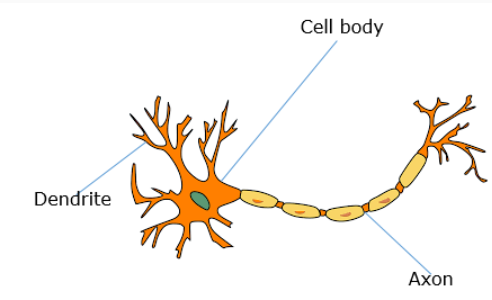
**Neural Networks**

Even with the advancements of technology and algorithms, there is still no match to the capabilities of the human brain which hasn't been fully understood, yet. The core components of our nervous system or the brain are the **neurons** or nerve cells which are also referred as “brain cells”. These cells are connected to one another to form a complex network structure known as “**Neural Network**”.

The above figure represents the structure of a neuron which has three major components, namely

1. Cell body
2. Dendrite
3. Axon

Dendrites receive signal from other neurons and bring it to the cell body, where the signal is processed and axons which are connected to dendrites of neighboring neurons, transmit these signals away from the cell body.

The study of **artificial neural network** (ANN) is inspired by attempts to simulate the biological neural system. ANN consists of interconnected **nodes,**analogous to human/biological neuron network. Just like a neuron receives an input signal, processes it and transmits to other neurons, in ANN, the node receives the input, process it using a function known as **activation function** and transmits the output to the other nodes.

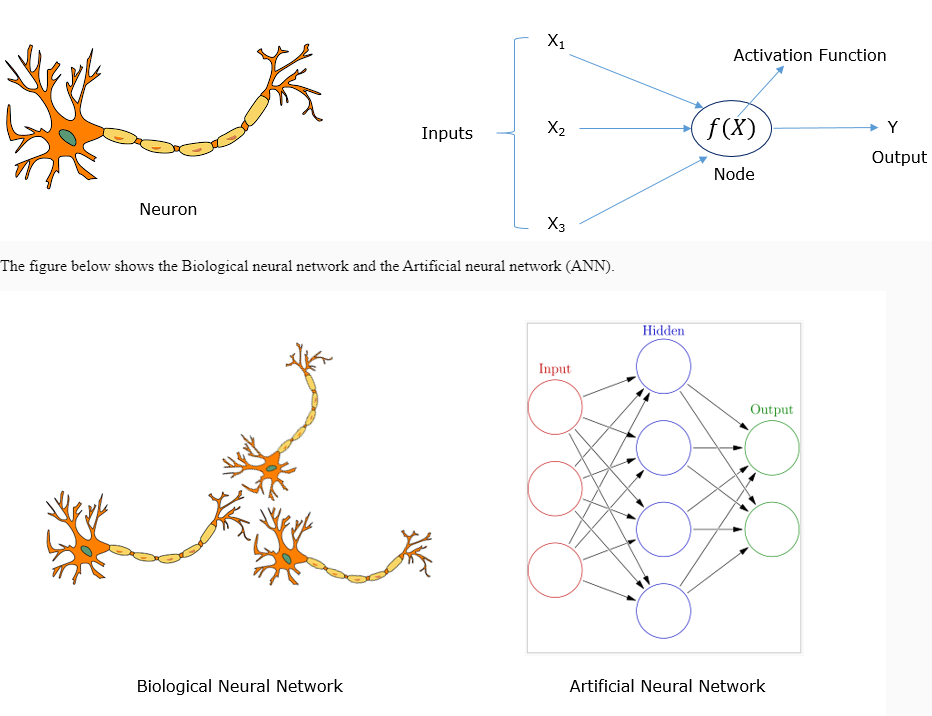
An ANN has 3 kinds of nodes, namely,

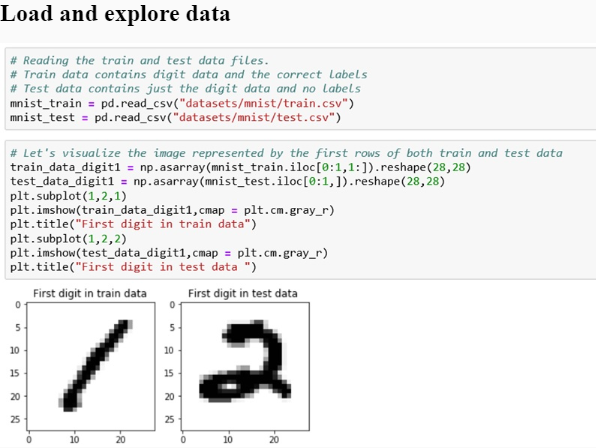
**Input Nodes:** These are the nodes, that receive the input.

**Hidden Nodes:** These nodes receive the input from the input nodes, process the input and pass it on to the output nodes.

**Output Nodes:** These nodes receive the processed inputs from the hidden nodes.

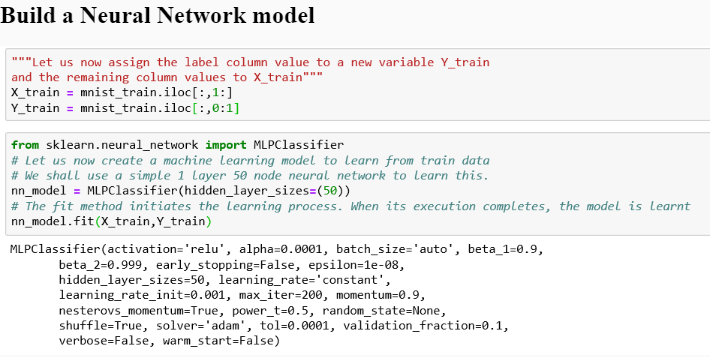
The figure below shows the analogy between a neuron of the human brain and the node of ANN.

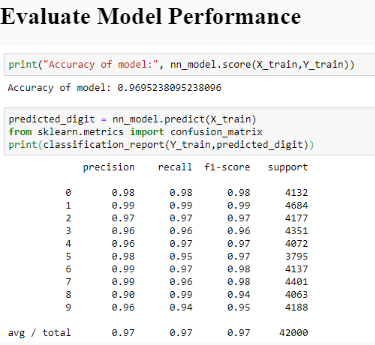
­­

Consider the handwritten data set that we used in part to understand clustering. Let us now look at how a Neural network can be used to classify the data.

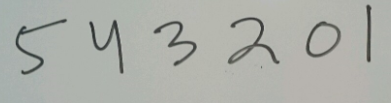
The below code demonstrates the usage of MLPClassifier in sklearn.neural\_network that helps us create a classifier using a neural network.

