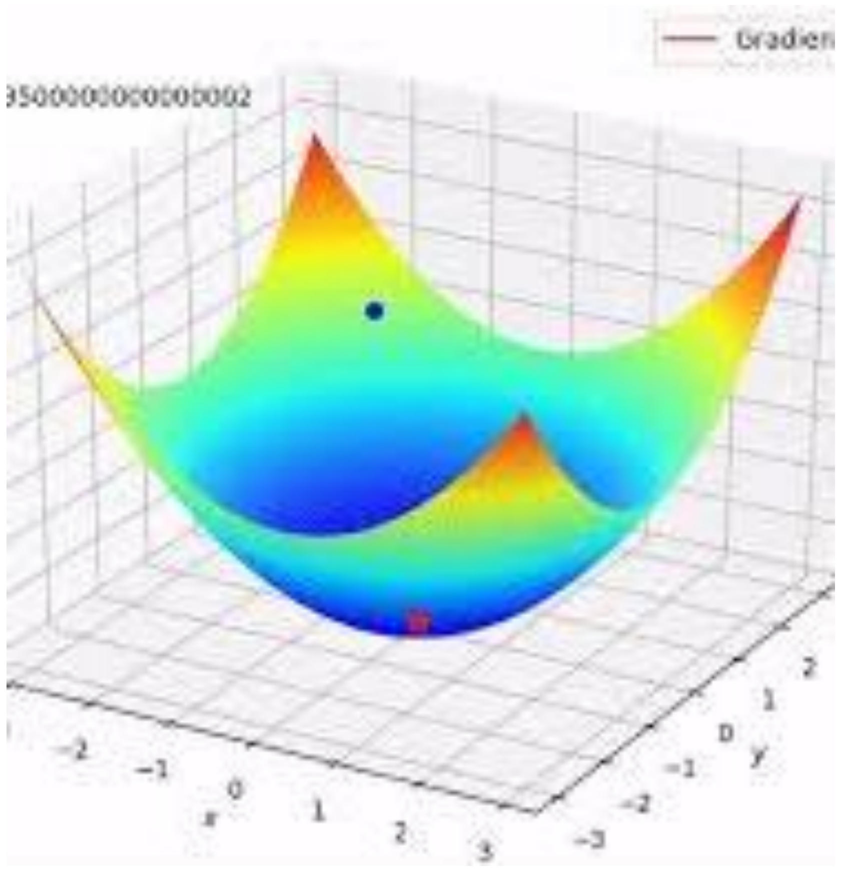
Module 2

- INTRODUCTION TO DEEP LEARNING : Feed Forward Neural Networks , Gradient Descent, Back Propagation Algorithm: Vanishing Gradient problem – Mitigation – RelU Heuristics for Avoiding Bad Local Minima – Heuristics for Faster Training – Nestors Accelerated Gradient Descent – Regularization – Dropout.

Gradient Descent

- Gradient descent is an optimization algorithm used in machine learning and neural networks to minimize the cost or loss function.
- ***cost or loss function*** measures the difference between the predicted output of the network and the actual output.
- The basic idea of gradient descent is to ***iteratively adjust the parameters of the neural network*** in the direction of the negative gradient of the cost function with respect to those parameters.
- This process continues until the cost function reaches a minimum or converges to a small value.



Gradient Descent Algorithm

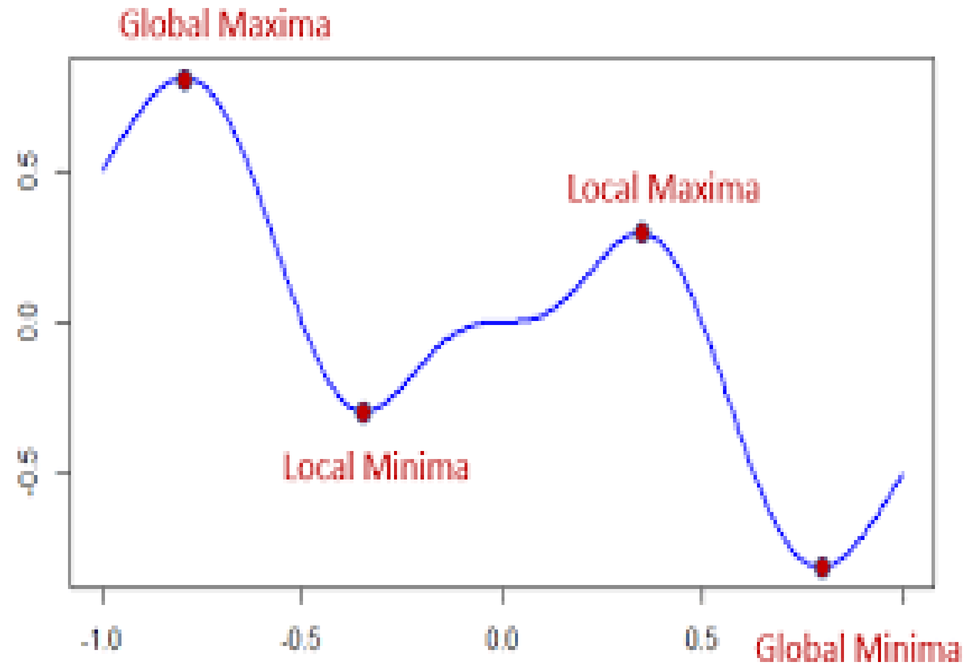
- Gradient Descent is defined as one of the most commonly used ***iterative optimization algorithms*** of machine learning to train the machine learning and deep learning models.

