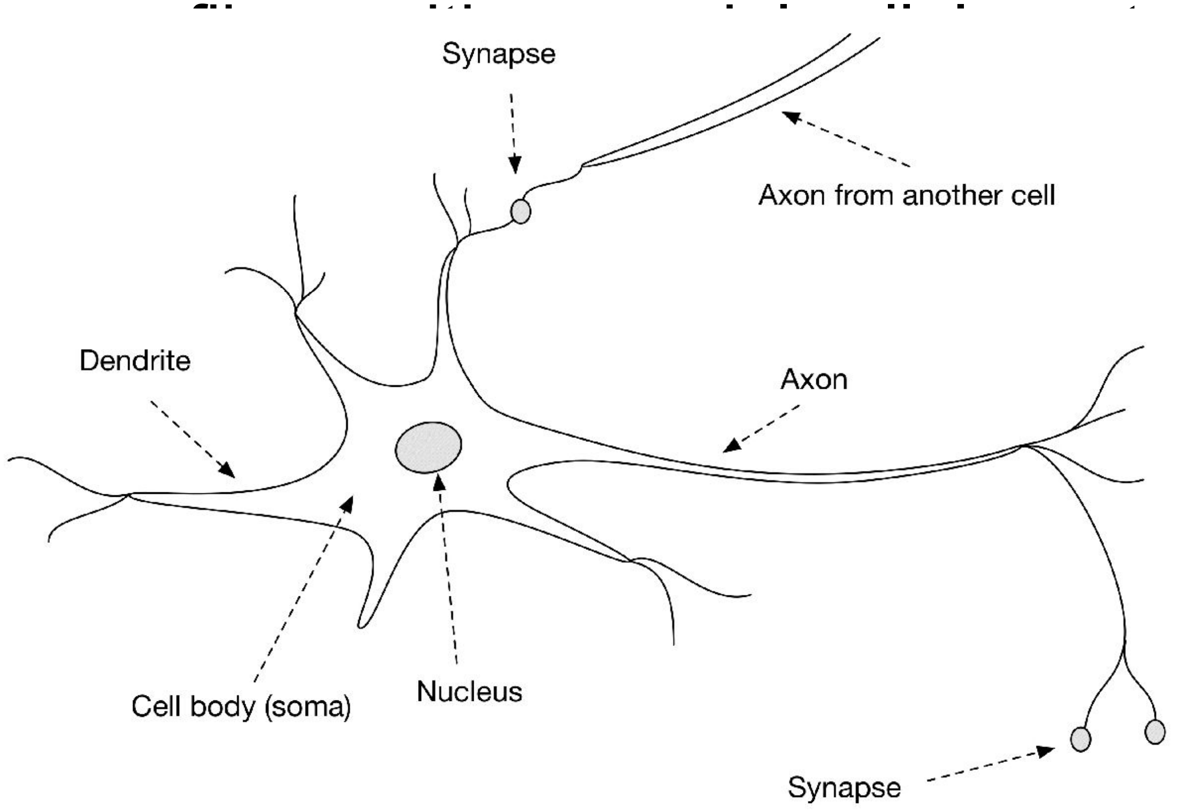
The Biological Neuron

- The biological neuron is a nerve cell that provides the fundamental functional unit for the nervous systems of all animals.
- Neurons exist to communicate with one another, and pass electro-chemical impulses across synapses, from one cell to the next, as long as the impulse is strong enough to activate the release of chemicals across a synaptic cleft.
- The strength of the impulse must surpass a minimum threshold or chemicals will not be released.

Major parts of the nerve cell:

- Soma
- Dendrites
- Axons
- Synapses

- The neuron is made up of a nerve cell consisting of a soma (cell body) that has many dendrites but only one axon. The single axon can branch hundreds of times, however. Dendrites are thin structures that arise from the main cell body.
- Axons are [illegible] nsion that comes from the cell body.



Biological Neuron

Synapses

- Synapses are the connecting junction between axon and dendrites.
- The majority of synapses send signals from the axon of a neuron to the dendrite of another neuron.
- The exceptions for this case are when a neuron might lack dendrites, or a neuron lacks an axon, or a synapse, which connects an axon to another axon.

Dendrites

- Dendrites have fibers branching out from the soma in a bushy network around the nerve cell.
- Dendrites allow the cell to receive signals from connected neighboring neurons and each dendrite is able to perform multiplication by that dendrite’s weight value.
- Here multiplication means an increase or decrease in the ratio of synaptic neurotransmitters to signal chemicals introduced into the dendrite.

Axons

- Axons are the single, long fibers extending from the main soma.
- They stretch out longer distances than dendrites and measure generally 1 centimeter in length (100 times the diameter of the soma).
- Eventually, the axon will branch and connect to other dendrites.
- Neurons are able to send electrochemical pulses through cross-membrane voltage changes generating *action potential*.
- This signal travels along the cell’s axon and activates synaptic connections with other neurons.

Information flow across the biological neuron

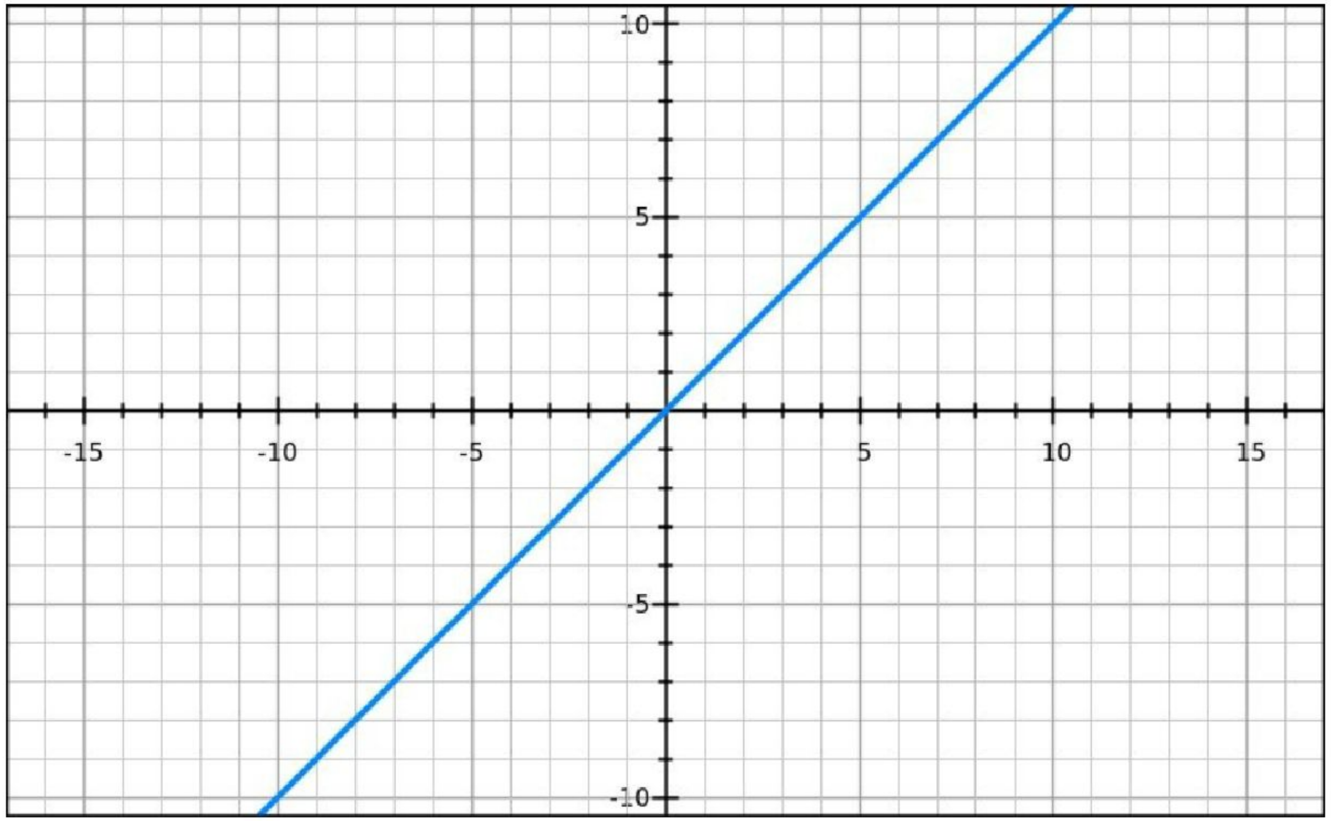
- Synapses that increase the potential are considered excitatory, and those that decrease the potential are considered inhibitory.
- Plasticity refers the long-term changes in strength of connections in response to input stimulus.
- Neurons also have been shown to form new connections over time and even migrate.
- These combined mechanics of connection change drive the learning process in the biological brain.

Activation Functions

- We use activation functions to propagate the output of one layer’s nodes forward to the next layer (up to and including the output layer).
- Activation functions are a scalar-to-scalar function, yielding the neuron’s activation.
- We use activation functions for hidden neurons in a neural network to introduce nonlinearity into the network’s modeling capabilities.
- Many activation functions belong to a logistic class of transforms that (when graphed) resemble an S.
- This class of function is called sigmoidal. The sigmoid family of functions contains several variations, one of which is known as the Sigmoid function.
- Let’s now take a look at some useful activation functions in neural networks.

Linear

- A linear transform is basically the identity function, and f(x) = Wx, where the dependent variable has a direct, proportional relationship with the independent variable.
- In practical terms, it means the function passes the signal through unchanged



Sigmoid

- Like all logistic transforms, sigmoids can reduce extreme values or outliers in data without removing them.
- A sigmoid function is a machine that converts independent variables of near infinite range into simple probabilities between 0 and 1, and most of its output will be very close to 0 or 1.
- This essentially means that when I have multiple neurons having sigmoid function as their activation function – the output is non linear as well.
- The function ranges from 0-1 having an S shape. This is vanishing gradient descent because the value reach zero two end in derivative calculation.

2. Tanh Function:

Description: *Similar to sigmoid but takes a real-valued number and scales it between -1 and 1.It is better than sigmoid as it is centred around 0 which leads to better convergence*

Formula: (e^x — e^-x) / (e^x + e^-x)

$f^1(x) = g(x) = 1=tanh^2(x)$

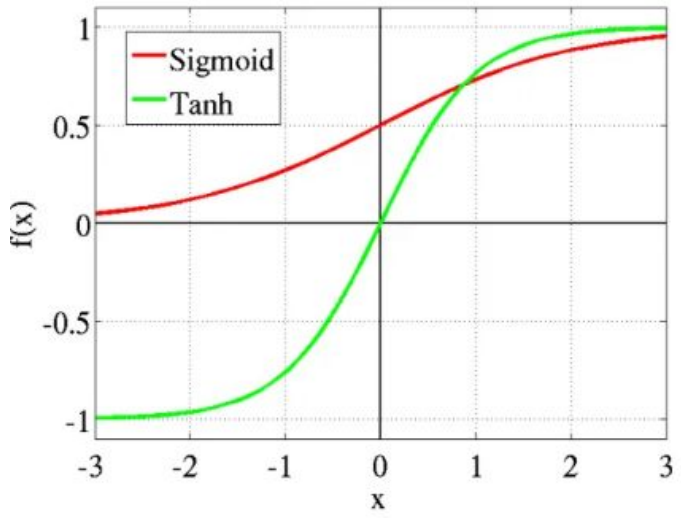
Range: (-1,1)

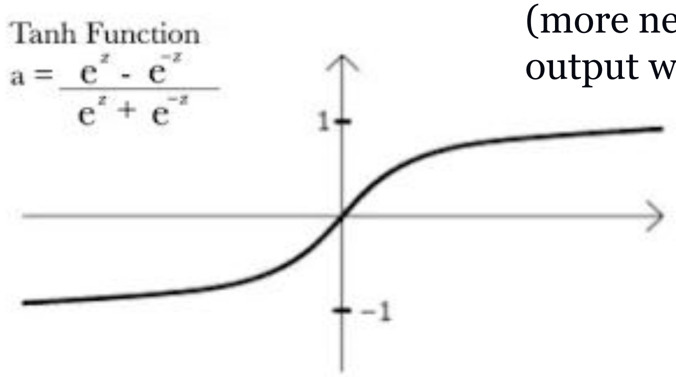
Pros: *The derivatives of the tanh are larger than the derivatives of the sigmoid which help us minimize the cost function faster*

Cons: *Similar to sigmoid, the gradient values become close to zero for wide range of values (this is known as vanishing gradient problem). Thus, the network refuses to learn or keeps learning at a very small rate.*

In Tanh, the larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.

Plot:





Hard Tanh

- Similar to tanh, hard tanh simply applies hard caps to the normalized range.
- Anything more than 1 is made into 1, and anything less than –1 is made into –1.
- This allows for a more robust activation function that allows for a limited decision boundary.

3. Softmax Function:

Description: *Softmax function can be imagined as a combination of multiple sigmoids which can returns the probability for a datapoint belonging to each individual class in a multiclass classification problem*

Formula:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

σ	= softmax
$\vec{z}$	= input vector
e^{z_i}	= standard exponential function for input vector
K	= number of classes in the multi-class classifier
e^{z_j}	= standard exponential function for output vector

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Range: (0,1), sum of output = 1

Pros: *Can handle multiple classes and give the probability of belonging to each class*

Cons: *Should not be used in hidden layers as we want the neurons to be independent. If we apply it then they will be linearly dependent.*

4. ReLU Function:

Description: *The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. This is the default function but modifying default parameters allows us to use non-zero thresholds and to use a non-zero multiple of the input for values below the threshold (called Leaky ReLU).*

Formula: max(0,x)

Range: (0,inf)

Pros: *Although RELU looks and acts like a linear function, it is a nonlinear function allowing complex relationships to be learned and is able to allow learning through all the hidden layers in a deep network by having large derivatives.*

Cons: *It should not be used as the final output layer for either classification/regression tasks*

Plot:

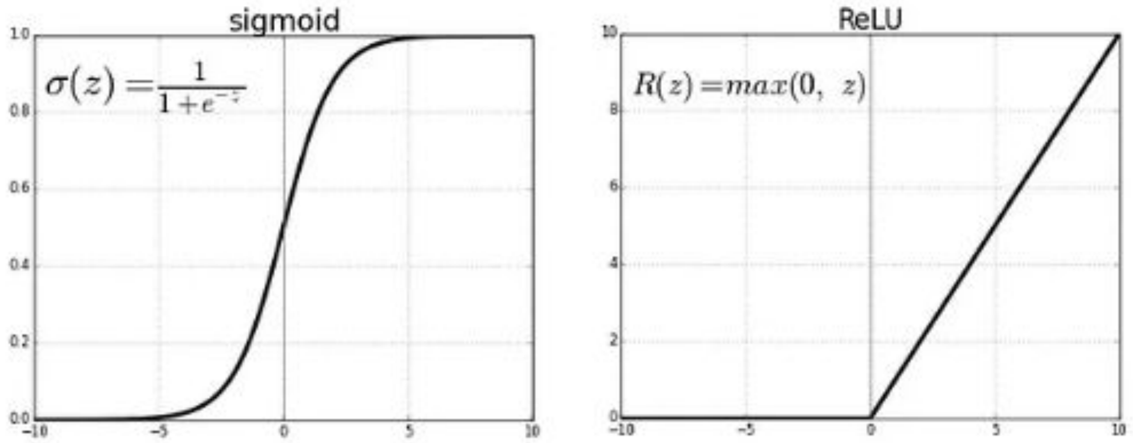
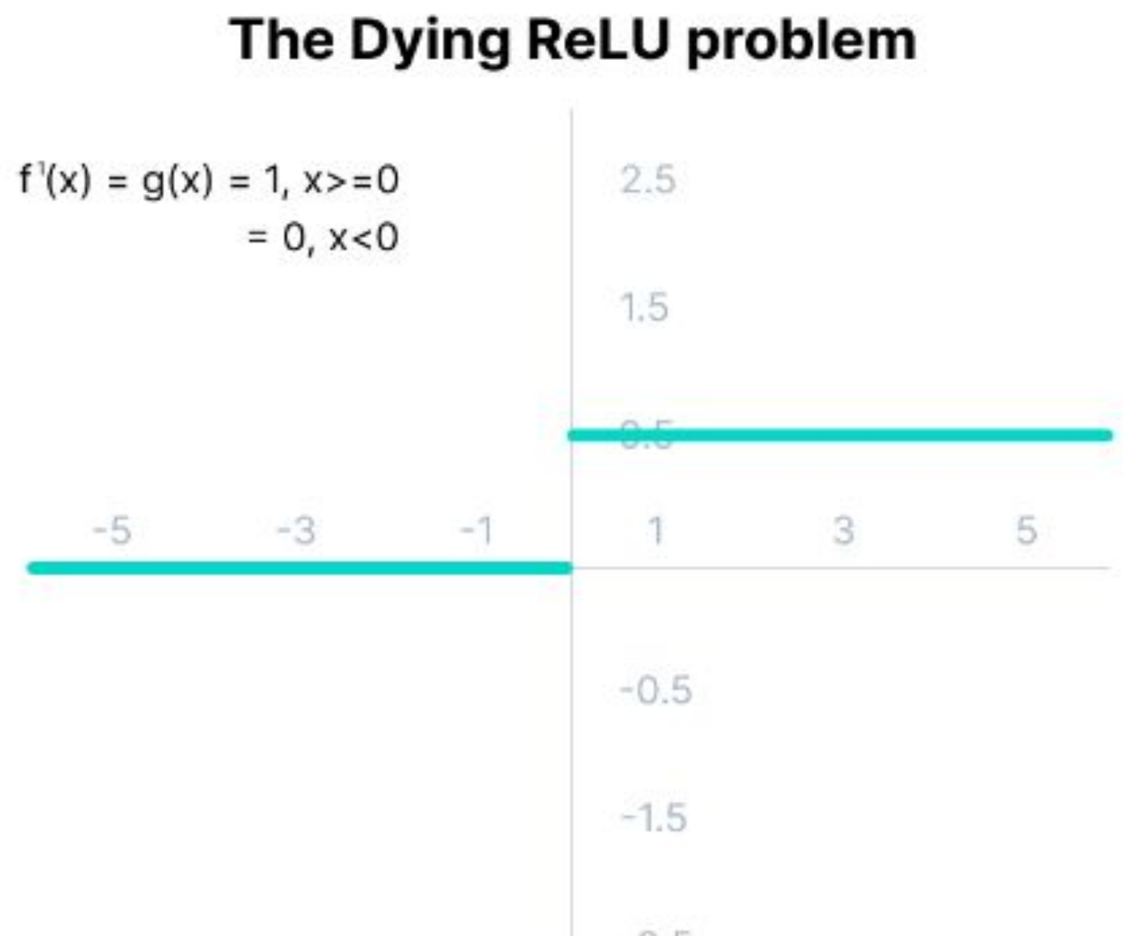


Fig: ReLU v/s Logistic Sigmoid

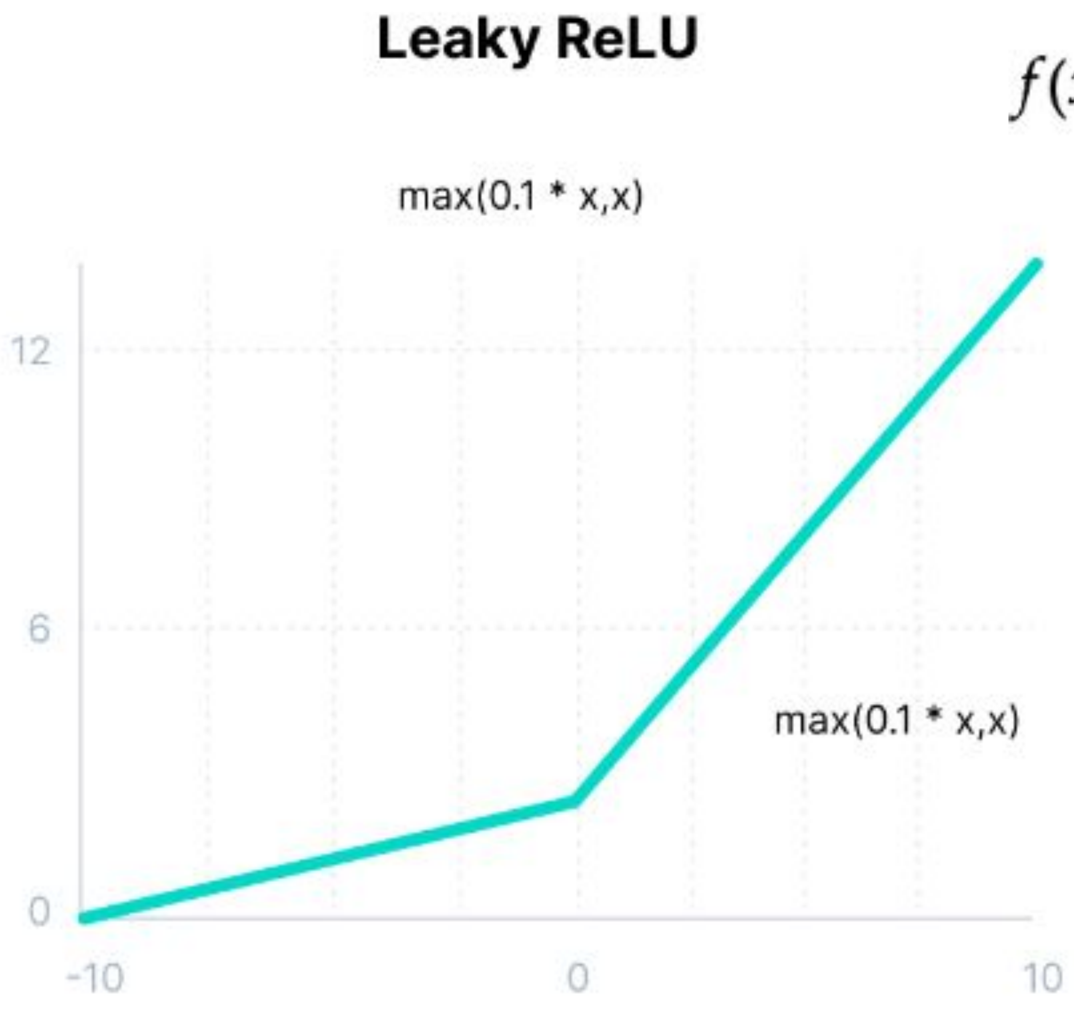
- The negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated.



- **Leaky ReLU Function -**Leaky ReLU is an improved version of ReLU function to **solve the Dying ReLU problem as it has a small positive slope in the negative area.**

Leaky ReLU

$$f(x) = max\ (0.1x, x)$$



- The advantages of Leaky ReLU are same as that of ReLU, in addition to the fact that **it does enable backpropagation, even for negative input values.**
- By making this minor modification for negative input values, the **gradient of the left side of the graph comes out to be a non-zero value.** Therefore, **we would no longer encounter dead neurons in that region.**
- **Here is the derivative of the Leaky ReLU function.**

Summary of Activation Function

- Sigmoid and softmax functions are now generally used in the output layer in case of binary and multi-class classification respectively.
- ReLU is now popularly used in the hidden layer as it is computationally faster and does not suffers from vanishing gradient problems.
- ReLU can suffer from dying neurons problem, to avoid this Leaky ReLU is used sometimes.
- While selecting an activation function, you must consider the problems it might face: vanishing and exploding gradients.
- Regarding the output layer, we must always consider the expected value range of the predictions. If it can be any numeric value (as in case of the regression problem) you can use the linear activation function or ReLU.
- Use Softmax or Sigmoid function for the classification problems.

1. **ReLU activation function should only be used in the hidden layers.**
2. **Sigmoid/Logistic and Tanh functions should not be used in hidden layers** as they make the model more susceptible to problems **during training (due to vanishing gradients)**.

Finally, a few rules for choosing the activation function for your output layer based on the type of prediction problem that you are solving:

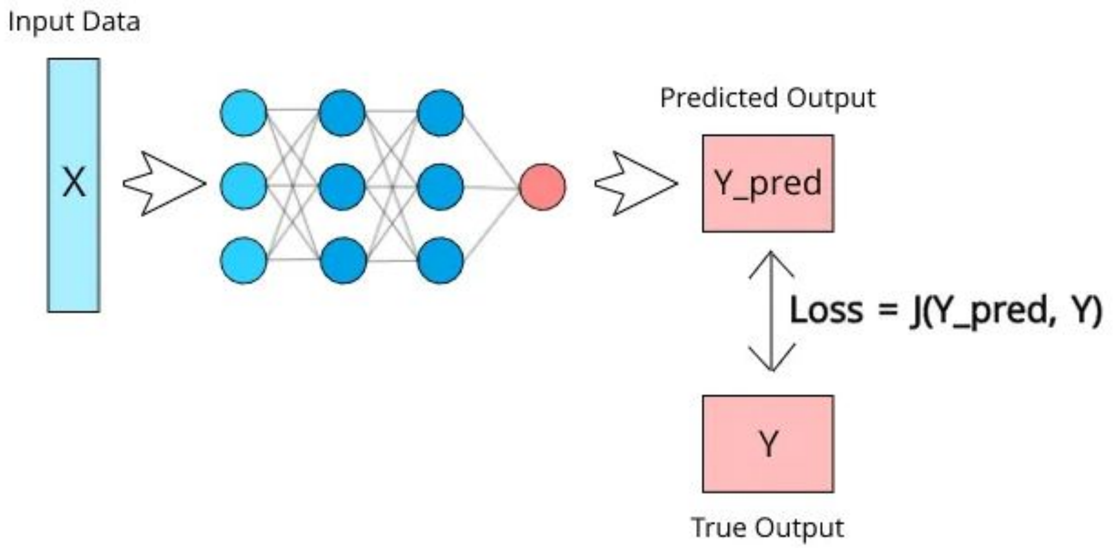
1. **Regression - Linear Activation Function**
2. **Binary Classification—Sigmoid/Logistic Activation Function**
3. **Multiclass Classification—Softmax**
4. **Multilabel Classification—Sigmoid**

The activation function used in hidden layers is typically chosen based on the type of neural network architecture.

5. **Convolutional Neural Network (CNN): ReLU activation function.**
6. **Recurrent Neural Network: Tanh and/or Sigmoid activation function.**

Loss Function

- The **loss function** is the function that **computes the distance between the current output of the algorithm and the expected output**.
- It's a method to evaluate how your algorithm models the data.
- It can be categorized into two groups. One for **classification** (discrete values, 0,1,2…) and the other for **regression** (continuous values).



Divergence generally means two things are moving apart while convergence implies that two forces are moving together.

Loss Function Notation

Equations in this section will use the notation described here:

- Consider the dataset gathered to train a neural net. Let "*N*" denote the number of samples (set of inputs with corresponding outcomes) that have been gathered.

- Consider the nature of the input and output collected. Each data point records some set of unique input features and output features. Let "*P*" denote the number of input features gathered and "*M*" denote the number of output features that have been observed.

- We will use (*X*,*Y*) to denote the input and output data we collected. Note that there will be *N* such pairs where the input is a collection of *P* values and the output *Y* is a collection of *M* values. We will denote the *ith* pair in the dataset as X_i and Y_i.

- We will use $\hat{Y}$ to denote the output of the neural net. Of course, $\hat{Y}$ is the network's guess at *Y* and therefore it will also have *M* features.

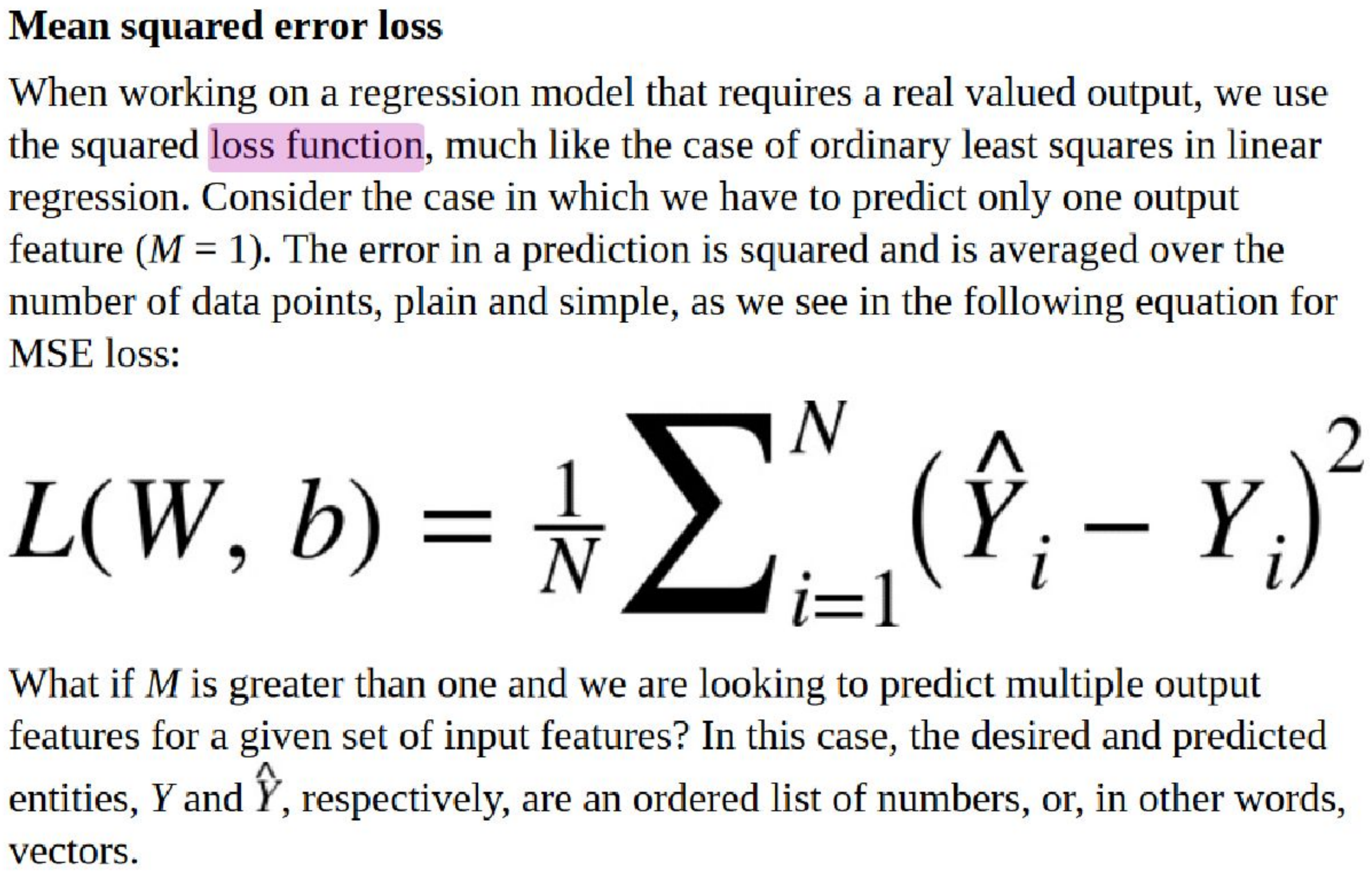
- We will use the notation $h(X_i) = \hat{Y}_i$ to denote the neural network transforming the input X_i to give the output $\hat{Y}_i$. We will alter this notation a little later to emphasize its dependence on weights and biases.

