Deep Learning

**Index**

| 1. Introduction to Deep Learning | 3 |
| --- | --- |
| 1. Introduction to Neural Network and Perceptron | 12 |
| 1. Gradient Descent | 20 |
| 1. Modeling Artificial Neural Network | 29 |

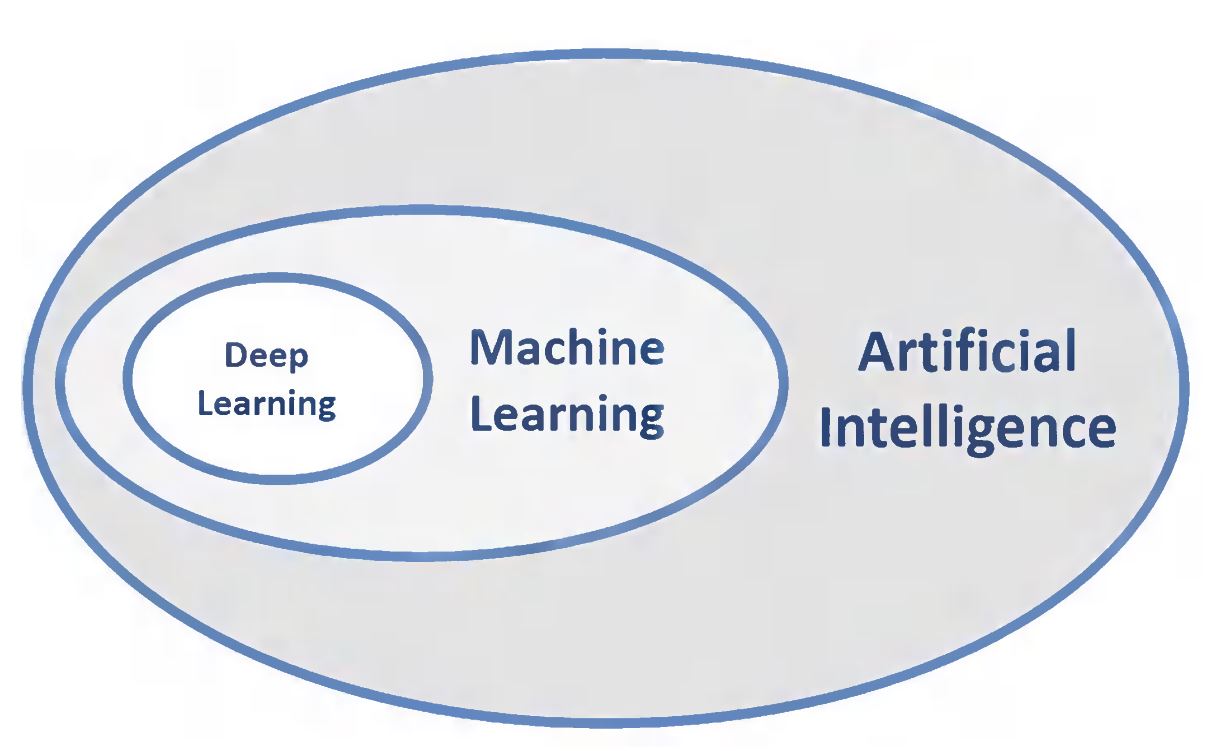
**01 Introduction to Deep Learning**

Before diving into Deep learning, let’s quickly refresh Machine Learning. Machine Learning is a field of study that provides machines the ability to learn from data without being programmed explicitly.

Machine learning has a lot of applications like fraud detection, churn prediction, and many more. Machine learning really does it’s a task well, but still, some human intervention is required especially for feature selection and engineering. Feature selection does play an important role in the success of a machine learning algorithm. Domain expertise plays an important role in selecting the right feature.

Now consider scenarios where you have a huge amount of data, or consider you have raw unstructured data like image, audio, and text. Feature extraction and selection in such scenarios can be hectic and time-consuming. On top of that, one simply cannot be a domain expert in multiple fields. Here enters Deep learning.

**Deep learning** is a subset of machine learning which is completely based on artificial neural networks. Deep Learning is a field of study which is focused on making machines mimic the human brain. In deep learning, we don’t need to explicitly program everything. No explicit feature selection and domain expertise are required here, still one is free to select features based on domain knowledge, that’s just an added advantage.



**1.1 Applications of deep learning**

* Self-driving cars
* Virtual assistance
* Language translation
* Face recognition

**1.2 Deep learning frameworks**

1. **TensorFlow** is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library that is used for machine learning applications like neural networks. Tensorflow provides both low and high-level APIs.
2. **Keras** is an open-source neural network library written in Python. It is capable of running on top of TensorFlow. It is designed to enable fast experimentation with deep neural networks. Keras provides only high-level APIs.

The Tensorflow version that we will use is **2.0.** The major difference between Tensorflow 1.0 and 2.0 is that version 2.0 has adopted the Keras API as a standard method of creating neural networks.

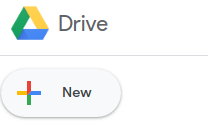
Other Deep learning frameworks are **PyTorch, Theano,** etc.

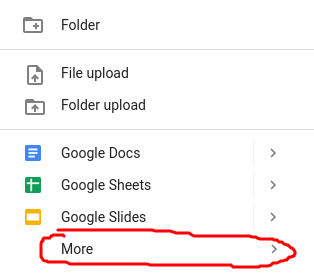
**1.3 Introduction to Google Colab**

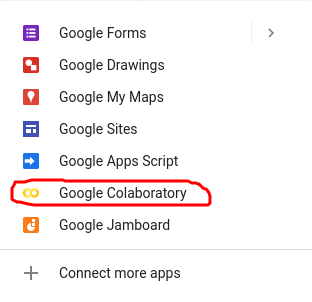
Google Collaboratory is a Jupyter notebook environment that needs no setup to use and runs completely on the cloud.

Following are the steps for creating google collab notebook -

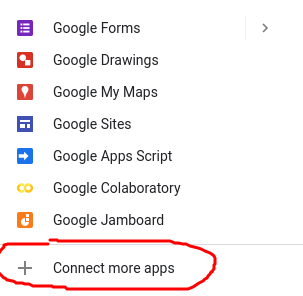
1. One must have a Google account in order to use Google Collaboratory.
2. Open Google drive -> new -> more -> Google Collaboratory.

