PERIYAR MANIAMMAI
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University)
Established Under Sec. 3 of UGC Act, 1956 • NAAC Accredited
think • innovate • transform

DEEP LEARNING
ASSIGNMENT -1

NAME :K.AJMALDEEN
REG.NO :121012012717
COURSE NAME:DEEP LEARNING

LIST OF LAYERS IN NEURAL NETWORK

1. *Input Layer (nnet.inputLayer)*: Represents the input data to the network.

2. *Fully Connected Layer (fullyConnectedLayer) or Dense Layer*: Each neuron in this layer is connected to every neuron in the previous layer and every neuron in the subsequent layer.

3. *Convolutional Layer (convolution2dLayer)*: Used in Convolutional Neural Networks (CNNs) for feature extraction from spatial data such as images.

4. *Max Pooling Layer (maxPooling2dLayer)*: Reduces the spatial dimensions of the input volume for down-sampling, typically used in CNNs.

5. *Average Pooling Layer (averagePooling2dLayer)*: Similar to max pooling but uses average instead of max values, also used for down-sampling.

6. *Recurrent Layer (lstmLayer or gruLayer)*: Used for processing sequences of data in Recurrent Neural Networks (RNNs), maintaining a state vector over time.

7. *Dropout Layer (dropoutLayer)*: Used for regularization by randomly setting a fraction of input units to zero during training to prevent overfitting.

8. *Batch Normalization Layer (batchNormalizationLayer)*: Normalizes the activations of the previous layer at each batch, helping in faster convergence and regularization.

9. *Global Average Pooling Layer (globalAveragePooling2dLayer)*: Computes the average value of each feature map in the previous layer, used to reduce the spatial dimensions of the input.

10. *Softmax Layer (softmaxLayer)*: Computes the softmax activation function, commonly used in the output layer of classification neural networks.

11. *Output Layer (nnet.outputLayer)*: Produces the final output of the network based on the computations performed in the hidden layers.

types of activation functions and when to use them

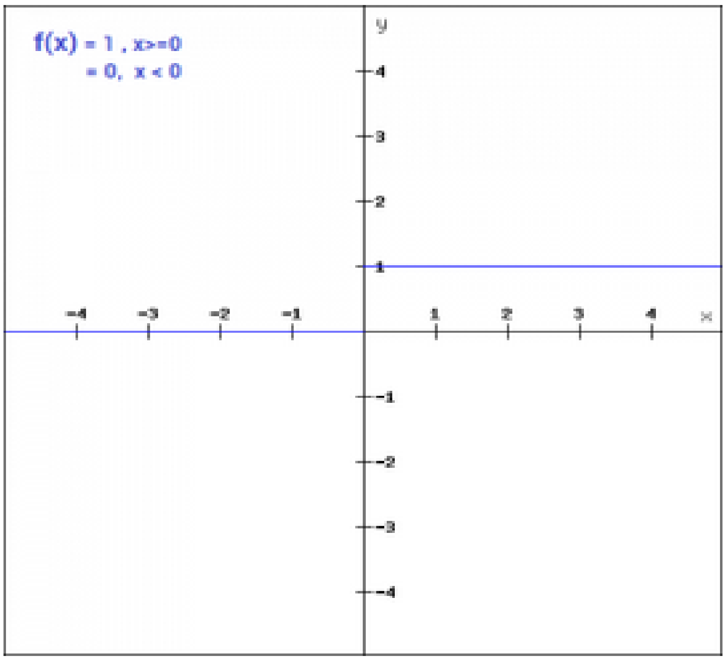
1. Binary Step Function

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)
```

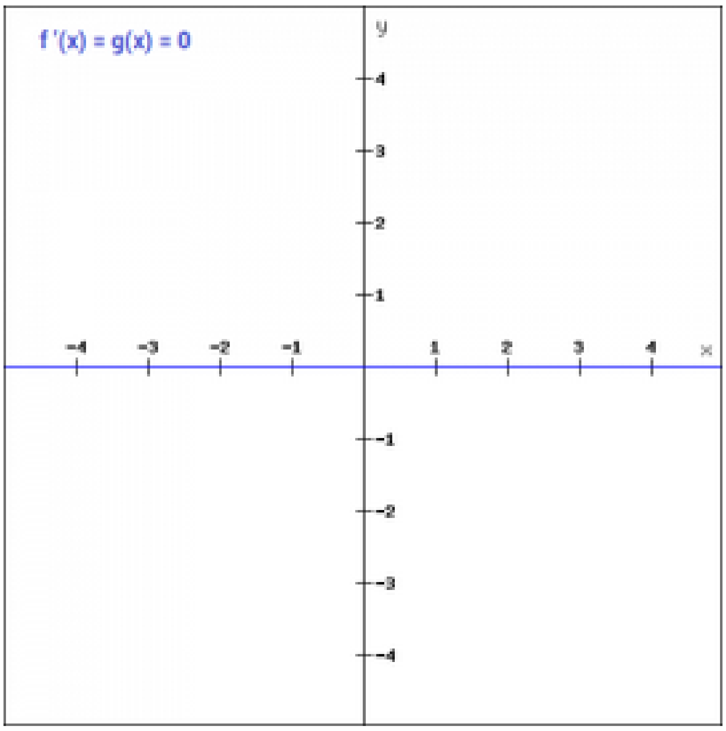
Output:

```
(1,0)
```

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

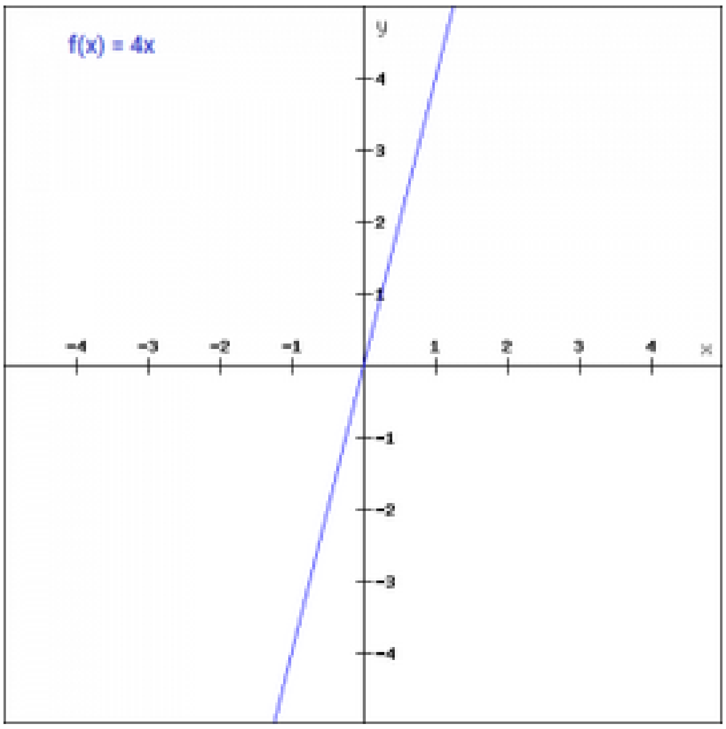


Gradients are calculated to update the weights and biases during the backprop process. Since the gradient of the function is zero, the weights and biases don't update.

2. Linear Function

We saw the problem with the step function, the gradient of the function became zero. This is because 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:

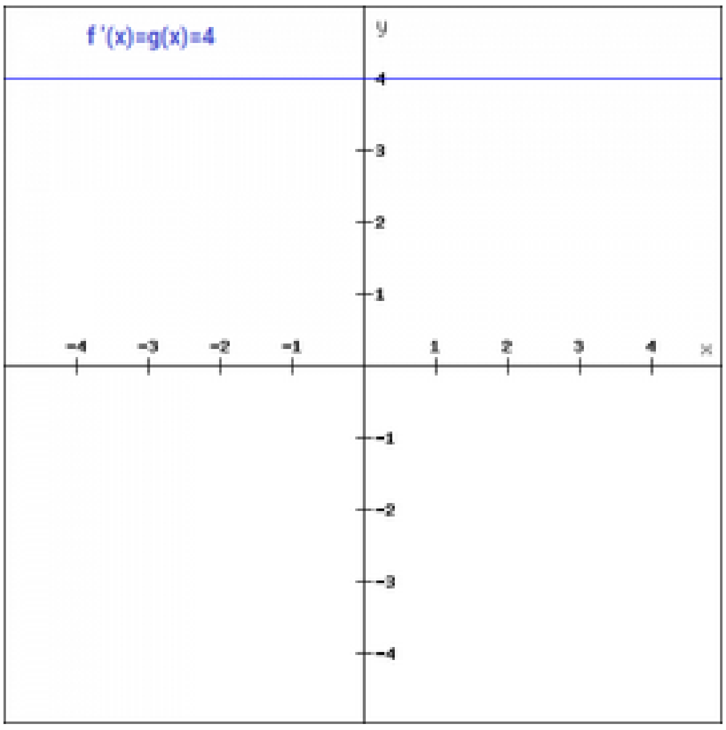
```
def linear_function(x):
    return 4*x
linear_function(4), linear_function(-2)
```

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.

$f'(x) = a$



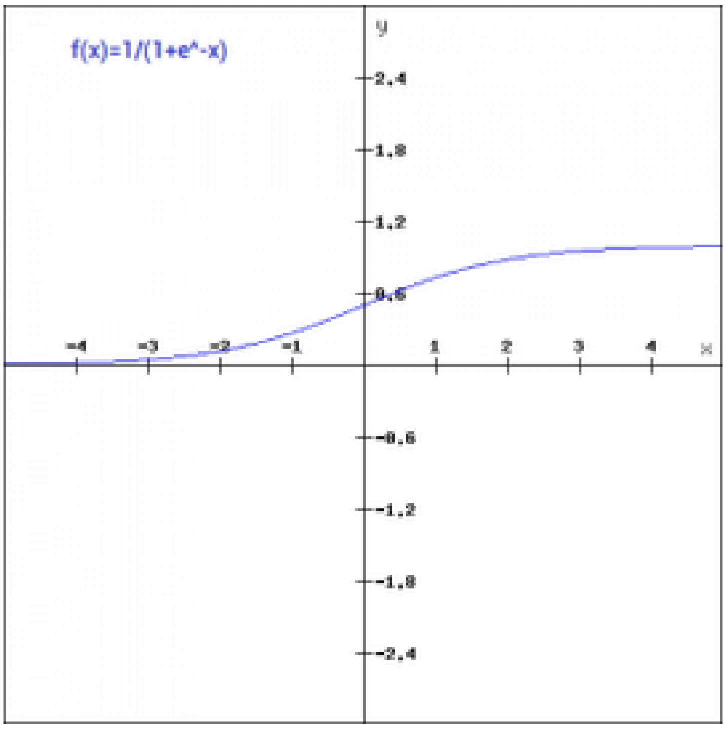
Although the gradient here does not become zero, but it is a constant which does not depend upon the input value x at all. This implies that the weights and biases will be updated during the backpropagation process but the updating factor would be the same.

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-

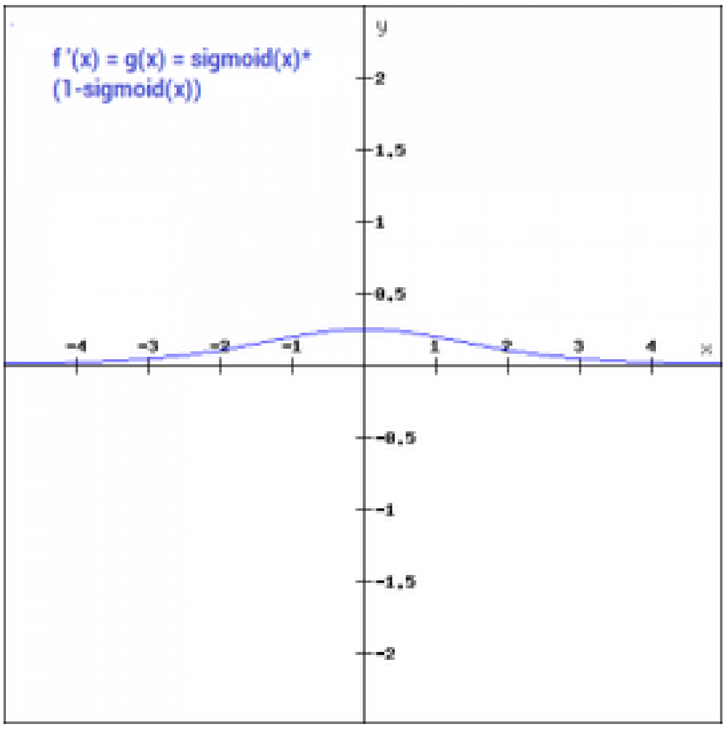
```
import numpy as np
def sigmoid_function(x):
    z = (1/(1 + np.exp(-x)))
    return z
sigmoid_function(7),sigmoid_function(-22)
```

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))



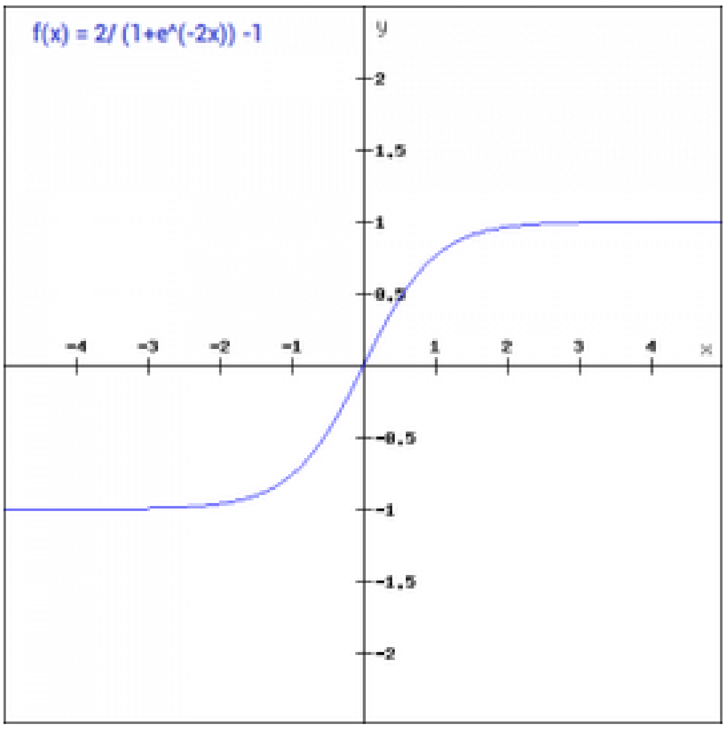
The gradient values are significant for range -3 and 3 but the graph gets much flatter in other regions. This implies that for values greater than 3 or less than -3, will have very small gradients. As the gradient value approaches zero, the network is not really learning.

