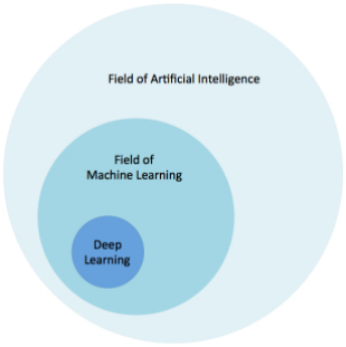
What is Deep Learning ??

To understand what Deep Learning is, we first need to understand the relationship that Deep learning has with Machine learning, Neural Networks, and Artificial Intelligence.

The best way to think of this relationship is to visualize them as concentric circles.



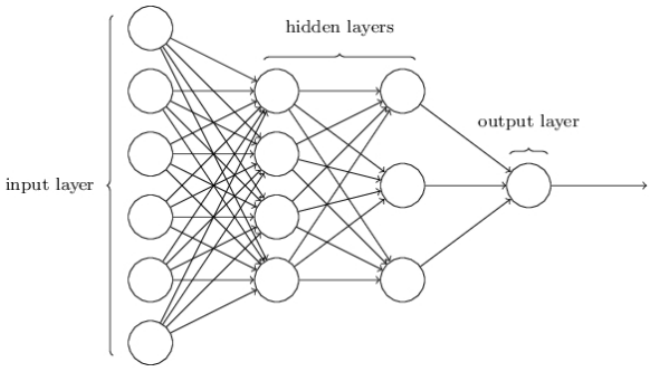
At the outer most ring you have Artificial Intelligence. Second layer inside of Artificial Intelligence is Machine learning and Deep learning is inside the Machine learning at the centre.

```
Broadly speaking, Deep Learning is a more approachable name for an Artificial Neural Network.
The “Deep” in Deep Learning refers to the depth of the network. An Artificial Neural Network can be very shallow.
```

Neural networks are inspired by the structure of the cerebral cortex. At the basic level is the perceptron, the mathematical representation of a biological neuron. Like in the cerebral cortex, there can be several layers of interconnected perceptrons.

The first layer is the input layer. Each node in this layer takes an input, and then passes its output as the input to each node in the next layer. There are generally no connections between nodes in the same layer and the last layer produces the outputs.

We call the middle part the hidden layer. These neurons have no connection to the outside (e.g. input or output) and are only activated by nodes in the previous layer.



Think of Deep Learning as the technique for learning in Neural networks that utilizes multiple layers of abstraction to solve pattern recognition problems. In the 1980s, most neural networks were a single layer due to the cost of computation and availability of data.

Machine learning is considered a branch or approach of Artificial intelligence, whereas Deep Learning is a specialized type of Machine Learning.

Machine learning involves Computer's intelligence that doesn't know the answers up front. Instead, the program will run against training data, verify the success of its attempts, and modify its approach accordingly.

Machine learning typical requires a sophisticated education, spanning software engineering and computer science to statistical methods and linear algebra.

There are two broad classes of machine learning methods:

- Supervised learning
- Unsupervised learning

In supervised learning, a machine learning algorithm uses a labeled dataset to infer the desired outcome. This takes a lot of data and time, since the data needs to be labeled by hand. Supervised learning is great for classification and regression problems.

For example, let's say that we were running a company and want to determine the effect of bonuses on employee retention. If we had historical data – i.e. employee bonus amount and tenure – we could use supervised machine learning.

With unsupervised learning, there aren't any predefined or corresponding answers. The goal is to figure out the hidden patterns in the data. It's usually used for clustering and associative tasks, like grouping customers by behavior. Amazon's "customers who also bought…" recommendations are a type of associative task.

While supervised learning can be useful, we often have to resort to unsupervised learning. Deep learning has proven to be an effective unsupervised learning technique.

Why is Deep Learning Important?

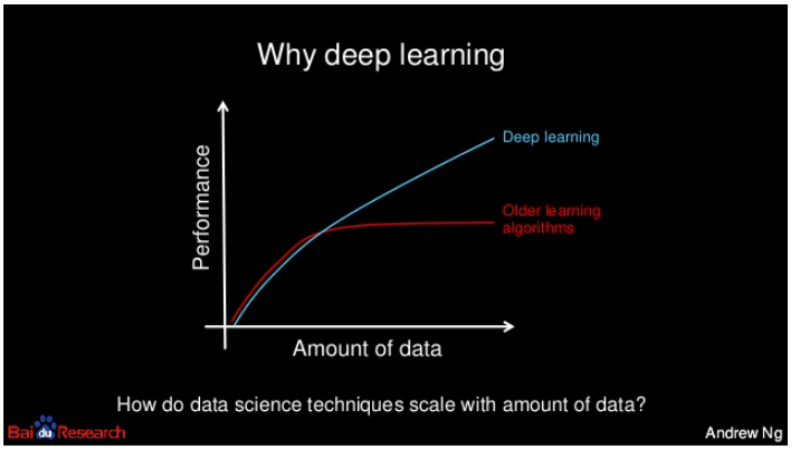
Computers have long had techniques for recognizing features inside of images. The results weren't always great. Computer vision has been a main beneficiary of Deep learning. Computer Vision using Deep Learning now rivals humans on many image recognition tasks.

Facebook has had great success with identifying faces in photographs by using deep learning. It's not just a marginal improvement, but a game changer: "Asked whether two unfamiliar photos of faces show the same person, a human being will get it right 97.53 percent of the time. New software developed by researchers at Facebook can score 97.25 percent on the same challenge, regardless of variations in lighting or whether the person in the picture is directly facing the camera."

Speech recognition is a another area that's felt deep learning's impact. Spoken languages are so vast and ambiguous. Baidu – one of the leading search engines of China – has developed a voice recognition system that is faster and more accurate than humans at producing text on a mobile phone. In both English and Mandarin.

What is particularly fascinating, is that generalizing the two languages didn't require much additional design effort: "Historically, people viewed Chinese and English as two vastly different languages, and so there was a need to design very different features," Andrew Ng says, chief scientist at Baidu. "The learning algorithms are now so general that you can just learn."

Google is now using deep learning to manage the energy at the company's data centers. They've cut their energy needs for cooling by 40%. That translates to about a 15% improvement in power usage efficiency for the company and hundreds of millions of dollars in savings.



Biological Neurons

Biological Neuron, BNN is an unusual-looking cell mostly found in animal cerebral cortexes (e.g., your brain), composed of a cell body containing the nucleus and most of the cell's complex components, and many branching extensions called Dendrites plus one very long extension called the Axon.

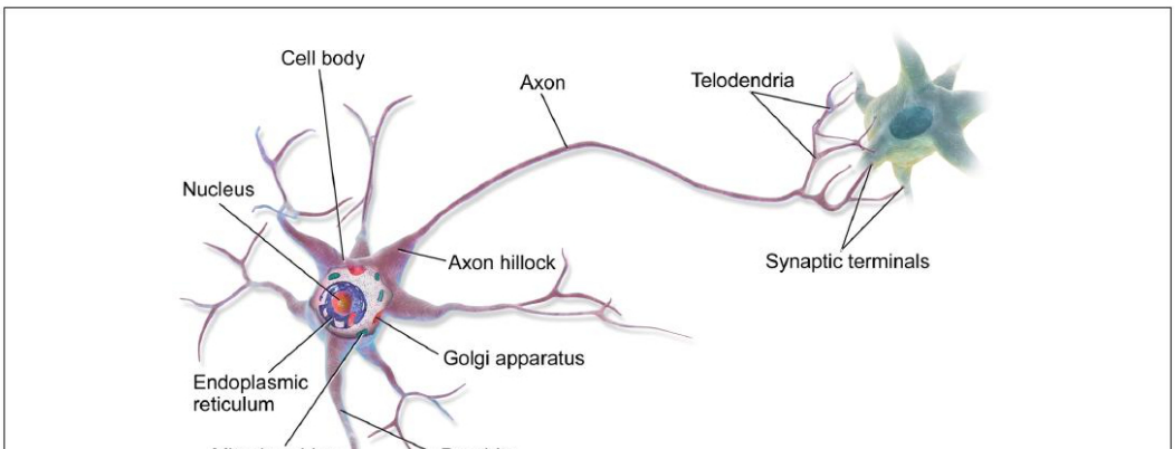
The Axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called Telodendria, and at the tip of these branches are minuscule structures called Synaptic Terminals (or simply Synapses), which are connected to the dendrites (or directly to the cell body) of other neurons.

Biological neurons receive short electrical impulses called Signals from other neurons via these Synapses. When a neuron receives a sufficient number of signals from other neurons within a few milliseconds, it fires its own signals.

Thus, individual Biological Neurons seem to behave in a rather simple way, but they are organized in a vast network of billions of neurons, each neuron typically connected to thousands of other neurons.

Highly complex computations can be performed by a vast network of fairly simple neurons, much like a complex ant hill can emerge from the combined efforts of simple ants.

The architecture of Bilogical Neural Network (BNN) is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers.



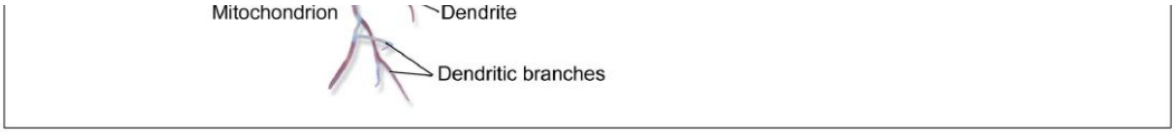


Figure 10-1. Biological neuron[3]

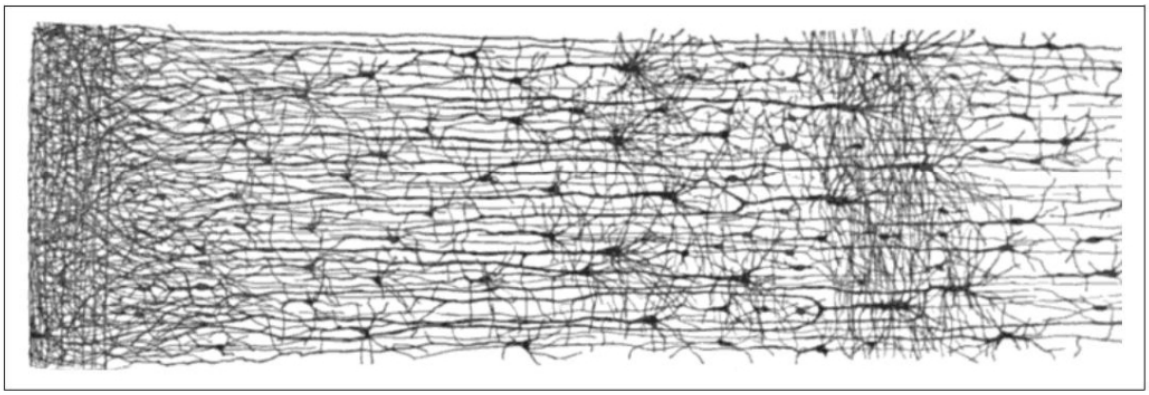


Figure 10-2. Multiple layers in a biological neural network (human cortex)[5]

Logical Computations with Neurons

Warren McCulloch and Walter Pitts proposed a very simple model of the Biological Neuron, which later became known as an Artificial Neuron. It has one or more binary (on/off) inputs and one binary output.

The artificial neuron simply activates its out-put when more than a certain number of its inputs are active.

McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want.

For example, let'sbuild a few ANNs that perform various logical computations assuming that a neuron is activated when at least two of its inputs are active.

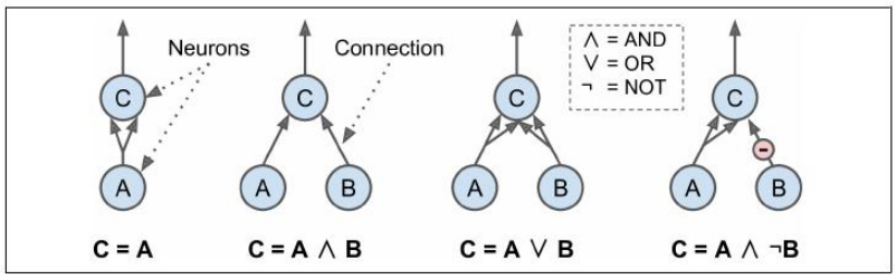


Figure 10-3. ANNs performing simple logical computations

- The first network on the left is simply the identity function: if neuron A is activated, then neuron C gets activated as well (since it receives two input signals from neuron A), but if neuron A is off, then neuron C is off as well.
- The second network performs a logical AND: neuron C is activated only when both neurons A and B are activated (a single input signal is not enough to activate neuron C).
- The third network performs a logical OR: neuron C gets activated if either neuron A or neuron B is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and if neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

Perceptrons : Threshhold Logic Unit (TLU)

```
=>  The Perceptrons are one of the simplest ANN architectures, invented in 1957 by Frank Rosen blatt.

=>  It is based on a slightly different artificial neuron called a Threshold Logic Unit (TLU), or sometimes
      a Linear Threshold Unit (LTU).

=>  The input and output are now numbers (instead of binary on/off values) and each input con-nection is
      associated with a weight.
```

```
=>  The TLU computes a weighted sum of its inputs (z = w1x1 + w2x2 + ⋯ + wnxn = xTw), then applies a
      Step function to that sum and outputs the result.

     ->  hw(x) = step(z), where z = xTw.          # in hw(x), w is index & in xTw, T is Transpose.
```