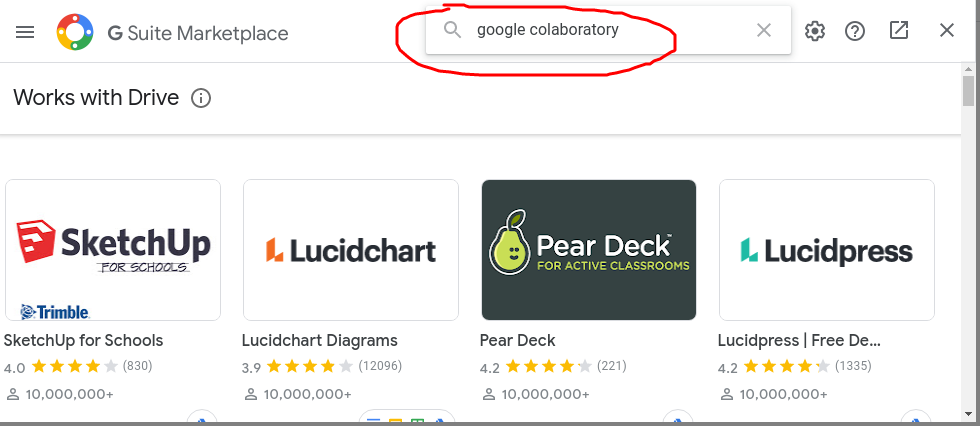




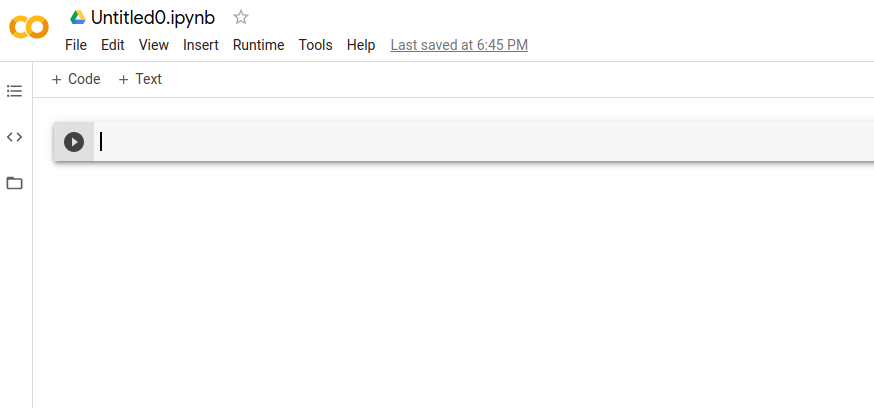


1. If Google Colaboratory is not visible, new -> more -> Connect more apps and search for Google Colaboratory.

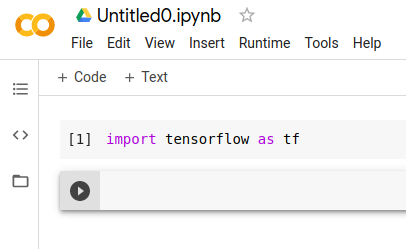




A Google Colaboratory notebook is ready to be used.

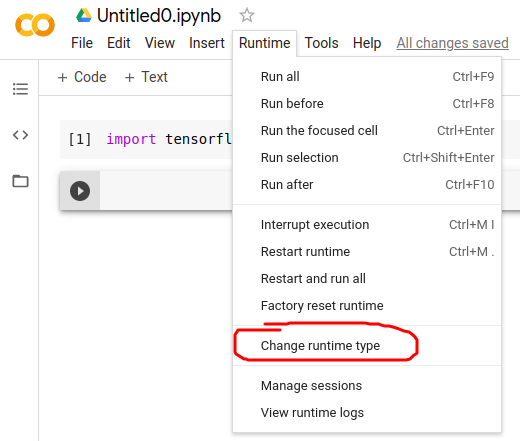


This notebook works exactly like the Jupyter notebook. One can write any Python code using a **code** cell, and markdown code using a **text** cell. Many Data analysis and Machine learning specific libraries like **Numpy, Pandas, Scikit Learn, Matplotlib, Seaborn, Scipy** are already available for use. Deep learning framework - **Tensorflow** is also available for use.



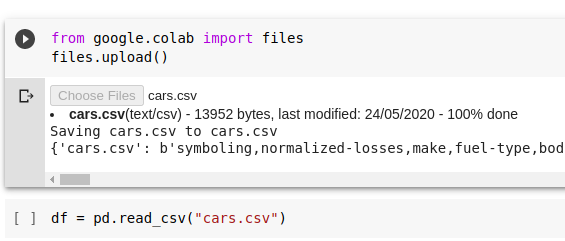
**Note** -

* One can also create a Google Collab notebook directly through this link - ***https://colab.research.google.com/*.**
* If one wishes to continue offline on a local system can install Tensorflow using the following command - ***pip install tensorflow****.* (Given - Machine must have a configuration that is suitable for training neural networks).
* Google Collab allows one to change runtime to **GPU** and **TPU**, which is suitable for training neural networks.

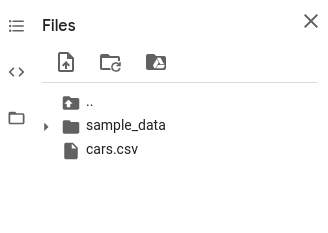


**Loading data to Google Collab -**

* One can simply load remote data using **read\_csv()** function of **Pandas** library and pass the url - *pd.read\_csv(csv\_file\_url).*
* One can also load a CSV file from the local system using the following code.

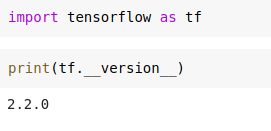


* Uploaded files are available in the files section on the left-hand side. Some sample\_data is also available for use.

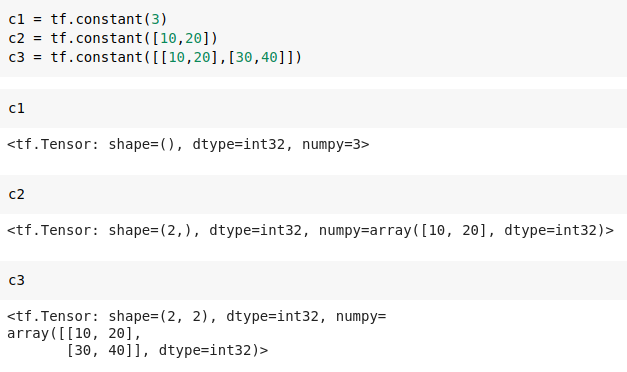


**1.4 Tensorflow basic syntax**

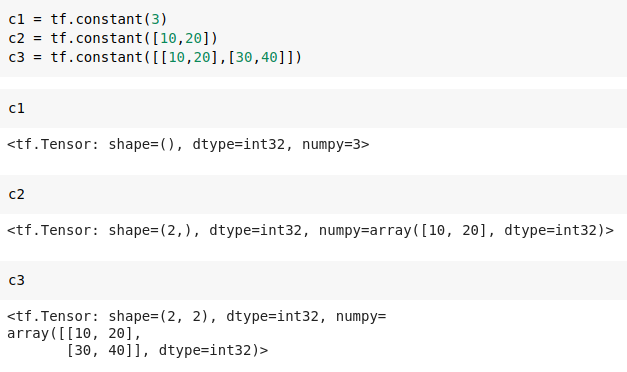
* Import and check the Tensorflow version.



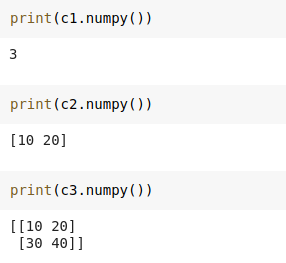
* Creating **Constants** using *tf.constant()* method.



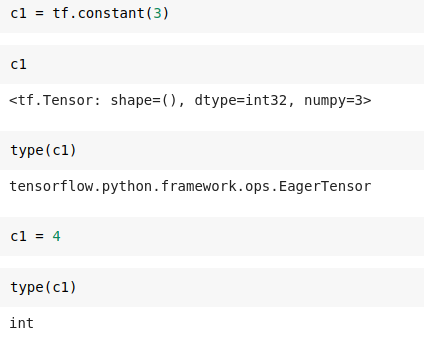
* A constant created is of type **Tensor**. A **Tensor** object can be a scalar, vector, or matrix.



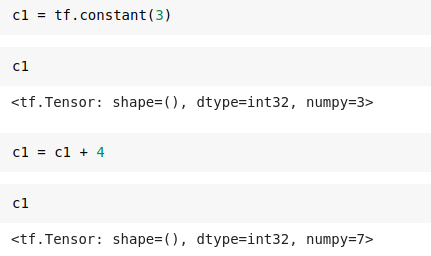
* Printing the NumPy version of the tensor object using *numpy()* function.



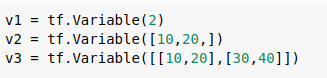
* A new value cannot be assigned to a Tensorflow constant.



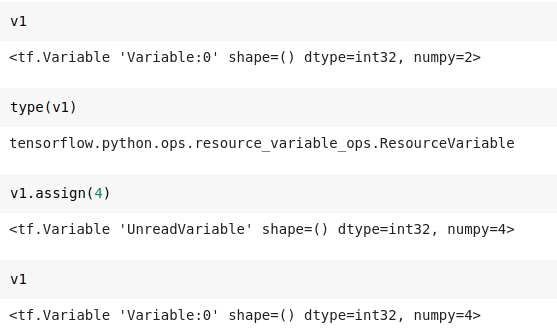
* A new value can be assigned only through operations.