**Data Set in present in sreekanthsreeraj- Google Drive**





In real life, ANNs are widely used for pattern recognition owing to their ability to respond to unexpected inputs/patterns such as to recognize handwritten letters or digits, identify objects from images, etc. At the core, pattern recognition is also a classification problem.

**Consider the following handwritten digits:**

While it is really easy for humans to recognize the above sequence of digits as 5 4 3 2 0 1, it is really difficult to write an algorithm, that can make a computer realize the same. This is because handwritings vary from person to person.

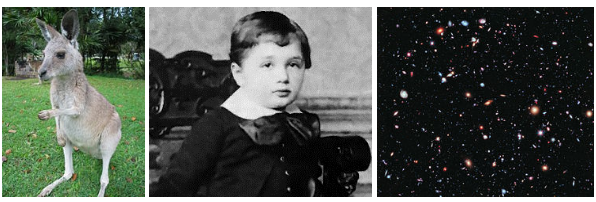
Neural network approaches this problem in an unconventional approach, as opposed to the classical algorithmic approach. It takes a set of handwritten texts or digits as training examples and automatically infers the rules for recognizing handwritten texts or digits. After the neural network is trained to recognize the handwritten texts, it can also recognize texts which might be really different from the training examples.

Banks and other financial institutions use a technique known as optical character recognition (OCR) to recognize the signature of customers in documents. Under the hood, OCR uses neural networks.

Post offices are also using OCR to automatically sort the letters by reading the handwritten zip codes on the letters.

In **optical character recognition** that we discussed previously, we dealt with a limited number of characters.

Consider the problem given below which involves recognizing whether a given image is of a human being or not.



This image processing or pattern recognition problem where a given image of an arbitrary object with the background, color and other details,  needs to be checked against the expected characteristics of a human image is inherently more complex than OCR.

If we try to write an algorithm for such a problem, we may divide our problem into a set of sub-problems. Each sub-problem being a yes/no question such as:

1. Does the image have an eye in the top left?
2. Does it have an eye in the top right?
3. Does it have a nose in the middle?
4. Does it have a mouth in the bottom middle?
5. Is there hair on top? , etc.

If the answer to most of these questions is yes, then we should be able to conclude that the image is likely to have a human being in it.

Additionally, each of the sub-problems might further be broken down and ultimately be posed with questions about the presence or absence of a particular shape at a particular pixel point in the image. What makes this further complex is each of the sub-problems might have to deal with a wide range of exceptions. For example, there could be a bald man who doesn’t have hair on top of his head, or the photo might have been taken from an angle where both the eyes are not visible etc. Hence, to make the machine predict properly, our algorithm must consider all such exceptions. This may be achieved by using multiple neural networks.

As a result, the complex question “Does the image have a human being in it?” is broken down into many simpler questions. Each of these questions could be answered using a neural network.

Such neural networks with many layers are known as deep neural networks and the machine learning involved is called **deep learning**.

# Deep Learning

Deep learning is a new area of research in the field of machine learning, which aims to be at par with Artificial Intelligence(AI). It is used in various areas such as computer vision, Natural Language Processing(NLP) and speech processing, robotics, etc. Deep learning finds applications in various schemes such as Self-driving cars by Google or Tesla, etc., Digital Assistants such as Siri or Alexa, Recommendation systems in e-commerce sites, etc.

TensorFlow is an open source library for machine learning for various tasks. It is used by various Google products such as Gmail, Google Photos, speech recognition and search.

Getting started with TensorFlow is fairly simple, all you would need is a compatible version of Python. The detailed instructions on how to download and setup TensorFlow are documented on the official TensorFlow ebsite.

TensorFlow can be used to realize deep learning with just a few lines of code. The below code snippet demonstrates the usage of TensorFlow APIs to classify handwritten digits using the MNIST dataset.

1. """Importing the data"""
2. from tensorflow.examples.tutorials.mnist import input\_data
3. mnist = input\_data.read\_data\_sets("MNIST\_data/", one\_hot=True)
4. """Importing tensorflow library"""
5. import tensorflow as tf
6. """Placeholder for x. This is the input that we are providing"""
7. x = tf.placeholder(tf.float32, [None, 784])
8. """Variable W and b are to be determined by the regression. Initially assigned to zero"""
9. W = tf.Variable(tf.zeros([784, 10]))
10. b = tf.Variable(tf.zeros([10]))
11. """Neural Network model to find W and b using softmax regression"""
12. y = tf.nn.softmax(tf.matmul(x, W) + b)
13. """This is the placeholder for the correct answer of input """
14. y\_ = tf.placeholder(tf.float32, [None, 10])
15. """Cross entropy function that needs to be minimized for a good model"""
16. cross\_entropy = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(y, y\_))
17. """Applying gradient descent optimizer to minimize the cross entropy. learning rate chosen as 0.5"""
18. train\_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross\_entropy)
19. """Initialize the variables"""
20. init = tf.global\_variables\_initializer()
21. """Launch a session. Until this point, no computations are actually performed"""
22. sess = tf.Session()
23. sess.run(init)
24. for i in range(1000):
25. batch\_xs, batch\_ys = mnist.train.next\_batch(100)
26. sess.run(train\_step, feed\_dict={x: batch\_xs, y\_: batch\_ys})
27. """Comparing the predicted value against the actual value"""
28. correct\_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y\_,1))
29. accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))
30. """Displaying the overall accuracy"""
31. print(sess.run(accuracy, feed\_dict={x: mnist.test.images, y\_: mnist.test.labels}))

The obtained output is as shown below. Here we can observe that the accuracy obtained is 90.59%.

**Deep Learning -2**

**Software requirements**

Anaconda distribution 4.4.0 with Python 3.6.1 or above.

Here's a list of packages version used while building this course:

* **imageio**: 2.5.0
* **keras**: 2.2.4
* **keras-lr-finder**: 0.1
* **keras-vis**: 0.4.1
* **matplotlib**: 2.0.2
* **mlxtend**: 0.15.0.0
* **numpy**: 1.16.4
* **pandas**: 0.24.1
* **Pillow**: 6.0.0
* **scikit-learn**: 0.21.2
* **scipy**: 1.2.1
* **tensorboard**: 1.12.2
* **tensorflow**: 1.12.0
* **vis**: 0.0.5

Why deep learning

The starting of 21st century has introduced us with data as the new fuel to drive today’s society. Our dependency on electronic devices has increased at an exploding rate. With the availability of such humongous data generated every second along with the ample computational resources, many complex tasks have been solved. Following are a few real world examples:

1. Facial Recognition Photo Tagging: Are you on Facebook? If yes, have you ever uploaded a picture of yours along with your friends? Did you notice that just by hovering over your friends’ faces, Facebook identifies their faces and asks you to tag them too? This is an example of deep learning image identification backed by the Deep Face (a deep learning architecture developed by Facebook, Inc.).
2. Natural Language Processing/Generation/Translation: Have you ever met a person from a region, whose language is difficult for you to decipher? What do you do if you have to converse with that person? Translation apps could be the answer! Google Translate is one of the targeted services to solve such frequent problems. The Google Translate product uses Google’s Neural Machine Translation System (a deep learning architecture) to provide near human-expertise in translating languages.
3. Self-driving cars – Have you heard of the autonomous cars being developed by Tesla, Google, Uber and every other major automaker? Have you wondered how their self-governing navigation system works? They use sensors and on-board analytics to learn to recognize any obstacles in the path and react to them appropriately using Deep Learning.

