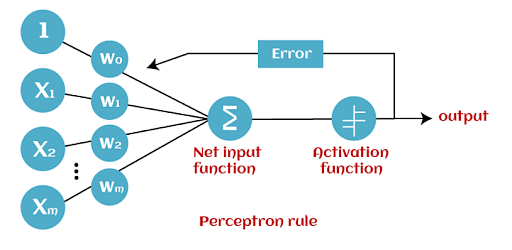
**1.**What is Perceptron?

* A machine-based algorithm used for supervised learning of various binary sorting tasks is called Perceptron. Furthermore, Perceptron also has an essential role as an Artificial Neuron or Neural link in detecting certain input data computations in business intelligence.
* A perceptron model is also classified as one of the best and most specific types of Artificial Neural networks. Being a supervised learning algorithm of binary classifiers, we can also consider it a single-layer neural network with four main parameters: input values, weights and Bias, net sum, and an activation function.

2. How does a Perceptron Works?

* Perceptron is considered a single-layer neural link with four main parameters. The perceptron model begins with multiplying all input values and their weights, then adds these values to create the weighted sum.
* Further, this weighted sum is applied to the activation function ‘f’ to obtain the desired output. This activation function is also known as the step function and is represented by ‘f.



* This step function or Activation function is vital in ensuring that output is mapped between (0,1) or (-1,1). Take note that the weight of input indicates a node’s strength. Similarly, an input value gives the ability the shift the activation function curve up or down.
* Step 1: Multiply all input values with corresponding weight values and then add to calculate the weighted sum. The following is the mathematical expression of it:
* ∑wi\*xi = x1\*w1 + x2\*w2 + x3\*w3+……..x4\*w4
* Add a term called bias ‘b’ to this weighted sum to improve the model’s performance.
* Step 2:  An activation function is applied with the above-mentioned weighted sum giving us an output either in binary form or a continuous value as follows:
* Y=f(∑wi\*xi + b)

**3. What is the Perceptron algorithm?**

* The [Perceptron algorithm](https://en.wikipedia.org/wiki/Perceptron) is a two-class (binary) classification machine learning algorithm.
* It is a type of neural network model, perhaps the simplest type of neural network model.
* It consists of a single node or neuron that takes a row of data as input and predicts a class label. This is achieved by calculating the weighted sum of the inputs and a bias (set to 1). The weighted sum of the input of the model is called the activation.
* **Activation** = Weights \* Inputs + Bias
* If the activation is above 0.0, the model will output 1.0; otherwise, it will output 0.0.
* **Predict 1**: If Activation > 0.0
* **Predict 0**: If Activation <= 0.0
* Given that the inputs are multiplied by model coefficients, like linear regression and logistic regression, it is good practice to normalize or standardize data prior to using the model.
* The Perceptron is a linear classification algorithm. This means that it learns a decision boundary that separates two classes using a line (called a hyperplane) in the feature space. As such, it is appropriate for those problems where the classes can be separated well by a line or linear model, referred to as linearly separable.
* The coefficients of the model are referred to as input weights and are trained using the stochastic gradient descent optimization algorithm.
* Examples from the training dataset are shown to the model one at a time, the model makes a prediction, and error is calculated.
* The weights of the model are then updated to reduce the errors for the example. This is called the Perceptron update rule. This process is repeated for all examples in the training dataset, called an [epoch](https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/). This process of updating the model using examples is then repeated for many epochs.
* Model weights are updated with a small proportion of the error each batch, and the proportion is controlled by a hyperparameter called the learning rate, typically set to a small value. This is to ensure learning does not occur too quickly, resulting in a possibly lower skill model, referred to as premature convergence of the optimization (search) procedure for the model weights.

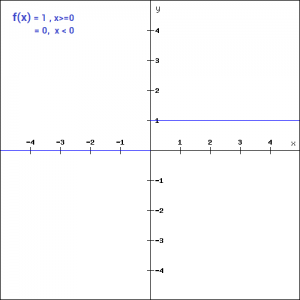
weights(t + 1) = weights(t) + learning\_rate \* (expected\_i – predicted\_) \* input\_i

* Training is stopped when the error made by the model falls to a low level or no longer improves, or a maximum number of epochs is performed.
* The initial values for the model weights are set to small random values. Additionally, the training dataset is shuffled prior to each training epoch. This is by design to accelerate and improve the model training process. Because of this, the learning algorithm is stochastic and may achieve different results each time it is run. As such, it is good practice to summarize the performance of the algorithm on a dataset using repeated evaluation and reporting the mean classification accuracy.
* The learning rate and number of training epochs are hyperparameters of the algorithm that can be set using heuristics or hyperparameter tuning.