* Creating **Variables** using *tf.Variable()* method.



* Tensorflow variables are of type **Variable** object. A new value can be assigned to a tensorflow variable using the *assign()* method.

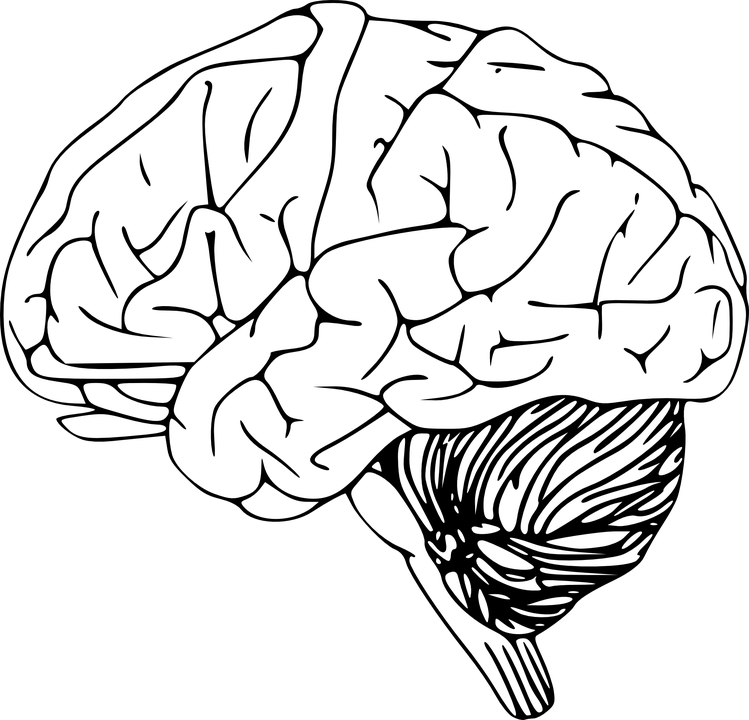


**02 Introduction to Neural Network and Perceptron**

**2.1 The Human Brain**

Humans perform complex tasks like vision, motor control, or language understanding very well. All the day-to-day activities done by a human is performed by the human brain.

The human brain is composed of neurons, glial cells, neural stem cells, and blood vessels.



**How does the brain work?**

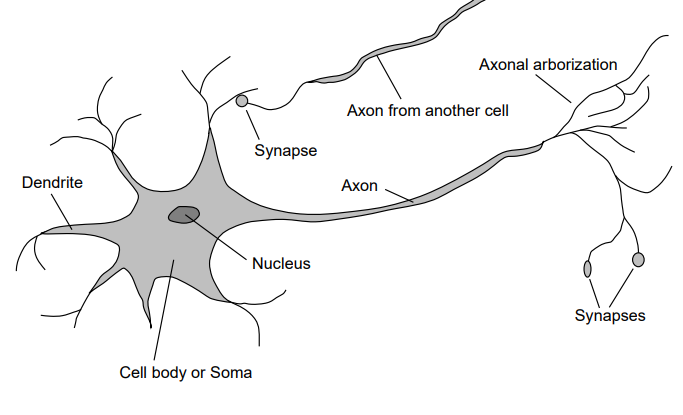
The brain works like a big computer. It processes information that it receives from the senses and body and sends messages back to the body. But the brain can do much more than a machine can: humans think and experience emotions with their brain, and it is the root of human intelligence.

The human brain is roughly the size of two clenched fists and weighs about 1.5 kilograms. From the outside, it looks a bit like a large

with folds and crevices. Brain tissue is made up of about 100 billion nerve cells (neurons) and one trillion supporting cells which stabilize the tissue.

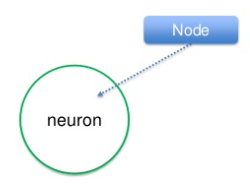
The neurons are ‘excitable’, this means that they produce electrical events called action potentials, which are also known as nerve impulses, or spikes. Nerve impulses are the basic currency of the brain. They allow neurons to communicate with each other, computations to be performed, and information to be processed.

When a neuron spikes it releases a neurotransmitter, a chemical that travels a tiny distance across a synapse before reaching other neurons. Any time a neuron spikes, neurotransmitters are released from hundreds of its synapses, resulting in communication with hundreds of other neurons. Neurons communicate through synapses.

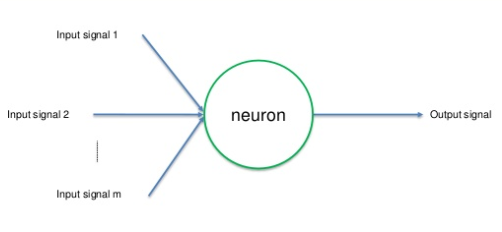


**2.2 Artificial neurons**

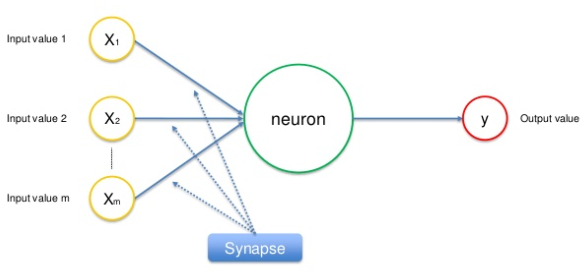
A circle represents a single Neuron.



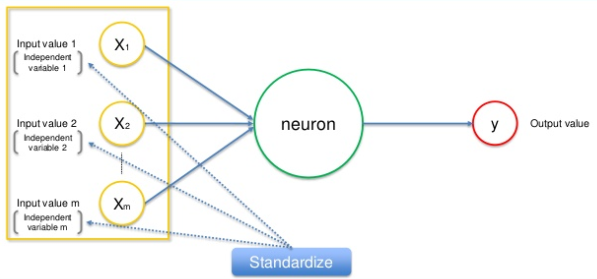
The input lines to this circle represent the signals a neuron gets. This signal can get more than one signal.



These signals are carried with the help of synapses.



After receiving the signals through the synapses the neuron would predict the output.



**Biological Neuron vs. Artificial Neuron**

The biological neuron is analogous to artificial neurons in the following terms:

| **Biological Neuron** | **Artificial Neuron** |
| --- | --- |
| Cell Nucleus (Soma) | Node |
| Dendrites | Input |
| Synapse | Weights or interconnections |
| Axon | Output |

Computational models inspired by the human brain:

* Massively parallel, distributed system, made up of simple processing units (neurons)
* Synaptic connection strengths among neurons are used to store the acquired knowledge.
* Knowledge is acquired by the network from its environment through a learning process

**2.3 The perceptron**

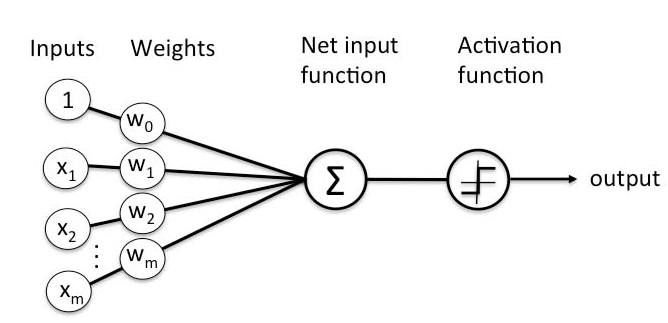
A perceptron is a neural network unit (an artificial neuron) that does certain computations to detect features or business intelligence in the input data. Perceptron was introduced by Frank Rosenblatt in 1957.

A Perceptron is an algorithm for supervised learning of binary classifiers. This algorithm enables neurons to learn and processes elements in the training set one at a time.