Some other major platforms which use deep learning include: real-estate companies, e-commerce websites, IoT applications, education platforms, etc., to address various day to day challenges.

The fact that we are all consumers to at least one of these deep learning products out there is undeniable and that is exactly why it is important for each one of us to be knowledgeable about Deep Learning.

Although Deep learning isn’t new in the market, however, it’s presence has been rapidly rising in the past few years due to the available computational resources and data.

“Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction”.

Let us try to make sense of this definition with the help of a simple example.

Consider the cat image.

Let us assume that we are trying to teach a child how to recognize cats for the very first time in life.

When the child is trying to learn to identify a cat, we never know exactly how the child is relating the word ‘cat’ to that specific mammal. Having said that, let us still try to reflect upon the various stages of learning of the child.

Stage 1: The child may wonder that all the mammals with fur are cats. So, initially the child tries to identify small features to arrive at the result. However, this could lead to the child incorrectly identifying a dog with fur as a cat. This tells us that the child still needs another stage of learning.

Similarly, a deep learning neural network starts with small features. With limited knowledge retaining capability (i.e. number of nodes and layers), it too tends to misidentify the target output initially.

Stage 2: To improve over the previous course of learning, the child starts combining few small features together like a mammal with long whiskers around nose. This way the child’s capability starts increasing as the child learns with more experience, utilizing more memory to piece and remember the features together.

Similarly, a deep learning network joins small features and look for improving its performance and the higher performance probability is still dependent upon ample data and computing resources.

Stage 3: With enough experience, a child connects various small features together to finally distinguish a cat from other mammals.

Likewise, a deep learning network built from scratch does various mistakes due to less knowledge (small features) but it gradually improves with more data (taking less time, if provided with high-end computational resources) to connect those small features to build an intuition to arrive at an output.

**Deep learning vs classical machine learning**

The notion of deep learning is built on the core of classical machine learning approaches (like regression analysis, classification algorithms, the use of optimization algorithms, etc.) and proves worthy over them in the following situations:

* when enormous amounts of data are available
* when enormous computational power is available
* a non-interpretable model is not detrimental

But how exactly does deep learning improve upon classical machine learning?  To answer this question let us consider the cat scenario again.

If a classical machine learning algorithm has to identify a cat’s face, then it would rely on feature engineering to build its model. The quality of features decides the accuracy of identification. On the contrary, a deep learning algorithm doesn’t need feature engineering from the user end! Instead, it asks for a sequence of input images and creates features (whisker, eyes, nose, hair pattern, etc.) out of them, all by itself! In this sense, we can say that a deep learning algorithm learns just as a child does.

The higher the number of cats a child observes, the more accurate his/her ability to identify a cat becomes. Similarly, the more the number of training data samples a deep learning algorithm gets, the higher its accuracy becomes in identifying cats becomes.

Of course, similar to a child, a deep learning algorithm does perform mistakes while starting its learning journey but due to the availability of enormous data and computational resources in the present times it has proven to be far better than classical machine learning architectures in solving certain complicated problems (as seen before) and in some cases competitive to the human experts.

The following graph sums up the story. Higher the amount of data, higher is the deep learning architecture performance as compared to classical machine learning architectures.

Let us now discuss the various components of a neural network:

**Neurons**: These are the elementary units in a neural network. An artificial neuron is a mathematical function conceived as a model of biological neurons.

**Layers**: These are the group of neurons at different levels. It can be broadly classified into the following three types:

Input Layer

* It is the first layer in any neural network.
* Input is fed to the network through this layer.
* It brings the initial data into the system for further processing by subsequent layers of artificial neurons.
* The input layer is the very beginning of the workflow for the artificial neural network.
* It is the leftmost layer in the architecture of a neural network.

Output Layer

* It is the last layer in any neural network.
* We get the output of the network through this layer.
* It gives the processed output by the previous layers of artificial neurons.
* The output layer is the last of the workflow for the artificial neural network.
* It is the rightmost layer in the architecture of a neural network.

Hidden Layer

* It is the layer between the input layer and the output layer in any neural network.
* It takes a set of weighted inputs and produces output through an activation function.
* It is a typical part of nearly any neural network through which engineers try to simulate the activities that occur in a human brain.
* There can be multiple hidden layers and each of these hidden layers can contain the same or a different number of neurons.
* The input after getting processed with different weights and biases at each neuron at different layers is used to give us the output.

**Weights and Biases**: Weights are numerical parameters which determine how strongly each of the neurons affects the other. A bias unit is an "extra" neuron added to each pre-output layer that stores the value of 1.

Weights are learned (or adjusted) to produce the desired outputs from the given inputs. Mathematical techniques like Stochastic Gradient Descent, Adam, etc. (which we will discuss later) are used to search for optimal weights that will make the model accurate in its prediction.

**Activation function**: An activation function is a mathematical function which converts the input to an output. It gives the nonlinearity property to neural networks and makes them the true universal functional approximately. Without activation functions, the working of neural networks will be comparable to the linear functions.

Commonly used activation functions are as follows:

* Sigmoid
* Tanh
* ReLU
* Softmax

Now, let us take an example to understand the relevance of each of the components that we just discussed, in building an efficient neural network. Let us consider the Fashion MNIST [dataset](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_0127785480690483207452/web-hosted/assets/fashionmnisttrain.csv) for the same. This dataset comprises of images corresponding to ten classes: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot.

So, before we start building the model let us first do some pre-processing using Keras, which we will discuss extensively later in this course.