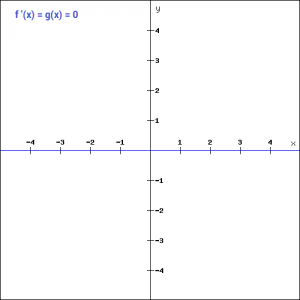
**Q4. How many activation functions are available? Explain all of them.**

1. Binary Step Function

* The first thing that comes to our mind when we have an activation function would be a threshold based classifier i.e. whether or not the neuron should be activated based on the value from the linear transformation.
* In other words, if the input to the activation function is greater than a threshold, then the neuron is activated, else it is deactivated, i.e. its output is not considered for the next hidden layer. Let us look at it mathematically-
* f(x) = 1, x>=0
* = 0, x<0



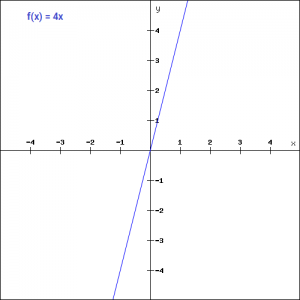
* This is the simplest activation function, which can be implemented with a single if-else condition in python
* def binary\_step(x):
* if x<0:
* return 0
* else:
* return 1
* binary\_step(5), binary\_step(-1)
* **Output:**
* (5,0)
* The binary step function can be used as an activation function while creating a binary classifier. As you can imagine, this function will not be useful when there are multiple classes in the target variable. That is one of the limitations of binary step function.
* Moreover, the gradient of the step function is zero which causes a hindrance in the back propagation process. That is, if you calculate the derivative of f(x) with respect to x, it comes out to be 0.
* f'(x) = 0, for all x



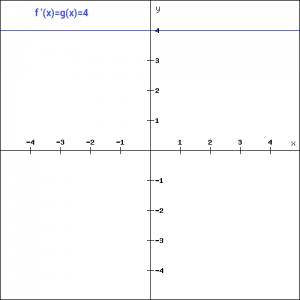
* Gradients are calculated to update the weights and biases during the backprop process. Since the gradient of the function is zero, the weights and biases don’t update.
* 2. Linear Function

There is no component of x in the binary step function. Instead of a binary function, we can use a linear function. We can define the function as-

* f(x)=ax



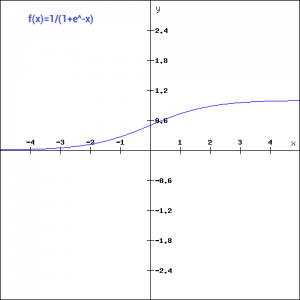
* Here the activation is proportional to the input.The variable ‘a’ in this case can be any constant value. Let’s quickly define the function in python:
* def linear\_function(x):
* return 4\*x
* linear\_function(4), linear\_function(-2)
* **Output:**
* (16, -8)
* What do you think will be the derivative is this case? When we differentiate the function with respect to x, the result is the coefficient of x, which is a constant.
* f'(x) = a



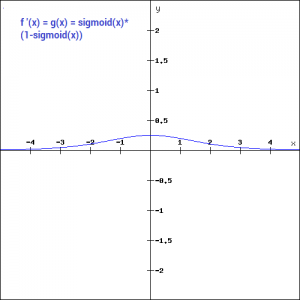
* Although the gradient here does not become zero, but it is a constant which does not depend upon the input value x at all. This implies that the weights and biases will be updated during the backpropagation process but the updating factor would be the same.
* In this scenario, the neural network will not really improve the error since the gradient is the same for every iteration. The network will not be able to train well and capture the complex patterns from the data. Hence, linear function might be ideal for simple tasks where interpretability is highly desired.

3. Sigmoid

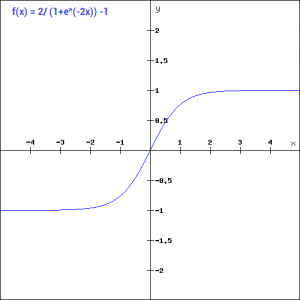
* The next activation function that we are going to look at is the Sigmoid function. It is one of the most widely used non-linear activation function. Sigmoid transforms the values between the range 0 and 1. Here is the mathematical expression for sigmoid-
* f(x) = 1/(1+e^-x)