It’s a type of linear classifier, which is a classification algorithm that makes predictions based on a linear predictor function which combines a set of weights with the feature vector.

A Linear classifier as defined trains data which is classified into corresponding classes. If you are applying classification for the two classes then all the training data must be in these two classes.

On the other hand, a binary classifier defines that there should be only 2 classes for classification. Therefore, the basic Perceptron algorithm is used for binary classification and all the training examples should lie in these classes. The basic unit in the Neuron is called the Perceptron.



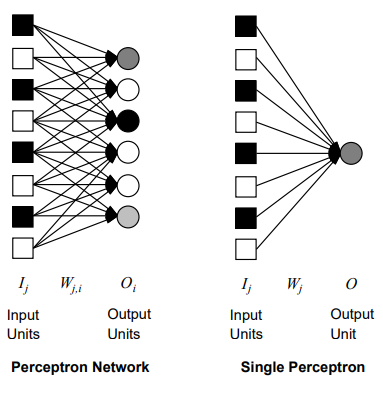
Following are the major components of a perceptron:

* **Input**: All the features become the input for a perceptron. We denote the input of a perceptron by [x1, x2, x3, ..,xn], where x represents the feature value and n represents the total number of features. We also have a special kind of input called the bias. In the image, we have described the value of the bias as w0.
* **Weights**:The values that are computed over the time of training the model. Initially, we start the value of weights with some initial value and these values get updated for each training error. We represent the weights for perceptrons by [w1,w2,w3,.. wn].
* **Bias**:A bias is an additional parameter that is adjusted along with the weighted sum of the inputs to the neuron. Bias helps the model in a way that it can fit best for the given data.
* **Weighted summation**: Weighted summation is the sum of the values that we get after the multiplication of each weight [wn] associated with each feature value [xn]. We represent the weighted summation by ∑wixi for all i -> [1 to n].
* **Step/activation function**: The role of the activation function is to convert the continuous value we get after the weighted summation of inputs into binary i.e 0 or 1.
* **Output**: The weighted summation is passed to the step/activation function and whatever value we get after computation is our predicted output.

There are two types of Perceptrons: Single layer and Multilayer.

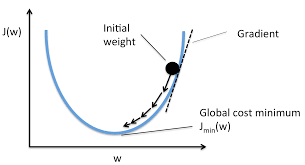
**Single-layer Perceptrons** can learn only linearly separable patterns.

**Multilayer Perceptrons** or **Feedforward neural networks** with two or more layers have greater processing power. We will learn feedforward neural networks in the coming chapters.



**03 Gradient Descent**

Gradient descent is an algorithm for finding the minimum of a differentiable function. Gradient descent is an iterative process, to find the minimum of a function we take steps proportional to the approximate gradient of the function at the current point.



The above image depicts a plot between w and J(w). Where J(w) is some differential function of w. The above plot depicts a convex function, meaning the function has only one minimum. In this scenario, one can easily reach the minima using gradient descent.

**3.1 The steps in gradient descent**

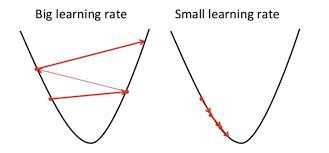
1. Start with any random **w** value.
2. Calculate gradient **G** of the function f(w) for that **w** value. A gradient is basically the slope which is either positive (moving upward) or negative (going downward).
3. Update the **w** value.

*w = w - nG*

Where **n** is the learning rate which depicts the magnitude of the step size. A

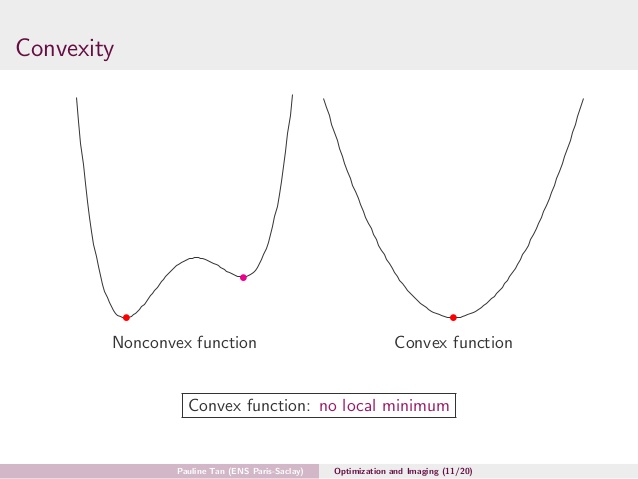
A small learning rate will take more time to reach the minimum than a larger learning rate. But one should be careful that a too-large learning rate might skip the minimum.

The above steps will repeat for **i** iterations. Both learning rate and no. of iterations are hyper-parameters. These hyperparameters must be set by the developer to fine-tune the entire process of finding the minimum.



**3.2 Convex vs non-convex function**

A convex function as discussed is the one which has only one minimum. But a non-convex function can have multiple minimums. Gradient descent works really well when the function is convex.



For a non-convex function, one might end up on local minima.

**3.3 Gradient descent in neural network**

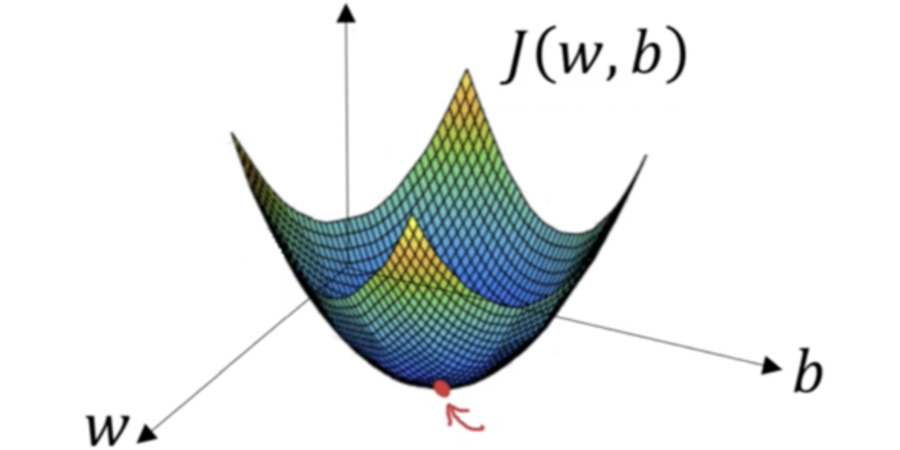
**Forward propagation**

In the previous chapter, we have seen how a single neuron can predict the output by performing the weighted sum of the inputs and passing the same through an activation function. This process is known as **forward propagation**.

But the output predicted with random weights cannot be accurate. A prediction is said to be more accurate when the value of the **loss function** for the given data is minimum. A loss function in an informal way is basically the difference between original and predicted value.

**Backward propagation**

Hence gradient descent can be used to find the optimal value of weight and bias that gives the minimum value of loss function. The process of updating the weight and bias with each iteration of gradient descent is known as **backward propagation**.



For **regression** problems, the most widely used loss function is the **mean squared error** and **log loss** for **classification** problems. We will discuss loss functions in detail in the next chapter.

**3.4 Types of gradient descent**

1. **Batch gradient descent**

* Start with any random **w** value.
* Calculate gradient **G** of the function f(w) for that **w** value, using all data points in the entire training data set.
* Update the **w** value.
* This will consume huge training time.