- It helps in finding the ***local minimum*** of a function.

- The best way to define the local minimum or local maximum of a function using gradient descent is as follows:

*-If we move towards a negative gradient or away from the gradient of the function at the current point, it will give the **local minimum** of that function.*

*-Whenever we move towards a positive gradient or towards the gradient of the function at the current point, we will get the **local maximum** of that function.*



Mathematical Representation

- ***Cost Function*** quantifies the error between predicted values and expected values and presents it in the form of a single real number.
- After making a hypothesis with initial parameters, we calculate the Cost function.
- ***Goal to reduce the cost function***, we modify the parameters by using the Gradient descent algorithm over the given data.

Hypothesis:	$h_\theta(x) = \theta_0 + \theta_1 x$
Parameters:	θ_0, θ_1
Cost Function:	$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$
Goal:	$\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

What is Gradient Descent?

Example:

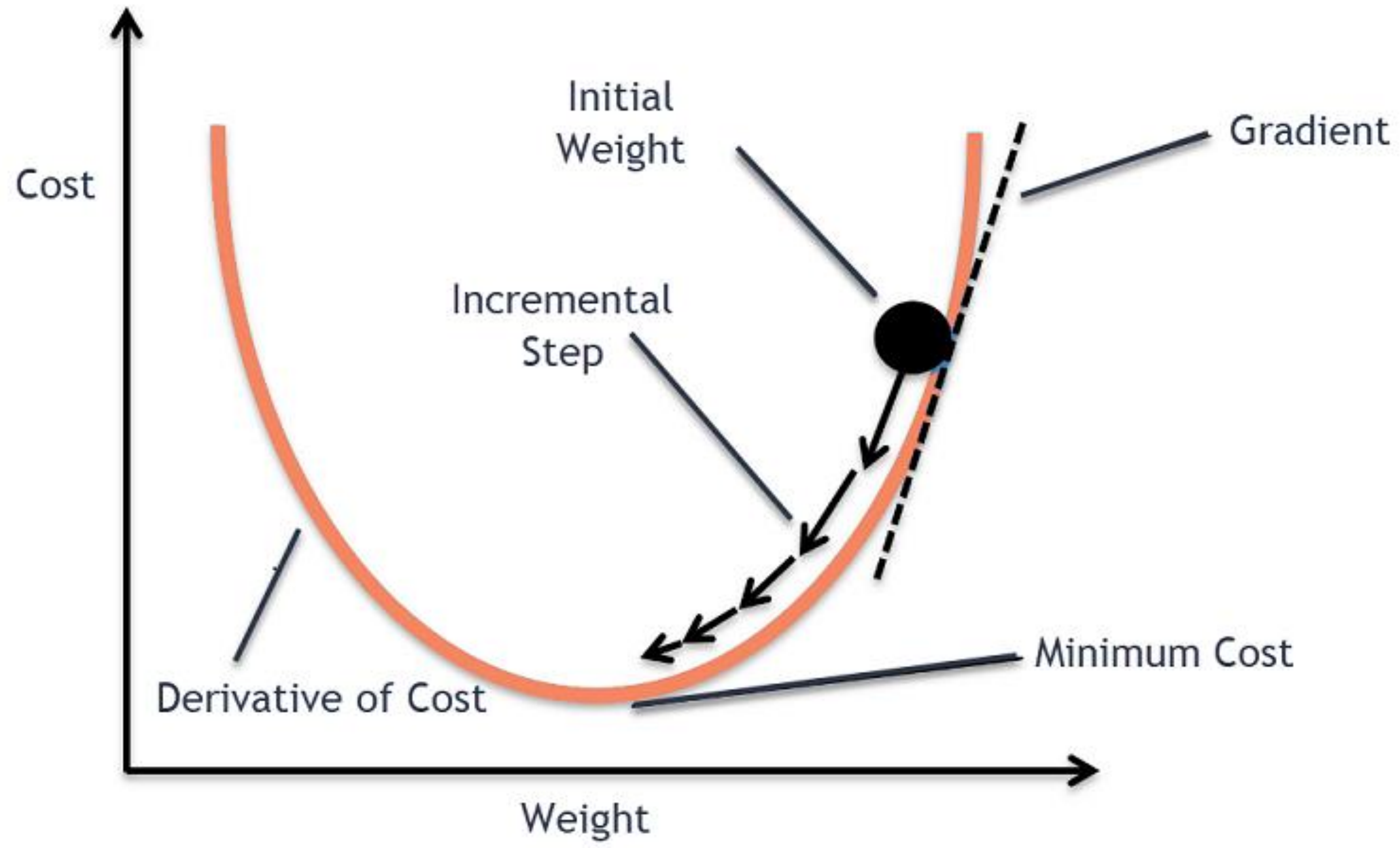
- *Let's say you are playing a game where the players are at the top of a mountain, and they are asked to reach the lowest point of the mountain. Additionally, they are blindfolded. So, what approach do you think would make you reach the lake?*
- The best way is to observe the ground and find where the land descends.
- From that position, take a step in the descending direction and iterate this process until we reach the lowest point.