- When referring to *jth* output feature, we will use it as a subscript firmly linking our notation to a matrix where the rows are different data points and the columns are the different unique features. Thus $y_{i,j}$ refers to the *jth* feature observed in the *ith* sample collected.

- We will represent the loss function by *L(W,b)*.

Given the data available, the loss function notation indicates that its value depends only on *W* and *b*, the weights and the biases of the neural network. This cannot be emphasized enough. In the universe of a given neural network with a set number of layers, configurations, and so on that will be trained on a given set of data, the value of the loss function depends exclusively on the state of the

Loss Functions for Regression



Mean squared error loss

When working on a regression model that requires a real valued output, we use the squared loss function, much like the case of ordinary least squares in linear regression. Consider the case in which we have to predict only one output feature ($M = 1$). The error in a prediction is squared and is averaged over the number of data points, plain and simple, as we see in the following equation for MSE loss:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{Y}_i - Y_i\right)^2$$

What if M is greater than one and we are looking to predict multiple output features for a given set of input features? In this case, the desired and predicted entities, Y and $\hat{Y}$, respectively, are an ordered list of numbers, or, in other words, vectors.

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{M}\sum_{j=1}^{M}(\hat{y}_{ij} - y_{ij})^2$$

- Note that *N*, the size of your dataset, and *M*, the number of features the network has to predict, are constants. So, consider these as simple scaling factors that you can account for in other ways (like by scaling the learning rate).
- In a lot of use cases the *M* is dropped and a division by two is added for mathematical convenience In the following equation, we see the version of MSE:

$$L(W, b) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\hat{y}_{ij} - y_{ij})^2$$

Other loss functions for regression

Although the MSE is used widely, it is quite sensitive to outliers, and this is something that you should consider when picking a loss function. When picking a stock to invest in, we want to take the outliers into account. But perhaps when buying a house we don’t.

In this case, what is of most interest is what most people would pay for it. In which case, we are more interested in the median and less so in the mean.

Mean absolute error loss

In a similar vein, an alternative to the MSE loss is the mean absolute error (MAE) loss, as shown in the following equation:

$$L(W, b) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M} | \hat{y}_{ij} - y_{ij} |$$

Mean squared log error loss
Another loss function used for regression is the mean squared log error (MSLE):

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\log \hat{y}_{ij} - \log y_{ij})^2$$

Mean absolute percentage error loss

- Finally, we have mean absolute percentage error (MAPE) loss:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\frac{100 \times |\hat{y}_{ij} - y_{ij}|}{y_{ij}}$$

Regression loss function discussion

- These are all valid choices, and there certainly is no single loss function that will outperform all other loss functions for every scenario. The MSE is very widely used and is a safe bet in most cases. So is the MAE.
- The MSLE and the MAPE are worth taking into consideration if our network is predicting outputs that vary largely in range.
- Suppose that a network is to predict two output variables: one in the range of [0, 10] and the other in the range of [0, 100]. In this case, the MAE and the MSE will penalize the error in the second output more significantly than the first.
- The MAPE makes it a relative error and therefore doesn’t discriminate based on the range. The MSLE squishes the range of all the outputs down, simply how 10 and 100 translate to 1 and 2 (in log base 10).

Loss Functions for Classification

- We can build neural networks to bin data points into different categories; for example, fraud|not fraud. However, when building neural networks for classification problems, the focus is often on attaching probabilities to these classifications (30 percent fraud|70 percent not fraud). These differing scenariosrequire different loss functions.

Hinge Loss

- Hinge loss is the most commonly used loss function when the network must be optimized for a hard classification. For example, 0 = no fraud and 1 = fraud, which by convention is called a 0-1 classifier. The 0,1 choice is somewhat arbitrary and –1, 1 is also seen in lieu of 0–1.
- Hinge loss is also seen in a class of models called maximum-margin classification models (e.g., support vector machines, a somewhat distant cousin to neural networks). Following is the equation for hinge loss when data points must be categorized as –1 or 1:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N} max(0,\ 1 - y_{ij} \times \hat{y}_{ij})$$

- The hinge loss is mostly used for binary classifications.

Logistic loss

- Logistic loss functions are used when probabilities are of greater interest than hard classifications. Great examples of these would be flagging potential fraud, with a human-in-the-loop solution or predicting the "probability of someone clicking on an ad," which can then be linked to a currency number. Predicting valid probabilities means generating numbers between 0 and 1.
- Predicting valid probabilities also means making sure the probability of mutually exclusive outcomes should sum to one. For this reason, it is essential that the very last layer of a neural network used in classification is a softmax.
- Note that the sigmoid activation function also will give valid values between 0 and 1. However, you cannot use it for scenarios in which the outputs are mutually exclusive, because it does not model the dependencies between the output values.
- Now that we have made sure our neural network will produce valid probabilities for the classes we have, we can dive headlong into the loss function and into the idea of what we should be optimizing here.
- We want to optimize for what is formally called the "maximum likelihood." In other words, we want to maximize the probability we predict for the correct class AND we want to do so for every single sample we have.

$$L(W, b) = -\sum_{i=1}^{N} y_i \times \log \hat{y}_i + (1 - y_i) \times \log (1 - \hat{y}_i)$$

Extending the loss function from two classes to *M* classes gives us the following equation for logistic loss:

$$L(W, b) = -\sum_{i=1}^{N} \sum_{j=1}^{M} y_{i,j} \times \log \hat{y}_{i,j}$$

Loss Functions for Reconstruction

- This set of loss functions relates to what is called *reconstruction*. The idea is simple. A neural network is trained to recreate its input as closely as possible. So, why is this any different from memorizing the entire dataset? The key here is to tweak the scenario so that the network is forced to learn commonalities and features across the dataset.
- In one approach, the number of parameters in the network is constrained such that the network is forced to compress the data and then re-create it. Another often-used approach is to corrupt the input with meaningless “noise” and train the network to ignore the noise and learn the data. Examples of these kinds of neural nets are restricted Boltzmann machines, autoencoders, and so on. These neural networks all use loss functions that are rooted in information theory.

Following is the equation for KL divergence:

$$D_{KL}(Y \,|\,|\, \hat{Y}) = -\sum_{i=1}^{N} Y_i \times \log\left(\frac{Y_i}{\hat{Y}_i}\right)$$

Kullback-Leibler Divergence(KL Divergence)

- KL-Divergence is something that allows us to measure *how far two distributions are apart*, this may seem a little bit strange at first, P||Q means event in Q over the probability of the event in P. Where the "||" operator indicates "*divergence*" or Ps divergence from Q. Importantly, the KL divergence score is not symmetrical, for example: KL(P || Q) != KL(Q || P)

Hyperparameters

- Hyperparameters are parameters whose values control the learning process and determine the values of model parameters that a learning algorithm ends up learning.
- you choose and set hyperparameter values that your learning algorithm will use before the training of the model even begins.
- Hyperparameters are used by the learning algorithm when it is learning but they are not part of the resulting model.
- At the end of the learning process, we have the trained model parameters which effectively is what we refer to as the model. The hyperparameters that were used during training are not part of this model.
- *Basically, anything in machine learning and deep learning that you decide their values or choose their configuration before training begins and whose values or configuration will remain the same when training ends is a hyperparameter.*

  Here are some common examples

  - Train-test split ratio
  - Learning rate in optimization algorithms (e.g. gradient descent)
  - Choice of optimization algorithm (e.g., gradient descent, stochastic gradient descent)

Hyperparameters

How to find a good set of hyper-parameters with a given dataset and architecture?

1. Learning rate (LR): Perform a learning rate range test to find the maximum learning rate.
2. Total batch size (TBS): A large batch size works well but the magnitude is typically constrained by the GPU memory.

Hyperparameters

- Choice of activation function in a neural network (nn) layer (e.g. Sigmoid, ReLU, Tanh)
- The choice of cost or loss function the model will use
- Number of hidden layers in a nn
- Number of activation units in each layer
- Number of iterations (epochs) in training a nn
- Kernel or filter size in convolutional layers
- Pooling size
- Batch size
- **Parameters:** Parameters on the other hand are internal to the model. That is, they are learned or estimated purely from the data during training as the algorithm used tries to learn the mapping between the input features and the labels or targets.

parameters

- Model training typically starts with parameters being initialized to some values (random values or set to zeros). As training/learning progresses the initial values are updated using an optimization algorithm (e.g. gradient descent). The learning algorithm is continuously updating the parameter values as learning progress but hyperparameter values set by the model designer remain unchanged.
- At the end of the learning process, model parameters are what constitute the model itself. **Examples of parameters**
  - The coefficients (or weights) of linear and logistic regression models.
  - Weights and biases of a nn
  - The cluster centroids in clustering

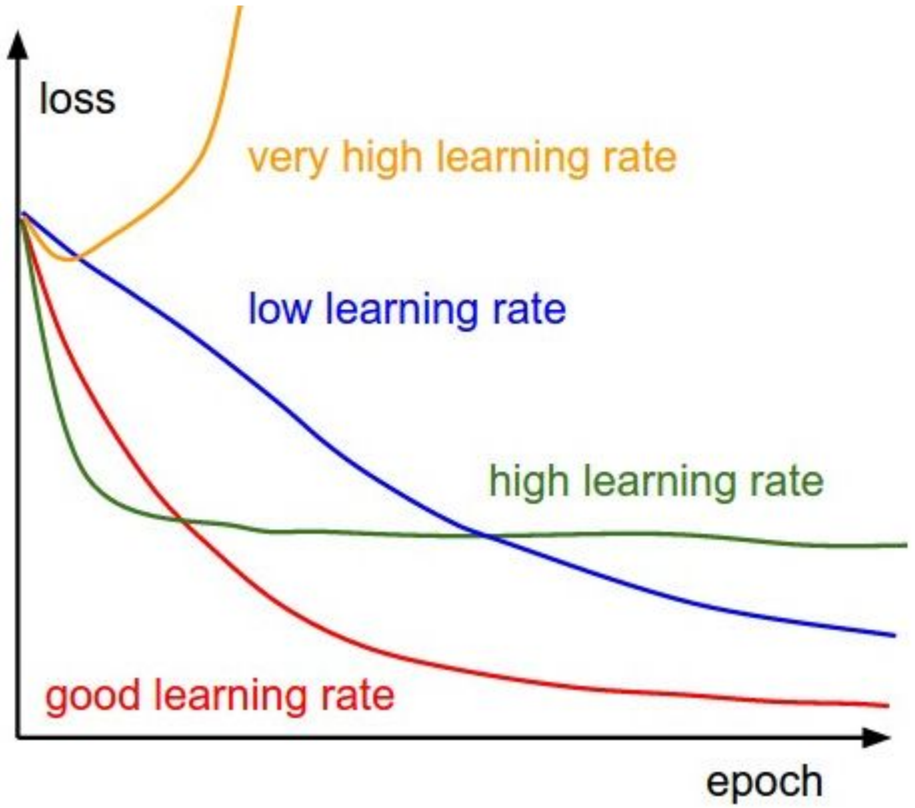
Therefore, setting the right hyperparameter values is very important because it directly impacts the performance of the model that will result from them being used during model training.

Learning Rate

- A neural network learns or approaches a function to best map inputs to outputs from examples in the training dataset.
- The learning rate hyperparameter controls the rate or speed at which the model learns. Precisely, it controls the amount of assigned error that the weights of the model are updated with each time they are updated.
- Given a perfectly configured learning rate, the model will learn to best estimate the function given number of layers and the number of nodes per layer in a given number of training epochs.
- Usually, **a large learning rate allows the model to learn faster,** at the cost of arriving on a sub-optimal final set of weights. **A smaller learning rate may allow the model to learn a more optimal or even globally optimal** set of weights but may take significantly longer time to train.
- At extremes, **a learning rate that is too large will result in weight updates that will be too large and the performance of the model will oscillate over training epochs.** Oscillating performance is said to be caused by weights that diverge. A learning rate that is too small may never converge or may get stuck on a suboptimal solution.

Learning Rate

- If the learning rate is high then it can overshoot the minimum and can fail to minimize the cost function.



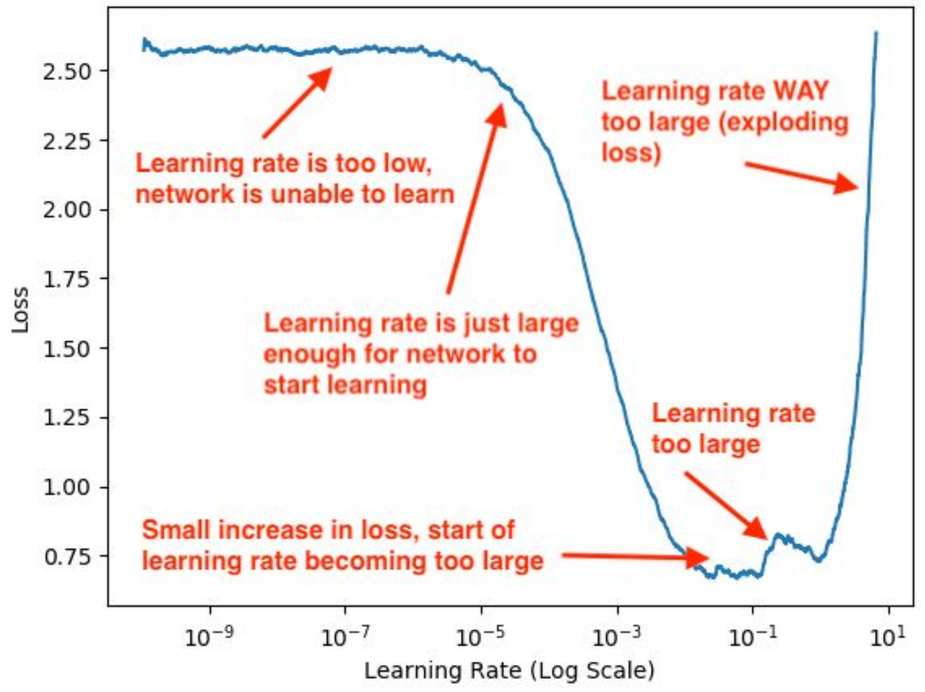


Figure 2: When inspecting a deep learning experiment, be sure to look at the loss landscape. This plot helps you identify when your learning rate is too low or too high.