1. **Stochastic gradient descent**

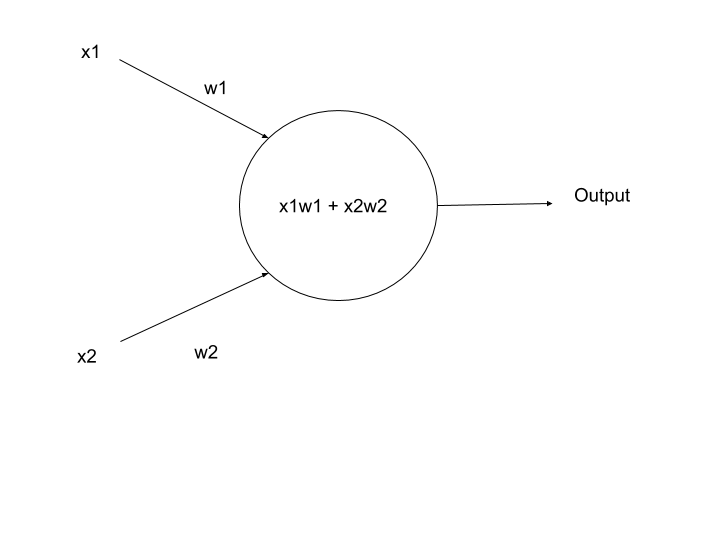
* Start with any random **w** value.
* Calculate gradient **G** of the function f(w) for that **w** value, using any one random data point from the entire training data set.
* Update the **w** value.
* Much faster than batch gradient descent.

1. **Mini-batch gradient descent**

Mini-batch gradient descent is a combination of the stochastic gradient descent and batch gradient descent. It splits the training data into small batches and performs the update on these batches. Therefore it results in a balance between the efficiency of batch gradient descent and the robustness of stochastic gradient descent.

**3.5 Regression using Single Neuron**

The following figure represents a single neuron for regression. Where **x1** and **x2** are inputs, **w1** and **w2** are weights multiplied with respective inputs. The final output is the weighted sum of both inputs. Since the output is continuous, there is no need for an activation function.

.

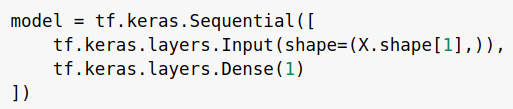
Gradient descent can be used to find the optimal value of the weights. The loss function in this situation can be the **mean squared error**. The other loss functions can be a **mean absolute error**, **root mean squared error**, **residual sum of squares,** etc.

**Practical single neuron regression**

1. Load dataset, perform basic cleaning like, **handling missing values**, **encoding categorical data**. Separating features ( X ) and target ( y ). Further splitting data for training ( X\_train, y\_train ) and testing ( X\_test, y\_test ).
2. Normalizing data is a very important step before training neural networks. It can be done using **scikit learn**.



1. Create the single neuron architecture, defining the input layer ( **Input** ) along with input **shape** and the output layer ( **Dense** ), using Tensorflow’s builtin **Keras API**.



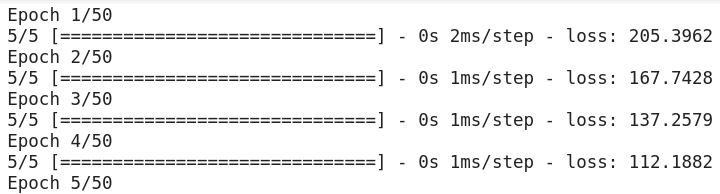
1. Define the optimizer and loss function using the **compile()** function. Here **sgd** stands for stochastic gradient descent and **mse** is mean squared error.



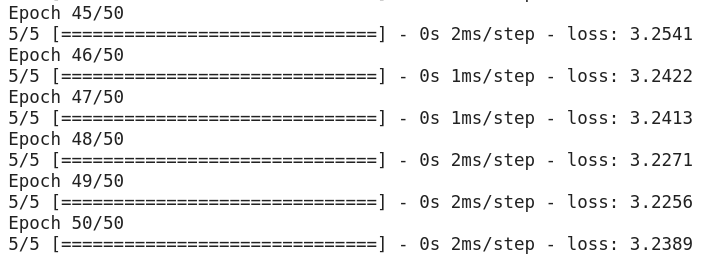
1. Train the model with X\_train and y\_train. Here **epochs** represent the number of times the weights should be updated to find the optimal weights.



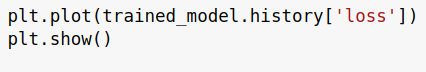
The output of the above code displays how the loss function value changes with each epoch.

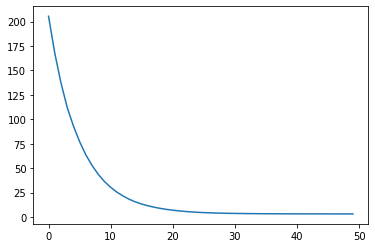


...



1. Plot the entire history of the loss function value, observe how the loss value decreased with each iteration.



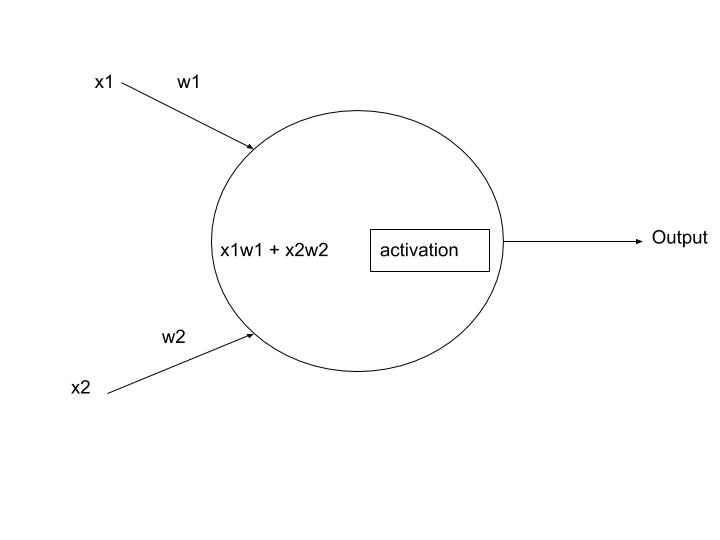


1. Finally, predict the output using **predict()** function.

****

**3.6 Classification using Single Neuron**

The following figure represents a single neuron for binary classification. Where **x1** and **x2** are inputs, **w1** and **w2** are weights multiplied with respective inputs. The weighted sum further is passed through an **activation function**, which is the final output. The **sigmoid function** can be used here which returns a probability value between 0 to 1.

****

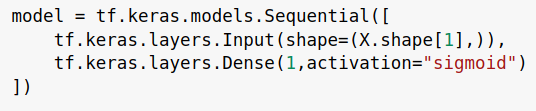
Gradient descent can be used to find the optimal value of the weights. The loss function in this situation can be the **binary cross-entropy** or **log loss** for binary classification.

**Practical single neuron classification**

1. Load dataset, perform basic cleaning like, **handling missing values**, **encoding categorical data**. Separating features ( X ) and target ( y ). Further splitting data for training ( X\_train, y\_train ) and testing ( X\_test, y\_test ).
2. Normalizing data is a very important step before training neural networks. It can be done using **scikit learn**.



1. Create the single neuron architecture, defining the input layer ( **Input** ) along with input **shape** and the output layer ( **Dense** ) along with the **activation function**.



1. Define the optimizer and loss function using the **compile()** function. Here **sgd** stands for stochastic gradient descent and the loss function is **binary\_crossentropy** for binary classification.