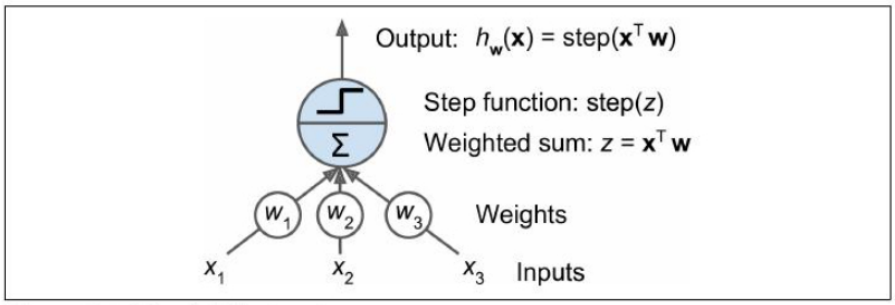


Figure 10-4. Threshold logic unit

```
 =>  A single TLU can be used for simple linear binary classification. It computes a linear combination of the
      inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the
        negative class (just like a Logistic Regression classifier or a linear SVM).

     For example, you could use a single TLU to classify iris flowers based on the petal length and width
       also adding an extra bias feature (x0 = 1).

=>  Training a TLU in this case means finding the right values for w0,w1, and w2.

=>  A Perceptron is simply composed of a single layer of TLUs, with each TLU connected to all the inputs.

=>  When all the neurons in a layer are connected to every neuron in the previous layer
     (i.e., its input neurons), it is common to draw special pass through neurons called input neurons and
    they just output whatever input they are fed.

=>  Moreover, an extra bias feature is generally added (x0 = 1), which is typically represented using a
     special type of Neuron called a Bias neuron which just outputs 1 all the time.

=>  A Perceptron with two inputs and three outputs is represented in below figure.
    This Perceptron can classifyinstances simultaneously into three different binary classes, which makes it
       a multi-output classifier.
```

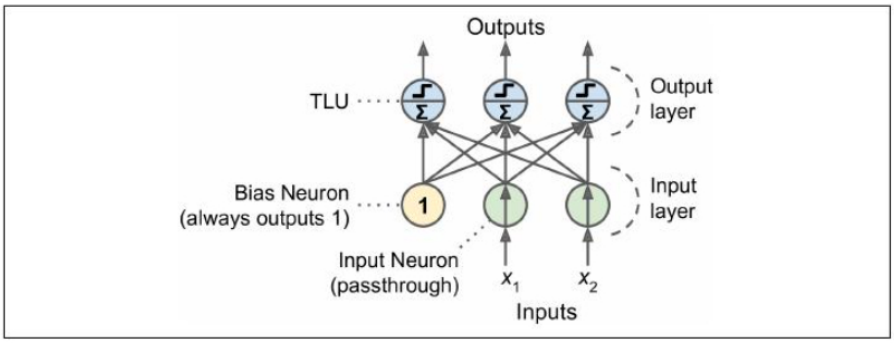


Figure 10-5. Perceptron diagram

Equation 10-2. Computing the outputs of a fully connected layer

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

- As always, **X** represents the matrix of input features. It has one row per instance, one column per feature.
- The weight matrix **W** contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector **b** contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The function ϕ is called the *activation function*: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).

So how is a Perceptron trained?

Training of a Perceptron

> The Perceptron training algorithm proposed by Frank Rosen blatt was largely inspired by Hebb's rule.

> In his book 'The Organization of Behavior' published in 1949, Donald Hebb suggested that - **when a Biological Neuron often triggers another neuron, the connection between these two neurons grows stronger**.

> This idea was later summarized by Siegrid Löwel in this catchy phrase - 'Cells that fire together, wire together' and this rule later became known as Hebb's rule or Hebbian learning.

> Scikit-Learn provides a Perceptron class that implements a single TLU network.

```
#  Implementing Perceptron class from sk-learn and training on iris dataset
#    Note that this solution is generally not unique: in general when the data are linearly separable,
#        there is an infinity of hyperplanes that can separate them.

import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)] # petal length, petal width
y = (iris.target == 0).astype(np.int) # Iris Setosa?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
y_pred
```

In [ ]:
```
          ###############   ##############   Multi-Layered Perceptrons    ################   ##############

 =>  An MLP is composed of one (passthrough) input layer, one or more layers of TLUs, called Hidden layers,
       and one final layer of TLUs called the Output layer.

 =>  The layers close to the input layer are usually called the lower layers, and the ones close to the outputs
        are usually called the upper layers.

 =>  Every layer except the output layer includes a Bias neuron and is fully connected to the next layer.
```

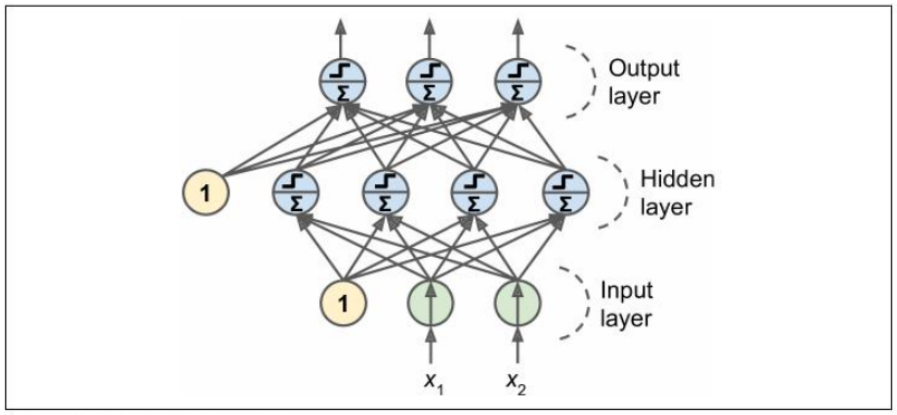
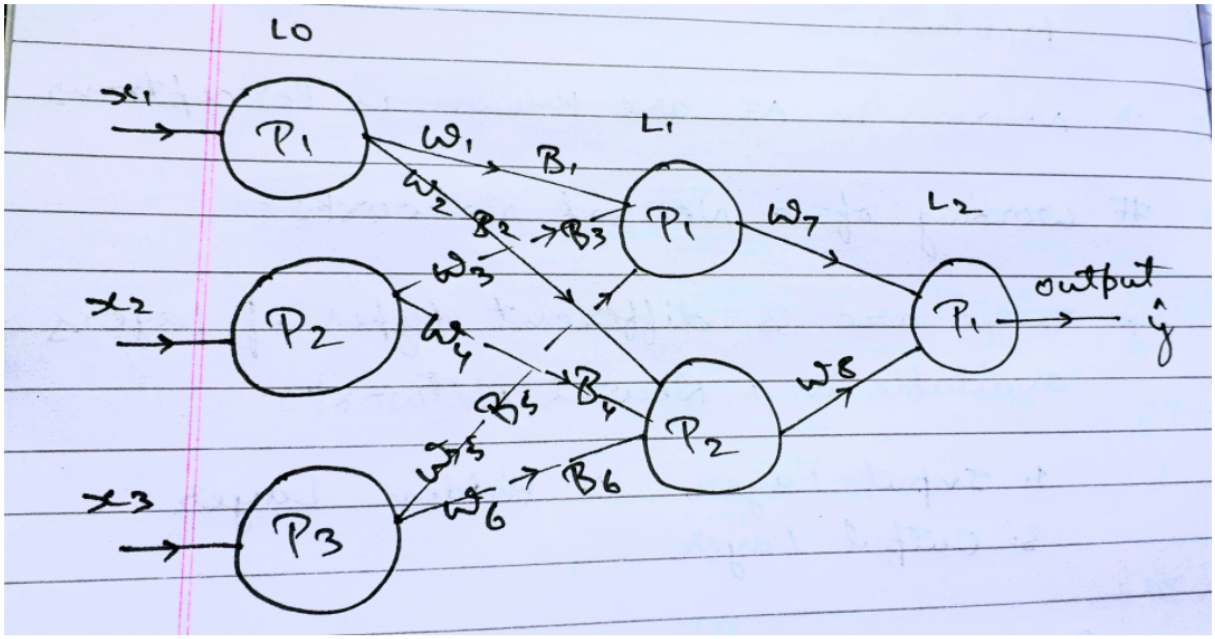


Figure 10-7. Multi-Layer Perceptron

Diagram of an Artificial Neural Networks

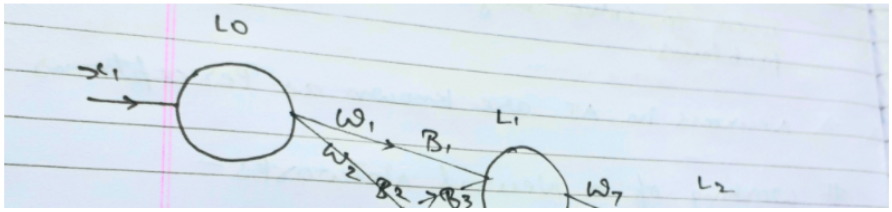
In [ ]:
```
 =>  There are 3 different types of layers lies inside a Neural Networks.

        * Input Layer     *  Hidden Layers     *  Output Layer

 =>  In Input Layer, Number of Perceptrons depends upon the total number of Features available in dataset.

 =>  Perceptrons of Input Layer receives the inputted data. In above case, Row 1 is going to be inputted in
      Input Layer, then 2nd Row, 3rd Row and so on .

 =>  The main function of Input layer is just only getting the inputs from dataset.

 =>  This Inputted data goes to 2nd Layer of Neural networks (Hidden layer) in such a way that Data from each
      Perceptron of Input layer goes inputted to each and every Perceptrons of Hidden layer.

 =>  When we try to pass our data from Input layer to Hidden layer, We multiply some randomly selected weights,
        W1 to inputs X1 and then add some randomly selected Bias, B1 to the inputs.
      Then this Linear mix of inputs (W1.X1 + B1) get entered into each Perceptrons of Hidden layer.

 =>  Weights and Bias are learnable parameters and they are added to get patterens by making relations
       in dataset more easily.

 =>  If we have only 2 Layers i.e. Input Layer and Output Layer, Whatever data we passes to Input Layer will
      be outputted based on Output Functions and with no parameters it is very difficult to find the
       relationships between the dataset.
```

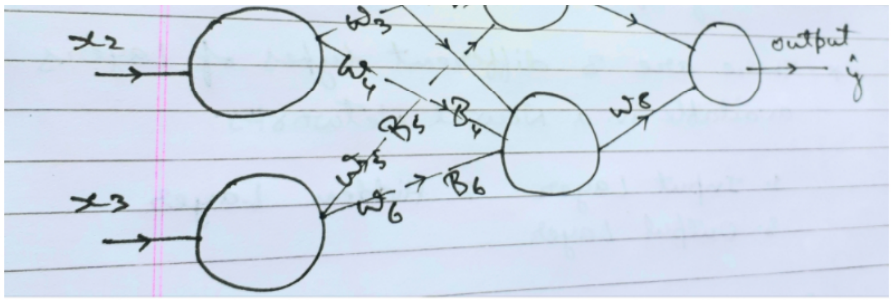


Feed - Forward Connection

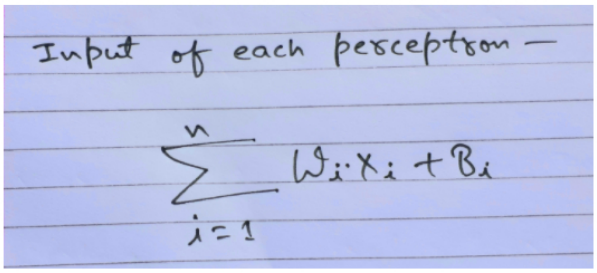
```
     ##############   #############   FFN in Training of Neural networks   ################   ##############

=>  Training of an Artificial Neural Networks has been classified into two diferent processes.

        * Feed Forward Connection (FFN)       &          * Backward Propagation

=>  In Feed Forward Connections, Data is getting inputted into the Input Layer o
      Neural Network and then
      it get multiplied and added with some randomly selected parameters like as 'weights' and 'Bias'.

=>  Weights & Bias are some Learnable parameters and they are getting introduced into the Networks for
      getting an ease of Acces in finding  Relations between data.

=>  After introducing Weights and Bias, a Linear mix of Inputs are getting inputted to the Perceptron of
      next layer of Network and there will be similar inputs from all other perceptrons of Input Layer.
    So a Summation of all Linear Mix of Inputs will be inputted to each perceptrons of Hidden Layer.

=>  Inside a Hidden layer, a Linear mix of data enters as Input, so we need some particular function which
     will be able to Filter, Restrict, Normalize and Non-Linearize the Inputted dataset, which is passed
      from Input layer to Hidden layer. ThereFore Activation Function comes into picture.
       We can add as many Hidden layers but in a certain limit.

=>  Activation functions are kind of functions, which always helps Perceptrons to fire itself in certain
     direction or in a certain range and also try to restricts some sort of dataset.

    Examples of Activation function -

       *  Sigmoid Activation Function         *  Tanh Activation Function
       *  RelU Activation Function            *  Leaky RelU Activation function etc.

=>  We can use a Single Activation Function inside a Hidden Layer.
     Activation function does some particular operations and give some outputs (say o1, o2), which get
       multiplied with some different weights again and then enters into Next Layer.


=>  In Output Layer, We again uses a Function which is known as Output Function.
   Here all Inputs are get added together by using Output function and then we get a final output or
     prediction.

=>  Output Functions are similar to Activation functions but it always getting used in the output Layer of
      the Network.

=>  There will be some differences between the Expected and Predicted outputs, and this difference is
     termed as Loss or cost.

=>  This whole process is known as Feed Forward Connection or Feed Forward Networks.
     In FFN, our Network does not try to learn something. Learning Never happens in FFN.
```