Learning Rate

Since Gradient descent can only find local minimum so, the lower learning rate may result in bad performance. To do so, it is better to start with the random value of the hyperparameter can increase model training time but there are advanced methods such as adaptive gradient descent can manage the training time.

There are lots of optimizer for the same task but no optimizer is perfect. It depends on some factors

1. **size of training data:** as the size of the training data increases training time for model increases. If you want to go with less training model time you can choose a higher learning rate but may result in bad performance.
2. **Optimizer(gradient descent) will be slow down whenever the gradient is small** then it is better to go with a higher learning rate.

PS. It is always better to go with different rounds of gradient descent

- **Gradient Descent is a method to find the optimum parameter of the hypothesis or minimize the cost function.**

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

where alpha is learning rate

Momentum

Momentum in neural networks is a **variant of the *stochastic gradient descent***. It replaces the gradient with a *momentum* which is an aggregate of gradients.

The momentum factor is a coefficient that is applied to an extra term in the weights update:

$$\underbrace{\Delta w_{ij}}_{\text{weight increment}} = (\underbrace{\eta}_{\text{learning rate}} * \underbrace{\frac{\partial E}{\partial w_{ij}}}_{\text{weight gradient}})$$

$$\Delta w_{ij} = (\eta * \frac{\partial E}{\partial w_{ij}}) + (\underbrace{\gamma}_{\text{momentum factor}} * \underbrace{\Delta w_{ij}^{t-1}}_{\text{weight increment, previous iteration}})$$

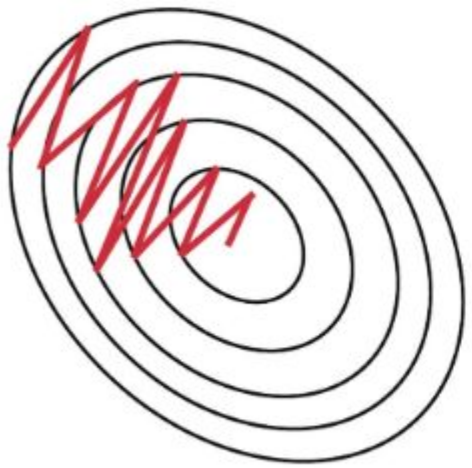
Advantages

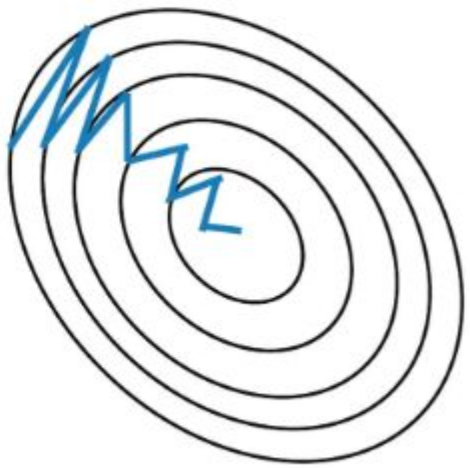
Beside others, momentum is known to speed up learning and to help not getting stuck in local minima.

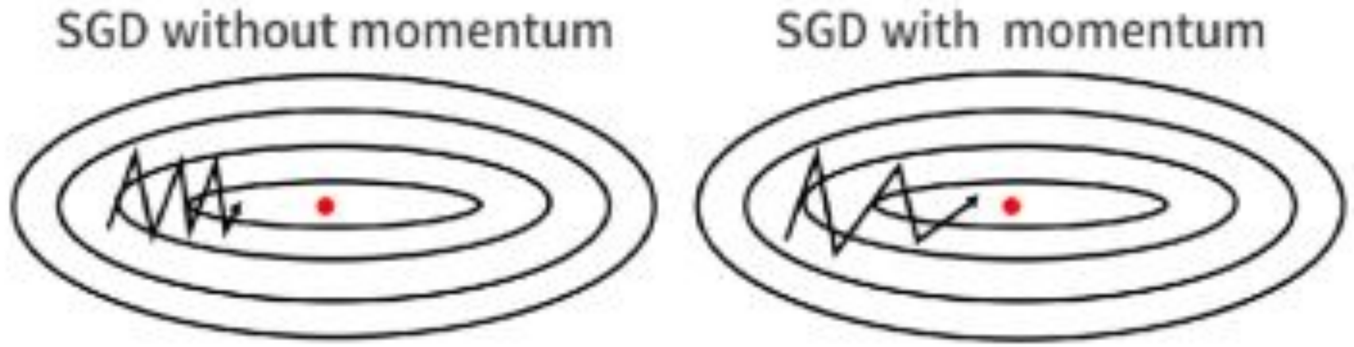
1. Momentum is faster than stochastic gradient descent the training will be faster than SGD.
2. Local minima can be an escape and reach global minima due to the momentum involved.

Momentum

- Momentum is a technique to prevent sensitive movement. When the gradient gets computed every iteration, it can have totally different direction and the steps make a zigzag path, which makes training very slow.

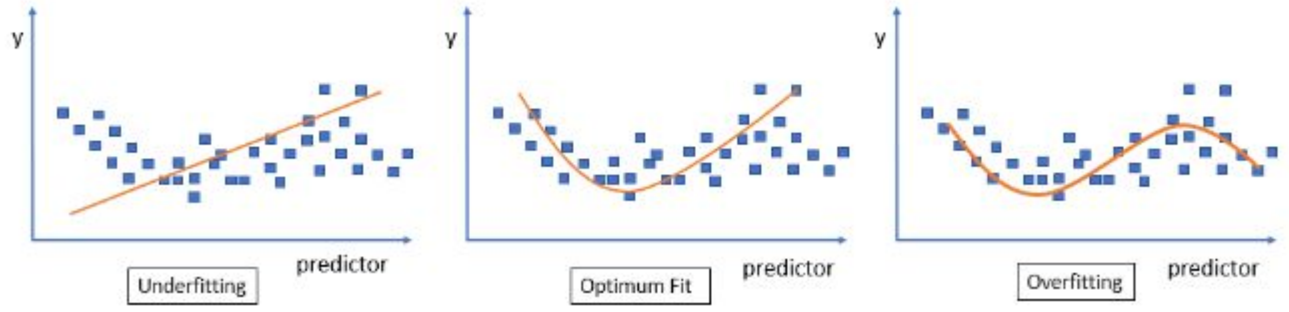

Stochastic Gradient Descent **withhout** Momentum

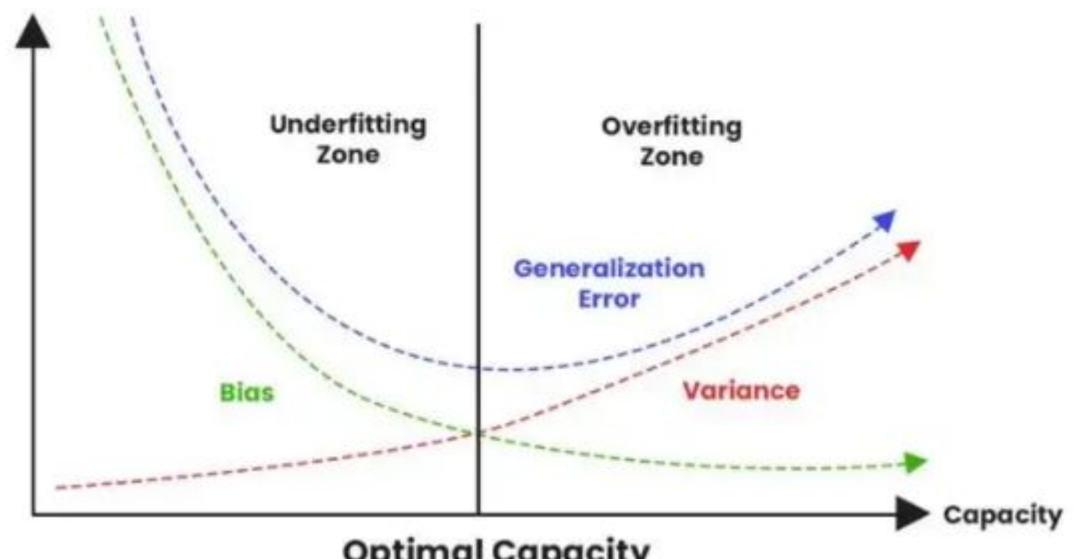

Stochastic Gradient Descent **with** Momentum



Regularization

- **Regularization**, in the context of neural networks, is a process of preventing a learning model from getting overfitted over training data. It involves a mechanism to reduce generalization errors of the learning model. optimum fit that presents the ability of a learning model to predict correct output for previously not seen data.





L1- Regularization

- Regularization is the process of preventing a learning model from getting overfitted over data.

  In L1 regularization, a penalty is introduced to suppress the learning model from getting overfitted. adding the absolute value of weight(Wj) parameters, while L2 regularization adds the squared value of weights(Wj) in the cost function.

$$Cost\ Function = Loss + L1\ Weight\ Penalty$$

$$= \underbrace{\sum_{i=1}^{M}\left(y_i - \sum_{j=1}^{N} x_{ij}w_j\right)^2}_{Squared\ Error} + \underbrace{\lambda \sum_{j=1}^{N} |w_j|}_{L1\ Regularization\ Term}$$

L1 Regularization

- ***L1 regularization tries to estimate the median of the data***

In this above equation, the L1 regularization term represents a magnitude of the coefficient value of the summation of the absolute value of weights or parameters of the neural network. ***Lambda is a hyperparameter.*** f lambda is zero, then it is equivalent to OLS. ***Ordinary Least Square or OLS, is a stats model which also helps us in identifying more significant features that can have a heavy influence on the output.*** But if lambda is very large, then it will add too much weight, and it will lead to under-fitting.

L1- Regularization

- **L1 Parameter -**Lasso Regression (**Least Absolute Shrinkage and Selection Operator**) adds **“Absolute value of magnitude”** of coefficient, as penalty term to the loss function.
- ***L2 regularization tries to estimate the mean of the data to avoid overfitting.***
- Lasso shrinks the less important feature’s coefficient to zero; thus, removing some feature altogether. **So, this works well for feature selection in case we have a huge number of features. The L1 regularizer basically looks for the parameter vectors that minimize the norm of the parameter vector (the length of the vector).** This is essentially the problem of how to optimize the parameters of a single neuron, a single layer neural network in general, and a single layer feed-forward neural network in particular.
- It just use the parameter vector and weights.
- **L1 regularization is robust in dealing with outliers. It creates sparsity in the solution** (most of the coefficients of the solution are zero), **which means the less important features or noise terms will be zero.** It makes L1 regularization robust to outliers.

L2- Regularization

Similar to L1 regularization, L2 regularization introduces a new cost function by adding a penalty term to a loss function called an L2 weight penalty.

$$\mathit{Cost\ Function} = \mathit{Loss} + \mathit{L2\ Weight\ Penalty}$$
$$= \underbrace{\sum_{i=1}^{M}\Big(y_i - \sum_{j=1}^{N} x_{ij}w_j\Big)^2}_{\mathit{Squared\ Error}} + \underbrace{\lambda \sum_{j=1}^{N} w_j^2}_{\mathit{L2\ Regularization\ Term}}$$

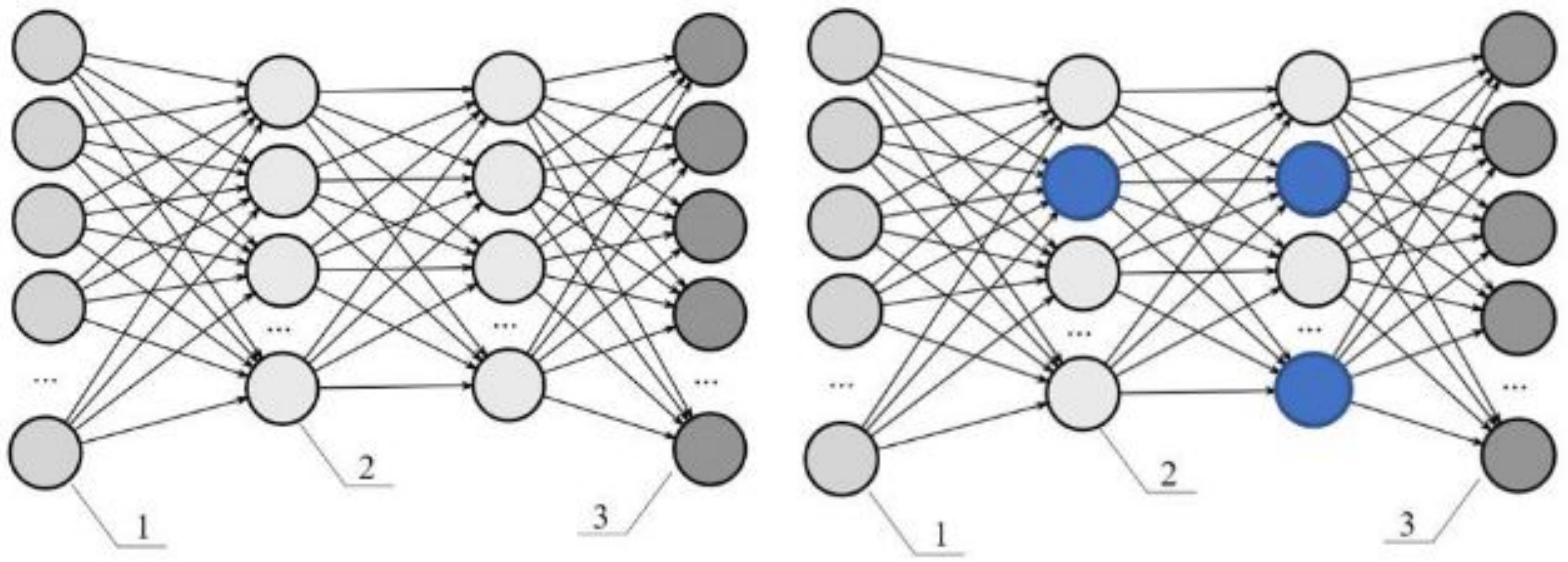
L2 regularization

As shown in the above equation, the L2 regularization term represents the weight penalty calculated by taking the squared magnitude of the coefficient, for a summation of squared weights of the neural network. **The larger the value of this coefficient, the higher is the penalty for complex features of a learning model. Here we take average of data distribution.**

Drop Out

A high number of nodes in each layer of a neural network capture the complex features of data along with noise which leads to overfitting of a learning model. In this regularization technique, the nodes are randomly selected and dropped to make a learning model more generalized so that it will perform better on newly arrived data as shown in the following diagram.

In the diagram above, the network on the left shows the fully connected dense neural network while the one on the right shows that few of the randomly selected nodes are dropped, which largely helps in preventing a learning model from getting overfitted on a training dataset and minimizes the regularization error.



Drop Out Regularization

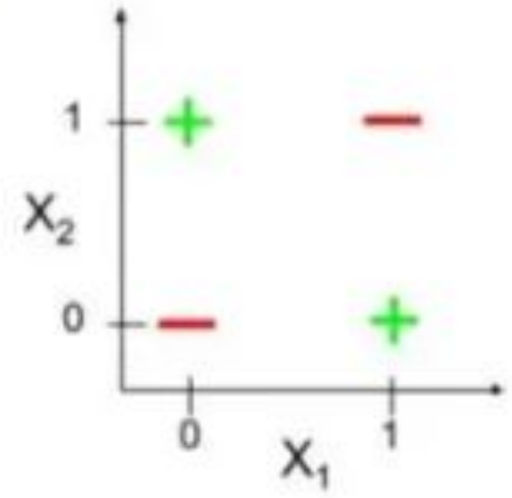
- The L2 regularization is the most common type of all regularization techniques and is also commonly known as weight decay or Ride Regression.