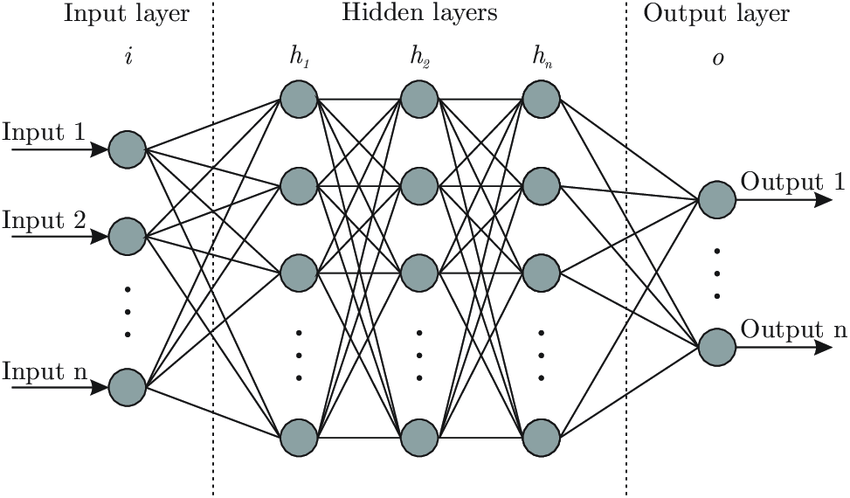


1. Further steps are similar to a regression, train the model, and predict the output.

**04 Modeling Artificial Neural Network**

**4.1 Artificial Neural Network**

An artificial neural network is a combination of multiple layers of neurons, each layer can have multiple neurons. These neurons are interconnected with the neurons in the other layers.



The **Input layer** has multiple neurons, one for each feature. The **Output layer** can either have one neuron for regression and binaryclassification or multiple neurons, one for each class in case of multi-classclassification. All the other layers in between are the **Hidden layers**.

The output from each layer of neurons becomes the input of the next layer of neurons.

Now that we have understood what an artificial neural network is, the question here is, *why do we need neural networks when a single neuron can be used for regression and classification?*

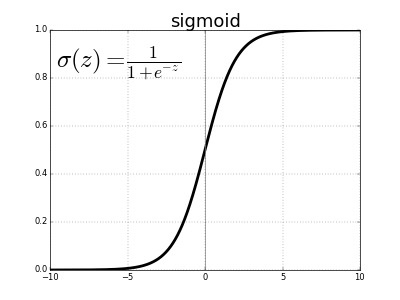
Well, a single neuron can only learn linearly separable patterns. Most of the time the data is non-linear in nature, this is where the neural networks come into the picture.

The first layer is a linear combination of the input. However, the **activation function** provides a non-linear output. The next layer combines the non-linear outputs of the first layer in a linear way i.e weighted sum of the values. Hence neural networks can fit nonlinear functions.

**4.2 Activation functions**

**Sigmoid function**

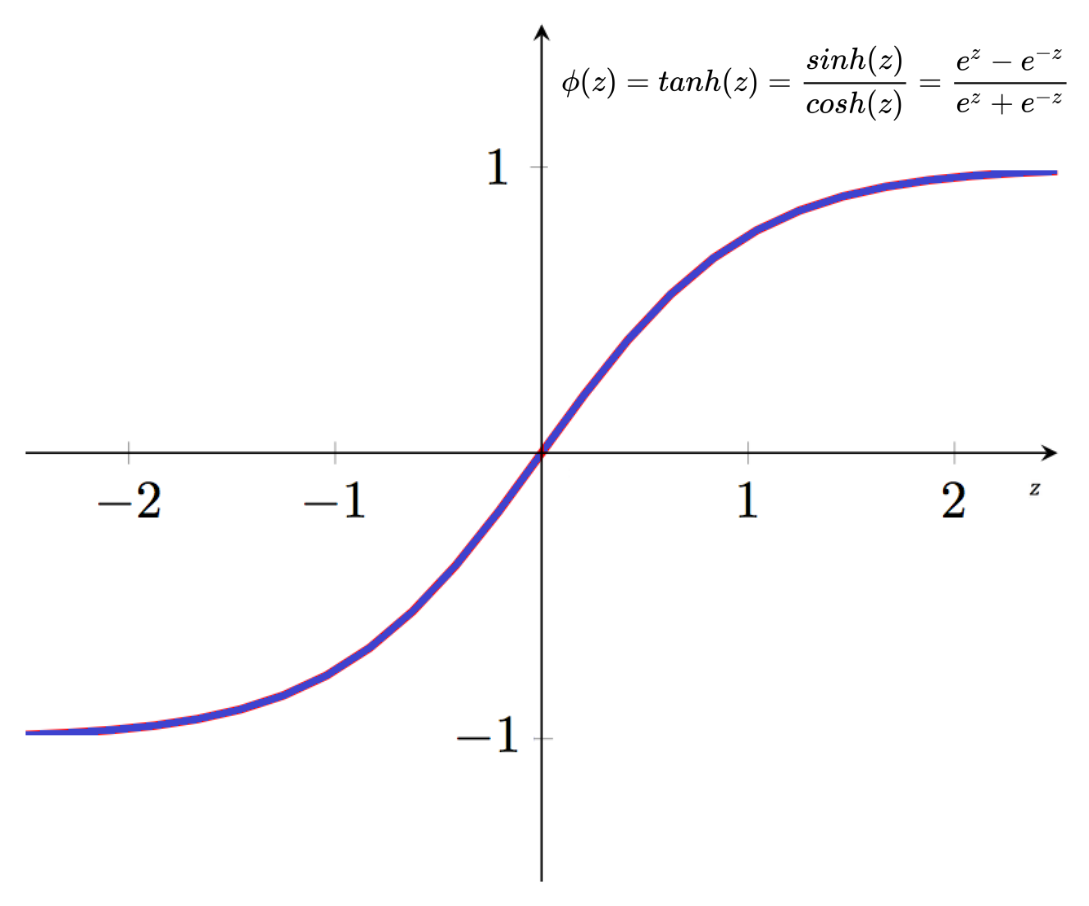
A sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve, the output of a sigmoid function is always between 0 - 1. Hence it can be used as an activation function for binary classification keeping a threshold value of 0.5, if sigmoid function output is >= 0.5 then it is considered a positive class 1 or else negative class 0.



The sigmoid function can be used as an activation function in the output layer, but should not be used in the hidden layers. The reason is, the range of the input value passed through the input layer lies between -1 to 1 ( after standardization ), if the sigmoid function is used in the hidden layers then the same value will be converted into a range of 0 to 1, breaking the uniformity of the neural network. This is where the hyperbolic tangent function comes into the picture.

**Hyperbolic tangent function**

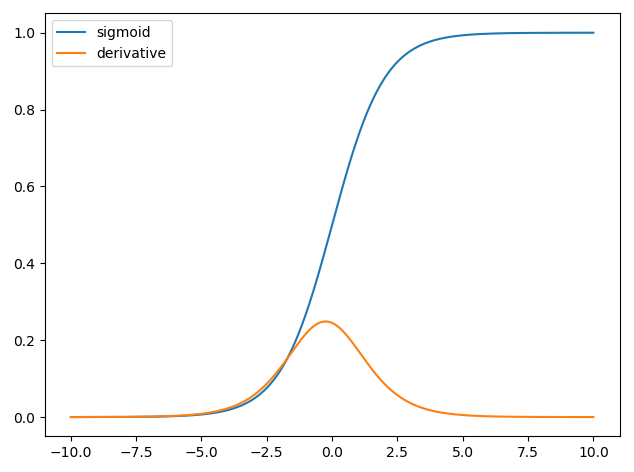
The hyperbolic tangent function is a mathematical function, this function is easily defined as the ratio between the hyperbolic sine and the cosine functions. The output of this function always lies between -1 to 1, making it suitable for the hidden layers to maintain the uniformity of neural networks.

****

But both sigmoid and hyperbolic tangent functions face the **vanishing gradient problem**.

The **vanishing gradient problem** is encountered when training artificial neural networks with gradient-based learning methods and backpropagation. In such methods, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that in the case of the sigmoid and hyperbolic tangent function, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.

The Maximum value of the derivative of the sigmoid function is just 0.25.



To avoid vanishing gradient problems, the Rectified Linear Unit function comes into the picture.