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-

$$\tanh(x)=2\text{sigmoid}(2x)-1$$



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:

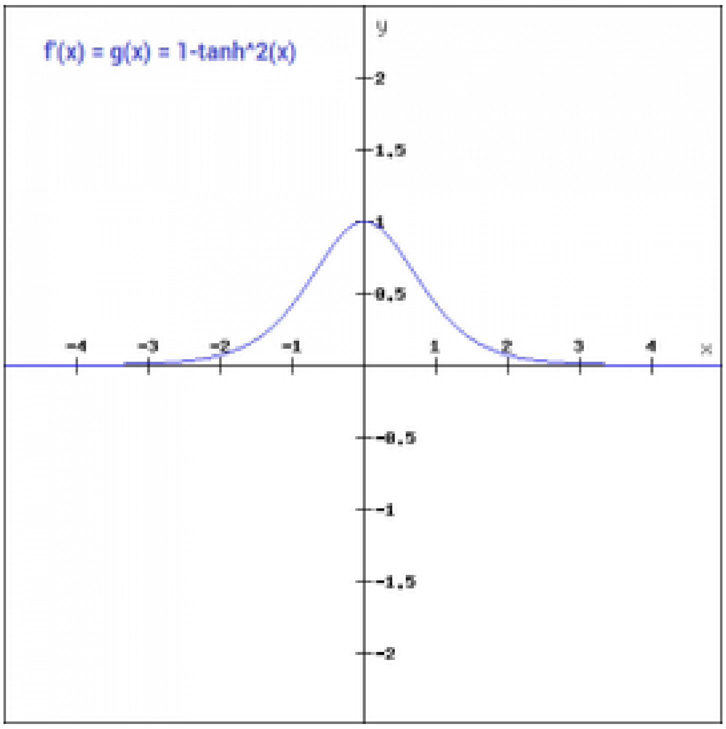
```
def tanh_function(x):
    z = (2/(1 + np.exp(-2*x))) -1
    return z
tanh_function(0.5), tanh_function(-1)
```

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.



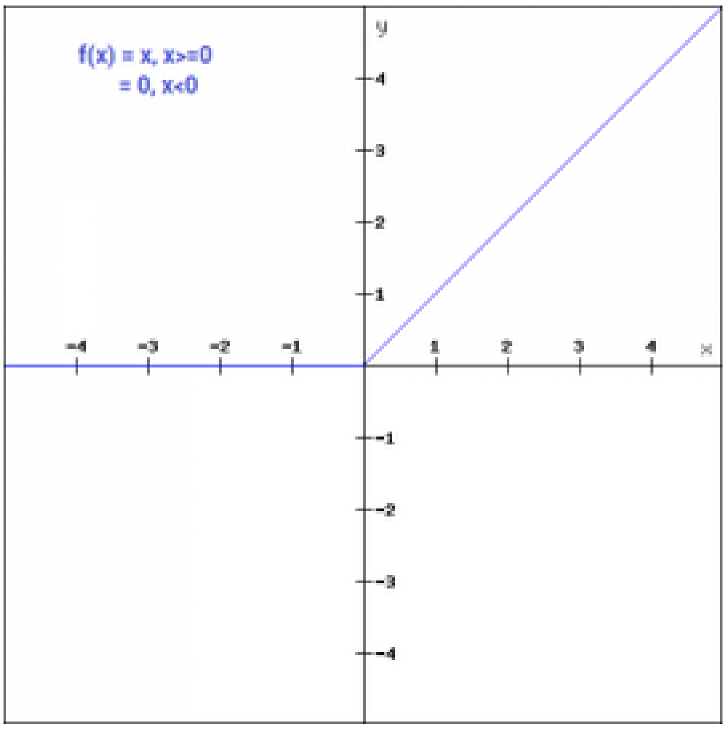
The gradient of the tanh function is steeper as compared to the sigmoid function. You might be wondering, how will we decide which activation function to choose? Usually tanh is preferred over the sigmoid function since it is zero centered and the gradients are not restricted to move in a certain direction.

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:

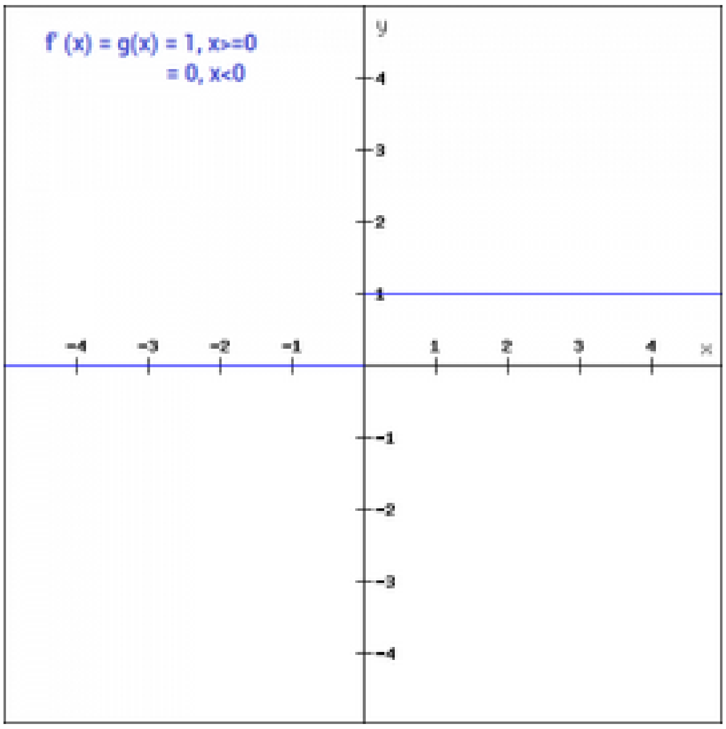
```
def relu_function(x):
    if x<0:
        return 0
    else:
        return x
relu_function(7), relu_function(-7)
```

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.

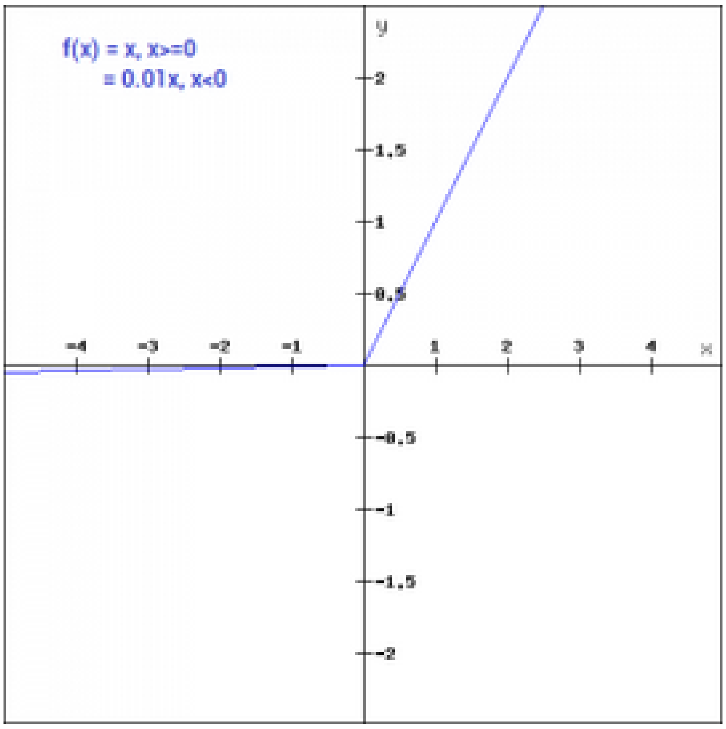
6. Leaky ReLU

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–

$f(x) = 0.01x, \; x<0$

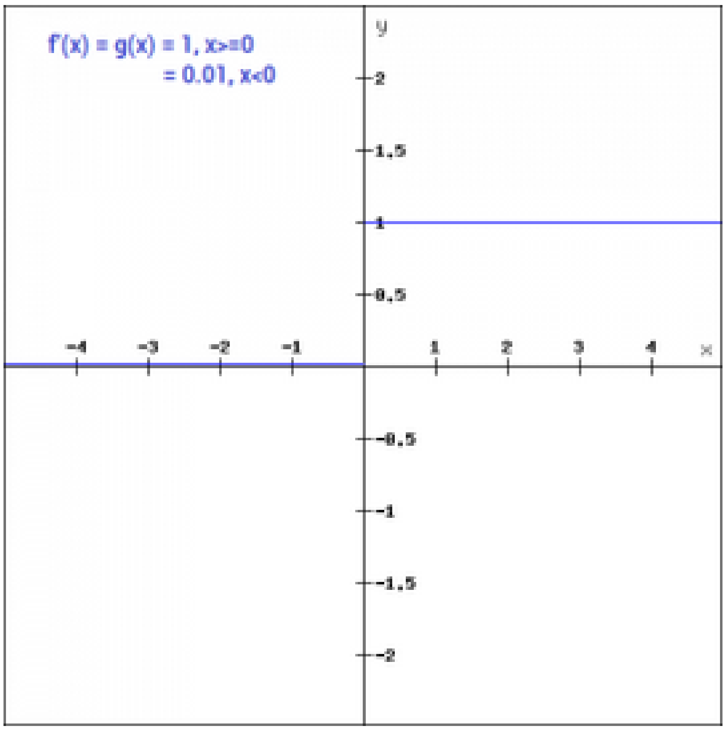
$\quad = \; x, \; x>=0$



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:

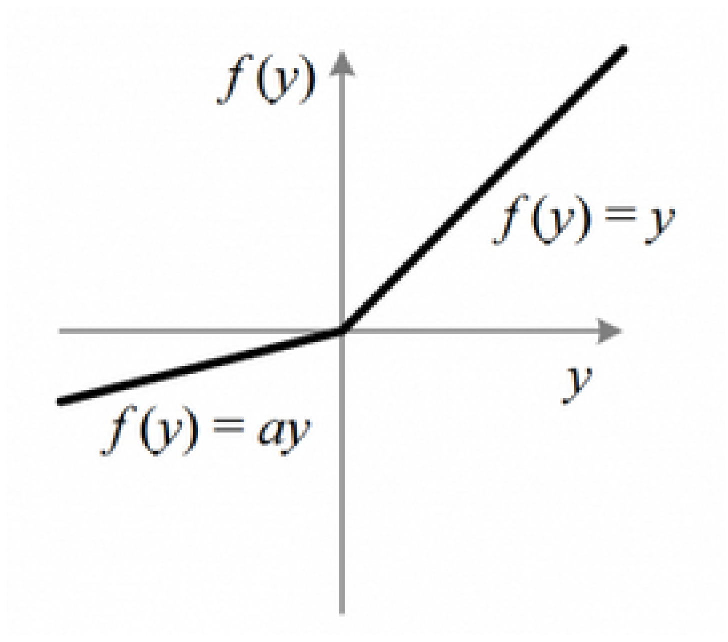
```
(7, -0.07)
```

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–

$$\begin{aligned} f(x) &= x, \quad x>=0 \\ &= ax, \quad x<0 \end{aligned}$$



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 replcaed with the value of a.