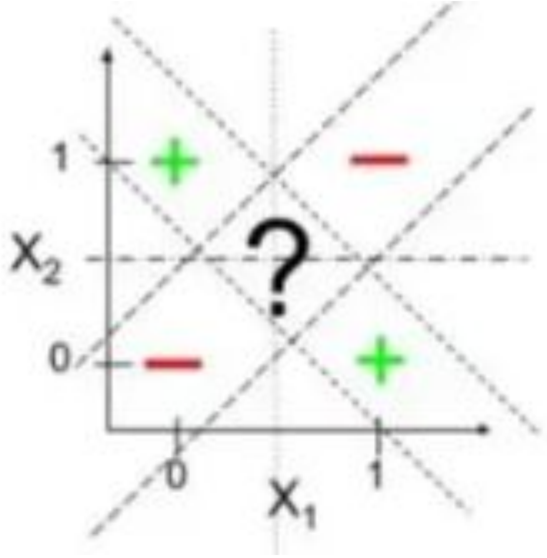
Sparsity

- The sparsity hyperparameter recognizes that for some inputs only a few features are relevant. For example, let’s assume that a network can classify a million images.
- Any one of those images will be indicated by a limited number of features. But to effectively classify millions of images a network must be able to recognize considerably more features, many of which don’t appear most of the time.
- An example of this would be how photos of sea urchins don’t contain noses and hooves. This contrasts to how in submarine images the nose and hoof features will be 0.
- The features that indicate sea urchins will be few and far between, in the vastness of the neural network’s layers. That’s a problem, because sparse features can limit the number of nodes that activate and impede a network’s ability to learn. In response to sparsity, biases force neurons to activate and the activations stay around a mean that keeps the network from becoming stuck.

Deep Feedforward Networks – Example of Ex OR,

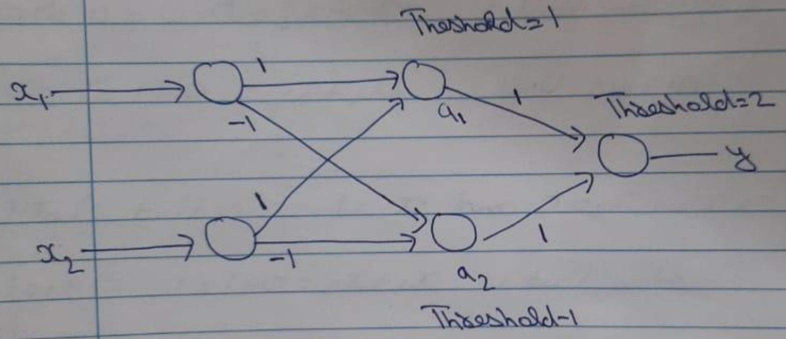
- Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models.
- These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. **There are no feedback connections in which outputs of the model are fed back into itself.** When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks





Deep Feedforward Networks- Example of EXOR

X	Y	O/P
0	0	0
0	1	1
1	0	1
1	1	0



Hidden Unit 1

$$a_1 = \begin{cases} 0 & x_1 + x_2 < 1 \\ 1 & x_1 + x_2 \geq 1 \end{cases}$$

Hidden Unit 2

$$a_2 = \begin{cases} 0 & -x_1 - x_2 < -1 \\ 1 & -x_1 - x_2 \geq -1 \end{cases}$$

Output Unit
$$y = \begin{cases} 0 & a_1+a_2 < 2 \\ 1 & a_1+a_2 \geq 2 \end{cases}$$

Case 1: $X_1 = 0$ and $X_2 = 0$, output should be 0

Hidden Unit 1 $x_1 + x_2 = 0 + 0 = 0 \Rightarrow a_1 = 0$

Hidden Unit 2 $-x_1 - x_2 = 0 - 0 = 0 \Rightarrow a_2 = 1$

Output Unit $a_1 + a_2 = 0 + 1 = 1 \Rightarrow y = 0$

Case 2: $x_1 = 0$ and $x_2 = 1$, output should by $y = 1$

Hidden Unit 1: $x_1 + x_2 = 0 + 1 = 1 \Rightarrow a_1 = 1$

Hidden Unit 2: $-x_1 - x_2 = 0 - 1 = -1 \Rightarrow a_2 = 1$

Output Unit $=$ $a_1 + a_2 = 1 + 1 = 2 \Rightarrow y = 1$

Case 3: ~~To~~ $x_1 = 1$ & $x_2 = 0$, output should be $y = 1$

Hidden Unit: $x_1 + x_2 = 1 + 0 = 1 \Rightarrow a_1 = 1$

Hidden Unit 2: $-x_1 - x_2 = -1 - 0 = -1 \Rightarrow a_2 = 1$

ZEAL
INSTITUTES
www.zealeducation.com

Output unit $a_1+a_2=1+1=2 \Rightarrow y=1$

Case 4: $x_1=1$ and $x_2=1$, output should be $y=0$

Hidden Unit 1: $x_1+x_2=1+1=2 \Rightarrow a_1=1$

Hidden Unit 2: $-x_1-x_2=-1-1=-2 \Rightarrow a_2=0$

Output unit : $a_1+a_2=1+0=1 \Rightarrow y=0$

Cost function

- It measures the difference, or error, between actual y and predicted y at its current position.
- This improves the machine learning model's efficacy by providing feedback to the model so that it can adjust the parameters to minimize the error and find the local or global minimum.
- It continuously iterates, moving along the direction of steepest descent (or the negative gradient) until the cost function is close to or at zero. At this point, the model will stop learning.
- Additionally, while the terms, cost function and loss function, are considered synonymous, there is a slight difference between them. It’s worth noting that a loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.

Error Backpropagation Learning Algorithm

- The error backpropagation learning algorithm is tool used during the training of neural networks. The main goal is to compute the gradient of the loss function (also known as the error function or cost function). These gradients are required for many optimization routines such as stochastic gradient descent and its many variants.

How does Error Backpropagation Work?

- Essentially, calculating the gradients relies entirely on the rules of differential calculus. As a neural network is a series of layers, for each data point the loss function is computed by passing a label data point through the network (feed forward). Next, the gradients are calculated starting from the final layer and then through use of the chain rule, the gradients can be passed backwards to calculate the gradients in the previous layers.
- The goal is to get the gradients for the loss function with respect to each model parameter (weights for each neural node connection as well as the bias weights). This point of this backwards method of error checking is to more efficiently calculate the gradient at each layer than the traditional approach of calculating each layer's gradient separately.

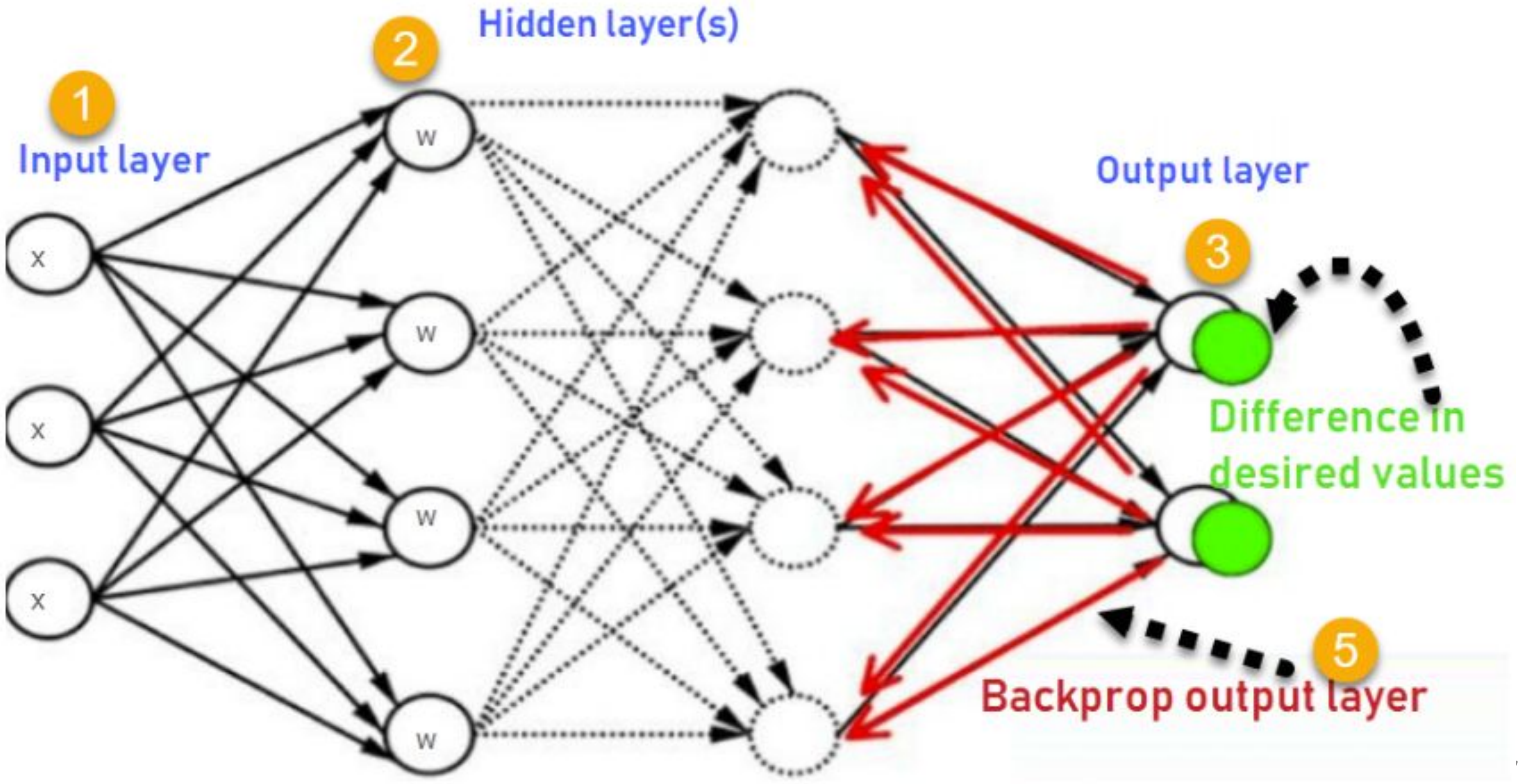
Error Backpropagation

- The principle behind the back propagation algorithm is to reduce the error values in randomly allocated weights and biases such that it produces the correct output. The system is trained in the supervised learning method, where the error between the system's output and a known expected output is presented to the system and used to modify its internal state.
- We need to update the weights such that we get the global loss minimum. This is how back propagation in neural networks works.
- **When the gradient is negative, an increase in weight decreases the error.**
- **When the gradient is positive, the decrease in weight decreases the error.**

Error Backpropagation Learning Algorithm

- **What are the Uses of Error Backpropagation?**

  Backpropagation is especially useful for deep neural networks working on error-prone projects, such as image or speech recognition. Taking advantage of the chain and power rules allows backpropagation to function with any number of outputs and better train all sorts of neural networks.



How Backpropagation Algorithm Works

Error Backpropagation

Steps-

- **Inputs X, arrive through the preconnected path**
- **Input is modeled using real weights W. The weights are usually randomly selected.**
- **Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.**
- **Calculate the error in the outputs**
- **Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.**

Why do the Gradients Vanish or Explode?

- **Vanishing** is when as backpropagation occurs, the gradients normally get smaller and smaller, gradually approaching zero. This leaves the weights of the initial or lower layers unchanged, causing the Gradient Descent to never converge to the optimum.
- For example, Activation Functions such as the sigmoid function have a very prominent difference between the variance of their inputs and outputs. They shrink and transform a large input space into a smaller output space, which lies between [0,1].
- Looking at the graph below of the Sigmoid Function, we can see that using larger inputs, regardless if they are negative or positive will classify at either 0 or 1. However, when the Backpropagation processes, it has no gradient to propagate backward in the Neural Network. The little gradient that does exist, will continuously keep diluting as the algorithm continues to process through the top layers, leaving nothing for the lower layers.

Gradient Descent

- Gradient descent is an optimization algorithm which is commonly-used to train machine learning models and neural networks.
- Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates.
- Until the function is close to or equal to zero, the model will continue to adjust its parameters to yield the smallest possible error.
- Once machine learning models are optimized for accuracy, they can be powerful tools for artificial intelligence (AI) and computer science applications.

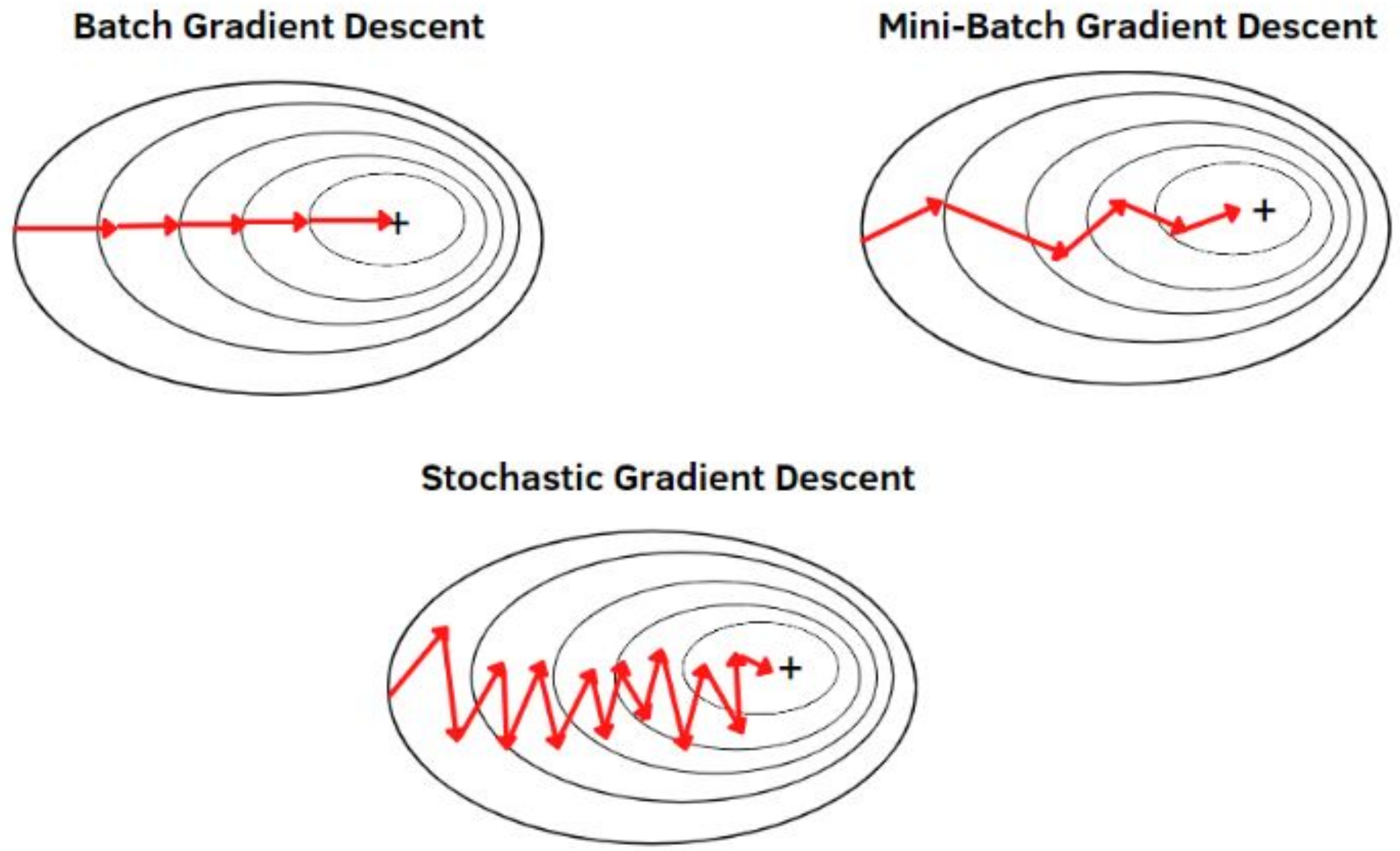
Gradient descent

- Gradient descent adjusts parameters to minimize particular functions to local minima. In linear regression, it finds weight and biases, and deep learning backward propagation uses the method.
- The algorithm objective is to identify model parameters like weight and bias that reduce model error on training data.
- A gradient measures how much the output of a function changes if you change the inputs a little bit.
- In machine learning, a gradient is a derivative of a function that has more than one input variable. Known as the slope of a function in mathematical terms, the gradient simply measures the change in all weights about the change in error.

$$dy/dx$$

- ***dy = change in y***
- ***dx = change in x***

Types Gradient descent



1.Batch Gradient Descent:

- Batch gradient descent (BGD) is used to find the error for each point in the training set and update the model after evaluating all training examples. This procedure is known as the training epoch. In simple words, it is a greedy approach where we have to sum over all examples for each update.
- **Advantages of Batch gradient descent:**
- It produces less noise in comparison to other gradient descent.
- It produces stable gradient descent convergence.
- It is Computationally efficient as all resources are used for all training samples.