1. *# Importing libraries*
2. import pandas as pd
3. import tensorflow as tf
4. import keras
5. *# Reading train and test data*
6. train\_data = pd.read\_csv('fashion-mnist\_train.csv')
7. *# Class names*
8. class\_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
9. 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
10. *# Creating validation data from test data*
11. val\_data = test\_data.iloc[:5000,:]
12. test\_data = test\_data.iloc[5000:,:]
13. *# Fetching the labels*
14. train\_labels = train\_data.label
15. val\_labels = val\_data.label
16. test\_labels = test\_data.label
17. *# Reshaping training data*
18. train\_images = train\_data.iloc[:,1:].values.reshape(60000, 28, 28)
19. *# Reshaping validation data*
20. val\_images = val\_data.iloc[:,1:].values.reshape(5000, 28, 28)
21. *# Scaling data in the range of 0-1*
22. train\_images = train\_images/255.0
23. val\_images = val\_images/255.0

Now, with the basic steps done, we can proceed with the building of the different variants of a neural network.

# Case I

In this first case, we will build a neural network with one hidden layer having one neuron and with the absence of a non-linear activation function as shown below. At this point do not worry about the details of the code, just concentrate on the outcome.

1. *# Defining multi-layer perceptron model with 1 hidden layer having 1 neuron*
2. model = keras.Sequential([
3. keras.layers.Flatten(input\_shape=(28, 28)), *# Perform conversion of higher dimensional data (here, 2-D) to 1-D data.*
4. keras.layers.Dense(1, activation=tf.keras.activations.linear), *# Hidden layer with 1 neuron and linear activation function*
5. keras.layers.Dense(10, activation=tf.keras.activations.linear) *# Output layer with linear activation function*
6. ])
7. *# Defining parameters like optimizer, loss function and evaluating metric*
8. model.compile(loss='sparse\_categorical\_crossentropy',
9. optimizer=keras.optimizers.Adam(),
10. metrics=['accuracy'])
11. model1 = model.fit(train\_images, train\_labels, epochs=5, validation\_data=(val\_images, val\_labels))

Here, the validation accuracy turns out to be **10.32%** which is very low.

# Case II

Next, let's build a network with one hidden layer having ten neurons as shown:

1. *# Defining multi-layer perceptron model with 1 hidden layer having 10 neurons*
2. model = keras.Sequential([
3. keras.layers.Flatten(input\_shape=(28, 28)), *# Perform conversion of higher dimensional data (here, 2-D) to 1-D data.*
4. keras.layers.Dense(10, activation=tf.keras.activations.linear), *# Hidden layer with 10 neurons and linear activation function*
5. keras.layers.Dense(10, activation=tf.keras.activations.linear) *# Output layer with linear activation function*
6. ])
7. *# Defining parameters like optimizer, loss function and evaluating metric*
8. model.compile(loss='sparse\_categorical\_crossentropy',
9. optimizer=keras.optimizers.Adam(),
10. metrics=['accuracy'])
11. model2 = model.fit(train\_images, train\_labels, epochs=5, validation\_data=(val\_images, val\_labels))

By increasing the number of neurons in a hidden layer, we can observe the increase in the validation accuracy to **43.3%**.

# Case III

# In the previous two cases, our models were performing linear regression due to the linear activation function. Now, let's introduce non-linearity to the above case to make it a classification model and thus classify ten classes.

1. *# Defining multi-layer perceptron model with 1 hidden layer having 10 neurons with non-linearity*
2. model = keras.Sequential([
3. keras.layers.Flatten(input\_shape=(28, 28)), *# Perform conversion of higher dimensional data (here, 2-D) to 1-D data.*
4. keras.layers.Dense(10, activation=tf.nn.relu), *# Hidden layer with 10 neurons and ReLU activation function*
5. keras.layers.Dense(10, activation=tf.nn.softmax) *# Output layer with softmax activation function*
6. ])
7. *# Defining parameters like optimizer, loss function and evaluating metric*
8. model.compile(loss='sparse\_categorical\_crossentropy',
9. optimizer=keras.optimizers.Adam(),
10. metrics=['accuracy'])
11. model3 = model.fit(train\_images, train\_labels, epochs=5, validation\_data=(val\_images, val\_labels))

As you can observe the validation accuracy has increased a lot higher compared to the previous case. It turns out to be 85.06% with the introduction of the non-linear activation function.

# Case IV

Let us try to increase the number of hidden layers. This time we will use three hidden layers, each with ten neurons and lastly accompanied by a non-linear activation function.

1. *# Defining multi-layer perceptron model with 3 hidden layer having 10 neurons each and with non-linearity*
2. model = keras.Sequential([
3. keras.layers.Flatten(input\_shape=(28, 28)), *# Perform conversion of higher dimensional data (here, 2-D) to 1-D data.*
4. keras.layers.Dense(10, activation=tf.nn.relu), *# Hidden layer with 10 neurons and ReLU activation function*
5. keras.layers.Dense(10, activation=tf.nn.relu), *# Hidden layer with 10 neurons and ReLU activation function*
6. keras.layers.Dense(10, activation=tf.nn.relu), *# Hidden layer with 10 neurons and ReLU activation function*
7. keras.layers.Dense(10, activation=tf.nn.softmax) *# Output layer with softmax activation function*
8. ])
9. *# Defining parameters like optimizer, loss function and evaluating metric*
10. model.compile(loss='sparse\_categorical\_crossentropy',
11. optimizer=keras.optimizers.Adam(),
12. metrics=['accuracy'])
13. model4 = model.fit(train\_images, train\_labels, epochs=5, validation\_data=(val\_images, val\_labels))

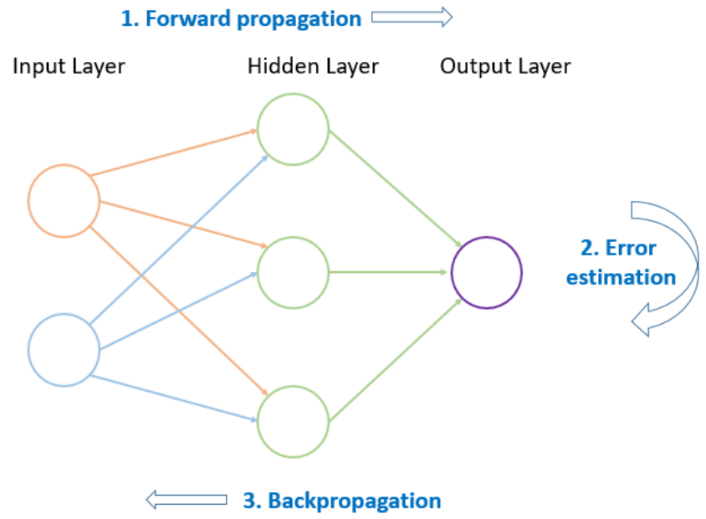
As we can observe, there is a slight improvement in the validation accuracy, 85.76%. We can build an optimum network by tuning each hyperparameter.