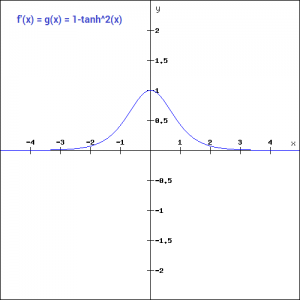
* A noteworthy point here is that unlike the binary step and linear functions, sigmoid is a non-linear function. This essentially means -when I have multiple neurons having sigmoid function as their activation function,the output is non linear as well. Here is the python code for defining the function in python-
* import numpy as np
* def sigmoid\_function(x):
* z = (1/(1 + np.exp(-x)))
* return z
* sigmoid\_function(7),sigmoid\_function(-22)
* **Output:**
* (0.9990889488055994, 2.7894680920908113e-10)
* Additionally, as you can see in the graph above, this is a smooth S-shaped function and is continuously differentiable. The derivative of this function comes out to be ( sigmoid(x)\*(1-sigmoid(x)). Let’s look at the plot of it’s gradient.
* f'(x) = sigmoid(x)\*(1-sigmoid(x))



* The gradient values are significant for range -3 and 3 but the graph gets much flatter in other regions. This implies that for values greater than 3 or less than -3, will have very small gradients. As the gradient value approaches zero, the network is not really learning.
* Additionally, the sigmoid function is not symmetric around zero. So output of all the neurons will be of the same sign. This can be addressed by scaling the sigmoid function which is exactly what happens in the tanh function. Let’s read on.
* 4. Tanh
* The tanh function is very similar to the sigmoid function. The only difference is that it is symmetric around the origin. The range of values in this case is from -1 to 1. Thus the inputs to the next layers will not always be of the same sign. The tanh function is defined as-
* tanh(x)=2sigmoid(2x)-1



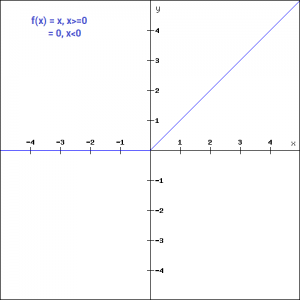
* In order to code this is python, let us simplify the previous expression.
* tanh(x) = 2sigmoid(2x)-1
* tanh(x) = 2/(1+e^(-2x)) -1
* And here is the python code for the same:
* def tanh\_function(x):
* z = (2/(1 + np.exp(-2\*x))) -1
* return z
* tanh\_function(0.5), tanh\_function(-1)
* **Output:**
* (0.4621171572600098, -0.7615941559557646)
* As you can see, the range of values is between -1 to 1. Apart from that, all other properties of tanh function are the same as that of the sigmoid function. Similar to sigmoid, the tanh function is continuous and differentiable at all points.
* Let’s have a look at the gradient of the tan h function.



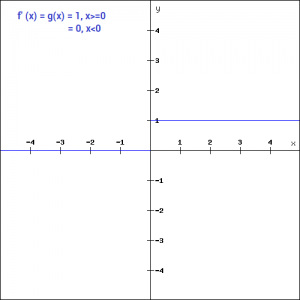
* The gradient of the tanh function is steeper as compared to the sigmoid function. You might be wondering, how will we decide which activation function to choose? Usually tanh is preferred over the sigmoid function since it is zero centered and the gradients are not restricted to move in a certain direction.

5. ReLU

* The ReLU function is another non-linear activation function that has gained popularity in the deep learning domain. ReLU stands for Rectified Linear Unit. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.
* This means that the neurons will only be deactivated if the output of the linear transformation is less than 0. The plot below will help you understand this better-
* f(x)=max(0,x)



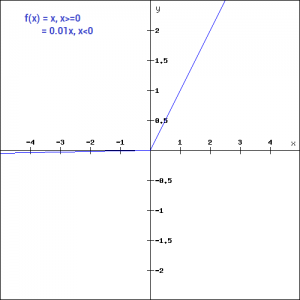
* For the negative input values, the result is zero, that means the neuron does not get activated. Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh function.  Here is the python function for ReLU:
* def relu\_function(x):
* if x<0:
* return 0
* else:
* return x
* relu\_function(7), relu\_function(-7)
* **Output:**
* (7, 0)
* Let’s look at the gradient of the ReLU function.
* f'(x) = 1, x>=0
* = 0, x<0

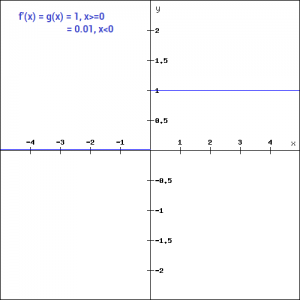


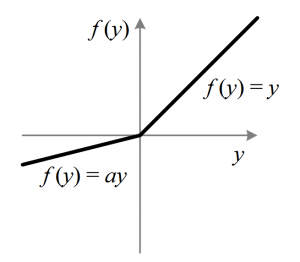
* If you look at the negative side of the graph, you will notice that the gradient value is zero. Due to this reason, during the backpropogation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated. This is taken care of by the ‘Leaky’ ReLU function.