2. Stochastic gradient descent

- Stochastic gradient descent (SGD) is a type of gradient descent that runs one training example per iteration. Or in other words, it processes a training epoch for each example within a dataset and updates each training example's parameters one at a time.
- As it requires only one training example at a time, hence it is easier to store in allocated memory. However, it shows some computational efficiency losses in comparison to batch gradient systems as it shows frequent updates that require more detail and speed.
- Further, due to frequent updates, it is also treated as a noisy gradient. However, sometimes it can be helpful in finding the global minimum and also escaping the local minimum.
- **Advantages of Stochastic gradient descent:**
- In Stochastic gradient descent (SGD), learning happens on every example, and it consists of a few advantages over other gradient descent.
- It is easier to allocate in desired memory.
- It is relatively fast to compute than batch gradient descent.
- It is more efficient for large datasets.

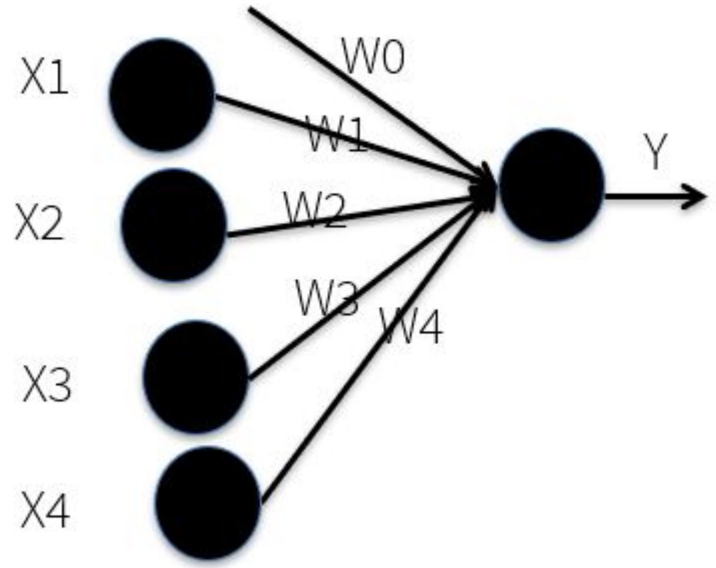
3. MiniBatch Gradient Descent:

- Mini Batch gradient descent is the combination of both batch gradient descent and stochastic gradient descent. It divides the training datasets into small batch sizes then performs the updates on those batches separately. Splitting training datasets into smaller batches make a balance to maintain the computational efficiency of batch gradient descent and speed of stochastic gradient descent. Hence, we can achieve a special type of gradient descent with higher computational efficiency and less noisy gradient descent.

Advantages of Mini Batch gradient descent:

- It is easier to fit in allocated memory.
- It is computationally efficient.
- It produces stable gradient descent convergence.

Implementation of Gradient Descent



- **Here is the simple linear formula:**
- Y = WX+ W0
- Here W0 is the bias term. If there are multiple
- Y = W1X1 + W2X2 + … WnXn + W0
- Using this formula let's check step by step, how to perform a gradient descent algorithm:
- 1. First, randomly initialize the Ws.
- 2. Using this random Ws calculate the predicted output Y_hat.
- 3. Take the Root mean square error of the Y and Y_hat.
- 4. Calculate the updated Ws using the formula: W= W— stepSize*derivative of the RMS, where stepSize needs to be chosen.
- 5. Keep repeating steps 2 to 4 till the convergence.

Gradient descent

- Suppose we have a data set consists of three features and we want to know what features are more important.
- So firstly we will give each feature a **WEIGHT** that weight shows how the feature is important if the weight is large then it’s important if it’s small then it’s not that important and will not affect the output.
- These Weights are firstly initialized randomly and will be updated after each iteration on the data set to get the optimum weights.
- So you may be asking now **WHERE IS THE GRADIENT DESCENT ??** Gradient descent is a way to update those weights and get their optimum value and minimizes the error.
- To get a better understanding of what is gradient descent follow these steps with me of how we update the weights of our features using gradient descent.

Gradient descent

Components of Gradient Descent :

- Learning Rate.
- Activation Function.
- Random Initialized Weights.

Step 1 :

We have 3 features so we will assume 3 Weights, one for each feature with any number. W1 = 0.3443, W2 = 0.3213, W3 = 0.5642

Step 2 :

Pick a suitable learning rate!!

Step 3 :

Calculate ‘Y’ the output of the system We calculate ‘Y’ by dot product the Features and Weights.

$$Y = X1*W1 + X2*W2 + X3*W3 + X4*W4$$

Gradient descent

Step 4 :

Choose an activation function, Here we'll be using a **Sigmoid Function** So we need to update our output 'y' to a new value. Since we're using sigmoid function then **Y = 1/1+e^-x.**

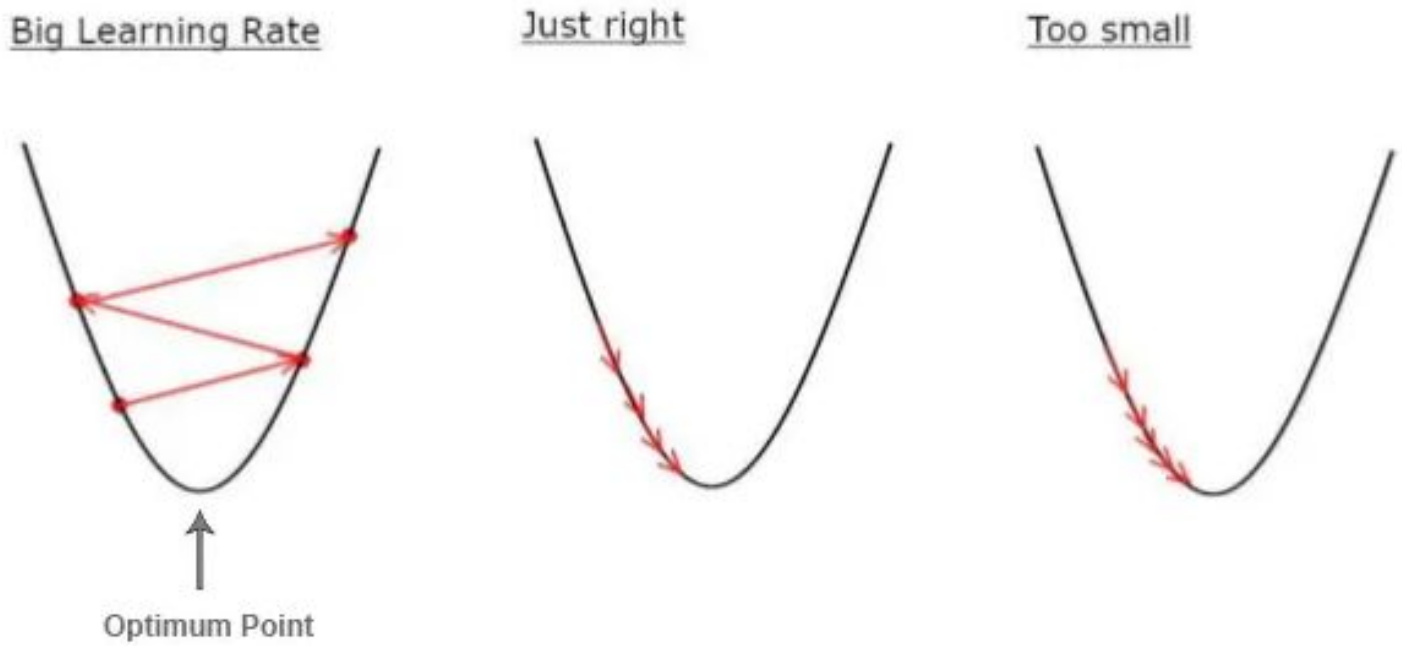
Step 5 :

Update the weights using gradient descent.

Gradient descent

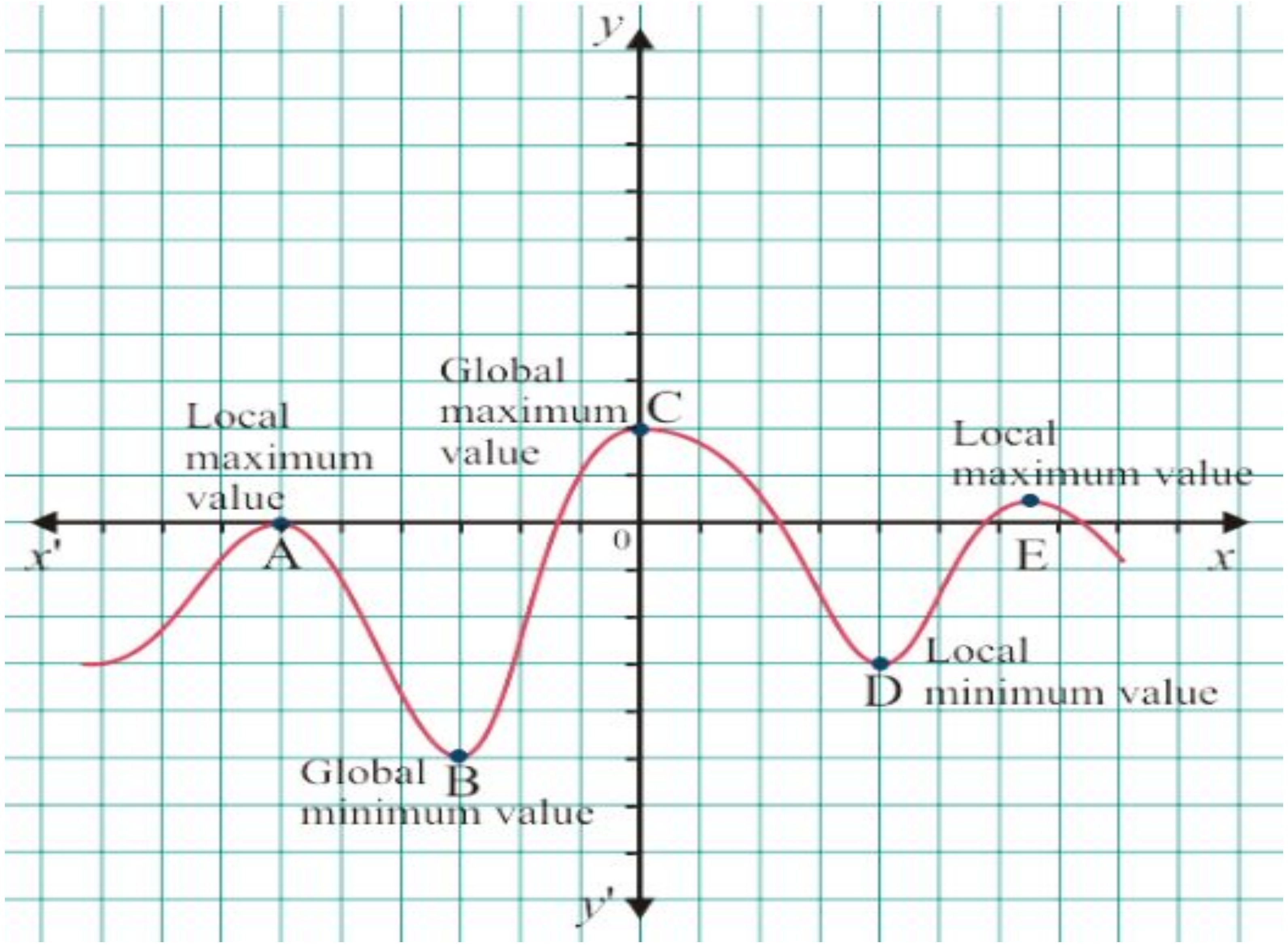
- **Step 6 :**
- Compute the error using **sum squared error over the whole test data** we will subtract the predicted value (From Step 4) and the actual value (Needed Output).

$$\mathrm{SSE} = \sum_{\underbrace{i=1}_{\text{test set}}}^{n} (\underbrace{y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i}_{\text{actual value}})^2$$