f'(x) = 1, x>=0

= a, x<0

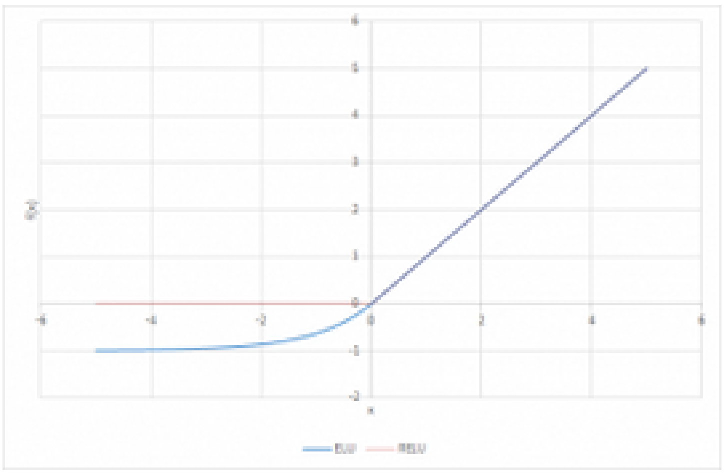
The parameterized ReLU function is used when the leaky ReLU function still fails to solve the problem of dead neurons and the relevant information is not successfully passed to the next layer.

8. Exponential Linear Unit

Exponential Linear Unit or ELU for short is also a variant of Rectiufied 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 .

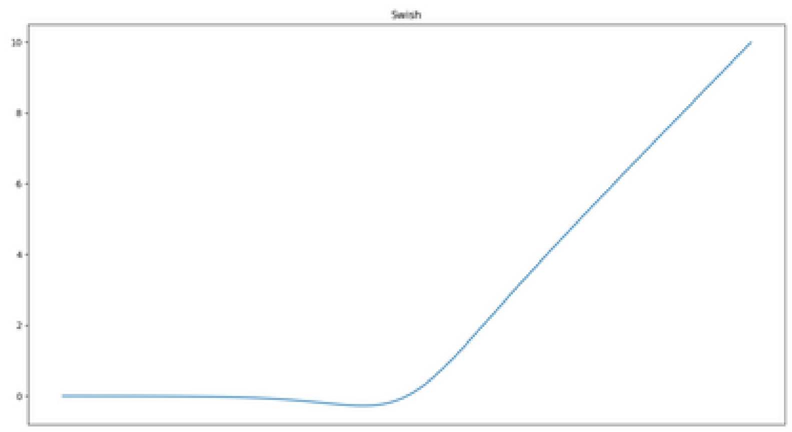
```
f'(x) = x,   x>=0
    = a(e^x), x<0
```

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 outoerforms 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–

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

DIFFERENT TYPES OF OPTIMIZERS

Introduction

In deep learning, we have the concept of loss, which tells us how poorly the model is performing at that current instant. Now we need to use this loss to **train** our network such that it performs better. Essentially what we need to do is to take the loss and try to **minimize** it, because a lower loss means our model is going to perform better. The process of minimizing (or maximizing) any mathematical expression is called **optimization.**

Optimizers are algorithms or methods used to change the attributes of the neural network such as **weights** and **learning rate** to reduce the losses. Optimizers are used to solve optimization problems by minimizing the function.

How do Optimizers work?

