(exp(W^o.H^l + b^o))$
    4. $L = -1/m * Y^T.log(P)$
    5. $DZ^{L+1} = P-Y$
    6. $DW^{L+1} = DZ^{L+1}.(H^L)^T$
    7. For l in [L,L-1,........1]:
        1. $dH^l = (W^l+1)^T.DZ^{l+1}$
        2. $DZ^l = dH^l \otimes \sigma'(Z^l)$
        3. $DW^l = DZ^l.(H^l-1)^T$
        4. $Db^l = DZ^l$
```

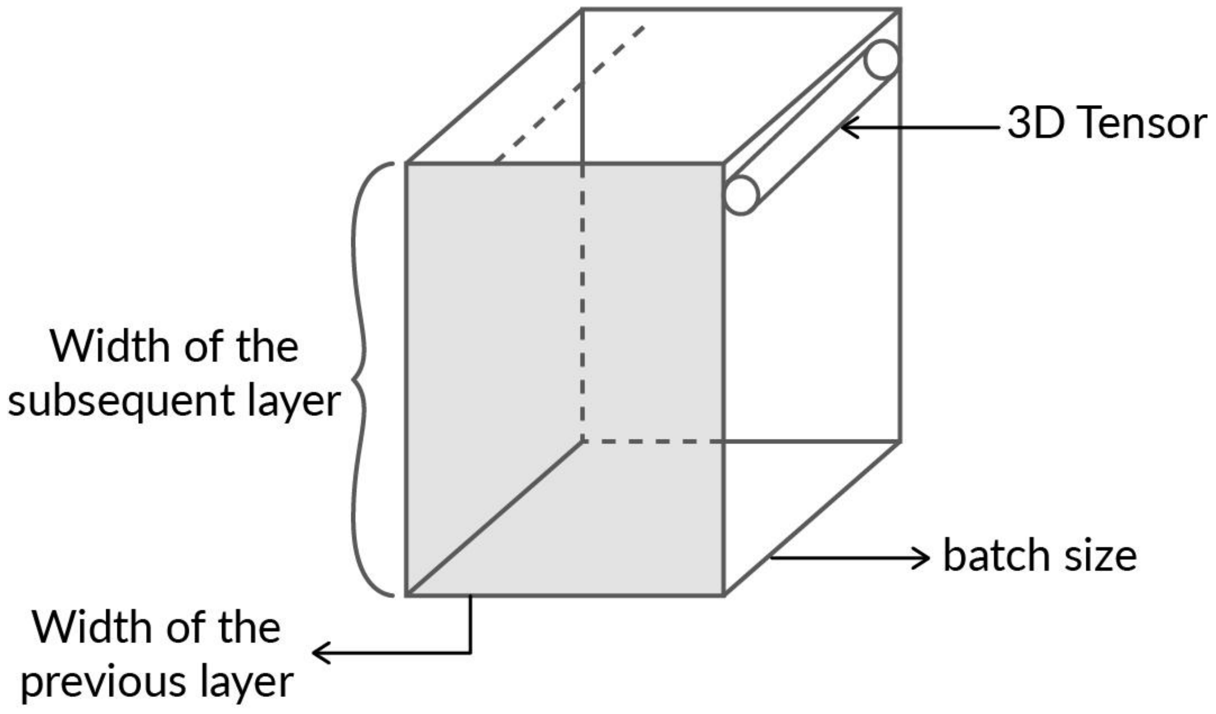
Note that, earlier, dz^l represented a vector. However, DZ^l now represents the matrix consisting of the dz^l vectors of all the data points in B (each dz^l representing a column of DZ^l). Similarly, dH^l, Y, P and H^l are all matrices of the corresponding individual vectors stacked side by side.

An interesting point to note here is that DW^l is a tensor, whereas Db^l is a matrix (try to think why this is so). This is something we do not want because in the update step, when we write the following:

$$W^l new = W^l old - \alpha . \partial L / \partial W^l$$

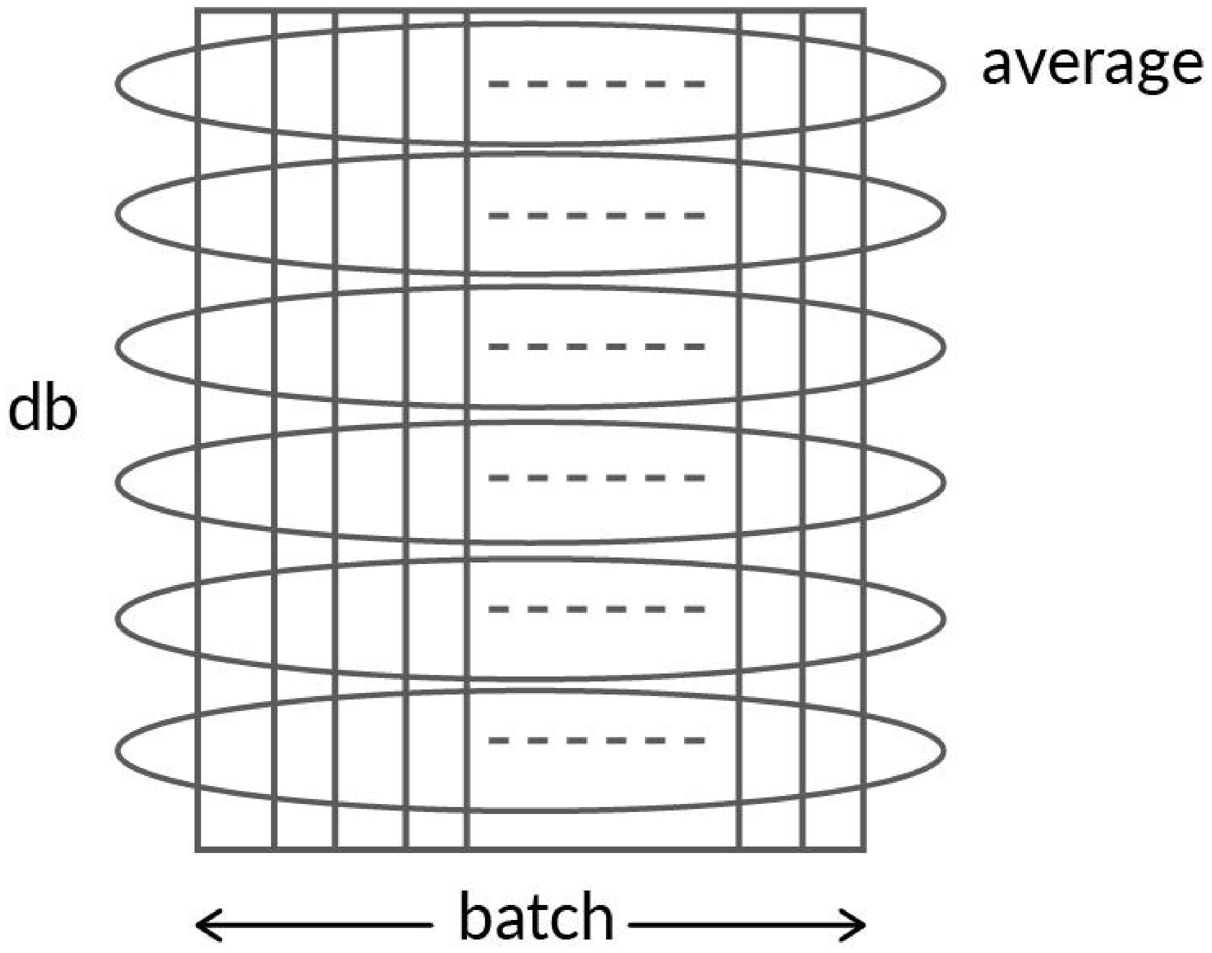
$$b^l new = b^l old - \alpha . \partial L / \partial b^l$$

Then, we want the shapes of $\partial L/\partial W^l$(currently a tensor DW^l) and W^l(a matrix) to be the same. Similarly, we want the shapes of $\partial L/\partial b^l$(currently a matrix Db^l) to be the same as that of b^l(a vector). From the image given below, we can see that the dot product $DW^l = DZ^l . (H^{l-1})^T$ is basically a **tensor**. This can be visualised as shown in the image below.



Each 'layer' along the 'batch size' dimension of this matrix contains the dW^l matrix for each data point. Remember that the loss function that we are minimising here is the average loss of data points in B, i.e., $1/m * \sum_{x \in B} L(p,y)$. Hence, we take the average of all the dW^l matrices across the batch to obtain the final weight gradient matrix for the update step. In other words, we collapse the tensor along the batch dimension using the average.

Similarly, Db^1 is a matrix, as shown in the following image, where each column corresponds to the vector db^1 for one data point in the batch.