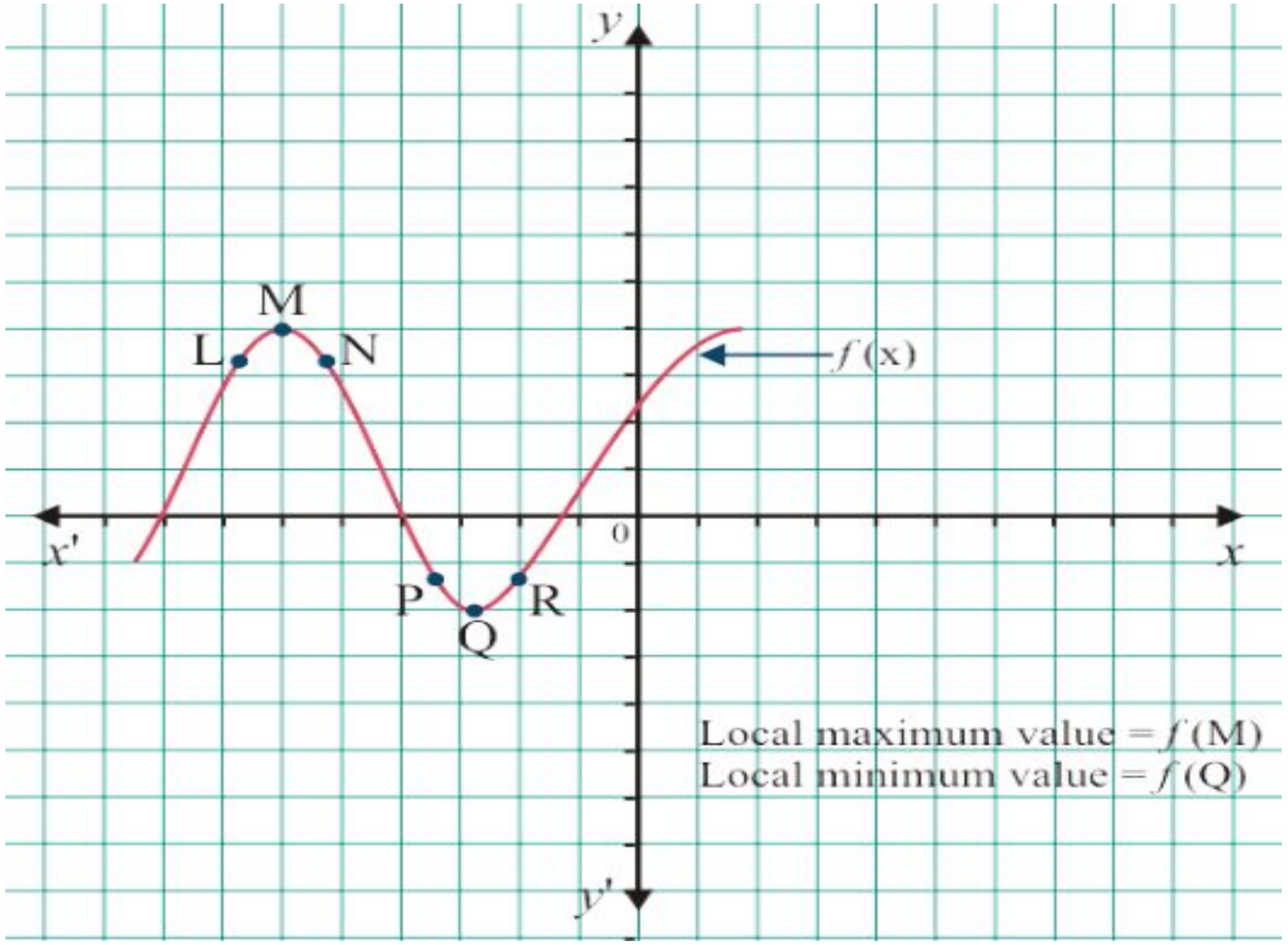


Local Minimum & Local Maximum

- On a graph, **many local maximum/minimum values may be possible,** but there will be **only one global maximum / minimum value.**



Local Minimum & Local Maximum



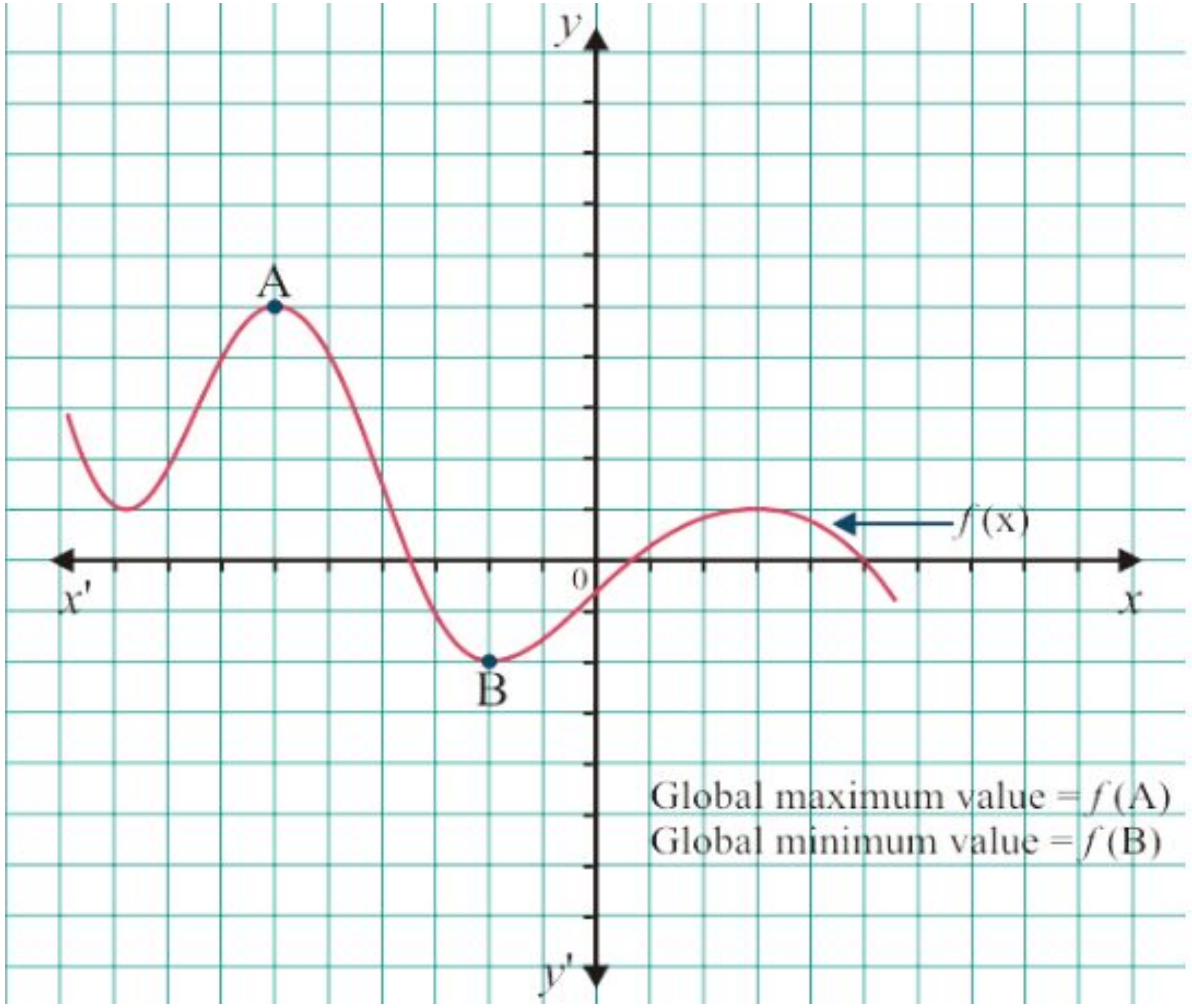
- **Local Maximum Value:**
- Local maximum value of a function $f\ (x)$ on a graph, is a value at a point (like M in the graph) which is greater than the values at the nearest adjacent points on left and right sides (like L and N in the graph)

  Thus, f (M) is the local maximum value of the function $f\ (x)$.

- **Local Minimum Value:**
- Local minimum value of a function $f\ (x)$ on a graph, is a value at a point (like Q in the graph). Which is lower than the values at the nearest adjacent points on left and right sides (like P and R in the graph).
- Thus, f (Q) is the local minimum value of the function $f\ (x)$.

Global Minimum & Global Maximum

- Global maximum and minimum values are also known as the absolute maximum and minimum values.



Global Minimum & Global Maximum

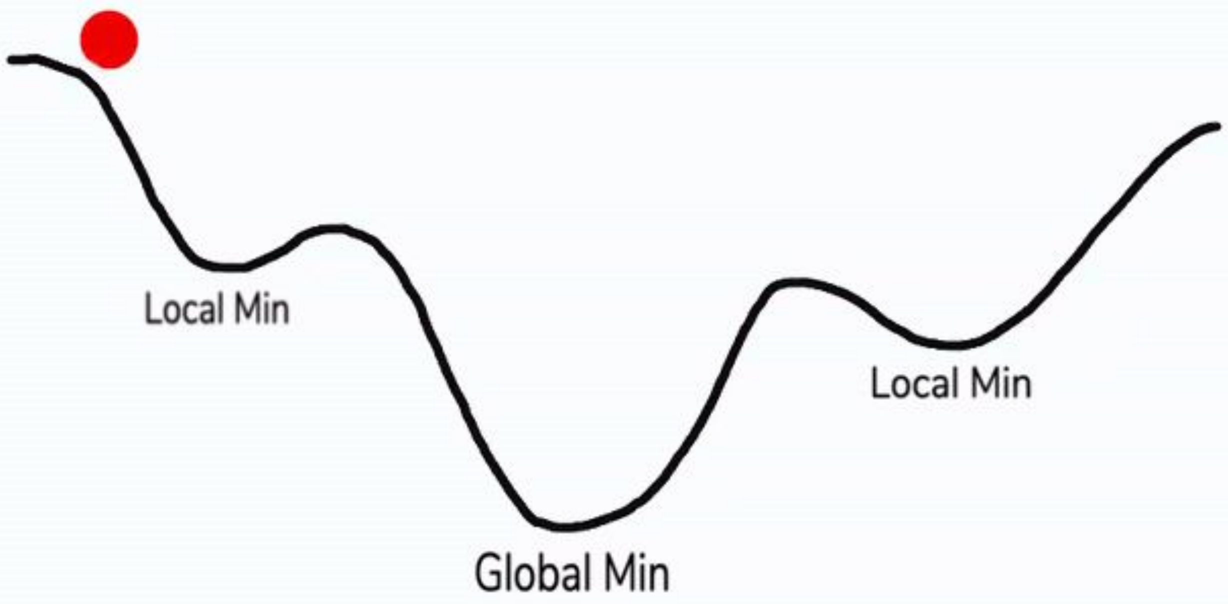
Global Maximum Value:

- Global maximum value of a function $f(x)$ on a graph, is a value at a point (say A) which is higher than the entire values of $f(x)$ at any other point.
- Thus, f (A) is the global/absolute maximum value of the function $f(x)$.

Global Minimum Value:

- Global minimum value of a function f (x) on a graph, is a value at a point (say B) which is lower than the entire values of $f(x)$ at any other point.
- Thus, f (B) is the global/absolute minimum value of the function $f(x)$.

- **The value of the derivative is the inclination of the slope at a specific point.**
- **If the derivative is positive, it means the slope goes up (when going to the right!)**
- **If the derivative is negative, it means the slope goes down.**
- **If the derivative is equal to 0**
- **0, it means it doesn't go up or down.**
- Once you have the value of the slope,



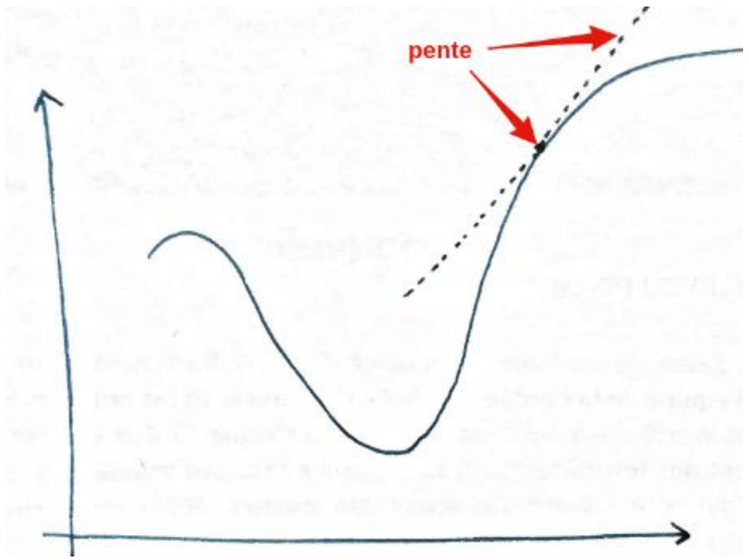
Concept

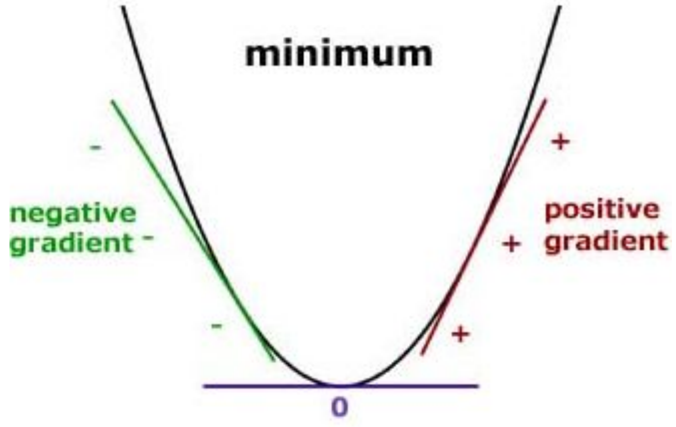
- Imagine you are a skier on a mountain.
- And you want to find the lowest point around you, i.e. finding the place with minimal altitude.
- Mathematically, the slope is the derivative:

what do you do?

You go in the opposite direction!

- **Positive derivative => Go to the left.**
- **Negative derivative => Go to the right.**



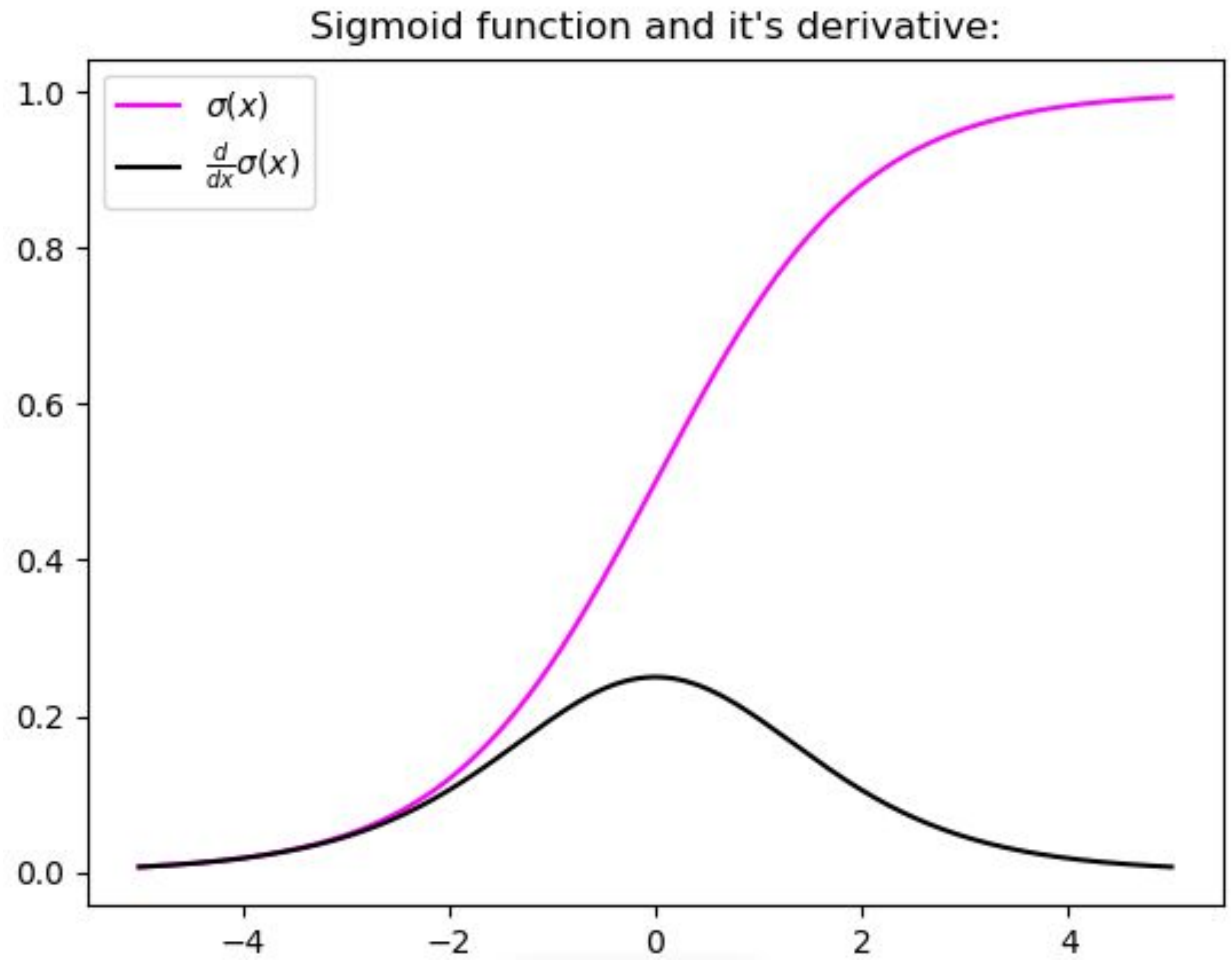


Why do the Gradients Vanish or Explode?