Assigning Weights and Bias To Inputs





Input of Perceptrons in Hidden layer



Backward Propagation

In [ ]:

```
=>  In FFN, we get some Losses due to introducing some randomly selected parameters and our whole objective
     and idea is to reduce this Loss. For reducing this loss, Backward Propagation comes into picture.

=>  Backward Propagation is a method for efficiently computing the gradient of the cost function of a
     Neural network with respect to its parameters. These partial derivatives can then be used to update the
       parameters using 'Gradient Descent'.

=>  The gradient shows how much the parameters 'weights' & 'Biases' needs to change (in positive or negative
       direction) to minimize the Cost function 'C' .

=>  In Backward Propagation, it will try to update the randomly selected parameters by using some
     'optimization' mechanism and it always occurs in backward direction and updates the parameters
    layer-by-layer in Network on the basis of losses.
     For updating the parameters, it uses a Differential equation.

=>  With updated parameters, it will start the Feed Forward Connection again and calculates some Losses
     and then it again starts the Backward Propagation to reduce the weights.
   It will keep updating the weights untill we get an optimal or reduced Loss and we get some pattern in
     dataset.

=>  One Feed Forward connection and one Backward Propagation togetherly makes a cycle, known as one Epoch.

=>  In Differential Eqns to update the parameters, we uses a Learning Rate to control the learning process.
     Learning Rate should not be too high or too low, it will be in range of (0.001 to 10).

=>  At beginning, We dont have any idea about any relation or patterns between the dataset, so we introduces
    some Learnable Parameters and in FFN, a Linear mix of Inputs goes to each and every units of
     the Hidden Layer. In Backward Propagation, We updates the  parameters on the basis of Loss.

=>  So with updated parameters, We repeat the FFN and this time Our network drops some data, which are not
      contributing in the result and this way, it learns the pattern between the dataset.

=>  Increment of Learnable Parameters makes our Network more flexible to learn the relationship b/w dataset.
    Introducing more units in Hidden Layers & more Hidden Layers will give a better result but they are
     just experimental ones.
```

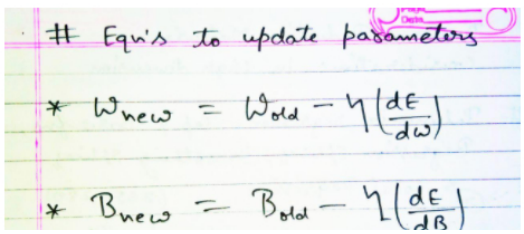
Using Chain Rule for updating the parameters

The chain rule is a way to compute the derivative of a function whose variables are themselves functions of other variables. If C is a scalar-valued function of a scalar z and z is itself a scalar-valued function of another scalar variable w, then the chain rule states that

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial z}\frac{\partial z}{\partial w}$$

For scalar-valued functions of more than one variable, the chain rule essentially becomes additive. In other words, if C is a scalar-valued function of N variables $z_1, \ldots, z_N$, each of which is a function of some variable w, the chain rule states that

$$\frac{\partial C}{\partial w} = \sum_{i=1}^{N} \frac{\partial C}{\partial z_i}\frac{\partial z_i}{\partial w}$$



Notation

In the following derivation, we'll use the following notation:

L - Number of layers in the network.

N^n - Dimensionality of layer $n \in \{0, \dots, L\}$. N^0 is the dimensionality of the input; N^L is the dimensionality of the output.

$W^m \in \mathbb{R}^{N^m \times N^{m-1}}$ - Weight matrix for layer $m \in \{1, \dots, L\}$. W_{ij}^m is the weight between the i^{th} unit in layer m and the j^{th} unit in layer $m-1$.

$b^m \in \mathbb{R}^{N^m}$ - Bias vector for layer m.

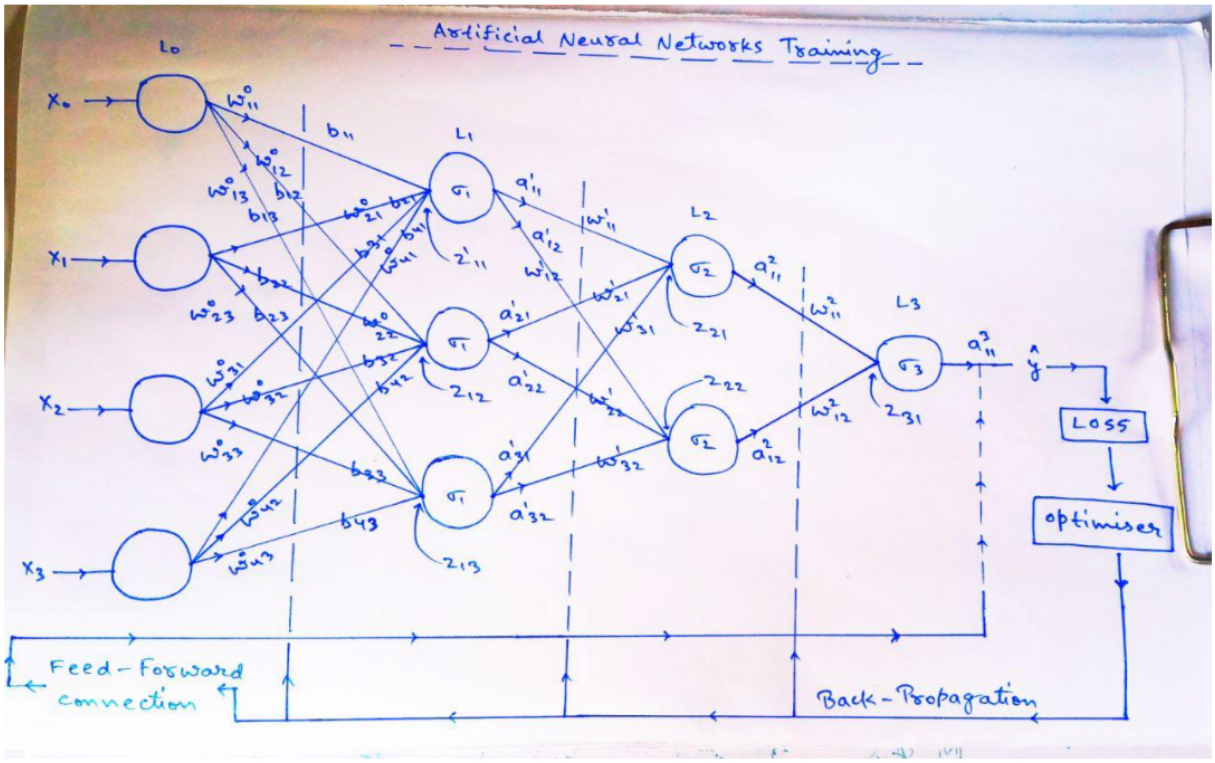
σ^m - Nonlinear activation function of the units in layer m, applied elementwise.

$z^m \in \mathbb{R}^{N^m}$ - Linear mix of the inputs to layer m, computed by $z^m = W^m a^{m-1} + b^m$.

$a^m \in \mathbb{R}^{N^m}$ - Activation of units in layer m, computed by $a^m = \sigma^m(h^m) = \sigma^m(W^m a^{m-1} + b^m)$. a^L is the output of the network. We define the special case a^0 as the input of the network.

$y \in \mathbb{R}^{N^L}$ - Target output of the network.

C - Cost/error function of the network, which is a function of a^L (the network output) and y (treated as a constant).



Neural Networks Training Blog

Backpropagation in general

In order to train the network using a gradient descent algorithm, we need to know the gradient of each of the parameters with respect to the cost/error function C; that is, we need to know $\frac{\partial C}{\partial W^m}$ and $\frac{\partial C}{\partial b^m}$. It will be sufficient to derive an expression for these gradients in terms of the following terms, which we can compute based on the neural network's architecture:

- $\frac{\partial C}{\partial a^L}$: The derivative of the cost function with respect to its argument, the output of the network
- $\frac{\partial a^m}{\partial z^m}$: The derivative of the nonlinearity used in layer m with respect to its argument

To compute the gradient of our cost/error function C to W_{ij}^m (a single entry in the weight matrix of the layer m), we can first note that C is a function of a^L, which is itself a function of the linear mix variables z_k^m, which are themselves functions of the weight matrices W^m and biases b^m. With this in mind, we can use the chain rule as follows:

$$\frac{\partial C}{\partial W_{ij}^m} = \sum_{k=1}^{N^m} \frac{\partial C}{\partial z_k^m} \frac{\partial z_k^m}{\partial W_{ij}^m}$$

Note that by definition

$$z_k^m = \sum_{l=1}^{N^m} W_{kl}^m a_l^{m-1} + b_k^m$$

It follows that $\frac{\partial z_k^m}{\partial W_{ij}^m}$ will evaluate to zero when $i \neq k$ because z_k^m does not interact with any elements in W^m except for those in the k^{th} row, and we are only considering the entry W^m. When $i = k$, we have

considering the entry W_{ij}. When $i = k$, we have

$$\begin{aligned}\frac{\partial z_i^m}{\partial W_{ij}^m} &= \frac{\partial}{\partial W_{ij}^m}\left(\sum_{l=1}^{N^m} W_{il}^m a_l^{m-1} + b_i^m\right)\\ &= a_j^{m-1}\\ \rightarrow \frac{\partial z_k^m}{\partial W_{ij}^m} &= \begin{cases} 0 & k \neq i\\ a_j^{m-1} & k = i\end{cases}\end{aligned}$$

The fact that $\frac{\partial C}{\partial a_k^m}$ is 0 unless $k = i$ causes the summation above to collapse, giving

$$\frac{\partial C}{\partial W_{ij}^m} = \frac{\partial C}{\partial z_i^m} a_j^{m-1}$$

or in vector form

$$\frac{\partial C}{\partial W^m} = \frac{\partial C}{\partial z^m} a^{m-1\top}$$

Similarly for the bias variables b^m, we have

$$\frac{\partial C}{\partial b_i^m} = \sum_{k=1}^{N^m} \frac{\partial C}{\partial z_k^m}\frac{\partial z_k^m}{\partial b_i^m}$$

As above, it follows that $\frac{\partial z_k^m}{\partial b_i^m}$ will evaluate to zero when $i \neq k$ because z_k^m does not interact with any element in b^m except b_k^m. When $i = k$, we have

$$\begin{aligned}\frac{\partial z_i^m}{\partial b_i^m} &= \frac{\partial}{\partial b_i^m}\left(\sum_{l=1}^{N^m} W_{il}^m a_l^{m-1} + b_i^m\right)\\ &= 1\\ \rightarrow \frac{\partial z_i^m}{\partial b_i^m} &= \begin{cases} 0 & k \neq i\\ 1 & k = i\end{cases}\end{aligned}$$

The summation also collapses to give

$$\frac{\partial C}{\partial b_i^m} = \frac{\partial C}{\partial z_i^m}$$

or in vector form

$$\frac{\partial C}{\partial b^m} = \frac{\partial C}{\partial z^m}$$

Now, we must compute $\frac{\partial C}{\partial z_k^m}$. For the final layer ($m = L$), this term is straightforward to compute using the chain rule:

$$\frac{\partial C}{\partial z_k^L} = \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_k^L}$$

or, in vector form

$$\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}$$

The first term $\frac{\partial C}{\partial a^L}$ is just the derivative of the cost function with respect to its argument, whose form depends on the cost function chosen. Similarly, $\frac{\partial a^m}{\partial z^m}$ (for any layer m includling L) is the derivative of the layer's nonlinearity with respect to its argument and will depend on the choice of nonlinearity. For other layers, we again invoke the chain rule:

$$\begin{aligned}\frac{\partial C}{\partial z_k^m} &= \frac{\partial C}{\partial a_k^m}\frac{\partial a_k^m}{\partial z_k^m}\\ &= \left(\sum_{l=1}^{N^{m+1}} \frac{\partial C}{\partial z_l^{m+1}}\frac{\partial z_l^{m+1}}{\partial a_k^m}\right)\frac{\partial a_k^m}{\partial z_k^m}\\ &= \left(\sum_{l=1}^{N^{m+1}} \frac{\partial C}{\partial z_l^{m+1}}\frac{\partial}{\partial a_k^m}\left(\sum_{h=1}^{N^m} W_{lh}^{m+1} a_h^m + b_l^{m+1}\right)\right)\frac{\partial a_k^m}{\partial z_k^m}\\ &= \left(\sum_{l=1}^{N^{m+1}} \frac{\partial C}{\partial z_l^{m+1}} W_{lk}^{m+1}\right)\frac{\partial a_k^m}{\partial z_k^m}\\ &= \left(\sum_{l=1}^{N^{m+1}} W_{kl}^{m+1\top}\frac{\partial C}{\partial z_l^{m+1}}\right)\frac{\partial a_k^m}{\partial z_k^m}\end{aligned}$$

where the last simplification was made because by convention $\frac{\partial C}{\partial z_l^{m+1}}$ is a column vector, allowing us to write the following vector form:

$$\frac{\partial C}{\partial z^m} = \left(W^{m+1\top}\frac{\partial C}{\partial z^{m+1}}\right)\circ\frac{\partial a^m}{\partial z^m}$$

Note that we now have the ingredients to efficiently compute the gradient of the cost function with respect to the network's parameters: First, we compute $\frac{\partial C}{\partial z_k^L}$ based on the choice of cost function and nonlinearity. Then, we recursively can compute $\frac{\partial C}{\partial z^m}$ layer-by-layer based on the term $\frac{\partial C}{\partial z^{m+1}}$ computed from the previous layer and the nonlinearity of the layer (this is called the "backward pass").