This matrix consists of the gradient vectors db^1 of each data point in the batch stacked side by side. Since we need to minimise the average loss of the batch, we take the average of these 'bias gradient vectors' to finally create a single vector.

Gradient Descent Variants

In this segment, you learnt about the different variants of the gradient descent algorithm. Some of the important points regarding these three variants can be summarised as follows:

1. Batch gradient descent:
    a. It is also referred to as ‘vanilla gradient descent’.

b. This algorithm takes into account the entire training data set and calculates the gradient for all the points in it before updating the weights and biases.

c. Due to this, the number of points required in memory becomes very large.

d. Also, the time required for calculating the gradient for the entire training set increases significantly.

2. Stochastic gradient descent:
    a. This algorithm calculates the gradient of each training example (x,y) one by one and simultaneously updates the weights and biases.

    b. It requires much less memory as compared to batch gradient descent. This is because only a certain training set is required in the memory at a particular point in time.

    c. Since updates are made after calculating the gradient of each training set, it requires frequent updates.

3. Mini-batch gradient descent:
    a. This algorithm divides the training set into batches and calculates the gradient of all the points in that particular batch before updating the weights and biases.

    b. Since the number of points in a batch is less than that in the entire training set, it requires less memory than the batch gradient descent algorithm but more than that required by the stochastic gradient descent algorithm.

    c. This is a widely used method, as it strikes a balance between the previous two methods, which means that you are neither required to go through the entire training set to make a single update nor do you need to make frequent updates.

The algorithm for gradient descent can be given as follows:

```
1. Batch size = m
2. For each epoch in [1,2,....,L]:
    a. Reshuffle the data set
    b. Number of batch = n/m
    c. For each batch in [1,2,...., number of batch]:
        i.  Compute the gradient for each input in the batch
            [$batch_{i-1}$ * m, $batch_i$* m]
    d. Average gradient $\nabla W$ = Sum of gradient $\nabla W$ / m
    e. Average gradient $\nabla b$ = Sum of gradient $\nabla b$ / m
    f. W = W - $\nabla W$, b = b - $\nabla b$
```

This algorithm can be used for any of the variants based on the value of n (total number of datapoints) in the following ways:

1. If m=1, the number of batches = n/m will be equal to n, i.e., the number of batches will be equal to the number of data points. Therefore, the algorithm used will be stochastic gradient descent.

2. If m=n, the number of batches = n/m will be equal to 1. Therefore, the algorithm used will be batch gradient descent.

3. If 1<m<n, the number of batches = n/m will lie between 1 and n. Therefore, the algorithm used will be mini-batch gradient descent.

Regularization Techniques

In this segment, you learnt about different regularisation techniques. Neural networks are usually large, complex models with tens of thousands of parameters; thus, they have a tendency to **overfit** the training data, usually when the amount of training data is less. As with many other ML models, **regularisation** is a common technique used in neural networks to address this problem.

One of the common regularisation techniques is to add a **regularisation term** to the objective function. This term ensures that the model does not capture the 'noise' in the data set, i.e., it does not overfit the training data. This, in turn, leads to a better **generalisability** of the model.

Whenever you build and train any ML or DL model, the aim is to have a model that can accurately predict results on unseen data. Additionally, you would also want your model to have minimal error. A model trained on the training data set tends to adapt to the data set, leading to overfitting, which means that the model is performing better on the training data set; however, it might not get good results on unseen data, i.e., the test data. This leads to a **lack of generalisability** of the model. However, the training error of the model for such a case will be low and vice versa.

Therefore, your model needs to strike a balance between controlling the training error and having good generalisability, which can be done using **regularisation techniques.** There are different types of regularisation techniques, and the one discussed above can be represented in a general form as follows:

```
Objective function = Loss function (Error term) + Regularization term
```

When you increase the regularisation term, it means you are giving more value to the regularisation terms, which, in turn, leads to reducing the weight values (as the regularisation term is a function of weights) as you need to keep the overall loss low. This ensures that the weights do not blow out of proportion. Also, biases are not included in the regularisation term as it turns out that regularising the function over the bias term is counterproductive.

You must, thus, keep in mind the following two important terms:

1. **Bias:**
   a. The bias of a model represents the amount of error that the model will commit on a given training data set, which essentially is a measure of the assumptions made by the model about the trends in the training data.

   b. If the assumptions made are a lot, take, for example, linear regression, the bias is high as the training error is high.

      1. As you include more variables in the linear regression model, some 2nd or 3rd order variables, you are reducing the assumptions that you are making about the model; hence, the training error decreases and the bias decreases.

      2. But there is always a chance for this new linear regression model to overfit and have a high variance.

   c. The same goes for deep learning models; however, the number of assumptions in the case of deep learning models is much less than that in linear regression.

   d. Bias quantifies **how accurate** the model is likely to be on the **future/ test data**.

2. **Variance:**
   a. The variance of the model measures to what extent the model changes when trained on a different data set.

   b. So, if the variation is very high in the model parameters when you train the model on different subsets of training data, it means that the model has high variance.

   c. In other words, the 'variance' of a model is the variance in its output on the same test data with respect to the changes in the training data. It is the degree of changes in the model itself with respect to changes in training data.

Hence, as you increase the regularisation parameter, you are forcing the weight values to reach close to zero, making the model simpler, essentially reducing the variance. Regularisation by making the model simple ensures that the model does not learn very minute patterns like noise, which, in turn, causes the bias to increase.

Next, you will revisit the concepts of L1 and L2 norm for which the objective function can be written as follows:

$$\text{Objective function} = L(F(xi),\theta) + \lambda f(\theta)$$

Where, L(F(xi),θ) is the loss function expressed in terms of the model output F(xi) and the model parameters θ. The second term λf(θ) has two components - the **regularisation parameter** λ and the **parameter norm** f(θ).

Broadly, two types of regularisation techniques are followed in neural networks:

1. L1 norm: $\lambda f(\theta) = ||\theta||_1$ is the sum of all the model parameters
2. L2 norm: $\lambda f(\theta) = ||\theta||_2$ is the sum of squares of all the model parameters

The 'parameter norm' regularisation discussed earlier is similar to what you studied in linear regression in almost every aspect. As in **lasso regression** (L1 norm), we get a **sparse weight matrix**, which is not the case with the L2 norm. Despite this fact, the L2 norm is more common because the sum of the squares term is easily **differentiable** at all points, which comes in handy during backpropagation.

Dropouts

In this segment, you learnt about another popularly used regularisation technique called **dropouts**, specifically for neural networks.

To summarise, the dropout operation is performed by multiplying the weight matrix W^1 with an α **mask vector** in the following manner:

$$W^1.\alpha$$

For example, let's consider the following weight matrix of the first layer.

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \\ w^1_{41} & w^1_{42} & w^1_{43} \end{bmatrix}$$

The shape of the vector α will be (3,1).

Now, if the value of q (probability of 1) is 0.66, then vector α will have two 1s and one 0. Hence, vector α can be any of the following three:

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

One of these vectors is then chosen randomly **in each mini-batch.** Let's suppose in a mini-batch, the mask $\alpha = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ is chosen. Therefore, the new (regularised) weight matrix will be as follows:

$$\begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \\ w^1_{41} & w^1_{42} & w^1_{43} \end{bmatrix} . \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & w^1_{13} \\ 0 & 0 & w^1_{23} \\ 0 & 0 & w^1_{33} \\ 0 & 0 & w^1_{43} \end{bmatrix}$$

As you saw above, all the elements in the last column become zero.

Some important points to note regarding dropouts:

- Dropouts can be applied only to some layers of the network. (This is a common practice; you can choose some layers arbitrarily to apply dropouts to.)
- Mask α is generated independently for each layer during feedforward, and the same mask is used in backpropagation.
- The mask changes with each mini-batch/iteration and is randomly generated in each iteration (sampled from a Bernoulli with some p(1)=q).

Dropouts help in **symmetry breaking**. There is every possibility of the creation of communities within neurons, which restricts them from learning independently. Hence, by setting some random set of the weights to zero in every iteration, this community/symmetry can be broken.

Note: A different mini-batch is processed in every iteration in an epoch, and dropouts are applied to each mini-batch.

Batch Normalization

The term '**batch normalisation**', as the name suggests, normalises the output of each batch. This can be done in the following way:

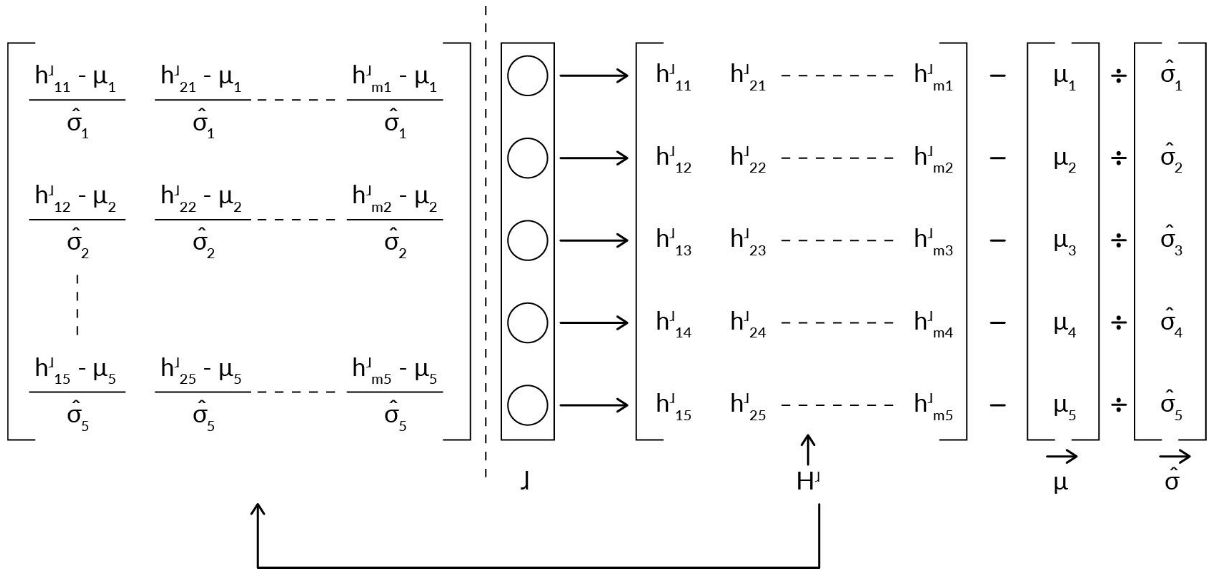
1. Computing the batch mean: $\mu_B \leftarrow 1/m \ \Sigma_{i=1}^{m} x_i$
2. Mini-batch variance: $\sigma_B^2 \leftarrow 1/m \ \Sigma_{i=1}^{m} (x_i - \mu_B)$

3. Normalisation: $x_i = (x_i - \mu_B) / (\sigma_B^2 + \varepsilon)^{½}$

4. Scale and shift: $y_i = (\gamma . x_i + \beta)$

Batch normalisation is performed on the output of the layers of each batch, H^l. It essentially involves normalising the matrix H^l across all data points in the batch. Each vector in H^l is normalised by the mean vector μ and the standard deviation vector ^σ computed across a batch.

The image given below shows the batch normalisation process for a layer l. Each column in matrix H^l represents the output vector of layer l, H^l, for each of the m data points in the batch. We need to compute vectors μ and ^σ, which represent the mean output from layer l across all points in the batch. We then need to normalise each column of the matrix H^l using μ and ^σ as shown below:



$\frac{h^l_{11} - \mu_1}{\hat\sigma_1}$ $\frac{h^l_{21} - \mu_1}{\hat\sigma_1}$ $\frac{h^l_{m1} - \mu_1}{\hat\sigma_1}$
$\frac{h^l_{12} - \mu_2}{\hat\sigma_2}$ $\frac{h^l_{22} - \mu_2}{\hat\sigma_2}$ $\frac{h^l_{m2} - \mu_2}{\hat\sigma_2}$
$\frac{h^l_{15} - \mu_5}{\hat\sigma_5}$ $\frac{h^l_{25} - \mu_5}{\hat\sigma_5}$ $\frac{h^l_{m5} - \mu_5}{\hat\sigma_5}$
l
h^l_{11} h^l_{21} h^l_{m1}
h^l_{12} h^l_{22} h^l_{m2}
h^l_{13} h^l_{23} h^l_{m3}
h^l_{14} h^l_{24} h^l_{m4}
h^l_{15} h^l_{25} h^l_{m5}
H^l
μ_1 μ_2 μ_3 μ_4 μ_5
$\vec{\mu}$
$\hat\sigma_1$ $\hat\sigma_2$ $\hat\sigma_3$ $\hat\sigma_4$ $\hat\sigma_5$
$\vec{\hat\sigma}$

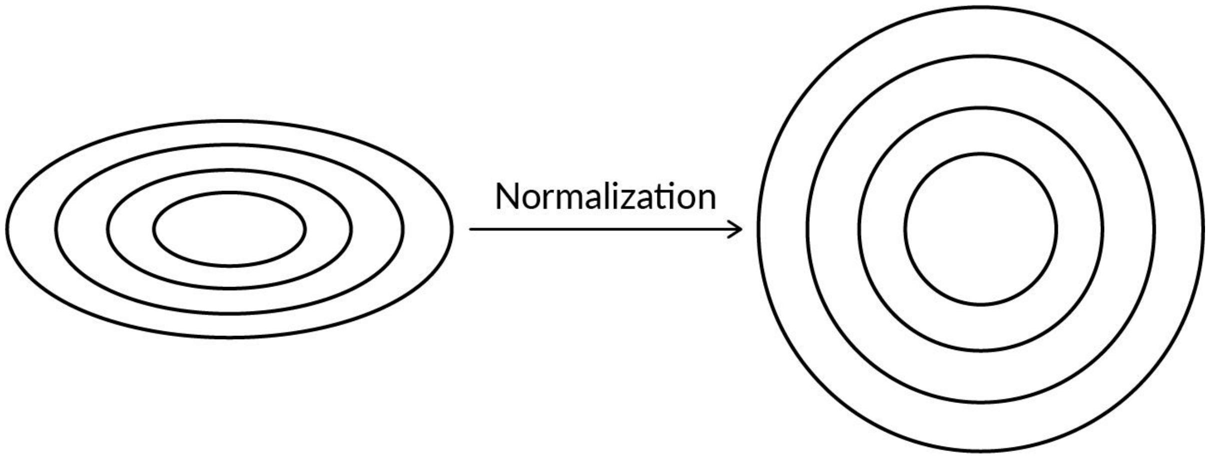
Hence, if layer l has five neurons, we will have H^l of the shape (5, m), where 'm' is the batch size, and vectors μ and ^σ are of shape (5,1). The first element of μ (μ_1) is the mean of the outputs of the first neuron for all the 'm' data points, the second element μ_2 is the mean of the outputs of the second neuron for all the 'm ' data points and so on. Similarly, we get vector ^σ as the standard deviation of the outputs of the five neurons across the m points. The normalisation step can be written as:

$$H^l = H^l - \mu / \hat{}\sigma$$

This step is performed by **broadcasting** μ and ^σ. The final H^l after batch normalisation is shown on the left side of the image given above. Batch normalisation is usually done for all the layer outputs, except the output layer.

However, one small problem is performing batch normalisation during **test time.** Test data points are fedforward one at a time, and there are no 'batches' during test time. Thus, we do not have any μ and ^σ with which we can normalise each test data point. So, we take an average, i.e., the **average of vectors μ and ^σ** of the different batches of the training set.

To understand how batch normalisation solves the problem of decoupling the weight interactions and improves the training procedure, let's reiterate why the normalisation of the input data works in the first place. The loss function contours the change after normalisation, as shown in the image below, and it is easier to find the minimum of the contours on the right compared to finding that on the left. The image below shows the normalisation process:



Note that the batch normalisation process can also be applied to the cumulative input vector into the layer Z^l instead of H^l. These are different heuristics. However, you need not worry about them since deep learning frameworks such as Keras use empirically proven techniques; all you need to do is write a simple line of code.

Libraries such as Keras use a slightly different form of batch normalisation. We can transform the above-mentioned equations as follows:

$$H^l = (H^l - \mu)/\hat{}\sigma = (H^l - \mu)/\sqrt{\hat{}\sigma^2 + \epsilon} = \gamma H^l + \beta$$

Here, constant ϵ ensures that the denominator does not become zero (when the variance is 0). The constants γ, β are hyperparameters. In Keras, you can implement batch normalisation as follows:

```
model.add(BatchNormalization(axis=-1, epsilon=0.001,
                    beta_initializer='zeros',
                    gamma_initializer='ones'))
```

The 'axis= -1' specifies that the normalisation should happen across the rows.

Summary

Fundamentals of TensorFlow

The objectives of this session are as follows:

1. Learn the capabilities of TensorFlow as an ML library, which specialises in deep learning
2. Apply the coding paradigm of TensorFlow to perform simple coding tasks

This session will focus on the fundamentals of TensorFlow, such as its data structure (tensors), and certain mathematical operations that can be performed using TensorFlow.

Introduction to TensorFlow

TensorFlow is an open-source platform for developing end-to-end ML solutions. The TensorFlow platform offers services such as TensorFlow.js, TensorFlow lite and TensorFlow Extended that are used for developing applications for browsers, mobile platforms and large production environments, respectively. TensorFlow has everything one might need to build an ML solution and deploy it on different platforms. Among all the available services in TensorFlow, TensorFlow machine learning library will be covered in this session.

In the previous session, you learnt about the advantages of neural networks. This session will help you understand how TensorFlow can help in building neural networks. TensorFlow offers two main advantages: 1) easy APIs to design complex models and 2) access to specialised hardware. Let's learn about them one by one.

1. **Easy APIs to design complex models:** As you already know, deep models (models with high multiple hidden layers) are good at recognising patterns in data. The TensorFlow API provides the ability to build such deep models with only a few lines of code, making the building process extremely easy. You also need not worry about writing the boiler plate code.
2. **Access to specialised hardware:** Although deep learning has great advantages, there are certain challenges associated with it. Deep learning algorithms need huge volumes of data for training, which means that the algorithms must be fast so that they can train on the huge volumes of data within a reasonable period of time. Certain standard data sets that are used to train computer vision models use 100 gigabytes of training data. This means that such huge volumes of data need to be processed quite fast so that the model can learn from the data efficiently; otherwise, the model will keep running forever.

TensorFlow increases the training speed by using specialised hardware, such as GPUs and TPUs, for processing. Let's understand the computational capabilities of the different hardware that are available for processing.

1. **Central processing unit (CPU):** This is the most commonly used processing hardware. CPUs are designed to process various types of computations quickly. Usually, CPUs have extremely high clock speeds and can perform serial calculations extremely fast. For example, think of a CPU as a supercar used for commuting from your house to the grocery shop to buy groceries; it can help you reach the store quickly.

2. **Graphics processing unit (GPU):** GPUs are general-purpose computing devices. Compared with CPUs, GPUs have slower clock speeds, but they are efficient at parallel computing. Although GPUs are frequently used in the gaming industry, they are quite efficient at data processing as well. For example, think of GPUs as a goods truck used for commuting from your house to the grocery store. Even though the goods truck will take more time than a supercar for each trip, it can carry much more goods, making it much more efficient than a CPU on an overall basis.

3. **Tensor processing unit (TPU):** A TPU is a specialised device developed by Google for processing data in the tensor format. It cannot be used for any other type of processing needs. Since TPUs are specifically developed for processing data in the tensor format, they are extremely fast. For example, think of a TPU as a special conveyor belt connecting your house to the grocery store. Even if it cannot be used to do anything else other than getting groceries, it is the fastest possible way to get groceries.

Apart from these, TensorFlow has a few other capabilities, which are as follows:

1. Calculating gradients of equations
2. APIs in Python and C++

With all these capabilities, TensorFlow is a library that can work well with DL models.

Tensors

A tensor is the fundamental data structure used in TensorFlow. It is a multi-dimensional array with a uniform data type. Earlier, you saw that DataFrames in Spark's structured API have the same data type for each column, whereas tensors have the same data type for the entire tensor. How does this affect the ML process? In the case of DataFrames, a single DataFrame can contain all the raw data loaded into it, such as integers, strings and floats. So, you can load raw data into the DataFrame first and then process the data in order to convert it into a numerical form for ML. In the case of tensors, you need to load data into another data structure and process it first, and when you are ready to learn from the data, you can load it into a tensor.

Similar to NumPy arrays, tensors are n-dimensional arrays. An important difference between the two is their performance; NumPy is a highly efficient library designed to work on CPUs, whereas TensorFlow can work on both CPUs and GPUs. So, if you have a compatible GPU, then it is highly likely that TensorFlow will outperform NumPy.

Although you can build tensors with as many dimensions as you want, the actual ML process is designed to work with 2D tensors. And, in reality, almost all the data can be represented in 4D tensors.

Almost all machine learning data sets can be represented in 2D, but you will need a 3D tensor to represent images and a 4D tensor to represent videos. The dimensions of a tensor are also called ranks.

Tensors are of two types: tf.constant (also known as tf.Tensor) and tf.Variable. Let's summarise the differences between these two types of tensors:

1. Once declared, constant tensors cannot change their values, whereas variable tensors can.
2. Constant tensors need to be initialised with a value while declaring itself, whereas variable tensors can be declared later using operations.
3. Another important difference between the two is that differentiation is calculated only for variable tensors, whereas the gradient operation (which will be explored later in the module) ignores constant tensors while differentiating.

Note: tf.constant is similar to tf.Tensor because both of them have immutable values, but tf.Variable is different from both. Whenever you declare a tensor with tf.constant, the tensor will be an object of type tf.tensor as compared to tf.Variable, which is a different object altogether. You can visit the tf.constant page to understand it better.

Declaring Tensors

You can declare a tensor by specifying the elements in the form of a list, as shown in the code given below.