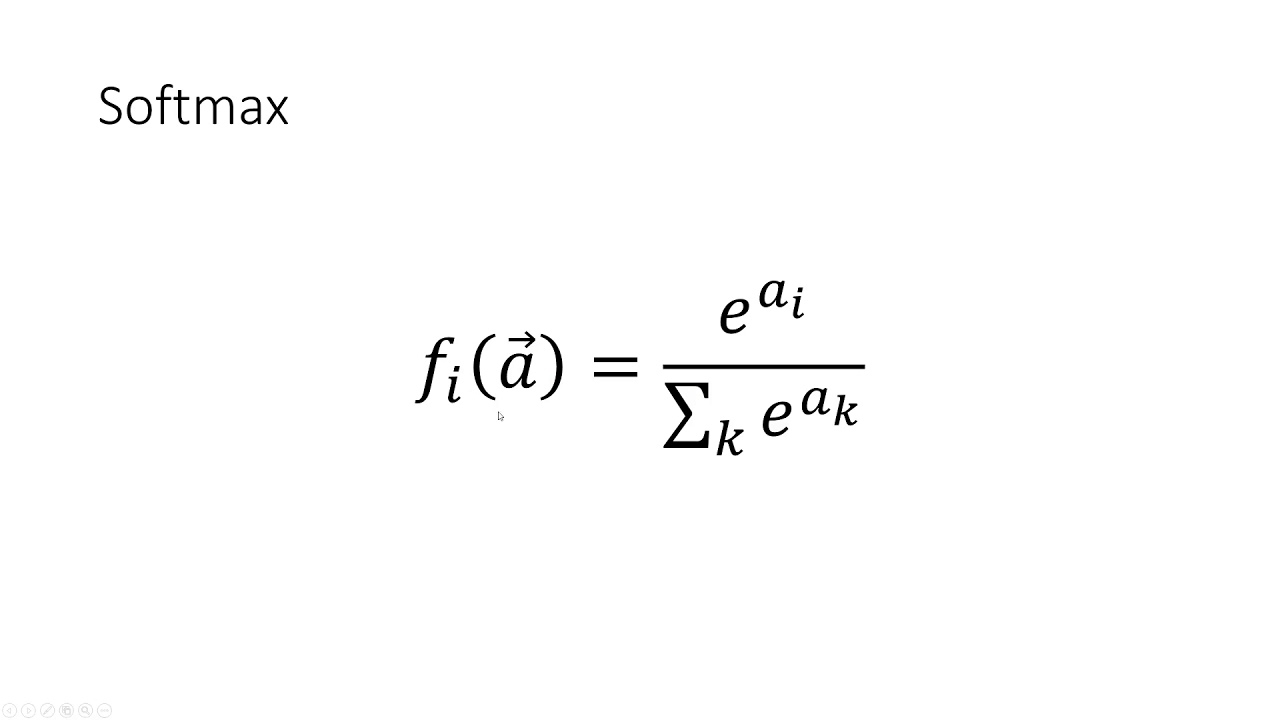
**Rectified linear unit**

The rectified linear activation function is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance.

****

**Softmax function**

The softmax function is a function that turns a vector of K real values into a vector of K real values that sum to 1. The input values can be positive, negative, zero, or greater than one, but the softmax transforms them into values between 0 and 1 so that they can be interpreted as probabilities. If one of the inputs is small or negative, the softmax turns it into a small probability, and if the input is large, then it turns it into a large probability, but it will always remain between 0 and 1.

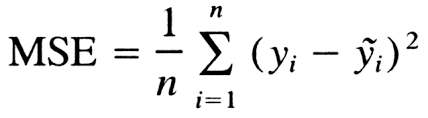
****

The softmax function can be used as an activation function in the output layer for the multi-class classification problems. The output layers must consist of n number of neurons for n different classes. The softmax function will return a probability value for each neuron in the output layer. Since each neuron represents a class, the final output will be the class with the highest probability value.

**4.3 Loss functions**

**Mean squared error**

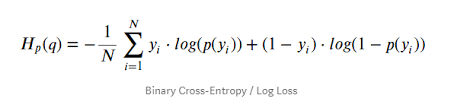
In statistics, the mean squared error (MSE) or mean squared deviation (MSD) of an estimator measures the average of the squares of the errors i.e the average squared difference between the estimated values and the actual value. The MSE is the loss function that measures the quality of an estimator, it is always non-negative, and values closer to zero are better. MSE can be used as a loss function for the **regression** problems.

****

**Binary cross-entropy**

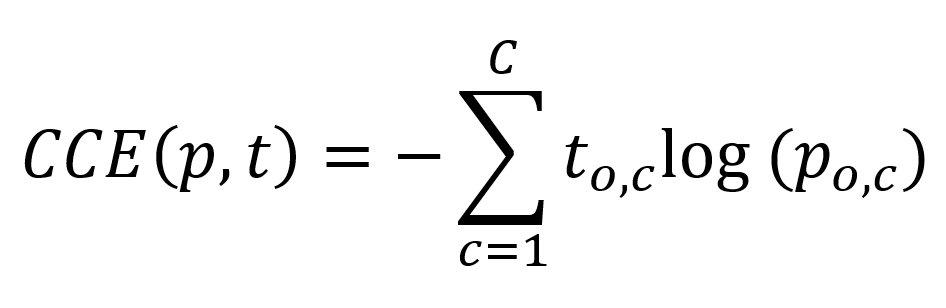
Since **MSE** measures the squared distance between actual and predicted value, it is not suitable for classification, as classification predicts binary or categorical output. Binary cross-entropy is used as a loss function for **binary classification**.

The cross-entropy value is high whenever there is a misclassification.

****

**Categorical cross-entropy**

Similarly, categorical cross-entropy is the loss function for **multi-class classification**.

****

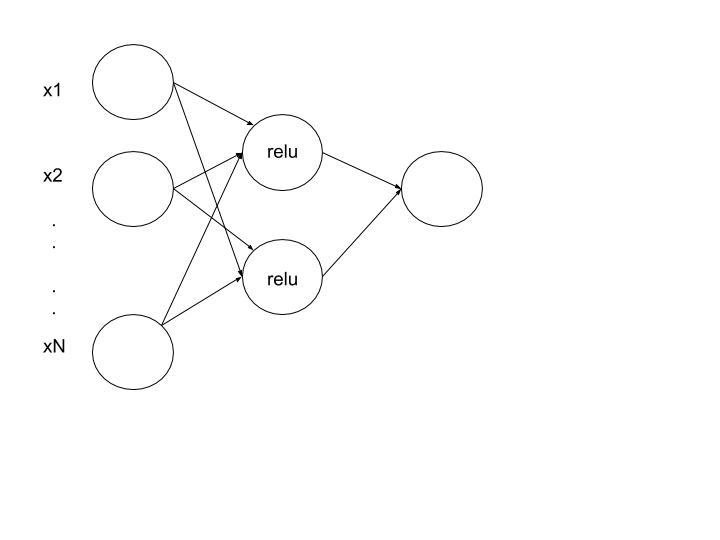
**4.4 Adama optimizer**

The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing.

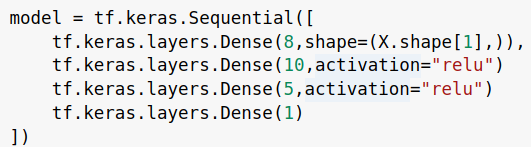
Stochastic gradient descent maintains a single [learning rate](https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/) for all weight updates and the learning rate does not change during training. On the other hand, **Adam** is an adaptive learning rate optimization algorithm. It is much faster than the stochastic gradient descent. Hence making is suitable for large neural networks.

**4.5 Regression using ANN**

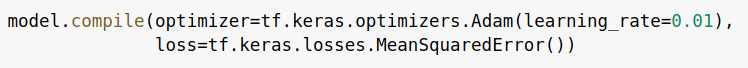
The following figure represents a neural network for regression. Input layer for **n** features. Hidden layer neurons must have activation functions like relu. The output layer must have only one neuron without any activation function.