The table below summarizes the above results:



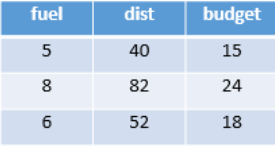
You can replicate the above results with the complete code given below in the Ipython notebook. Note, results may slightly differ due to random parameter initialization.

**Note book Present in : GitHub M1R3NNoverview.ipynb & Data present in Sreekanthsreeraj Google drive**

Let us next discuss how a neural network exactly learns its parameters.

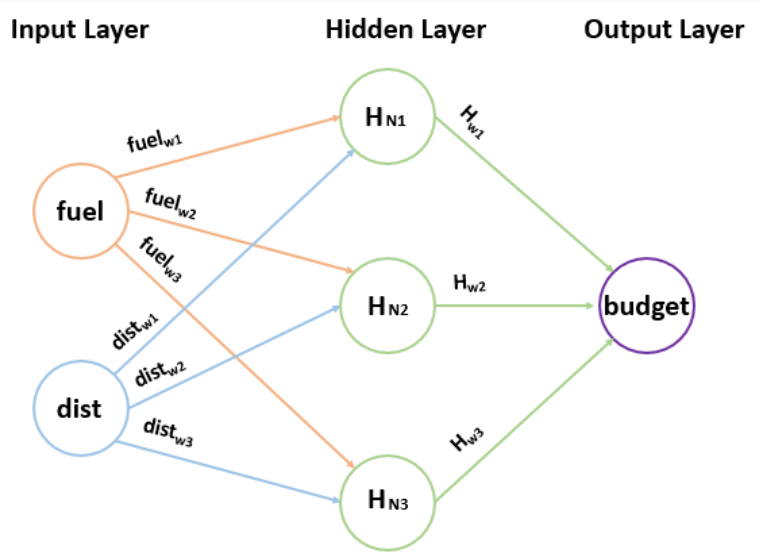
The model learns its parameters (weights and biases) through a process of forward propagation of data and back propagation of errors. To start with, random weights and biases are assumed. The input is then fed forward through the network. Each hidden layer modifies the input to it, until it reaches the output. This output achieved is compared to the expected output. Error is recorded as their difference. This error is then propagated backward through the layers to adjust the parameters. The forward propagation and the back propagation continue over a number of iterations, till a desirable low level of error is achieved. The neural network is then said to have learned its parameters.

Project : Budget Detection for Automobiles Data

Let us now implement forward propagation and back propagation on a sample data set. Consider two independent variables, fuel consumption in litres (fuel) and distance travelled in kilometers (dist) along with a dependent variable budget in dollars (budget).The sample data set to be used is given below:

1. *ecessary libraries to be used*
2. import numpy as np
3. import matplotlib.pyplot as plt
4. from matplotlib import animation
5. %matplotlib notebook
6. *# X: (Feature 1, Feature 2)*
7. X = np.array([[5, 40],
8. [8, 82],
9. [6, 52]], dtype=float)
10. *# y: Target*
11. y = np.array([[15], [24], [18]], dtype=float)
12. *# Scaling units*
13. X = X/np.max(X, axis=0) *# maximum of X array*
14. y = y/max(y) *# maximum of y array*

Next, let us define the architecture of the neural network to be built. We have one input layer assigned with two features with dimensions 3 x 2, hence two nodes. Then we have an output layer consisting of a quantitative output with dimensions 3 x 1, and hence we assign it with one node. Next, we take a hidden layer and assign it with three nodes. The overall model architecture resembles the diagram shown below.



Let us initialize the size of each layer with the following code:

1. class Neural\_Network(object):
2. def \_\_init\_\_(self):
3. *# Parameters*
4. self.inputSize = 2 *# Two nodes*
5. self.outputSize = 1 *# Single node*
6. self.hiddenSize = 3 *# Three nodes*

Next, let us take some known weights and try to understand how we can arrive at the output through the above model.

Let us start by scaling the values by dividing with the maximum value:

1. fuel = 5/8 = 0.625,
2. dist = 40/82 = 0.487,
3. budget = 15/24 = 0.625

Let us consider some weights corresponding to fuel node

1. fuelw1 = 0.3, fuelw2 = 0.2, fuelw3 = 0.6

and, dist node

1. distw1 = 0.22, distw2 = 0.56, distw3 = 0.7

With these weights and associated fuel and dist variables value, let us find the values for hidden layer nodes.

1. Hidden layer node 1 = HN1 = (0.625 \* 0.3) + (0.487 \* 0.22) = 0.29464
2. Hidden layer node 2 = HN2 = (0.625 \* 0.2) + (0.487 \* 0.56) = 0.39772
3. Hidden layer node 3 = HN3 = (0.625 \* 0.6) + (0.487 \* 0.7) = 0.7159

Now, let us apply non-linearity on these values to get the final hidden layer node values. Here, we will consider the sigmoid activation function whose code is shown below:

1. def sigmoid(self, s):
2. return 1/(1+np.exp(-s))

which produces the following values:

1. sigmoid(HN1) = 0.5731
2. sigmoid(HN2) = 0.5981
3. sigmoid(HN3) = 0.6717

Next, let us consider some weights between hidden layer and output layer;

1. Hw1 = 0.21, Hw2 = 0.45, Hw3 = 0.85

Then, the final value on the output layer is:

1. HN1 \* Hw1 + HN2 \* Hw2 + HN3 \* Hw3 = 0.5731 \* 0.21 + 0.5981 \* 0.45 + 0.6717 \* 0.85 = 0.9604

We can observe that the predicted output with assumed weights is **0.9604** whereas the actual output is **0.625**. The difference between the predicted and actual output is **0.3354**. However, this may or may not be the best prediction. This is where backpropagation comes handy which keeps on adjusting the loss to find optimum parameters.

As our aim is to decrease the loss (ideally to zero), and therefore we need to find the rate of change of the loss with respect to the weights which will help us to figure out the direction to alter the weights.