Backpropagation in practice

As discussed above, the exact form of the updates depends on both the chosen cost function and each layer's chosen nonlinearity. The following two table lists the some common choices for nonlinearities and the required partial derivative for deriving the gradient for each layer:

Nonlinearity	$a^m = \sigma^m(z^m)$	$\frac{\partial a^m}{\partial z^m}$	Notes
Sigmoid	$\frac{1}{1+e^{z^m}}$	$\sigma^m(z^m)(1-\sigma^m(z^m)) = a^m(1-a^m)$	"Squashes" any input to the range $[0, 1]$
Tanh	$\frac{e^{z^m}-e^{-z^m}}{e^{z^m}+e^{-z^m}}$	$1-(\sigma^m(z^m))^2 = 1-(a^m)^2$	Equivalent, up to scaling, to the sigmoid function
ReLU	$\max(0, z^m)$	$0, z^m < 0;\ 1, z^m \geq 0$	Commonly used in neural networks with many layers

Similarly, the following table collects some common cost functions and the partial derivative needed to compute the gradient for the final layer:

Cost Function	C	$\frac{\partial C}{\partial a^L}$	Notes
Squared Error	$\frac{1}{2}(y-a^L)^\top(y-a^L)$	$y-a^L$	Commonly used when the output is not constrained to a specific range
Cross-Entropy	$(y-1)\log(1-a^L) - y\log(a^L)$	$\frac{a^L-y}{a^L(1-a^L)}$	Commonly used for binary classification tasks; can yield faster convergence

In practice, backpropagation proceeds in the following manner for each training sample:

1. Forward pass: Given the network input a^0, compute a^m recursively by
$$a^1 = \sigma^1(W^1 a^0 + b^1), \ldots, a^L = \sigma^L(W^L a^{L-1} + b^L)$$
2. Backward pass: Compute
$$\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}$$
for the final layer based on the tables above, then recursively compute
$$\frac{\partial C}{\partial z^m} = \left(W^{m+1\top}\frac{\partial C}{\partial z^{m+1}}\right) \circ \frac{\partial a^m}{\partial z^m}$$
for all other layers. Plug these values into
$$\frac{\partial C}{\partial W^m} = \frac{\partial C}{\partial z_i^m} a^{m-1\top}$$
and
$$\frac{\partial C}{\partial b^m} = \frac{\partial C}{\partial z^m}$$
to obtain the updates.

Example: Sigmoid network with cross-entropy loss using gradient descent

A common network architecture is one with fully connected layers where each layer's nonlinearity is the sigmoid function $a^m = \frac{1}{1+e^{z^m}}$ and the cost function is the cross-entropy loss $(y-1)\log(1-a^L) - y\log(a^L)$. To compute the updates for gradient descent, we first compute (based on the tables above)

$$\begin{aligned}\frac{\partial C}{\partial z^L} &= \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L} \\ &= \left(\frac{a^L - y}{a^L(1-a^L)}\right) a^L(1-a^L) \\ &= a^L - y\end{aligned}$$

From here, we can compute

$$\begin{aligned}\frac{\partial C}{\partial z^{L-1}} &= \left(W^{L\top}\frac{\partial C}{\partial z^L}\right) \circ \frac{\partial a^{L-1}}{\partial z^{L-1}} \\ &= W^{L\top}(a^L - y) \circ a^{L-1}(1-a^{L-1}) \\ \frac{\partial C}{\partial z^{L-2}} &= \left(W^{L-1\top}\frac{\partial C}{\partial z^{L-1}}\right) \circ \frac{\partial a^{L-2}}{\partial z^{L-2}} \\ &= W^{L-1\top}\left(W^{L\top}(a^L - y) \circ a^{L-1}(1-a^{L-1})\right) \circ a^{L-2}(1-a^{L-2})\end{aligned}$$

and so on, until we have computed $\frac{\partial C}{\partial z^m}$ for $m \in \{1, \ldots, L\}$. This allows us to compute $\frac{\partial C}{\partial W_{ij}^m}$ and $\frac{\partial C}{\partial b_i^m}$, e.g.

$$\begin{aligned}\frac{\partial C}{\partial W^L} &= \frac{\partial C}{\partial z^L} a^{L-1\top} \\ &= (a^L - y)a^{L-1\top} \\ \frac{\partial C}{\partial W^{L-1}} &= \frac{\partial C}{\partial z^{L-1}} a^{L-2\top} \\ &= W^{L\top}(a^L - y) \circ a^{L-1}(1-a^{L-1})a^{L-2\top}\end{aligned}$$

and so on. Standard gradient descent then updates each parameter as follows:

$$\begin{aligned}W^m &= W^m - \lambda\frac{\partial C}{\partial W^m} \\ b^m &= b^m - \lambda\frac{\partial C}{\partial b^m}\end{aligned}$$

where λ is the learning rate. This process is repeated until some stopping criteria is met.

Gradient Descent

The simplest algorithm for iterative minimization of differentiable functions is known as just **gradient descent**. Recall that the gradient of a function is defined as the vector of partial derivatives:

$$\nabla f(x) = [\partial f x_1, \partial f x_2, \ldots, \partial f x_n]$$

and that the gradient of a function always points towards the direction of maximal increase at that point.

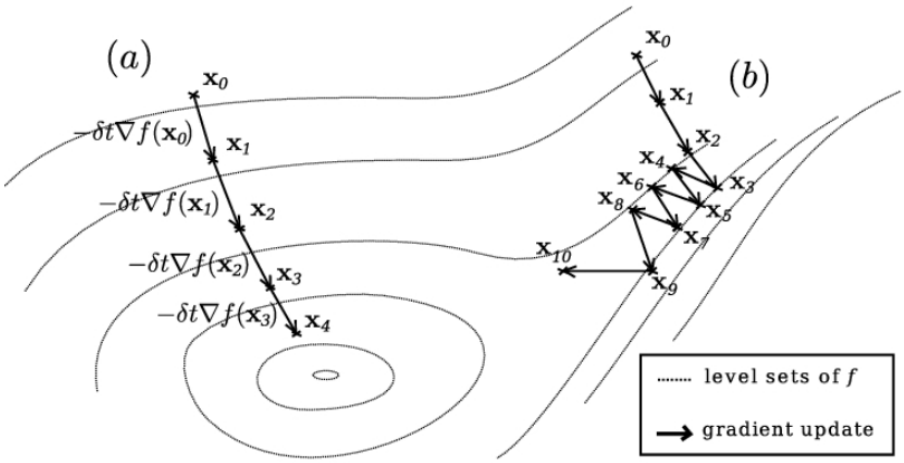
Equivalently, it points *away* from the direction of maximum decrease - thus, if we start at any point, and keep moving in the direction of the negative gradient, we will eventually reach a local minimum.

This simple insight leads to the Gradient Descent algorithm. Outlined algorithmically, it looks like this:

1. Pick a point x_0 as your initial guess.

2. Compute the gradient at your current guess: $v_i = \nabla f(x_i)$
3. Move by α (your step size) in the direction of that gradient: $x_{i+1} = x_i + \alpha v_i$
4. Repeat steps 1-3 until your function is close enough to zero (until $f(x_i) < \varepsilon$ for some small tolerance ε)

Note that the step size, α, is simply a parameter of the algorithm and has to be fixed in advance.



Notice that the hyperbolic tangent function asymptotes at -1 and 1, rather than 0 and 1, which is sometimes beneficial, and its derivative is simple:

$$\frac{d \tanh(x)}{dx} = 1 - \tanh^2(x)$$

Performing gradient descent will allow us to change the weights in the direction that optimially reduces the error. The next trick will be to employ the **chain rule** to decompose how the error changes as a function of the input weights into the change in error as a function of changes in the inputs to the weights, mutliplied by the changes in input values as a function of changes in the weights.

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial h} \frac{\partial h}{\partial w}$$

This will allow us to write a function describing the activations of the output weights as a function of the activations of the hidden layer nodes and the output weights, which will allow us to propagate error backwards through the network.

The second term in the chain rule simplifies to:

$$\begin{aligned} \frac{\partial h_k}{\partial w_{jk}} &= \frac{\partial \sum_l w_{lk} a_l}{\partial w_{jk}} \\ &= \sum_l \frac{\partial w_{lk} a_l}{\partial w_{jk}} \\ &= a_j \end{aligned}$$

where a_j is the activation of the jth hidden layer neuron.

For the first term in the chain rule above, we decompose it as well:

$$\frac{\partial E}{\partial h_k} = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial h_k} = \frac{\partial E}{\partial g(h_k)} \frac{\partial g(h_k)}{\partial h_k}$$

The second term of this chain rule is just the derivative of the activation function, which we have chosen to have a conveneint form, while the first term simplifies to:

$$\frac{\partial E}{\partial g(h_k)} = \frac{\partial}{\partial g(h_k)} \left[\frac{1}{2} \sum_k (t_k - y_k)^2 \right] = t_k - y_k$$

Combining these, and assuming (for illustration) a logistic activiation function, we have the gradient:

$$\frac{\partial E}{\partial w} = (t_k - y_k) y_k (1 - y_k) a_j$$

Which ends up getting plugged into the weight update formula that we saw in the single-layer perceptron:

$$w_{jk} \leftarrow w_{jk} - \eta (t_k - y_k) y_k (1 - y_k) a_j$$

Note that here we are *subtracting* the second term, rather than adding, since we are doing gradient descent.

We can now outline the MLP learning algorithm:

1. Initialize all w_{jk} to small random values
2. For each input vector, conduct forward propagation:
   - compute activation of each neuron j in hidden layer (here, sigmoid):

$$h_j = \sum_i x_i v_{ij}$$

$$a_j = g(h_j) = \frac{1}{1 + \exp(-\beta h_j)}$$

   - when the output layer is reached, calculate outputs similarly:

$$h_k = \sum_k a_j w_{jk}$$

$$y_k = g(h_k) = \frac{1}{1 + \exp(-\beta h_k)}$$

$$1 + \exp(-\rho n_k)$$

3. Calculate loss for resulting predictions:
   - compute error at output:

$$\delta_k = (t_k - y_k) y_k (1 - y_k)$$

4. Conduct backpropagation to get partial derivatives of cost with respect to weights, and use these to update weights:
   - compute error of the hidden layers:

$$\delta_{hj} = \left[\sum_k w_{jk} \delta_k \right] a_j (1 - a_j)$$

   - update output layer weights:

$$w_{jk} \leftarrow w_{jk} - \eta \delta_k a_j$$

   - update hidden layer weights:

$$v_{ij} \leftarrow v_{ij} - \eta \delta_{hj} x_i$$

Return to (2) and iterate until learning completes. Best practice is to shuffle input vectors to avoid training in the same order.

Its important to be aware that because gradient descent is a hill-climbing (or descending) algorithm, it is liable to be caught in local minima with respect to starting values. Therefore, it is worthwhile training several networks using a range of starting values for the weights, so that you have a better chance of discovering a globally-competitive solution.

One useful performance enhancement for the MLP learning algorithm is the addition of **momentum** to the weight updates. This is just a coefficient on the previous weight update that increases the correlation between the current weight and the weight after the next update. This is particularly useful for complex models, where falling into local mimima is an issue; adding momentum will give some weight to the previous direction, making the resulting weights essentially a weighted average of the two directions. Adding momentum, along with a smaller learning rate, usually results in a more stable algorithm with quicker convergence. When we use momentum, we lose this guarantee, but this is generally seen as a small price to pay for the improvement momentum usually gives.

A weight update with momentum looks like this:

$$w_{jk} \leftarrow w_{jk} - \eta \delta_k a_j + \alpha \Delta w_{jk}^{t-1}$$

where α is the momentum (regularization) parameter and Δw_{jk}^{t-1} the update from the previous iteration.

The multi-layer pereptron is implemented below in the `MLP` class. The implementation uses the scikit-learn interface, so it is uses in the same way as other supervised learning algorithms in that package.

Activation Functions

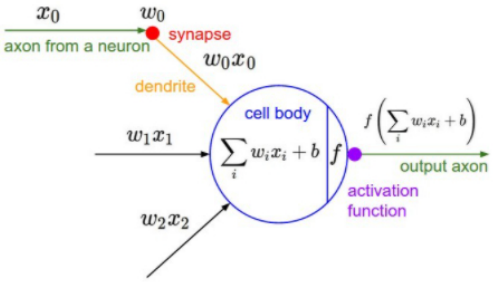
> Activation Functions are a kind of function, which helps a Neuron to fire itself in a certain direction, in a certain bandwidth, or in a certain range. It also try to restrict some sort of dataset.

> Activation functions helps to determine the output of a Neural Network. These type of functions are attached to each Neurons in the Network, and determines whether it should be activated or not, based on whether each neuron's input is relevant for the model's prediction.

> Activation Function also helps to normalize the output of each neuron to a range between (1 and 0) or between (1 and -1).

> Neural Networks use non-linear Activation Functions, which can help the network learn complex data, compute and learn almost any function representing a question, and provide accurate predictions.

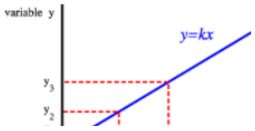
> The activation function is a mathematical "gate" in between the input feeding the current neuron and its output going to the next layer. It can be as simple as a step function that turns the neuron output on and off, depending on a rule or threshold.

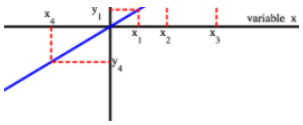


There are 3 different types of Activation Functions available to be used in Deep Learning.

Linear Activation functions

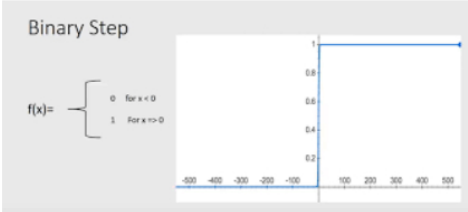
> **Linear Activation functions** always defines a Linear relationship 'Y = m.X + c'. Differentials of a Linear function gives some constant, so it can be used in Back-propagation. But a linear activation function is not able to handle the complexity of data like as it will get failed in cases of Image recognitions or more.





Binary Step Activation functions

> **Binary Step Activation functions** always gives it's output in a range (0 and 1). For Negative X, it always gives 0 and for Positive X, i gives 1. It can handle 2 classes. But it's derivative is always 0, so this function is not proper for using in Back-propagation.