```
# Vector in TensorFlow
vector_tf = tf.constant([0, 1, 2, 3, 4])

# Vector in Numpy
vector_np = np.array([0, 1, 2, 3, 4])

print("Shapes:", vector_tf.shape, vector_np.shape)

""" A matrix is a 2-dimensional tensor"""

# Matrix in TensorFlow
matrix_tf = tf.constant([[0, 1], [2, 3], [4, 5]])

# Matrix in Numpy
```

```
matrix_np = np.array([[0, 1], [2, 3], [4, 5]])

print("Shapes:", matrix_tf.shape, matrix_np.shape)

"""3-dimensional tensor"""

# 3-d tensor in TensorFlow
t3_tf = tf.constant([[[0., 1.], [2, 3], [4, 5]]]*5)

# 3-d matrix in Numpy
t3_np = np.array([[[0, 1], [2, 3], [4, 5]]]*5)

print("Shapes:", t3_tf.shape, t3_np.shape)
```

You can note two important points here: the number of square brackets and the use of multiplication. Apart from the brackets needed to declare the 2D array, there is one extra pair of brackets in the code. This extra pair of brackets informs TensorFlow that the rank of the tensor being initialised is three. By multiplying the array by 5, the same array is repeated five times to give values to the rank-three tensor. Similarly, you can use a single element or a single row of values to create a tensor of your desired dimension.

1. You will notice the following elements whenever a tensor is printed:
    1. Its values
    2. Its shape
    3. Its data type

2. TensorFlow is able to auto-detect the data type of a tensor based on its values. If there is variability in the data types of the given values, then the following two things can happen:
    1. The different data types can be combined into one. For example, if some of the declared numbers are integers and some are floats, then TensorFLow will make all of them float.
    2. Data types cannot be combined. For example, strings and floats cannot be combined, and TensorFlow will throw an error.

However, the entire tensor needs to have the same data type.

You can declare tf.constant and tf.Variable as exactly the same. Take a look at some observations regarding declaring tensors:

1. You can use tf.Variable exactly like tf.constant; in order to initialise the tensor with values, you can pass in a list. However, in the case of variables, you can change the

values later.

2. You can also specify the data type while initialising the tensor. By doing this, you can be sure of the data type. Although doing so might seem a bit trivial now, in the upcoming segments, you will understand the importance of declaring the data type of a tensor.

3. You can access the values of a tensor directly using the .numpy() function. This function will return the values of the tensor as a NumPy array.

Mathematical Operations on Tensors

TensorFlow has all the mathematical capabilities that you will need to build an ML model, such as basic operations, linear algebra, computing gradients and many more. Let's go through these capabilities one by one.

TensorFlow supports all the basic mathematical operators, and you can call these operators by simply using the respective operators. To use the operator commands, you need to ensure that both the tensors on which operations are to be carried out have the same dimensions. If their dimensions are different, then an error will be thrown because the operations are performed element-wise. Another point to note is that when you divide any number by zero, TensorFlow is smart enough to show the infinity value.

The same operations can also be performed using functions from the TensorFlow library. For example, the tf.add() function can be used instead of the addition operator. Similarly, the tf.subtract(), tf.multiply() and tf.divide() functions work exactly how you would expect them to. You can read about all the mathematically available functions here.

Linear Algebra:

1. The linalg module from the TensorFlow library has all the necessary functions for processing matrices. You can read about the available functions here.

2. Many matrix operations require shape compatibility. For instance, in the case of the matrix multiplication, the number of columns in the first matrix must be equal to that in the second one. Such conditions must be met while performing respective operations on tensors. If they are not met, then TensorFlow will throw an error.

3. It is possible that some linear operations are not defined. For example, all matrices cannot be inverted. In such cases, TensorFlow will throw an error. So, in order to execute a matrix operation, the operation should be mathematically possible. For example, calculating the inverse of a matrix with determinant zero is not possible; hence, it will not be possible in TensorFlow as well.

4. TensorFlow uses numerical algorithms for carrying out matrix operations; hence, it is important to have the tensor of the float data type.

Reshaping and Broadcasting

As discussed earlier in a segment, shape-modifying functions are important because a lot of matrix operations require shape compatibility. It is likely that you will have to modify the shapes of tensors to ensure that computations in ML algorithms do not throw errors. Some of the methods to ensure that mathematical operations are performed efficiently are as follows:

Reshaping: As the name suggests, reshaping changes the dimension of a tensor. However, there are certain limitations to the reshape function, one of which is that it can reshape a tensor but cannot remove or add new elements. For example, a tensor with the shape (3, 4) will have 12 elements. In this case, the reshape function can change the shape of this tensor to (4, 3) or (6, 2) or even (12, 1) but not to (4, 4). If you want to change the shape to (4,4), then you will need 16 elements in the tensor, but there are only 12 elements in it; hence, the reshape operation will throw an error.

Broadcasting: You can perform an operation on tensors with mismatching dimensions by broadcasting the elements to more dimensions. For example, you can use a matrix with shape (1) and multiply it by a matrix with shape (3,3). Now, even though the shapes are not compatible, the multiplication will be executed because the smaller matrix is repeated over all the elements of the larger matrix. Such a repetition of one matrix is called broadcasting. In order to perform the broadcast operation, certain conditions must be met, and the shapes of the tensors are said to be compatible only if these conditions are met. Compatible tensors are defined by the following rule: Two shapes are compatible if the dimensions for each dimension pair are either equal or if one of them is (1). While trying to broadcast a tensor to a shape, the computation starts with the trailing dimensions and works its way forward.