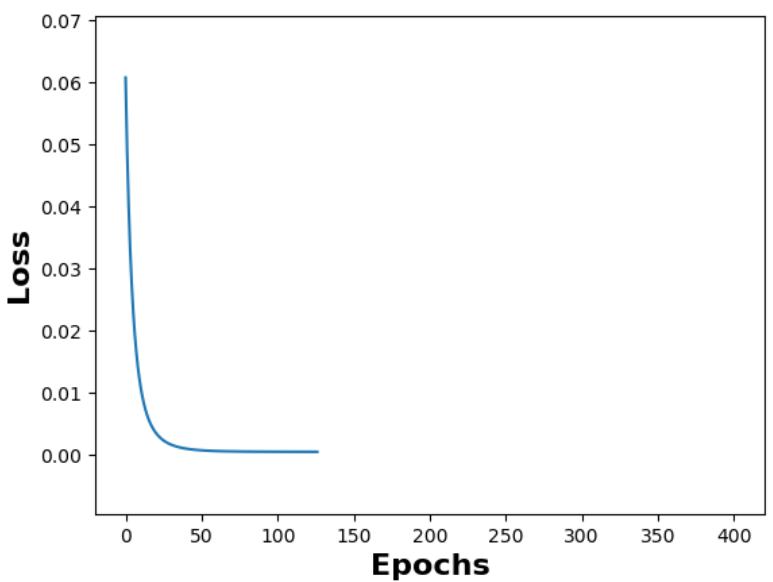
This process uses gradient descent which helps us to decrease the loss. We are going to learn more about gradient descent in the next section.

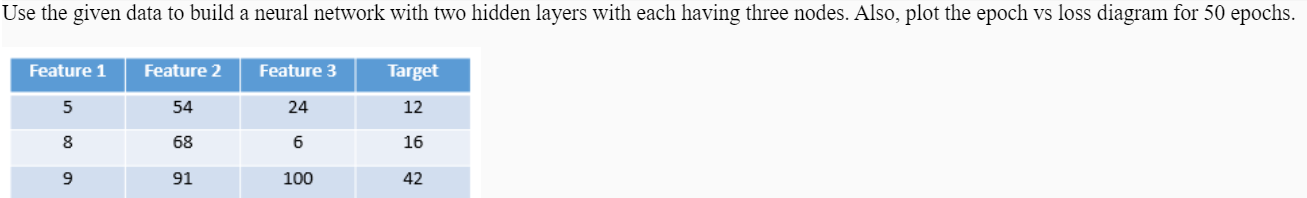
To find the change in the weights we first find the error margin of the output layer by taking the difference of predicted and actual output. Next, you need to apply the derivative of the sigmoid activation function to the output layer to figure out how much hidden layer contributed to the output error.

This process has been coded below:

1. def sigmoidPrime(self, s):
2. *#derivative of sigmoid*
3. return s \* (1 - s)
4. def backward(self, X, y, o):
5. *# backward propagation through the network*
6. self.o\_error = y - o *# error in output*
7. self.o\_delta = self.o\_error\*self.sigmoidPrime(o) *# applying derivative of sigmoid to error*
8. self.z2\_error = self.o\_delta.dot(self.W2.T) *# z2 error: how much our hidden layer weights contributed to output error*
9. self.z2\_delta = self.z2\_error\*self.sigmoidPrime(self.z2) *# applying derivative of sigmoid to z2 error*
10. self.W1 += X.T.dot(self.z2\_delta) *# adjusting first set (input --> hidden) weights*
11. self.W2 += self.z2.T.dot(self.o\_delta) *# adjusting second set (hidden --> output) weights*

This will lead you to the following result which illustrates that with every next epoch, the loss is being reduced. Here, the iterations are stopped at epoch 150 as the loss doesn't show any significant decrease post this.

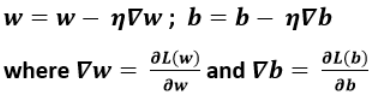




Optimization Algorithms

**Gradient Descent**

In the previous resource, we have learned how a basic neural network works. We observed that during the back propagation the weights are adjusted using the loss and an optimization algorithm.

In general, gradient descent can be considered as the base optimization algorithm. For updating the weights, it follows the following update rule:

For a simple linear regression with equation y = mX + c, having parameters m and c, the above rules can be implemented in python as shown below:

1. for i in range(epochs):
2. Y\_pred = m\*X + c *# The current predicted value of Y*
3. D\_m = (-2/n) \* sum(X \* (Y - Y\_pred)) *# Derivative wrt m*
4. D\_c = (-2/n) \* sum(Y - Y\_pred) *# Derivative wrt c*
5. m = m - L \* D\_m *# Update m*
6. c = c - L \* D\_c *# Update c*

While considering gradient descent, there are multiple drawbacks such as getting stuck at a local minimum, slow process due to the consideration of the complete data set at every epoch, etc. Therefore, many variants of gradient descent have been invented which we are going to discuss next.

**Variants of gradient descent algorithm**

# Stochastic Gradient Descent: The first variant of gradient descent is the Stochastic Gradient Descent where all the process remains similar to gradient descent except that only one sample is taken into consideration at every epoch which reduces the time taken to find the optimum value.

It follows a greedy approach and due to its frequent fluctuations and noisy updates, it approximates the gradient by overshooting in different directions and thus trying to find the optimum value.

# Mini-Batch Gradient Descent: Mini-Batch Gradient Descent neither takes a complete data set at every epoch nor it takes one sample at every epoch. Instead what it does is take a specified size of the batch defined by the user. This helps in better convergence as the updates tackle less noise and also suits the hardware by finding the optimum value in given memory (governed by batch size).

# Momentum: The momentum extends the functionality of SGD by accelerating its movement when stuck in a local minimum. With the introduction of velocity in SGD, the progress takes places only in relevant direction and thus skipping the oscillations from irrelevant directions which actually degrades the result in SGD. Movement in a relevant direction also leads to an increase in the step length. Due to the progress in relevant direction, the convergence takes place quickly, however, with increased velocity and bigger step it may happen that this algorithm can even bypass an optimum minimum.

# Nesterov Accelerated Gradient (NAG): The drawback of the momentum is that it may miss an optimum minimum due to high velocity and thus taking larger steps. This problem is identified and resolved by Yurii Nesterov who suggested that an optimization algorithm should initiate with a high step in the direction of previously accumulated gradient. Next, it should measure the gradient and make a correction which should result in an update. This way, one can prevent from going too fast and such a foresight update can result in finding the optimum minimum.

**Algorithms with adaptive learning rate:** One of the most difficult hyper-parameter to set in neural networks is the learning rate. Out of all the previously discussed gradient descent variants, momentum algorithm is found to mitigate this problem but with an introduction of another hyper parameter, momentum. There are various algorithms which have been introduced recently to adapt the learning rate for the model parameters.

# Adagrad : Adagrad individually adapts the learning rate for all the model parameters. For instance, a high learning rate is considered for the parameters with smaller updates whereas a small learning rate is considered for the parameters with larger updates. In other words, learning rate decreases rapidly for the parameters with the largest partial derivative of the loss and vice-versa.