Non-linear Activation functions

> Non Linear Activation functions are able to understand the complexity of dataset and also it gives some differentials, which gets used in Back-propagation to update the parameters. So We generally uses Non-linear Activation functions in Deep learning.

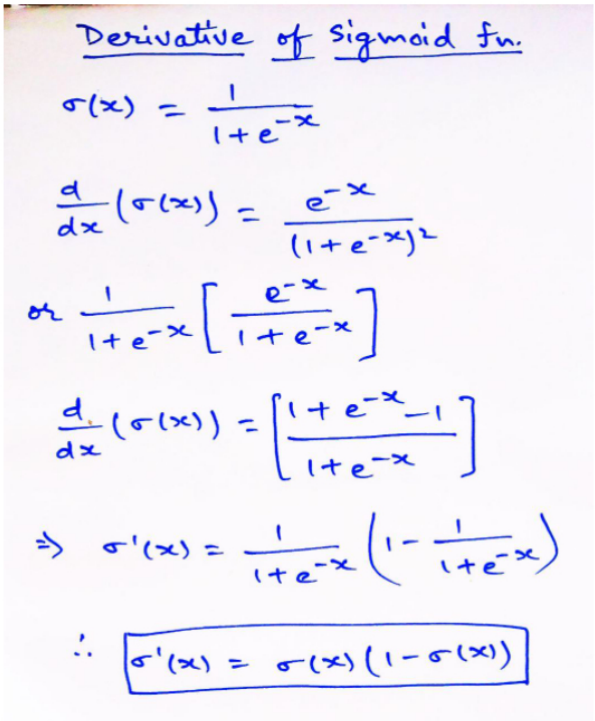
```
####  Some popular Activation functions used in Deep Learning

     * Sigmoid Function                      * Tanh Function
     * ReLU Function                         * Leaky RelU Function
     * PReLU Function                        * ELU Function
     * Softmax Function                      * SoftPlus Function
     * Swish Function                        * Maxout Function
```

Sigmod or Logistic Activation Function

> The Sigmoid function is the most frequently used Activation function in the beginning of Deep learning. For every Input value, it gives Output in a range ϵ (0,1).

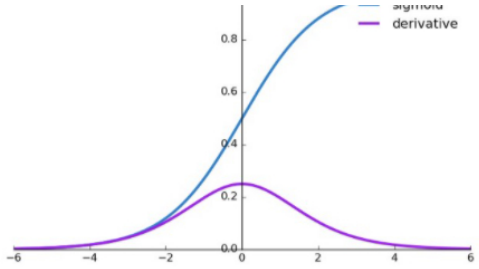
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Calculation of Derivative of Sigmoid function



$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

Graph of Sigmoid Function and it's Derivative



> Advantages of Sigmoid Function : -

1. Smooth gradient, preventing “jumps” in output values.
2. Output values bound between 0 and 1, normalizing the output of each neuron.
3. Clear predictions, i.e very close to 0 or 1.

> Sigmoid has 4 major disadvantages:

- Leads to Vanishing Gradient problem
- Leads to Gradient Saturation or dispersing
- Function output is not zero-centered
- Time consuming and Slower for computers due to using Exponential operations.

Vanishing Gradient Problem

> In Back Propagation, Derivatives of each previous layers have some contribution in updating the weights of next layers and We uses Chain rule to update the weights. And Derivatives of Sigmoid Functions ranges between (0 to 0.25).

> When Sigmoid functions are getting used as Activation functions in Deep neural networks, it's Derivative are getting decreases during updating of weights in each layers. In case of a Deep neural networks, When Back-propagation reaches to the last layer, Derivative of Sigmoid Function gives a very small value (near to 0) and so there will be very minute change or no changes in the weights in the last layer of Network. This problem is known as Vanishing Gradient problem.

> In 1980s or 1990s, Sigmoid function is the only Activation function, that people uses and they don't have too many layers in their Neural networks. They just deal with 2 or 3 layered Neural networks to avoiding this Vanishing Gradient problem.

> Visit Blog wikipedia

Non Zero - Centric Problem

> Functions having it's centre lying on the origin are Termed as Zero - Centric Functions. Sigmoid function is not Zero - Centric Function.

> When we uses a Zero - Centric function as Activation Function, it will try to control the changes or updates in gradients in backward direction.

> In case of a Non Zero-centric Function, Gradient updates will goes in different - different directions and it will make our optimisation much more harder because it is not able to maintain the consistency in Gradient updates and so every time it's value fluctuates and gives a different value.

Gradient Saturation or Gradient Dispersing Problem

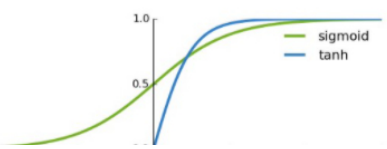
> When inputs are away from the coordinate origin, the gradient of the function becomes very small, almost zero. In the process of neural network backpropagation, we all use the chain rule of differential to calculate the differential of each weight w. When the backpropagation passes through the sigmod function, the differential on this chain is very small. Moreover, it may pass through many sigmod functions, which will eventually cause the weight w to have little effect on the loss function, which is not conducive to the optimization of the weight. This The problem is called Gradient Saturation or Gradient Dispersion.

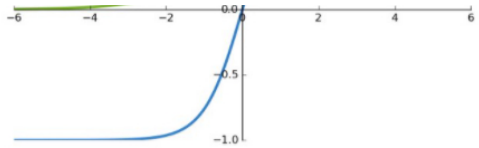
visit here

Tanh or Hyperbolic Tangent Function

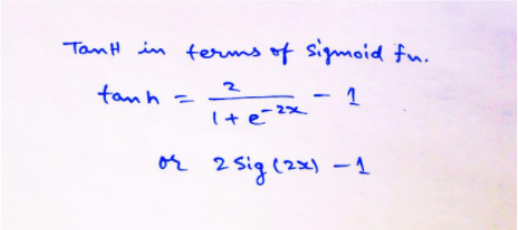
> Tanh or Hyperbolic Tangent function is modified version of the Sigmoid Function. The output interval of tanh is [-1 & 1] and the whole function is 0-centric, which is better than Sigmoid.

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





Tanh function can be represented in terms of Sigmoid function.

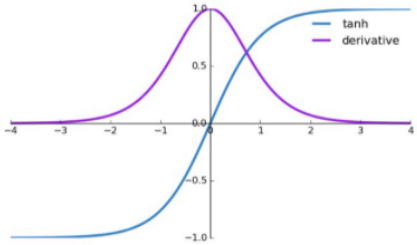


Derivative of TanH Function

For calculating the derivative of Tanh Function, we uses u/v rule.

$$a = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\frac{da}{dz} = 1 - a^2$$

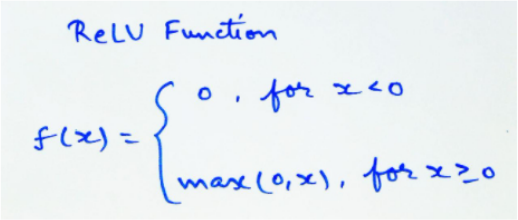


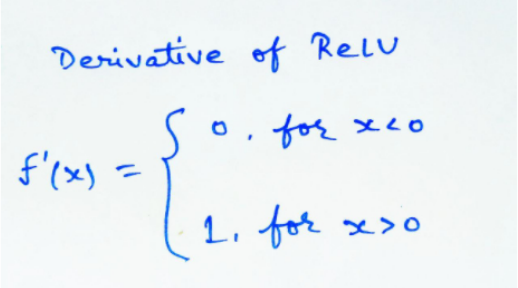
In general Binary classification problems, Tanh function is used for the hidden layer and the Sigmoid function is used for the output layer. However, these are not static, and the specific activation function to be used must be analyzed according to the specific problem, or it depends on debugging.

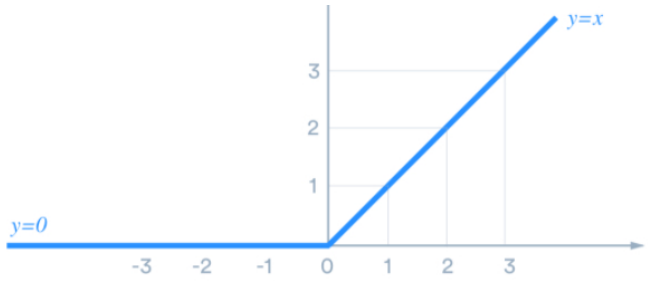
Comparing to Sigmoid Function, Tanh is a Zero centerd function but still it leads to Vanishing Gradient problem.

Rectified Linear Unit or ReLU Function

ReLU or Rectified Linear Unit function is defined as a function that takes the maximum value and outputs in a range (0, +infinity), and this function is not fully interval-derivable, but we can take sub-gradients.







> Advantages of ReLU Function.

- Overcome with Gradient Saturation or Vanishing Gradient problem. When the input is Positive, it always gives a Gradient but in case of Negative input, it gives Y as 0, so no any Gradient.
- ReLU Function not activates all Neurons at a time. Means that, If any Linear combination of inputs are negative, then it will restrict that Neuron to fire itself and gives output as 0. May be in Next iteration, if there will be Positive input, then it will Fire the Neuron. This property is very important in Feed Forward Connection.
- ReLU calculation speed is much faster because it has only a Linear relationship (forward or backward) and so it is much faster than Sigmod and Tanh. (Sigmod and tanh need to calculate the exponent, which results in slower calculation speed.)

> Disadvantages of ReLU Function

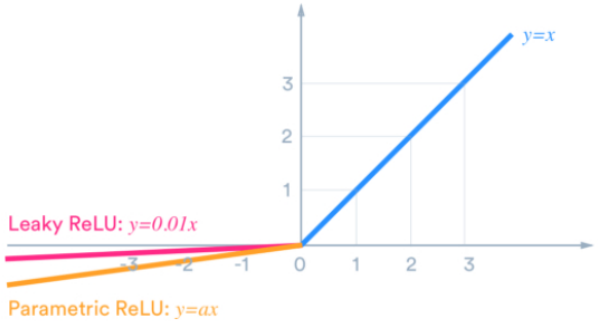
- When the input is Negative, ReLU is completely inactive, which means that once a negative number is entered, ReLU will die. In this way, in the forward propagation process, it is not a problem. Some areas are sensitive and some are insensitive. But in the back propagation process, if you enter a negative number, the gradient will be completely zero, which has the same problem as the Sigmoid function and Tanh function.
- ReLU Function is not a Zero Centric function.

Leaky ReLU Function

> Leaky ReLU Function is a modified version of ReLU Function. For Negative Inputs, It gives some values of Y. In order to solve the Dead ReLU Problem, people proposed to set the first half of ReLU 0.01x instead of 0.

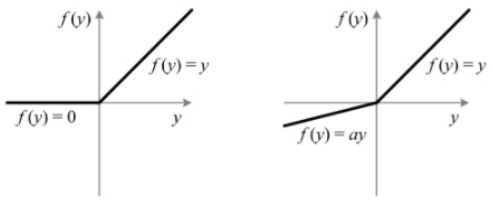
> Leaky ReLU has all the advantages of ReLU, plus there will be no problems with Dead ReLU, but in actual operation, it has not been fully proved that Leaky ReLU is always better than ReLU.

$$f(x) = \max(0.01x, x)$$



PRelu (Parametric ReLU)

> Parametric ReLU or PReLU Function is improvised version of ReLU Function. In the negative region, PReLU has a small slope, which can also avoid the problem of ReLU death. Compared to ELU, PReLU is a linear operation in the negative region. Although the slope is small, it does not tend to 0, which is a certain advantage.



ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

> In PRelu, alpha 'α' is a learnable parameter, which it learns by tuning..

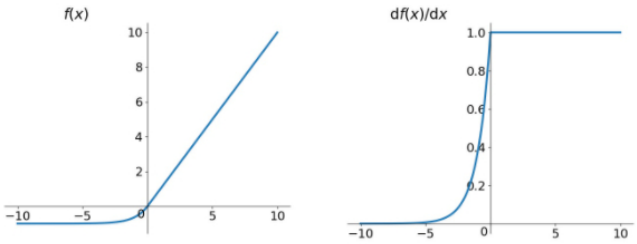
- if α = 0, f becomes ReLU
- if α > 0, f becomes Leaky ReLU

- If α is a learnable parameter, f becomes PReLU

ELU (Exponential Linear Units) function

> ELU is also proposed to solve the problems of ReLU.

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{otherwise} \end{cases}$$



> Advantages of ELU Function

- No dead Neurons issues, if input is Negative
- ELU is a Zero-centric Function means that Mean of the output is close to 0.

> Disadvantages of ELU Functions

- For negative inputs, ELU computation is expensively

Softmax function

> Softmax function is used for Multi class classifications in output layers. It is used for getting the probabilities for different multiple classes among all. It always gives an output in a range [0,1].

Softmax Function is defined as

$$S(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}, j = 1, 2, \ldots, K$$

> Softmax is different from the normal max function: the max function only outputs the largest value, and Softmax ensures that smaller values have a smaller probability and will not be discarded directly. It is a "max" that is "soft".

> The denominator of the Softmax function combines all factors of the original output value, which means that the different probabilities obtained by the Softmax function are related to each other.