- As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. This is known as the **vanishing gradients** problem.

- **Vanishing** is when as backpropagation occurs, the gradients normally get smaller and smaller, gradually approaching zero. This leaves the weights of the initial or lower layers unchanged, causing the Gradient Descent to never converge to the optimum.
- Observing the graph in the next slide of the Sigmoid function, we can see that for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero. Thus, when the backpropagation algorithm chips in, it virtually has no gradients to propagate backward in the network, and whatever little residual gradients exist keeps on diluting as the algorithm progresses down through the top layers. So, this leaves nothing for the lower layers.

Why do the Gradients Vanish or Explode?



Explode

- On the contrary, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the ***exploding gradients*** problem.

- Exploding is the opposite of Vanishing and is when the gradient continues to get larger which causes a large weight update and results in the Gradient Descent to diverge.
- **Exploding gradients occur due to the weights in the Neural Network, not the activation function.**
- The gradient linked to each weight in the Neural Network is equal to a product of numbers. If this contains a product of values that is greater than one, there is a possibility that the gradients become too large.
- The weights in the lower layers of the Neural Network are more likely to be affected by Exploding Gradient as their associated gradients are products of more values. This leads to the gradients of the lower layers being more unstable, causing the algorithm to diverge.

Why do the Gradients Vanish or Explode?

Weight Initialization

Weight Initialization is the process of setting the weights of a Neural Network to small random values that help define the starting point for the optimization of the model.

- **Initializing weights to zero**
- **Initializing weights randomly**
- **Initializing weights using Heuristic**

Sentiment Analysis

- Sentiment analysis is a type of market analysis that includes the use of text analysis, biometrics, natural language processing (NLP), and computational linguistics to recognize the state of the said information.
- In simple terms, it's the process of determining whether a piece of content – email, social media post, or article – is negative, positive, or neutral.
- Sentiment analysis enables you to ascertain public opinion and understand consumer experiences.
- But why should you even bother about sentiment analysis? For starters, it's extremely helpful in social media monitoring.
- It helps you gauge public opinion on certain topics on an enormous scale.

- Besides, it can play a pivotal role in market research and customer service.
- With sentiment analysis, you can see what people think about your products or your competitors' products.
- This helps you understand customer attitudes and preferences, enabling you to make informed decisions.

Types of Sentiment Analysis

People have a wide range of emotions – sad or happy, interested or uninterested, and positive or negative. Different sentiment analysis models are available to capture this variety of emotions. Let’s look at the most important types of sentiment analysis.

Fine-Grained

- This sentiment analysis model helps you derive polarity precision. You can conduct a sentiment analysis across the following polarity categories: very positive, positive, neutral, negative, or very negative. Fine-grained sentiment analysis is helpful for the study of reviews and ratings.
- For a rating scale from 1 to 5, you can consider 1 as very negative and five as very positive. For a scale from 1 to 10, you can consider 1-2 as very negative and 9-10 as very positive.

2. Aspect-Based

- While fine-grained analysis helps you determine the overall polarity of your customer reviews, aspect-based analysis delves deeper.
- It helps you determine the particular aspects people are talking about. Let’s say; you’re a mobile phone manufacturer, and you get a customer review stating, “the camera struggles in artificial lighting conditions.”
- With aspect-based analysis, you can determine that the reviewer has commented on something “negative” about the “camera.”

3. Emotion Detection

As the name suggests, emotion detection helps you detect emotions. This can include anger, sadness, happiness, frustration, fear, worry, panic, etc. Emotion detection systems typically use lexicons – a collection of words that convey certain emotions.

Some advanced classifiers also utilize robust machine learning (ML) algorithms. It’s recommended to use ML over lexicons because people express emotions in a myriad of ways. Take this line, for example: “This product is about to kill me.”

This line may express feelings of fear and panic. A similar line – this product is killing it for me – has an entirely different and positive meaning. But the word “kill” might be associated with fear or panic in the lexicon. This may lead to inaccurate emotion detection.

4. Intent Analysis

- Accurately determining consumer intent can save companies time, money, and effort. So many times, businesses end up chasing consumers that don’t plan to buy anytime soon.
- Accurate intent analysis can resolve this hurdle. The intent analysis helps you identify the intent of the consumer – whether the customer intends to purchase or is just browsing around.
- If the customer is willing to purchase, you can track them and target them with advertisements. If a consumer isn’t ready to buy, you can save your time and resources by not advertising to them.

How to Perform Sentiment Analysis?

Modern-day sentiment analysis approaches are classified into three categories: knowledge-based, statistical, and hybrid. Here’s how to perform sentiment analysis.

- **Knowledge-Based**: This approach included the classification of text based on words that emanate emotion.
- **Statistical**: This approach utilizes machine learning algorithms like latent semantic analysis and deep learning for accurate sentiment detection.
- **Hybrid**: This approach leverages both knowledge-based and statistical techniques for on-point sentiment analysis.

Deep Learning with Pytorch , Jupyter and colab

- PyTorch is an optimized Deep Learning tensor library based on Python and Torch and is mainly used for applications using GPUs and CPUs. PyTorch is favored over other Deep Learning frameworks like TensorFlow and Keras since it uses dynamic computation graphs and is completely Pythonic. It allows scientists, developers, and neural network debuggers to run and test portions of the code in real-time. Thus, users don’t have to wait for the entire code to be implemented to check if a part of the code works or not.

The two main features of PyTorch are:

- Tensor Computation (similar to NumPy) with strong GPU (Graphical Processing Unit) acceleration support
- Automatic Differentiation for creating and training deep neural networks

Basics of PyTorch

The basic PyTorch operations are pretty similar to Numpy.

- **Introduction to Tensors**

In machine learning, when we represent data, we need to do that numerically. A tensor is simply a container that can hold data in multiple dimensions. In mathematical terms, however, a tensor is a fundamental unit of data that can be used as the foundation for advanced mathematical operations.

It can be a number, vector, matrix, or multi-dimensional array like Numpy arrays. Tensors can also be handled by the CPU or GPU to make operations faster. There are various types of tensors like Float Tensor, Double Tensor, Half Tensor, Int Tensor, and Long Tensor, but PyTorch uses the 32-bit Float Tensor as the default type.

Mathematical Operations

- The codes to perform mathematical operations are the same in PyTorch as in Numpy. Users need to initialize two tensors and then perform operations like addition, subtraction, multiplication, and division on them.

Matrix Initialization and Matrix Operations

- To initialize a matrix with random numbers in PyTorch, use the function randn() that gives a tensor filled with random numbers from a standard normal distribution. Setting the random seed at the beginning will generate the same numbers every time you run this code. Basic matrix operations and transpose operation in PyTorch are also similar to NumPy.

Common PyTorch Modules

- In PyTorch, modules are used to represent neural networks.

Autograd

- The autograd module is PyTorch's automatic differentiation engine that helps to compute the gradients in the forward pass in quick time. Autograd generates a directed acyclic graph where the leaves are the input tensors while the roots are the output tensors.

Optim

- The Optim module is a package with pre-written algorithms for optimizers that can be used to build neural networks.

nn

- The nn module includes various classes that help to build neural network models. All modules in PyTorch subclass the nn module.

Solving an Image Classification Problem Using PyTorch

- Step 1 – Initialize the input and output using tensor.
- Step 2 – Define the sigmoid function that will act as an activation function. Use a derivative of the sigmoid function for the backpropagation step.
- Step 3 – Initialize the parameters such as the number of epochs, weights, biases, learning rate, etc., using the randn() function. This completes the creation of a simple neural network consisting of a single hidden layer and an input layer, and an output layer.
- The forward propagation step is used to calculate output, while the backward propagation step is used for error calculation. The error is used to update the weights and biases.
- Next, we have our final neural network model based on a real-world case study, where the PyTorch framework helps create a deep learning model.

The task at hand is an image classification problem, where we find out the type of apparel by looking at different apparel images.

Step 1 – Classify the image of apparel into different classes.

- There are two folders in the dataset – one for the training set and the other for the test set. Each folder contains a .csv file that has the image id of any image and the corresponding label name. Another folder contains the images of the specific set.

Step 2 – Load the Data

- Import the required libraries and then read the .csv file. Plot a randomly selected image to better understand how the data looks. Load all training images with the help of the train.csv file.

Step 3 – Train the Model

- Build a validation set to check the performance of the model on unseen data. Define the model using the import torch package and the needed modules. Define parameters like the number of neurons, epochs, and learning rate. Build the model, and then train it for a particular number of epochs. Save training and validation loss in case of each epoch—plot, the training, and validation loss, to check if they are in sync.

Jupyter

- Project Jupyter is a suite of software products used in interactive computing. IPython was originally developed by Fernando Perez in 2001 as an enhanced Python interpreter. A web based interface to IPython terminal in the form of IPython notebook was introduced in 2011. In 2014, Project Jupyter started as a spin-off project from IPython.
- Packages under Jupyter project include −
- **Jupyter notebook** − A web based interface to programming environments of Python, Julia, R and many others
- **QtConsole** − Qt based terminal for Jupyter kernels similar to IPython
- **nbviewer** − Facility to share Jupyter notebooks
- **JupyterLab** − Modern web based integrated interface for all products.
- Standard distribution of Python comes with a **REPL (Read-Evaluate-Print Loop)** environment in the form of Python shell with **>>>** prompt. IPython (stands for Interactive Python) is an enhanced interactive environment for Python with many functionalities compared to the standard Python shell.

Jupyter Notebook

- Jupyter notebooks basically provides an interactive computational environment for developing Python based Data Science and deep learning applications. They are formerly known as ipython notebooks. The following are some of the features of Jupyter notebooks that makes it one of the best components of Python ML ecosystem −
- Jupyter notebooks can illustrate the analysis process step by step by arranging the stuff like code, images, text, output etc. in a step by step manner.
- It helps a data scientist to document the thought process while developing the analysis process.
- One can also capture the result as the part of the notebook.
- With the help of jupyter notebooks, we can share our work with a peer also.

- The Jupyter Notebook is an open source web application that you can use to create and share documents that contain live code, equations, visualizations, and text. Jupyter Notebook is maintained by the people at Project Jupyter.

COLAB

- Colaboratory by Google (Google Colab in short) is a Jupyter notebook based runtime environment which allows you to run code entirely on the cloud.
- This is necessary because it means that you can train large scale ML and DL models even if you don’t have access to a powerful machine or a high speed internet access.
- Google Colab supports both GPU and TPU instances, which makes it a perfect tool for deep learning and data analytics enthusiasts because of computational limitations on local machines.
- Since a Colab notebook can be accessed remotely from any machine through a browser, it’s well suited for commercial purposes as well.

- colab notebooks allow you to combine **executable code** and **rich text** in a single document, along with images,
- HTML, LaTeX and more.
- When you create your own Colab notebooks, they are stored in your Google Drive account.
- You can easily share your Colab notebooks with co-workers or friends, allowing them to comment on your notebooks or even edit them.
- We can use Google Colabs like Jupyter notebooks.
- They are really convenient because Google Colab hosts them, so we don’t use any of our own computer resources to run the notebook.
- We can also share these notebooks so other people can easily run our code, all with a standard environment since it’s not dependent on our own local machines.
- However, we might need to install some libraries in our environment during initialization.

- Google Colab is a great platform for deep learning enthusiasts, and it can also be used to test basic machine learning models, gain experience, and develop an intuition about deep learning aspects such as hyperparameter tuning, preprocessing data, model complexity, overfitting and more.

- Deep learning is a computationally expensive process, a lot of calculations need to be executed at the same time to train a model. To mitigate this issue, Google Colab offers us not only the classic CPU runtime but also an option for a GPU and TPU runtime as well.
- The CPU runtime is best for training large models because of the high memory it provides.
- The GPU runtime shows better flexibility and programmability for irregular computations.
- The TPU runtime is highly-optimized for large batches and CNNs and has the highest training throughput.
- If you have a smaller model to train, I suggest training the model on GPU/TPU runtime to use Colab to its full potential.
- To create a GPU/TPU enabled runtime, you can click on runtime in the toolbar menu below the file name. From there, click on “**Change runtime type**”, and then select GPU or TPU under the Hardware Accelerator dropdown menu.

Why is Google Colab the Ultimate Tool

- Because your system’s configurations don’t matter AT ALL! With Colab, you can train your complex machine learning and deep learning models without worrying about the computational resources they require. GPUs and even much more expensive TPUs are provided by the platform free of cost!

A plethora of Pre-installed Libraries

- The Anaconda Jupyter Notebook comes with several pre-installed Python libraries such as **Pandas**, **NumPy**, **Matplotlib**, etc. Google Colab offers these libraries and on top of these, provides even more pre-installed ML libraries such as **Keras**, **PyTorch**, and **TensorFlow**.

Collaboration Capabilities

- Similar to working on a Google Sheet, multiple developers can co-code on the same workbook simultaneously. This is particularly helpful if it is a group project, right? You can also share your work with your fellow developers.

Cloud Storage Solution

- When you use Jupyter Notebook as your working environment, all your work is stored on the local machine. Opting for Colab, on the other hand, lets you access your workbooks seamlessly from any other device, as they are all saved in your Google Drive.

GPU and TPU Access

- Training complex models might take hours on Jupyter Notebook because CPUs are slow to execute. Colab lets you use its dedicated GPUs and TPUs for your personal ML project implementation which takes hardly a few minutes.
- Colab allows you to run your code for 24 hours without interruptions, and 12 hours if you connect to an interactive GPU-based VM runtime. For most folks, this is sufficient enough time to meet their computation needs. You can run multiple CPU, GPU, and TPU instances simultaneously, but the resources would be shared between these instances.