Expanding dimensions: If you notice the possible broadcasting operations carefully, you will realise that in a few cases, you are essentially increasing the dimensions of a tensor by broadcasting it to a larger tensor. You can also do this by using the expand_dimms method. Using this method, you can add a dimension to a tensor without adding elements to the new dimension.

Computational Graphs and Gradients

A computational graph is a representation of the mathematical operations in a TensorFlow program. Tensors and computational graphs are two of the most fundamental concepts of TensorFlow. As you already know, tensors are the data structures that are used to store data in TensorFlow, whereas computational graphs are used to identify the 'flow' using different mathematical operations through which a tensor will pass. In a computational graph, the edges represent the data (in the form of tensors), whereas the nodes represent the mathematical operations that need to be performed on the tensors.

Computational graphs offer the following benefits:

1. **Visualisation:** Computational graphs help in visualising the algorithm. Visualising an algorithm makes it easier for you to develop and maintain complex algorithms. This is helpful especially in the case of neural networks because neural network models are quite complex.

2. **Gradient calculations:** Calculating gradients is one of the important abilities of TensorFlow. Computational graphs are used to trace the dependencies of variables on one another. First, a path is traced from the dependent variable to the independent variable; after this, all the intermediate gradients are calculated and used to compute the expected gradient using the chain rule of differentiation.

3. **Distributed architecture:** Computational graphs also help in distributing the training process on a cluster of machines. In the next segment, you will learn how this distributed architecture exactly works.

Now, let's learn how to find the gradient of a given equation. The steps involved in calculating the gradient of any function are given below:

1. Initialise the independent variables. The dependent and independent variables need to be of the tf.Variable type tensors for the gradient to work.

2. Create a context of the GradientTape() method and record the equations that relate the different variables inside the context.

3. To find the derivative of an equation that is recorded in the gradient tape context, use the .gradient() method outside the context and pass in the variable to differentiate as well as the variable with respect to which the differentiation will happen.

The GradientTape() method has a few new capabilities. Let's take a look at them here:

Chain rule: In a function representing y = h(x), in which x is defined as x = g(r), you can directly calculate dy/dr. The GradientTape() method can calculate the gradient based on the chain rule independently, without the need to specify the internal dependencies. To revise the chain rule, you can visit the optional module titled '**Math for Machine Learning**' in the course '**Machine Learning I**'.

Persistence of variables: Consider the system of equations given below.

p = a(x); q = b(p); y = c(q);

After defining all the equations inside the GradientTape() method, all the intermediate variables inside the tape, such as p and q, will be lost. You can use the chain rule to calculate the derivative of y with respect to q, p or x. However, you cannot calculate dq/dx unless you mention the parameter ‘persistent=True’ in the GradientTape() method.

You can also make the variable non-trainable to ensure that you do not calculate the gradient of that particular variable.

TensorFlow Architecture

The core of TensorFlow is the C++ engine, which is responsible for communicating with the other engines, such as the distributed master (this is explained later in this segment), the kernel implementations, and the actual devices on which the computations will be run.

Additionally, a Python API is built on top of the C++ core. The Python API provides the developers with a high-level language for passing instructions to the core. Using the Python API, you can build high-level modules such as tf.math and tf.linalg. These modules provide basic functionalities out of the box and save a lot of efforts that might have otherwise been wasted in writing boilerplate low-level code. Now, libraries that make ML tasks easier are built on top of modules such as tf.math and tf.linalg. One such library is called Keras, to which you will be introduced in the next session.

Distributed Architecture

Throughout this session, all the demonstrations were performed on a single device provided by the Google Colab platform. Regardless of the size of the provided machine, you are bound to encounter data so big that a single machine will not be able to process it. So, it becomes important to be able to distribute tasks across multiple machines or devices. TensorFlow is capable of distributing the learning process.

Distributed computations in TensorFlow can be divided into the following three main steps:

1. **Client:** The job of the client is to build a computational graph using the available APIs. As you learnt earlier in this segment, the heart of TensorFlow is the C++ core, and different APIs are built on top of it. The objective of the client is to make sure all the tensors and the operations to be carried out on the tensors are communicated to the core of TensorFlow.

2. **Distributed master:** The distributed master receives the complete computational graph from the client-side. First, it runs some rule-based optimisations, such as combining constant nodes, if they are present. The master then creates subgraphs from the computational graph so that workers can individually work in the subgraphs.

3. **Workers:** There are two types of worker processes: a parameter server and the actual computation work. The parameter server, as the name suggests, is responsible for holding all the necessary parameters and data. Its sole job is to keep all the parameters

updated for other workers to use and update.

The second type of job that the workers do is the actual computation. All the workers access the data from the parameter server and perform the operations that are defined in the subgraph given to each of them. After the computations are complete, the workers update the parameter server so that the remaining process can be carried out.

Data flows from the client to the master and then to the workers. The workers communicate among themselves and complete their job. After the required computation is complete, the data is sent back to the client for the user to access.

Finally, you also implemented the neural network to classify the handwritten digits of the MNIST data set using TensorFlow.