> In the case of binary classification, for Sigmoid, there are:

$$p(y=1|x) = \frac{1}{1+e^{-\theta^T x}}$$

$$p(y=0|x) = 1 - p(y=1|x) = \frac{e^{-\theta^T x}}{1+e^{-\theta^T x}}$$

> For Softmax with K = 2, there are:

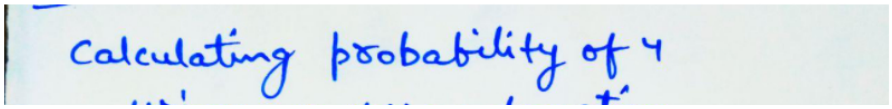
$$p(y=1|x) = \frac{e^{\theta_1^T x}}{e^{\theta_0^T x} + e^{\theta_1^T x}} = \frac{1}{1+e^{(\theta_0^T - \theta_1^T)x}} = \frac{1}{1+e^{-\beta x}}$$

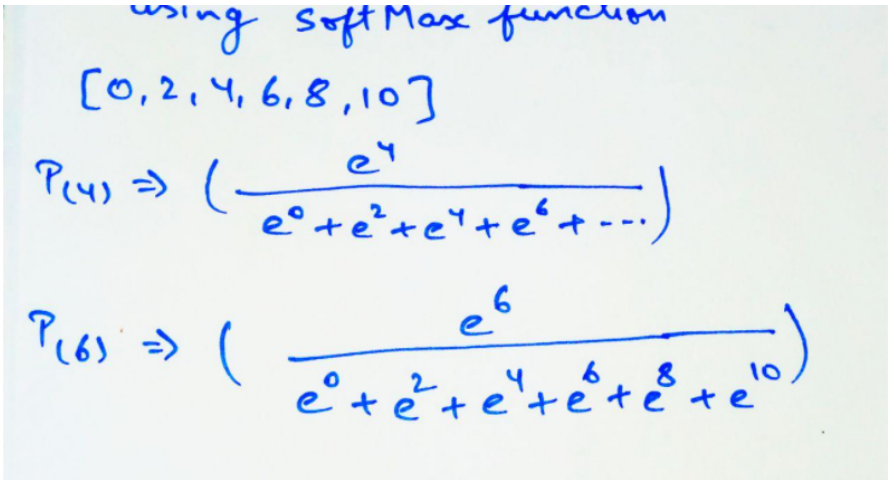
$$p(y=0|x) = \frac{e^{\theta_0^T x}}{e^{\theta_0^T x} + e^{\theta_1^T x}} = \frac{e^{(\theta_0^T - \theta_1^T)x}}{1+e^{(\theta_0^T - \theta_1^T)x}} = \frac{e^{-\beta x}}{1+e^{-\beta x}}$$

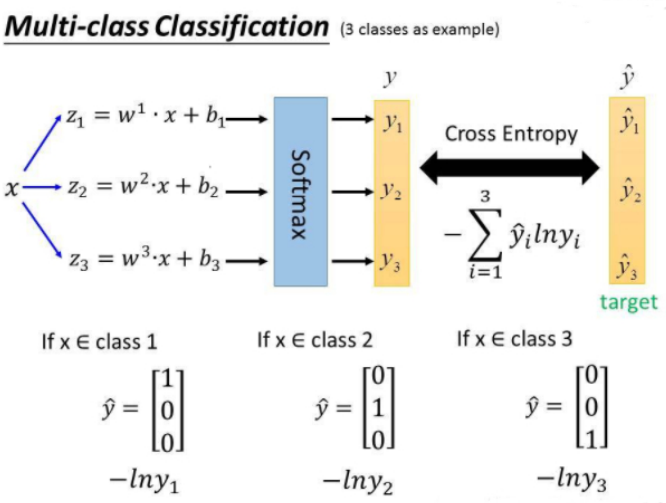
> It can be seen that in the case of binary classification, Softmax is degraded to Sigmoid.

$$\beta = -(\theta_0^T - \theta_1^T)$$

Calculation of probabilities using Softmax Function





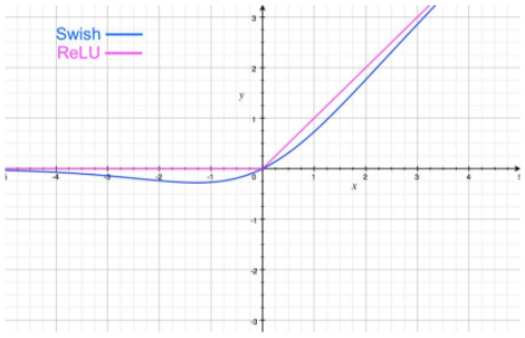


Swish (A Self-Gated) Function

> Swish Function is derived from the Sigmoid Function itself and it is defined as => **y = x * sigmoid (x)**

> Swish's design was inspired by the use of sigmoid functions for gating in LSTMs and Highway networks. We use the same value for gating to simplify the gating mechanism, which is called **self-gating**.

> The advantage of self-gating is that it only requires a simple scalar input, while normal gating requires multiple scalar inputs. This feature enables self-gated activation functions such as Swish to easily replace activation functions that take a single scalar as input (such as ReLU) without changing the hidden capacity or number of parameters.



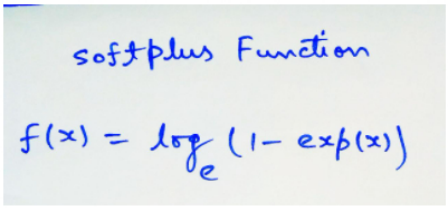
Advantages of Swish Function

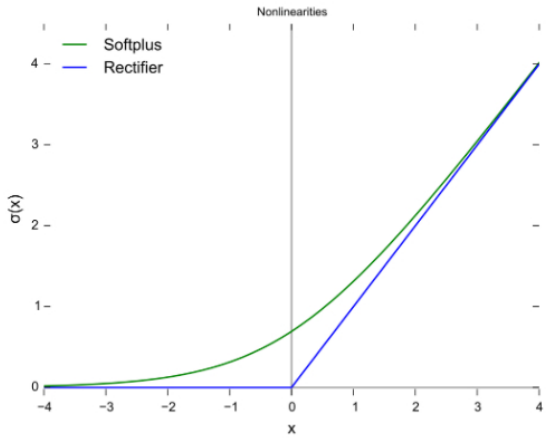
- Unboundedness (unboundedness) is helpful to prevent gradient from gradually approaching '0' during slow training, causing Saturation. At the same time, being bounded has advantages, because bounded active functions can have strong regularization, and larger negative inputs will be resolved.
- At the same time, smoothness also plays an important role in optimization and generalization.

SoftPlus Activation Function

> The softplus function is similar to the ReLU function, but it is relatively smooth.It is unilateral suppression like ReLU. It has a wide acceptance

range (0, + inf).





Maxout Activation Function

The Maxout activation function don't goes with the linear combination inputs, but it gives the maximumm input among all different inputs coming from various Neurons.

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

One relatively popular choice is the Maxout neuron (introduced recently by Goodfellow et al.) that generalizes the ReLU and its leaky version. Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have w1,b1 =0).The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks

The Maxout activation is a generalization of the ReLU and the leaky ReLU functions. It is a learnable activation function.

Maxout can be seen as adding a layer of activation function to the deep learning network, which contains a parameter k. Compared with ReLU, sigmoid, etc., this layer is special in that it adds k neurons and then outputs the largest activation value. value.

Generally speaking, these activation functions have their own advantages and disadvantages. There is no statement that indicates which ones are not working, and which activation functions are good. All the good and bad must be obtained by experiments.

Blog

Cost Functions used in Deep Learning

L1 and L2 loss

L1 and *L2* are two common Loss functions which are mainly used to minimize the error.

- **L1 loss function** is also known as **Least Absolute Deviations** in short **LAD**.
- **L2 loss function** is also known as **Least Square Errors** in short **LSE**.

L1 Loss function

It is used to minimize the error which is the sum of all the absolute differences in between the Actual value and the Predicted value.

$$L1LossFunction = \sum_{i=1}^{n} |y_{true} - y_{predicted}|$$

L2 Loss Function

> It is also used to minimize the error which is the sum of all the squared differences in between the Actual value and the Predicted value.

$$L2LossFunction = \sum_{i=1}^{n}(y_{true} - y_{predicted})^2$$

> **L1 Loss is more sensitive to outliers and controls them, so it is always advisable to use L1 Loss in case of outliers, otherwise go with L2 Loss.**

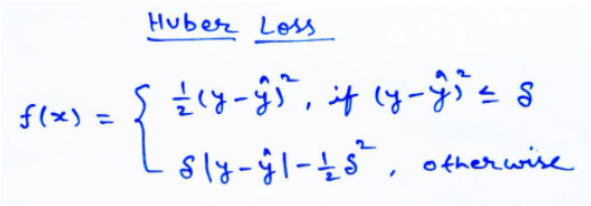
Python Implementation for L1 and L2 losses

> **l1_loss = tf.abs((y_pred - y_actual))**

> **l2_loss = tf.square((y_pred - y_actual))**

Huber Loss

> Huber Loss is a combination of L1 & L2 losses and this uses a threshold value to detect the outliers. In case of outliers, it uses L1 Loss with some regularizations otherwise it uses the L2 loss.



> Huber Loss is often used in Regression problems. Compared with L2 loss, Huber Loss is less sensitive to outliers (because if the residual is too large, it is a Piecewise function, loss is a linear function of the residual). Among them, δ is a set parameter, y represents the real value, and $f(x)$ represents the predicted value.

> The advantage of this is that when the residual is small, the loss function is L2 norm, and when the residual is large, it is a linear function of L1 norm.

> The Huber loss is quadratic when the error is smaller than a thres-hold δ (typically 1), but linear when the error is larger than δ. Thismakes it less sensitive to outliers than the mean squared error, andit is often more precise and converges faster than the mean abso-lute error.

Pseudo-Huber Loss function

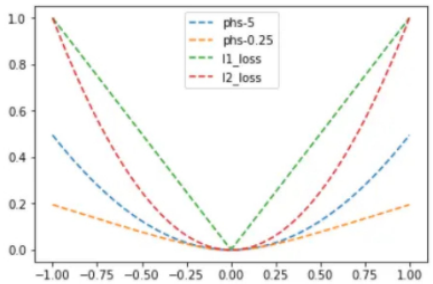
> A smooth approximation of Huber loss to ensure that each order is differentiable.

$$L_\delta(a) = \delta^2(\sqrt{1 + (a/\delta)^2} - 1).$$

As such, this function approximates $a^2/2$ for small values of a, and approximates a straight line with slope δ for large values of a.

While the above is the most common form, other smooth approximations of the Huber loss function also exist.[5]

> Where δ is the set parameter, the larger the value, the steeper the linear part on both sides.



Hinge Loss

> Hinge loss is often used for Binary classification problems, such as ground true: t = 1 or -1, predicted value y = wx + b.

> In other words, the closer the y is to t, the smaller the loss will be.

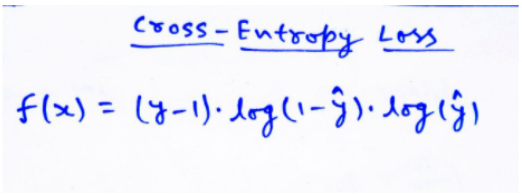
$$\ell(y) = \max(0, 1 - t \cdot y)$$

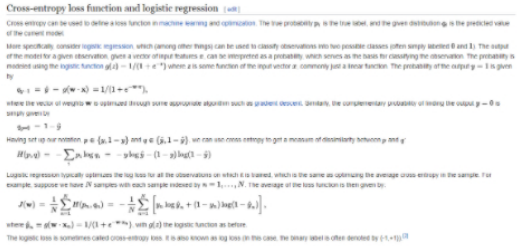
> For getting more about Hinge Loss

```
###  Implementation of Hinge Loss in Python

x_guess2 = tf.linspace(-3.,5.,500)
x_actual2 = tf.convert_to_tensor([1.]*500)

#Hinge loss
#hinge_loss = tf.losses.hinge_loss(labels=x_actual2, logits=x_guess2)
hinge_loss = tf.maximum(0.,1.-(x_guess2*x_actual2))
0with tf.Session() as sess:
    x_,hin_ = sess.run([x_guess2, hinge_loss])
    plt.plot(x_,hin_,'--', label='hin_')
    plt.legend()
    plt.show()
```

Cross Entropy Loss

> Cross Entropy Loss is mainly applied to Binary classification problems. The predicted value is a probability value and the loss is defined according to the cross entropy. Note the value range of the above value: the predicted value of y should be a probability and the value range is [0,1] .





Cross-entropy loss function and logistic regression [edit]

Cross-entropy can be used to define a loss function in machine learning and optimization. The true probability p_i is the true label, and the given distribution q_i is the predicted value of the current model.

More specifically, consider logistic regression, which (among other things) can be used to classify observations into two possible classes (often simply labelled 0 and 1). The output of the model for a given observation, given a vector of input features x, can be interpreted as a probability, which serves as the basis for classifying the observation. The probability is modeled using the logistic function $g(z) = 1/(1 + e^{-z})$ where z is some function of the input vector x, commonly just a linear function. The probability of the output $y = 1$ is given by

$$q_{y=1} = \hat{y} = g(\mathbf{w} \cdot \mathbf{x}) = 1/(1 + e^{-\mathbf{w} \cdot \mathbf{x}}),$$

where the vector of weights $\mathbf{w}$ is optimized through some appropriate algorithm such as gradient descent. Similarly, the complementary probability of finding the output $y = 0$ is simply given by

$$q_{y=0} = 1 - \hat{y}$$

Having set up our notation, $p \in \{y, 1-y\}$ and $q \in \{\hat{y}, 1-\hat{y}\}$, we can use cross-entropy to get a measure of dissimilarity between p and q:

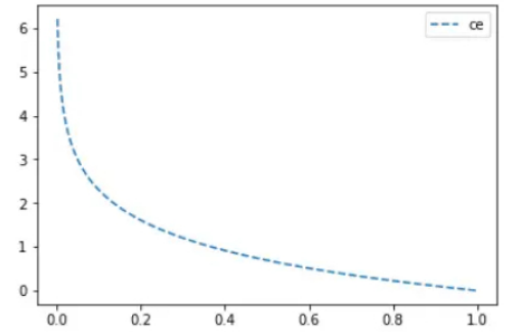
$$H(p,q) = -\sum_i p_i \log q_i = -y \log \hat{y} - (1-y) \log (1-\hat{y})$$

Logistic regression typically optimizes the log loss for all the observations on which it is trained, which is the same as optimizing the average cross-entropy in the sample. For example, suppose we have N samples with each sample indexed by $n = 1, \ldots, N$. The average of the loss function is then given by:

$$J(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N} \sum_{n=1}^{N} \left[y_n \log \hat{y}_n + (1-y_n) \log (1-\hat{y}_n) \right],$$

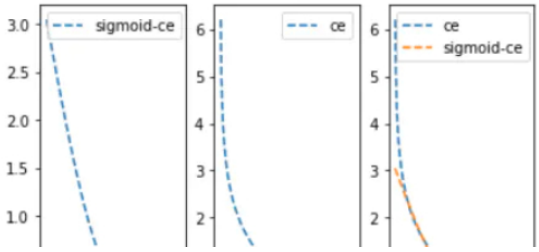
where $\hat{y}_n \equiv g(\mathbf{w} \cdot \mathbf{x}_n) = 1/(1 + e^{-\mathbf{w} \cdot \mathbf{x}_n})$, with $g(z)$ the logistic function as before.