# The parameter weights update follows the rule given below:

where v(t) accumulates the running sum of the square of the gradients. Due to the accumulation of these squared gradients from the beginning of the training, an excessive decrease in learning rate can take place.

# Adadelta : Adadelta overcomes the problem of Adagrad’s squared gradients by limiting its window of the squared gradient to some fixed size.

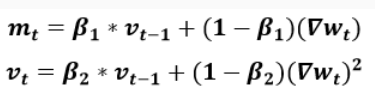
# RMSProp: RMSProp controls the rapid growth of v(t) and decreasing learning rate which we see in the Adagrad, by changing the gradient accumulation into an exponentially weighted moving average.

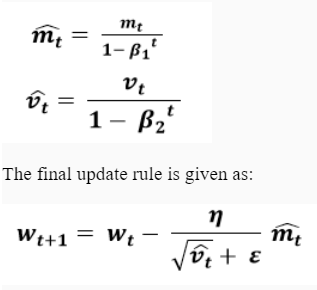
Given below is the parameter weights update rule:

As can be observed from the above equations, v(t) represents an exponentially decaying average of all the previous squared gradients. It is observed that RMSProp may have a high bias in early training.

# Adaptive Moment Estimation (Adam)

Adaptive Moment Estimation (Adam) is built on the combined concept of RMSProp and Momentum algorithms. Further, it also adds the bias-correction through its second moment which is a drawback we noticed in the early stages of training in RMSProp.

As you can recall, in momentum optimization algorithm, the current gradient is computed using the previous gradients whereas, in RMSProp learning rate is adjusted based on previous gradients. With these two benefits combined, Adam deals with adaptive learning rate as well as adaptive momentum.Given below is the parameter weight update rule:

Here, RMSProp ensures that v(t) does not grow rapidly to avoid the chances of decaying learning rate and m(t) based on momentum ensuring that it calculates the exponentially decaying average of previous gradients, not the current gradient.

The second moment can be given as follows:

It rectifies most of the optimization algorithm problems such as vanishing learning rate, slow convergence, and high variance.

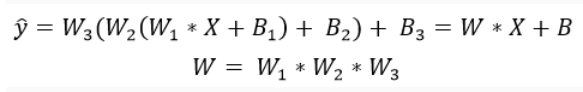
# Conclusion

In practice, adaptive learning algorithm like Adam provides fast convergence and better results (especially in the optimization end when gradient becomes sparser). It also mitigates the need of learning rate tuning. On the flip side, gradient variant like SGD tries to find the best minimum but consequently takes more time relative to other algorithms.

Activation functions

Recall that previously we had encountered and used an activation function to add non-linearity to the hidden layer nodes. Let us now try to understand activation functions in detail and explore the various functions.

Consider a neural network with two hidden layers having weights from input layer till the output layer as W1, W2, and W3 respectively. The equation for such a network can be represented as follows:

Here, no activation function has been used or in other words we can say that a linear activation function is present in the network. So, the final output corresponds to a linear approximation of the weights and biases and thus can only represent a linear relationship.

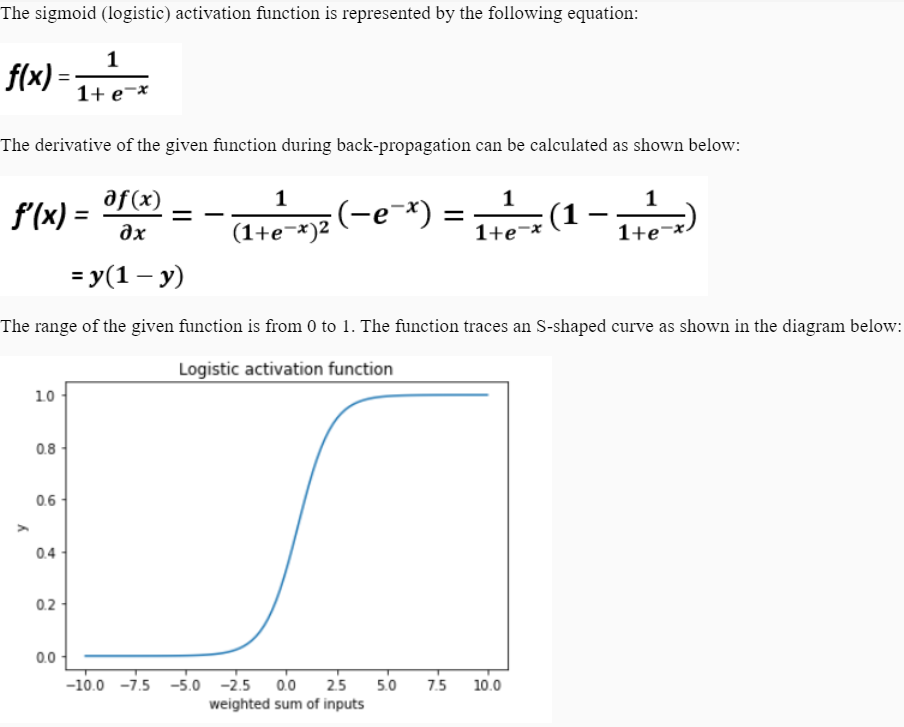
To introduce a non-linear relationship in the network, we are going to learn the following non-linear activation functions:

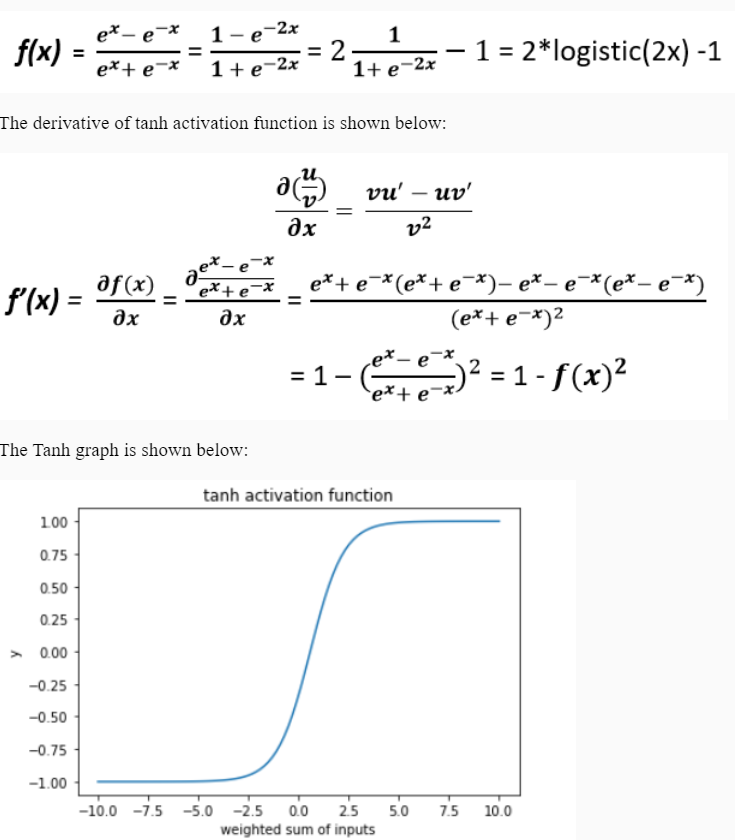
1. Sigmoid
2. Tanh
3. Rectified Linear Unit (ReLU)
4. Leaky Rectified Linear Unit (Leaky ReLU)