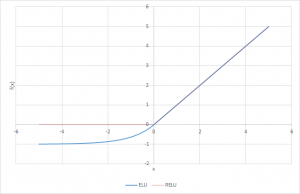
6. Leaky ReLU

* Leaky ReLU function is nothing but an improved version of the ReLU function. As we saw that for the ReLU function, the gradient is 0 for x<0, which would deactivate the neurons in that region.
* Leaky ReLU is defined to address this problem. Instead of defining the Relu function as 0 for negative values of x, we define it as an extremely small linear component of x. Here is the mathematical expression-
* f(x)= 0.01x, x<0
* = x, x>=0



* By making this small modification, the gradient of the left side of the graph comes out to be a non zero value. Hence we would no longer encounter dead neurons in that region. Here is the derivative of the Leaky ReLU function
* f'(x) = 1, x>=0
* =0.01, x<0
* 
* Since Leaky ReLU is a variant of ReLU, the python code can be implemented with a small modification-
* def leaky\_relu\_function(x):
* if x<0:
* return 0.01\*x
* else:
* return x
* leaky\_relu\_function(7), leaky\_relu\_function(-7)
* **Output:**
* (7, -0.07)
* Apart from Leaky ReLU, there are a few other variants of ReLU, the two most popular are – Parameterised ReLU function and Exponential ReLU.
* 7. Parameterised ReLU
* This is another variant of ReLU that aims to solve the problem of gradient’s becoming zero for the left half of the axis. The parameterised ReLU, as the name suggests, introduces a new parameter as a slope of the negative part of the function. Here’s how the ReLU function is modified to incorporate the slope parameter-
* f(x) = x, x>=0
* = ax, x<0



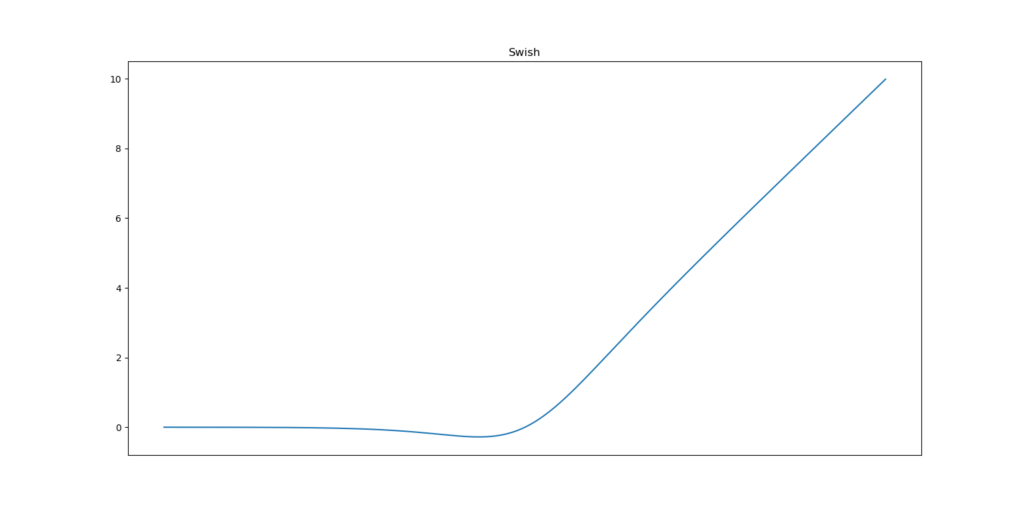
* When the value of a is fixed to 0.01, the function acts as a Leaky ReLU function. However, in case of a parameterised ReLU function, ‘**a**‘ is also a trainable parameter. The network also learns the value of ‘**a**‘ for faster and more optimum convergence.
* The derivative of the function would be same as the Leaky ReLu function, except the value 0.01 will be replaced with the value of a.
* f'(x) = 1, x>=0
* = a, x<0
* The parameterized ReLU function is used when the leaky ReLU function still fails to solve the problem of dead neurons and the relevant information is not successfully passed to the next layer.
* 8. Exponential Linear Unit
* Exponential Linear Unit or ELU for short is also a variant of Rectified Linear Unit (ReLU) that modifies the slope of the negative part of the function. Unlike the leaky relu and parametric ReLU functions, instead of a straight line, ELU uses a log curve for defning the negatice values. It is defined as
* f(x) = x, x>=0
* = a(e^x-1), x<0
* 
* Let’s define this function in python
* def elu\_function(x, a):
* if x<0:
* return a\*(np.exp(x)-1)
* else:
* return x
* elu\_function(5, 0.1),elu\_function(-5, 0.1)