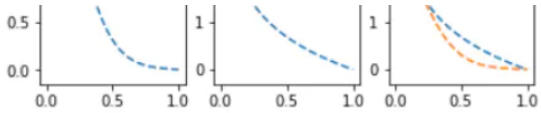
The logistic loss is sometimes called cross-entropy loss. It is also known as log loss (in this case, the binary label is often denoted by {−1,+1}).[3]



Sigmoid Cross-entropy Loss

> The above cross-entropy loss requires that the predicted value is a probability. Generally, we calculate $scores = x * w + b$. Entering this value into the sigmoid function can compress the value range to (0,1).





> It can be seen that the sigmoid function smoothes the predicted value(such as directly inputting 0.1 and 0.01 and inputting 0.1, 0.01 sigmoid and then entering, the latter will obviously have a much smaller change value), which makes the predicted value of sigmoid-ce far from the label loss growth is not so steep.

Softmax Cross-entropy Loss

> First, the softmax function can convert a set of fraction vectors into corresponding probability vectors. Here is the definition of softmax function

---

In mathematics, the **softmax function**, or **normalized exponential function**,[1]:198 is a generalization of the logistic function that "squashes" a K-dimensional vector $\mathbf{z}$ of arbitrary real values to a K-dimensional vector $\sigma(\mathbf{z})$ of real values in the range (0, 1] that add up to 1. The function is given by

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

> As above, softmax also implements a vector of 'squashes' k-dimensional real value to the [0,1] range of k-dimensional, while ensuring that the cumulative sum is 1.

> According to the definition of cross entropy, probability is required as input.Sigmoid-cross-entropy-loss uses sigmoid to convert the score vector into a probability vector, and softmax-cross-entropy-loss uses a softmax function to convert the score vector into a probability vector.

> According to the definition of cross entropy loss.

$$H(p, q) = -\sum_{x} p(x) \log q(x)$$

> where $p(x)$ represents the probability that classification x is a correct classification, and the value of p can only be 0 or 1. This is the prior value $q(x)$ is the prediction probability that the x category is a correct classification, and the value range is (0,1)

> So specific to a classification problem with a total of C types, then $p(x_j)$, $(0 <= j <= C)$ must be only 1 and C-1 is 0(because there can be only one correct classification, correct the probability of classification as correct classification is 1, and the probability of the remaining classification as correct classification is 0)

> Then the definition of softmax-cross-entropy-loss can be derived naturally.

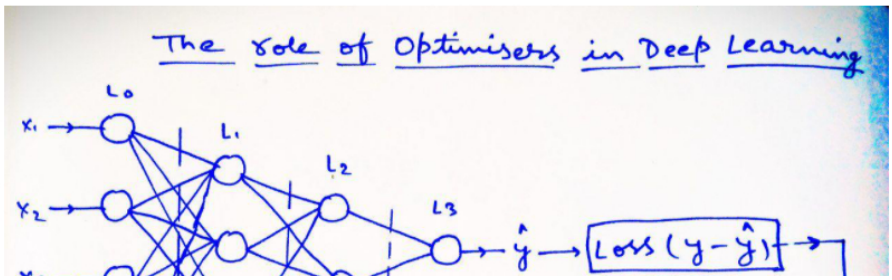
$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \qquad \text{or equivalently} \qquad L_i = -f_{y_i} + \log\sum_j e^{f_j}$$

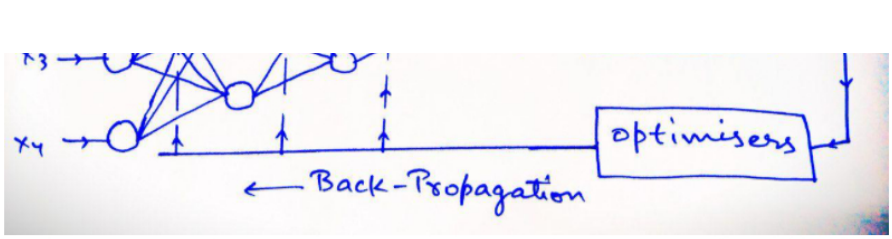
> Where f_j is the score of all possible categories, and f_{y_i} is the score of ground true class

The Role of Optimisers in Deep Learning

> Optimisers played a vital role in Neural Networks training. When data enters inside the network with Input Nodes, Feed Forward Connection starts and it will end up with getting some losses.

> Then Backward Propagation starts and it updates the randomly selected parameters layer-by-layer in the Network, by using some 'optimization' mechanism in backward direction.





> There are 3 basic optimisers based on **Gradient Descent** approach, that we uses in Deep learning.

* Batch Gradient Descent or (BGD)
* Stochastic Gradient Descent or (SGD)
* Mini Batch Gradient Descent or (MBGD)

Batch Gradient Descent

> BGD uses the entire Training set to calculate the **Gradient** of the **Cost function** to the parameters.

> In BGD, We try to send the whole dataset at a time and Records are selected randomly and enters to the network one-by-one. Then we calculate a Combined Loss or Cost by taking Summation of all losses, and then we send the Cost for optimisation for finding out the Gradients.

> In BGD, it will be able to do **Optimisation**, if it will be able to pass entire dataset to the **Neural network**.

> **Batch Gradient Descent can converge to a global minimum for convex functions and to a local minimum for non-convex functions.**

> **Disadvantages of BGD**

- Memory Consumption is too high
- Calculation of Gradients will be slow
- Computationally very expensive.
- Not good to update the model in real time

Mini-batch Gradient Descent

> In MBGD, We make batches of 'n' (n ε (50 ~ 256 advisable) records and then these mini-batches are selected randomly and enters to the Neural network one-by-one. Then we calculates Costs for each mini-batches and send for **optimisation**.

> In a single Iteration, one mini-batch passes thru the Network and we calculate the Cost and send it for **optimisation** and then update the parameters. Then in next Iteration, some other mini-batches will be passes and we calculates the cost and update the weights.

> MBGD uses a small batch of samples, that is, n samples to calculate each time. In this way, it can reduce the variance when the parameters are updated, and the convergence is more stable. It can make full use of the highly optimized matrix operations in the **Deep Learning** library for more efficient gradient calculations.

> **MBGD is modified version of BGD but with multiple mini-batches with 'n ε (50 ~ 256)' records.

> **In MBGD, if batch size is equal to Total no. of records in datset, it becomes BGD.**

> **Compare to BGD, MBGD is resource efficient, consumes low memory and calculation is faster.

Disadvantages with Mini-Batch Gradient Descent

> **Mini-batch Gradient Descent** does not gives a guarantee that we will be able to do a good convergence of Data or Error in a better way, causes due to randomly selection of batches, because Samples are extracted from dataset don't represents the properties of Entire dataset and so we don't get a good Convergence and so also not a **Absolute Global Minima or Local Minima** points.

> If **Learning rate, η** is tto small, my convergence rate falls and Time to take for finding out the Absolute minima will increase and in case of a high **Learning rate, η**, it will not be able to achieve Absolute Mnima and it will keep oscillating between Maximum values and minimum values, causes due to different errors getting for different mini-batches.

> We should control the **Learning rate, η** in case of MBGD. In addition, this method is to apply the **same learning rate** to all parameter updates. If our data is sparse, we would prefer to update the features with lower frequency.

> In addition, for **non-convex functions**, it is also necessary to avoid trapping at the local minimum or saddle point, because the error around the saddle point is the same, the gradients of all dimensions are close to 0, and SGD is easily trapped here.

> **Saddle points** are the curves, surfaces, or hypersurfaces of a saddle point neighborhood of a smooth function are located on different sides of a tangent to this point. For example, this two-dimensional figure looks like a saddle: it curves up in the x-axis direction and down in the y-axis direction, and the saddle point is (0,0).

Stochastic Gradient Descent

> In **Stochastic Gradient Descent**, a single record is selected and send to the **Neural networks**. We calculates the Loss for this single record and send it for **optimisation** and the update the weights. Then in next Iterations, some other records are sent to the network individually and we calculates the loss and the update the weights.

> In SGD, **Loss function** and **optimisers** are not supposed to wait for the entire dataset to calculate themselves.

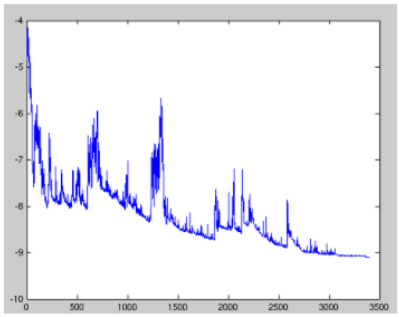
> Comparing to BGD, SGD works just with a single iteration and so requires less computational resources but takes more time to train the network.

> IN BGD, it will get converged to **Local Minima** point, but in SGD, it will always oscillates or vary between one point to other for each and every dataset, so it's very difficult to get **Absolute Minima** point.

> IN SGD, gets multiple Minimum values for entire dataset and we get some **zig-zag** type curve and minima's will keep fluctuating.

> For SGD, **Learning rate, η** should be lesser comparing to BGD and MBGd.

> SGD is faster than BGD and MBGD.



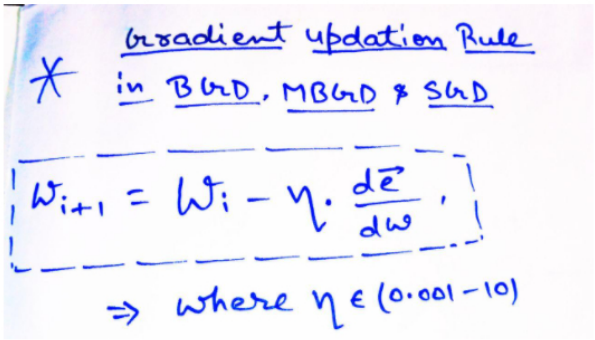
Fluctuations in SGD

Disadvantages:

> However, because SGD is updated more frequently, the cost function will have severe oscillations. BGD can converge to a local minimum, of course, the oscillation of SGD may jump to a better local minimum.

> When we decrease the learning rate slightly, the convergence of SGD and BGD is the same.

BGD, MBGD and SGD are some ways to pass the data from Neural networks.



Optimizers are ways to calculate the Gradients and update the parameters.

Concept of Momentum in Deep Learning

> **Momentum is momentum**, which simulates the inertia of an object when it is moving, that is, the direction of the previous update is retained to a certain extent during the update, while the current update gradient is used to fine-tune the final update direction. In this way, you can increase the stability to a certain extent, so that you can learn faster, and also have the ability to get rid of local optimization.

> Using **Momentum-based Optimizers**, We get smooth convergence of data and takes very less time to calculate the gradients.

SGD SGD

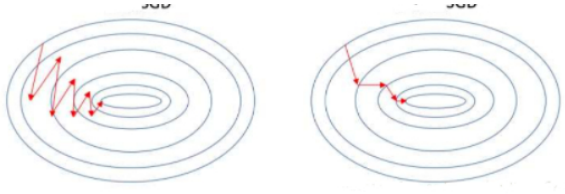


Figure :- SGD without Momentum &&& SGD with Momentum

Adaptive Momentum-based Gradient Descent optimiser or AdaGrad

> **Adagrad** is an algorithm for gradient-based optimization, which adapts the **Learning rates** to the parameters, using low **Learning rate** for parameters associated with frequently occurring features, and using high **Learning rates** for parameters associated with infrequent features. So, it is well-suited for dealing with **sparse data**.

> In very beginning, We starts from a high **Learning rate** and gradually as I will increase my Iterations, I will keep decreasing my **Learning rate** because in very beginning, our **Neural network** will not be aware about dataset and Every time we receives new variances or records of data, but in Later stage, once my **Neural network** is able to pass data for certain epochs (or Iterations) and so my model has alreday adjusted the weights and if their will be some changes in features are not frequent, we will slow down the **Learning rate**. In this case, I am providing my model a better possibility to converge for optimise the Losses.

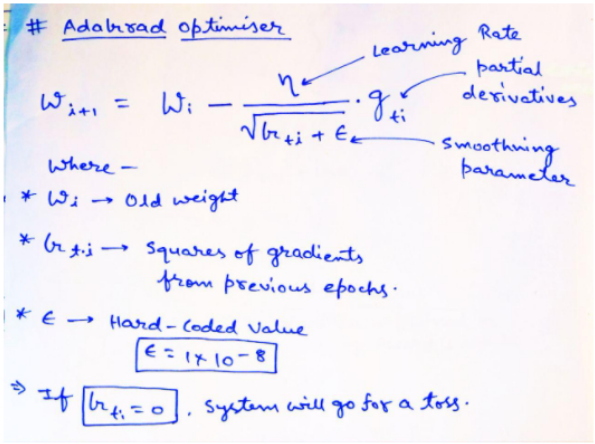
> I will be able to define the **Accuracy** of my model, if it will converge and this way, we uses to keep changing the **Learning rate** and will not use a constant **Learning rate** throughout the entire learning period.