Sigmoid Activation Function :Logistic (Sigmoid) activation function suffers with the below given drawbacks:

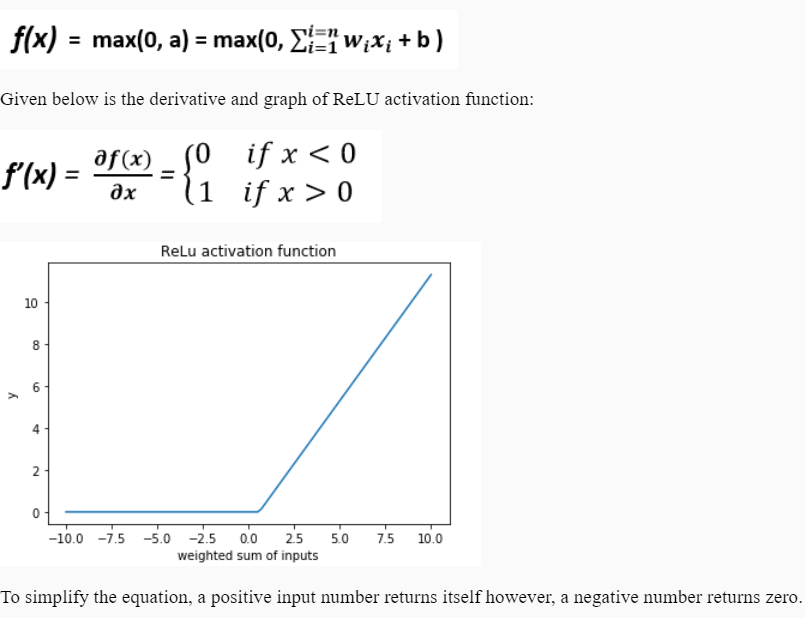
A saturated sigmoid neuron can cause the gradient to vanish.

* The output is always positive and hence it doesn’t follow a zero-centered approach where the output can flow in either positive or negative direction, thus making the optimization hard.
* The convergence gets slower due to the factor e-x which requires a heavy computation.



Tanh (Hyperbolic tangent): As we noted down that logistic function slows down learning due to its non-zero mean, therefore, an anti-symmetric activation function is needed and that’s where tanh comes into picture. The activation function tanh is built upon sigmoid activation function as can be understood from the given mathematical equation and graph:

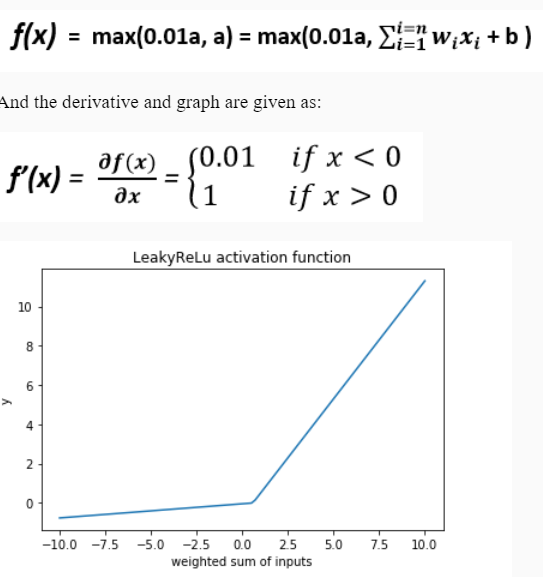
The advantage that we receive from this function is that it is a zero-centered function as it ranges between -1 to 1. However, it still lags due to vanishing gradient problem and computation issues.

ReLU (Rectified Linear Unit): ReLU is quite faster and much more efficient in learning for high-dimensional data. It doesn’t require intensive computation as it computes the function f(x)= max(0, x) which simply thresholds input matrix to zero. It doesn’t saturate i.e. no vanishing or exploding gradient and hence suitable for deep networks (multiple cascaded layers).

Mathematically, it is represented as:

On the other side, ReLU is also responsible for creating dying neurons (a dying neuron is the one which never fires). Dying neurons are generated when no gradient flows backward through ReLU. There are two solutions to mitigate this problem:

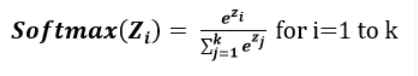
* Assign small learning rate
* Use leaky ReLU which allows a small negative slope when the unit is not active

Leaky ReLU: As mentioned in the ReLU section, leaky ReLU is created to fix the dying neurons problem of ReLU. With a negative weighted sum of inputs, it introduces a small slope to keep the update alive for the neurons.

Mathematically, it is defined as:

From the result we can observe that, a positive input number returns itself whereas a negative input number returns a negative value scaled by 0.01 (or any other small value).

By using this activation function, we avoid saturation issue as it works for both negative and positive sides, no vanishing gradient, and easier computation.

Softmax activation function results in probability values for a multiclass classification problem, the same way as sigmoid works for a binary classification problem.

Generally, it is used in the output layer as it is where we are required to get the probabilities of each class.

# Conclusion

You can start with ReLU activation function for your hidden layers and switch to others while performing hyperparameter tuning if you observe any other function performing better than ReLU. However, to avoid dead neurons, leaky ReLU should be preferred with all the advantages of ReLU. For the output layer, you should choose an activation function depending upon the problem you're solving.

Variants of Neural Networks

So far we have discussed the basic nuances of a neural network. The concept of neural nets ranges back to the 20th century and since then multiple variants of neural nets have been invented. In this course, we are going to learn about the following variants with the help of a few of their real-world applications:

1. **Multi-Layer Perceptron (MLP):** Used in regression, classification, etc.
2. **Convolutional Neural Network (CNN):** Used in image-processing, time-series prediction, etc.
3. **Recurrent Neural Network (RNN):** Used in next word prediction, speech recognition, etc.
4. **Auto-Encoders:** Used in data compression, watermark removal, etc.