**Output:**

* (5, -0.09932620530009145)
* The derivative of the elu function for values of  x greater than 0 is 1, like all the relu variants. But for values of x<0, the derivative would be  a.e^x .
* f'(x) = x, x>=0
* = a(e^x), x<0

9.Swish

* Swish is a lesser known activation function which was discovered by researchers at Google. Swish is as computationally efficient as ReLU and shows better performance than ReLU on deeper models.  The values for swish ranges from negative infinity to infinity. The function is defined as –
* f(x) = x\*sigmoid(x)
* f(x) = x/(1-e^-x)



* As you can see, the curve of the function is smooth and the function is differentiable at all points. This is helpful during the model optimization process and is considered to be one of the reasons that swish out performs ReLU.
* A unique fact about this function is that swich function is not monotonic. This means that the value of the function may decrease even when the input values are increasing. Let’s look at the python code for the swish function
* def swish\_function(x):
* return x/(1-np.exp(-x))
* swish\_function(-67), swish\_function(4)
* **Output:**
* (5.349885844610276e-28, 4.074629441455096)

10. Softmax

* Softmax function is often described as a combination of multiple sigmoids. We know that sigmoid returns values between 0 and 1, which can be treated as probabilities of a data point belonging to a particular class. Thus sigmoid is widely used for binary classification problems.
* The softmax function can be used for multiclass classification problems. This function returns the probability for a datapoint belonging to each individual class. Here is the mathematical expression of the same-
* 
* While building a network for a multiclass problem, the output layer would have as many neurons as the number of classes in the target. For instance if you have three classes, there would be three neurons in the output layer. Suppose you got the output from the neurons as [1.2 , 0.9 , 0.75].
* Applying the softmax function over these values, you will get the following result – [0.42 ,  0.31, 0.27]. These represent the probability for the data point belonging to each class. Note that the sum of all the values is 1. Let us code this in python
* def softmax\_function(x):
* z = np.exp(x)
* z\_ = z/z.sum()
* return z\_

softmax\_function([0.8, 1.2, 3.1])

* **Output:**
* array([0.08021815, 0.11967141, 0.80011044])
* Choosing the right Activation Function
* Now that we have seen so many activation  functions, we need some logic / heuristics to know which activation function should be used in which situation. Good or bad – there is no rule of thumb.
* However depending upon the properties of the problem we might be able to make a better choice for easy and quicker convergence of the network.
* Sigmoid functions and their combinations generally work better in the case of classifiers
* Sigmoids and tanh functions are sometimes avoided due to the vanishing gradient problem
* ReLU function is a general activation function and is used in most cases these days
* If we encounter a case of dead neurons in our networks the leaky ReLU function is the best choice
* Always keep in mind that ReLU function should only be used in the hidden layers
* As a rule of thumb, you can begin with using ReLU function and then move over to other activation functions in case ReLU doesn’t provide with optimum results.

**5.What is Forward propagation?**

* As the name suggests, the input data is fed in the forward direction through the network. Each hidden layer accepts the input data, processes it as per the activation function and passes to the successive layer.

## Why Feed-forward network?

* In order to generate some output, the input data should be fed in the forward direction only. The data should not flow in reverse direction during output generation otherwise it would form a cycle and the output could never be generated. Such network configurations are known as feed-forward network. The feed-forward network helps in forward propagation.
* At each neuron in a hidden or output layer, the processing happens in two steps:
* **Preactivation:** it is a weighted sum of inputs i.e. the linear transformation of weights w.r.t to inputs available. Based on this aggregated sum and activation function the neuron makes a decision whether to pass this information further or not.
* **Activation:** the calculated weighted sum of inputs is passed to the activation function. An activation function is a mathematical function which adds non-linearity to the network. There are four commonly used and popular activation functions — sigmoid, hyperbolic tangent(tanh), ReLU and Softmax.
* Forward propagation refers to storage and calculation of input data which is fed in forward direction through the network to generate an output. Hidden layers in neural network accepts the data from the input layer, process it on the basis of activation function and pass it to the output layer or the successive layers.