> A constant **Learning rate** may not be suitable for all parameters. For example, some parameters may have reached the stage where only fine-tuning is needed, but some parameters need to be adjusted a lot due to the small number of corresponding samples.

> Adagrad proposed as an algorithm that adaptively assigns different **Learning rates** to various parameters among them.

> **GloVe word embedding uses AdaFrad where infrequent words required a greater update and frequent words require smaller updates.**

> **AdaGrad eliminates the need to manually tune the learning rate.**

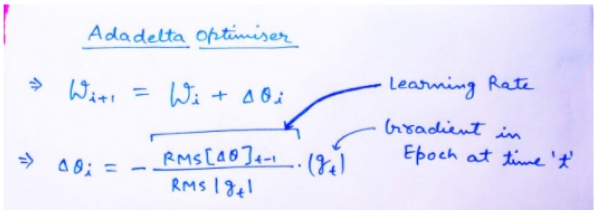


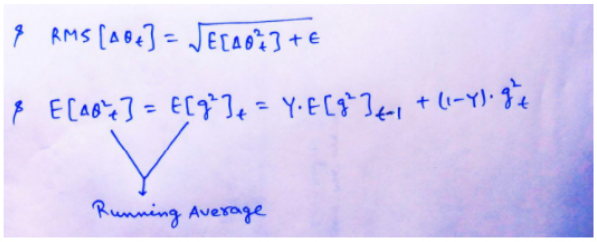
Disadvantages in Adagrad optimiser

- **The learning rate is monotonically decreasing.**
- **The learning rate in the late training period is very small.**
- **It requires manually setting a global initial learning rate.**
- **Computationally very Expensive**

Adadelta

> **Adadelta** is an extension of **Adagrad** and it tries to reduce Adagrad's property of aggressively and monotonically reducing the **Learning rate**.





> In **AdaGrad optimiser**, we uses summation of squares of gradients from all previous epochs, but In **Adadelta optimiser**, we uses **root-mean-square** of gradient of this current epoch and uses **Learning rate** from previous epoch.

> In **Adadelta**, we take the ratio of the **Running average** of the previous epochs to the current gradient and we do not need to set the default **Learning rate**.

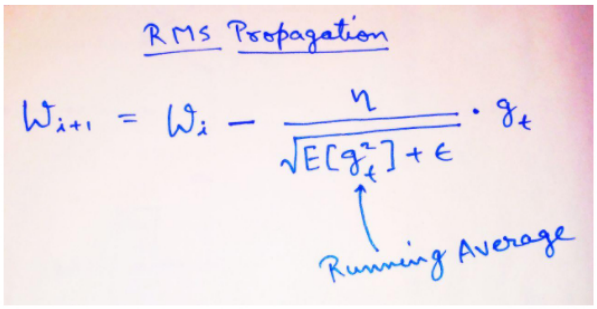
> So,this way! I am able to eliminate the dependencies of selecting a Hard-coded value as **Learning rate** and this **Learning rate** is dynamic and will not be decreased to a small value because learning rate will be changed in each & every epochs.

RMSProp Optimiser

> The full name of RMSProp algorithm is **Root Mean Square Propagation**, which is an adaptive learning rate optimization algorithm proposed by **Geoffrey Hinton**.

> **Geoff Hinton** is known as **Father of Deep learning** and get **Alan Turing award** in 2018.

> **RMSProp** tries to resolve Adagrad's radically diminishing **Learning rates** by using a moving average of the squared gradient. It utilizes the magnitude of the recent gradient descents to normalize the gradient.



> **Adagrad** will accumulate all previous gradient squares, but **RMSProp** just calculates the corresponding average value, so it can alleviate the problem that the learning rate of the **Adagrad algorithm** drops quickly.

> The difference is that **RMSProp** calculates the **differential squared weighted average of the gradients** and this makes the network functions converge faster.

> In **RMSProp**, **Learning rate** gets adjusted automatically and it chooses a different learning rate for each parameter.

> **RMSProp** divides the learning rate by the average of the exponential decay of squared gradients

Adam

Adaptive Moment Estimation optimiser, Adam is another method that computes adaptive learning rates for each parameter by storing an exponentially decaying average of past squared gradients like **Adadelta** and **RMSprop**.

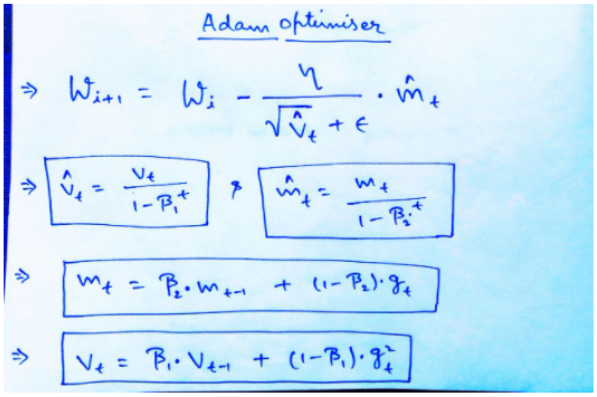
> **Adam** also keeps an exponentially decaying average of past gradients, similar to momentum.

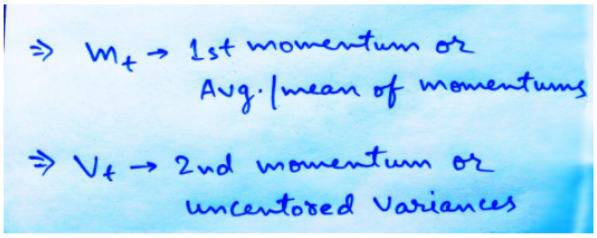
> **Adam** can be viewed as a combination of **Adagrad** and **RMSprop** so that **Adagrad** works well on sparse gradients and **RMSProp** works well in online and non-stationary settings repectively.

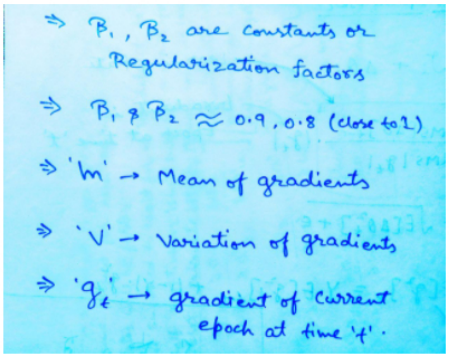
> **Adam** implements the **Exponential moving average of the gradients** to scale the **Learning rate** instead of a simple average as in **Adagrad**. It keeps an exponentially decaying average of past gradients.

> **Adam optimizer** is computationally efficient and has very less memory requirement and itis one of the most popular and famous **Gradient descent-based Optimization** algorithms.

> **Adam optimizer** comes with **SGD**, by default but also can be used with other.







> **Uncentored Variances** are those variances which, are not able to be defined in both directions.

> **AdaMax Optimizer**, **N_Adam optimizer** & **MS Grad optimizer** are some more good optimizers.

Comparisions

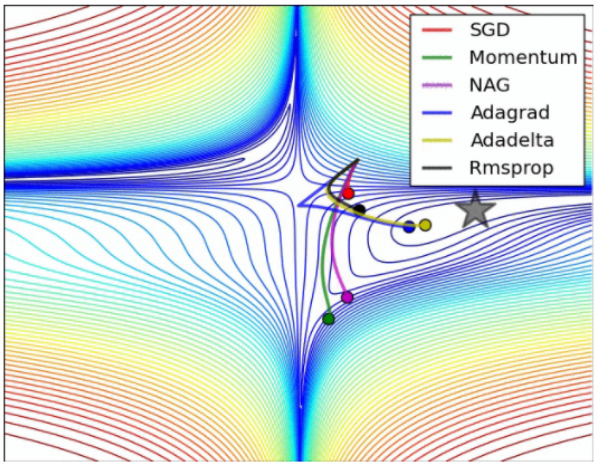
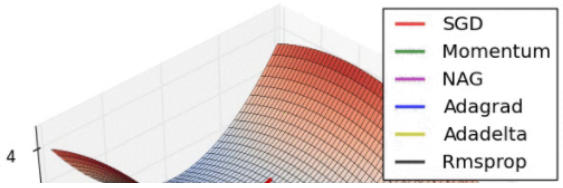


Figure :- SGD optimization on loss surface contours



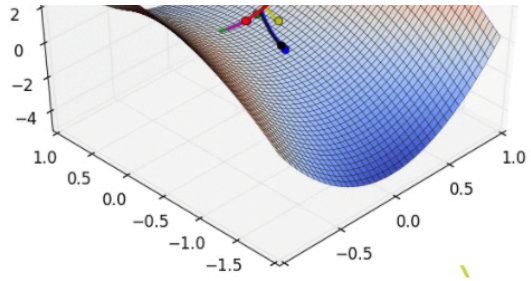


Figure :- SGD optimization on saddle point

How to choose optimizers?

- If the data is sparse, use the self-applicable methods, namely Adagrad, Adadelta, RMSprop, Adam.
- RMSprop, Adadelta, Adam have similar effects in many cases.
- Adam just added bias-correction and momentum on the basis of RMSprop,
- As the gradient becomes sparse, Adam will perform better than RMSprop.

Overall, Adam is the best choice.

> SGD is used in many papers, without momentum, etc. Although SGD can reach a minimum value, it takes longer than other algorithms and may be trapped in the saddle point.

- If faster convergence is needed, or deeper and more complex neural networks are trained, an adaptive algorithm is needed.

Multi-layered Perceptrons Architectures

Architecture of a Neural network used for Regression

> MLPs can be used for Regression tasks. If you want to predict a single value (e.g.,the price of a house), then you just need a single output neuron, its output is the predicted value.

> For Multivariate Regression or to predict multiple values at once, you need one output neuron per output dimension.

> For example, to locate the center of an object on an image, you need to predict 2D coordi-nates, so you need two output neurons. If you also want to place a bounding boxaround the object, then you need two more neurons for the 'width' and 'height' of the object and you end up with 4 output neurons.

> In general, when building an MLP for Regression, you do not want to use any Activation function for the output neurons, so they are free to output any range of values. However, if you want to guarantee that the output will always be positive, then you can use the ReLU activation function, or Softplus Activation function in the output layer.

> Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the Logistic function or the Hyperbolic Tangent, and scale the labels to the appropriate range (0 to 1) for the logistic function, or (–1 to 1) for the Hyperbolic Tangent function.

> The Loss function to use during training is typically the Mean Squared Error, but if you have lot of outlier in the training set, you may prefer to use the Mean Absolute Error instead. Alternatively, you can use the Huber loss, which is a combination of both.

Table 10-1. Typical Regression MLP Architecture

Hyperparameter	Typical Value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem. Typically 1 to 5.
# neurons per hidden layer	Depends on the problem. Typically 10 to 100.
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification MLP Architecture ¶

> MLPs can also be used for Classification tasks.

For a Binary classification problem, you just need a single output neuron using the Logistic Activation function and the output will be a number between (0 & 1), which you can interpret as the estimated probability of the Positive class. Obviously, the estimated probability of the Negative class is equal to one minus that number.

MLPs can also easily handle multi-labeled Binary classification tasks. For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or non-urgent email.

In this case, you would need two Output neurons, both using the Logistic Activation function. The first would output the probability that the email is spam and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each Positive class. Note that the output probabilities do not necessarily add up to one. This lets the model output any combination of labels. You can have non-urgent ham, urgent ham, non-urgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of 3 or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the Softmax Activation function for the whole output layer.

The Softmax function will ensure that all the estimated probabilities are between 0 and 1 and that they add up to one (which is required if the classes are exclusive). This is called Multiclass classification.

As Loss function, Cross - entropy (or Log loss) function is used.

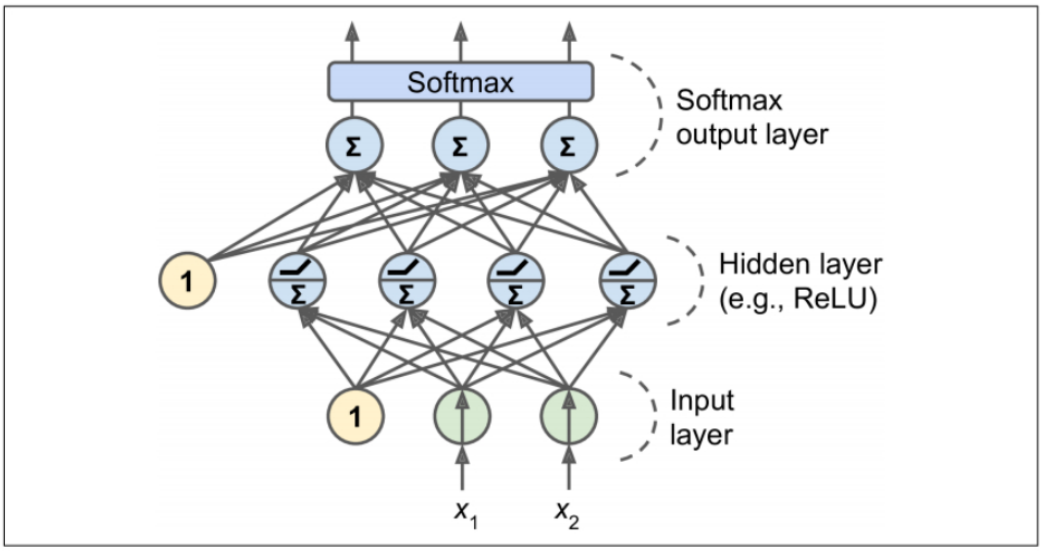


Figure 10-9. A modern MLP (including ReLU and softmax) for classification

Table 10-2. Typical Classification MLP Architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross-Entropy	Cross-Entropy	Cross-Entropy