For a useful mental model, you can think of a hiker trying to get down a mountain with a blindfold on. It’s impossible to know which direction to go in, but there’s one thing she can know: if she’s going down (making progress) or going up (losing progress). Eventually, if she keeps taking steps that lead her downwards, she'll reach the base.

Similarly, it's impossible to know what your model's weights should be right from the start. But with some trial and error based on the loss function (whether the hiker is descending), you can end up getting there eventually.

How you should change your weights or learning rates of your neural network to reduce the losses is defined by the optimizers you use. Optimization algorithms are responsible for reducing the losses and to provide the most accurate results possible.

Various optimizers are researched within the last few couples of years each having its advantages and disadvantages. Read the entire article to understand the working, advantages, and disadvantages of the algorithms.

We'll learn about different types of optimizers and how they exactly work to minimize the loss function.

1. Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini Batch Stochastic Gradient Descent (MB-SGD)
4. SGD with momentum
5. Nesterov Accelerated Gradient (NAG)
6. Adaptive Gradient (AdaGrad)
7. AdaDelta
8. RMSprop
9. Adam

Gradient Descent

Gradient descent is an optimization algorithm that's used when training a machine learning model. It's based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum.

WHAT IS GRADIENT DESCENT? Gradient Descent is an optimization algorithm for finding a local minimum of a differentiable function. Gradient descent is simply used to find the values of a function's parameters (coefficients) that minimize a cost function as far as possible.

You start by defining the initial parameter's values and from there gradient descent uses calculus to iteratively adjust the values so they minimize the given cost-function.

The weight is initialized using some initialization strategies and is updated with each epoch according to the update equation.

The above equation computes the gradient of the cost function **J(θ)** w.r.t. to the parameters/weights **θ** for the entire traiing dataset:

Our aim is to get to the bottom of our graph(Cost vs weights), or to a point where we can no longer move downhill–a local minimum.

Okay now, what is Gradient?

"A gradient measures how much the output of a function changes if you change the inputs a little bit." —Lex Fridman (MIT)

Image source

Importance of Learning rate

How big the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slows we will move towards the optimal weights.

For gradient descent to reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high. This is important because if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent (see left image below). If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a whil (see the right image).

So, the learning rate should never be too high or too low for this reason. You can check if you’re learning rate is doing well by plotting it on a graph.

In code, gradient descent looks something like this:

```
for i in range(nb_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

For a pre-defined number of epochs, we first compute the gradient vector params_grad of the loss function for the whole dataset w.r.t. our parameter vector params.

Advantages:

1. Easy computation.
2. Easy to implement.
3. Easy to understand.

Disadvantages:

1. May trap at local minima.
2. Weights are changed after calculating the gradient on the whole dataset. So, if the dataset is too large then this may take years to converge to the minima.
3. Requires large memory to calculate the gradient on the whole dataset.

Stochastic Gradient Descent (SGD)

SGD algorithm is an extension of the Gradient Descent and it overcomes some of the disadvantages of the GD algorithm. Gradient Descent has a disadvantage that it requires a lot of memory to load the entire dataset of n-points at a time to compute the derivative of the loss function.

In the SGD algorithm derivative is computed

SGD performs a parameter update for *each* training example **x(i)** and label **y(i)**:

$$\boldsymbol{\theta = \theta - \alpha \cdot \partial(J(\theta;x(i),y(i)))/\partial\theta}$$

where **{x(i) ,y(i)}** are the training examples.

To make the training even faster we take a Gradient Descent step for each training example. Let's see what the implications would be in the image below. On the left, we have Stochastic Gradient Descent (where m=1 per step) we take a Gradient Descent step for each example and on the right is Gradient Descent (1 step per entire training set).

1. SGD seems to be quite noisy, at the same time it is much faster but may not converge to a minimum.
2. Typically, to get the best out of both worlds we use Mini-batch gradient descent (MGD) which looks at a smaller number of training set examples at once to help (usually power of 2 - 2^6 etc.).
3. Mini-batch Gradient Descent is relatively more stable than Stochastic Gradient Descent (SGD) but does have oscillations as gradient steps are being taken in the direction of a sample of the training set and not the entire set as in BGD.

It is observed that in SGD the updates take more number iterations compared to gradient descent to reach minima. On the right, the Gradient Descent takes fewer steps to reach minima but the SGD algorithm is noisier and takes more iterations.

Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example.

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

Advantage:

Memory requirement is less compared to the GD algorithm as the derivative is computed taking only 1 point at once.

Disadvantages:

1. The time required to complete 1 epoch is large compared to the GD algorithm.
2. Takes a long time to converge.
3. May stuck at local minima.

Mini Batch Stochastic Gradient Descent (MB-SGD)

MB-SGD algorithm is an extension of the SGD algorithm and it overcomes the problem of large time complexity in the case of the SGD algorithm. MB-SGD algorithm takes a batch of points or subset of points from the dataset to compute derivate.

It is observed that the derivative of the loss function for MB-SGD is almost the same as a derivate of the loss function for GD after some number of iterations. But the number of iterations to achieve minima is large for MB-SGD compared to GD and the

The update of weight is dependent on the derivate of loss for a batch of points. The updates in the case of MB-SGD are much noisy because the derivative is not always towards minima.MB-SGD divides the dataset into various batches and after every batch, the parameters are updated.

$$\boldsymbol{\theta = \theta - \alpha \cdot \partial(J(\theta;B(i)))/\partial\theta}$$

where **{B(i)}** are the batches of training examples.