Create the neural network architecture for regression, defining the first hidden layer ( **Dense** with k neurons where k>=1 ) along with **input** **shape = n** (no of features ),hidden layers with relu activation function ( **Dense** with k neurons where k>= 1 ) and the output layer ( **Dense** with 1 neuron ) without any activation function, using Tensorflow’s builtin **Keras API**.



Define the optimizer and loss function using the **compile()** function. An alternative optional way of passing the optimizer is to pass *tf.keras.optimizers.Adam()* object along with the custom learning rate. Similarly, loss function can be passed using *tf.keras.losses.MeanSquaredError()* object.

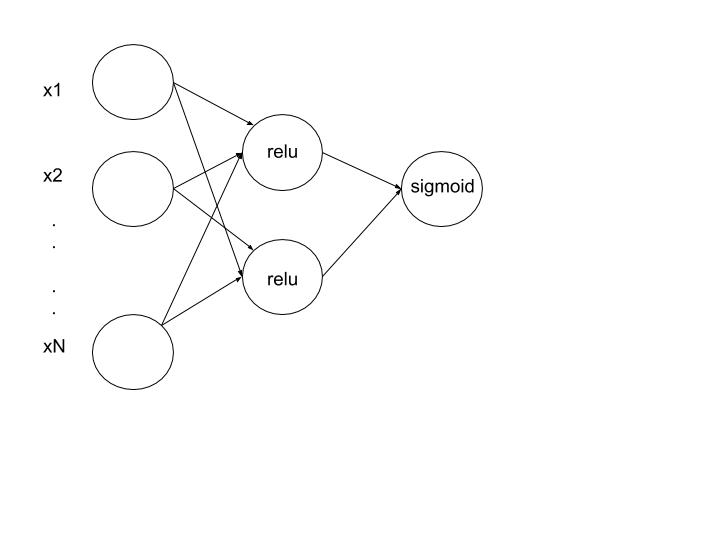


Train the model with X\_train and y\_train. Here **epochs** represent the number of times the weights should be updated to find the optimal weights. The **batch\_size** refers to the size of the batch (number of training data) for one iteration of adam optimizer.

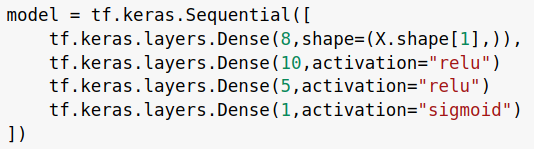


**4.6 Classification using ANN**

The following figure represents a neural network for binary classification. Input layer for **n** features. Hidden layer neurons must have activation functions like **relu**. The output layer must have only one neuron with a **sigmoid** activation function.



Create the neural network architecture for binary classification, defining the first hidden layer ( **Dense** with k neurons where k>=1 ) along with **input** **shape = n** (no of features ),hidden layers with relu activation function ( **Dense** with k neurons where k>= 1 ) and the output layer ( **Dense** with 1 neuron ) with sigmoid activation function, using Tensorflow’s builtin **Keras API**.



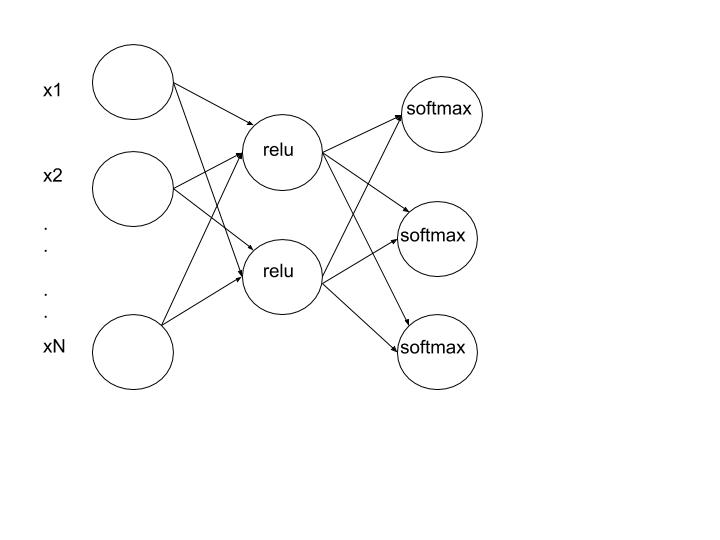
Define the optimizer and loss function using the **compile()** function. Here **adam** stands for adam optimizer and the loss function is **binary\_crossentropy** for binary classification.



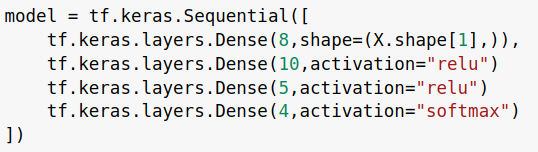
After that, the training process remains the same.

**4.7 Multi-class classification using ANN**

The following figure represents a neural network for multi-class classification. Input layer for **n** features. Hidden layer neurons must have activation functions like **relu**. The output layer must have **n** number of neurons for **n** number of classes with a **softmax** activation function.



Create the neural network architecture for multi-class classification, defining the first hidden layer ( **Dense** with k neurons where k>=1 ) along with **input** **shape = n** (no of features ),hidden layers with relu activation function ( **Dense** with k neurons where k>= 1 ) and the output layer ( **Dense** with **n** neurons for **n** classes ) with softmax activation function, using Tensorflow’s builtin **Keras API**.



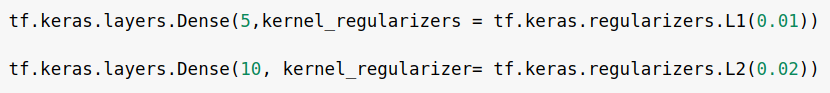
Define the optimizer and loss function using the **compile()** function. Here **adam** stands for adam optimizer and the loss function is **categorical\_crossentropy** for multi-class classification.



**4.8 Regularization**

Since neural networks are very complex, there is a high chance of overfitting. This is where regularization comes into the picture. We are already aware of L1 and L2 regularization techniques.

**L1 and L2 regularization**

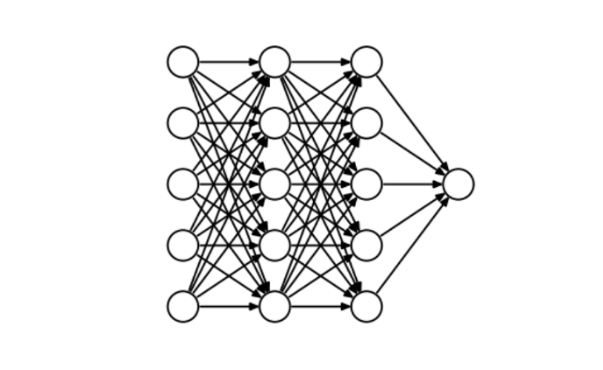
****

Lasso (L1) and Ridge (L2) regularization can be done by passing *kernel\_regularizer* argument to the Keras layers using *tf.keras.regularizers* along with the alpha value (hyperparameter).

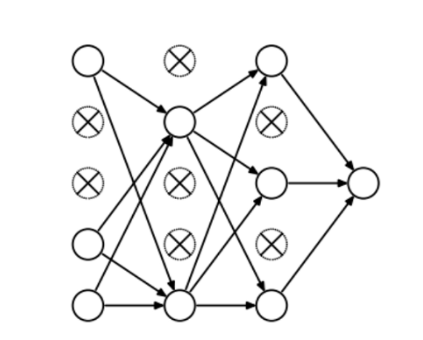
**Dropout** **technique**

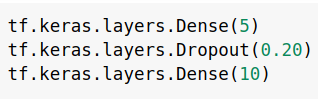
This is one of the most interesting types of regularization techniques. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.

To understand dropout, consider the following figure as our neural network architecture.



At every iteration, the dropout technique randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below.





The above code shows the probability *(0.20)* of choosing how many nodes should be dropped is the hyperparameter of the **dropout layer**.