In code, instead of iterating over examples, we now iterate over mini-batches of size 50:

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for batch in get_batches(data, batch_size=50):
        params_grad = evaluate_gradient(loss_function, batch, params)
        params = params - learning_rate * params_grad
```

Advantages:

Less time complexity to converge compared to standard SGD algorithm.

Disadvantages:

1. The update of MB-SGD is much noisy compared to the update of the GD algorithm.
2. Take a longer time to converge than the GD algorithm.
3. May get stuck at local minima.

SGD with momentum

A major disadvantage of the MB-SGD algorithm is that updates of weight are very noisy. SGD with momentum overcomes this disadvantage by denoising the gradients. Updates of weight are dependent on noisy derivative and if we somehow denoise the derivatives then converging time will decrease.

The idea is to denoise derivative using exponential weighting average that is to give more weightage to recent updates compared to the previous update.

It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum symbolized by ‘**γ**’.

$$\mathbf{V(t) = \gamma . V(t-1) + \alpha . \partial(J(\theta))/\partial\theta}$$

Now, the weights are updated by $\boldsymbol{\theta = \theta - V(t)}$.

The momentum term **γ** is usually set to 0.9 or a similar value.

Momentum at time ‘t’ is computed using all previous updates giving more weightage to recent updates compared to the previous update. This leads to speed up the convergence.

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma<1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster converg

The diagram above concludes SGD with momentum denoises the gradients and converges faster as compared to SGD.

Advantages:

1. Has all advantages of the SGD algorithm.
2. Converges faster than the GD algorithm.

Disadvantages:

We need to compute one more variable for each update.

Nesterov Accelerated Gradient (NAG)

The idea of the NAG algorithm is very similar to SGD with momentum with a slight variant. In the case of SGD with a momentum algorithm, the momentum and gradient are computed on the previous updated weight.

Momentum may be a good method but if the momentum is too high the algorithm may miss the local minima and may continue to rise up. So, to resolve this issue the NAG algorithm was developed. It is a look ahead method. We know we'll be using $\boldsymbol{\gamma.V(t-1)}$ for modifying the weights so, $\boldsymbol{\theta-\gamma V(t-1)}$ approximately tells us the future location. Now, we'll calculate the cost based on this future parameter rather than the current one.

$$\boldsymbol{V(t) = \gamma.V(t-1) + \alpha.\ \partial(J(\theta - \gamma V(t-1)))/\partial\theta}$$

and then update the parameters using $\boldsymbol{\theta = \theta - V(t)}$

Again, we set the momentum term γγ to a value of around 0.9. While Momentum first computes the current gradient (small brown vector in Image 4) and then takes a big jump in the direction of the updated accumulated gradient (big brown vector), NAG first makes a big jump in the direction of the previously accumulated gradient (green vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (red vector). This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs on a number of tasks.

Both NAG and SGD with momentum algorithms work equally well and share the same advantages and disadvantages.

Adaptive Gradient Descent(AdaGrad)

For all the previously discussed algorithms the learning rate remains constant. So the key idea of
AdaGrad is to have an adaptive learning rate for each of the weights.

It performs smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequently occurring features.

For brevity, we use gt to denote the gradient at time step t. **gt,i** is then the partial derivative of the objective function w.r.t. to the parameter **θi** at time step **t, η** is the learning rate and ∇θ is the partial derivative of loss function **J(θi)**

In its update rule, Adagrad modifies the general learning rate **η** at each time step **t** for every parameter **θi** based on the past gradients for **θi**:

where **Gt** is the sum of the squares of the past gradients w.r.t to all parameters **θ.**

The benefit of AdaGrad is that it eliminates the need to manually tune the learning rate; most leave it at a default value of 0.01.

Its main weakness is the accumulation of the squared gradients(**Gt**) in the denominator. Since every added term is positive, the accumulated sum keeps growing during training, causing the learning rate to shrink and becoming infinitesimally small and further resulting in a vanishing gradient problem.

Advantage:

No need to update the learning rate manually as it changes adaptively with iterations.

Disadvantage:

As the number of iteration becomes very large learning rate decreases to a very small number which leads to slow convergence.

AdaDelta

The problem with the previous algorithm
AdaGrad was learning rate becomes very small with a large number of iterations which leads to slow convergence. To avoid this, the AdaDelta algorithm has an idea to take an exponentially decaying average.

Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done. Compared to Adagrad, in the original version of Adadelta, you don't have to set an initial learning rate.

Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average **E[g2]t** at time step t then depends only on the previous average and current gradient:

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

RMSprop

RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests **γ** be set to 0.9, while a good default value for the learning rate **η** is 0.001.

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates

Adaptive Moment Estimation (Adam)

Adam can be looked at as a combination of
RMSprop and Stochastic Gradient Descent with momentum.

Adam computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients **vt** like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients **mt**, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.

Hyper-parameters **β1, β2 ∈ [0, 1)** control the exponential decay rates of these moving averages. We compute the decaying averages of past and past squared
gradients **mt** and **vt** respectively as follows:

mt and **vt** are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.