Gradient Descent

Gradient descent

- Gradient descent is an iterative optimization algorithm for finding the local minimum of a function.
- To find the local minimum of a function using gradient descent, we must take steps proportional to the negative of the gradient (move away from the gradient) of the function at the current point.
- If we take steps proportional to the positive of the gradient (moving towards the gradient), we will approach a local maximum of the function, and the procedure is called **Gradient Ascent.**

Gradient descent



Steps in Gradient Descent

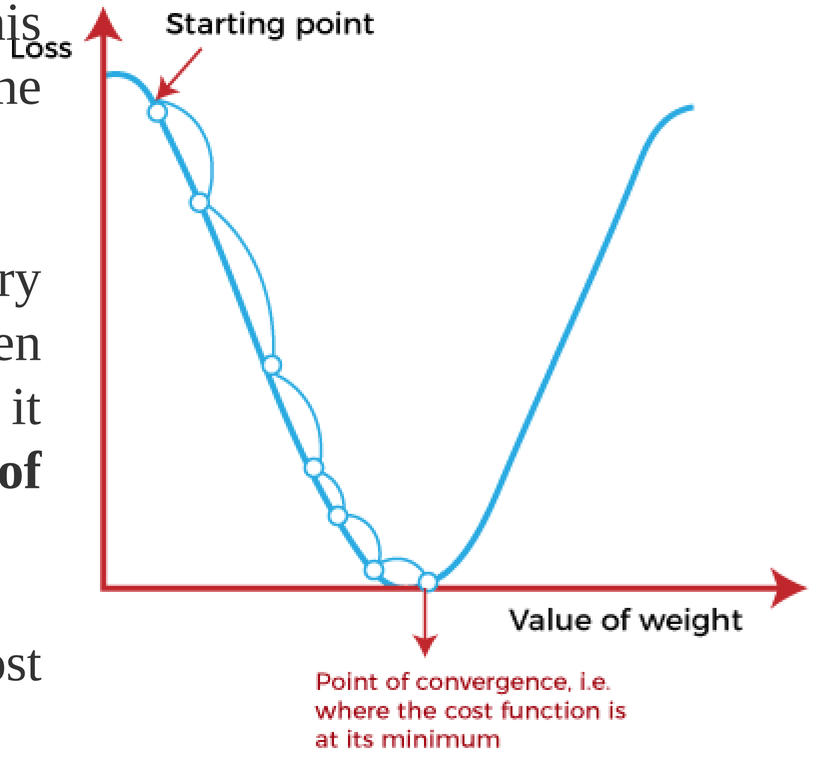
- The goal of the gradient descent algorithm is to minimize the given function (say cost function). To achieve this goal, it performs two steps iteratively:

1. **Compute the gradient** (slope), the first order derivative of the function at that point
2. **Make a step (move) in the direction opposite to the gradient**, opposite direction of slope increase from the current point by alpha times the gradient at that point

- The starting point is considered just as an arbitrary point
- At this starting point, we will derive the first derivative or slope and then use a tangent line to calculate the steepness of this slope. Further, this slope will inform the updates to the parameters (weights and bias).

- The slope becomes steeper at the starting point or arbitrary point, but whenever new parameters are generated, then steepness gradually reduces, and at the lowest point, it approaches the lowest point, which is called **a point of convergence.**

- The main objective of gradient descent is to minimize the cost function or the error between expected and actual.


Starting point
Loss
Value of weight
Point of convergence, i.e. where the cost function is at its minimum

Gradient descent algorithm

$$\text{repeat until convergence } \{$$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$(\text{for } j = 1 \text{ and } j = 0)$$

$$\}$$

Initialize the parameters of the model with random values.

Calculate the gradient of the cost function with respect to each parameter.

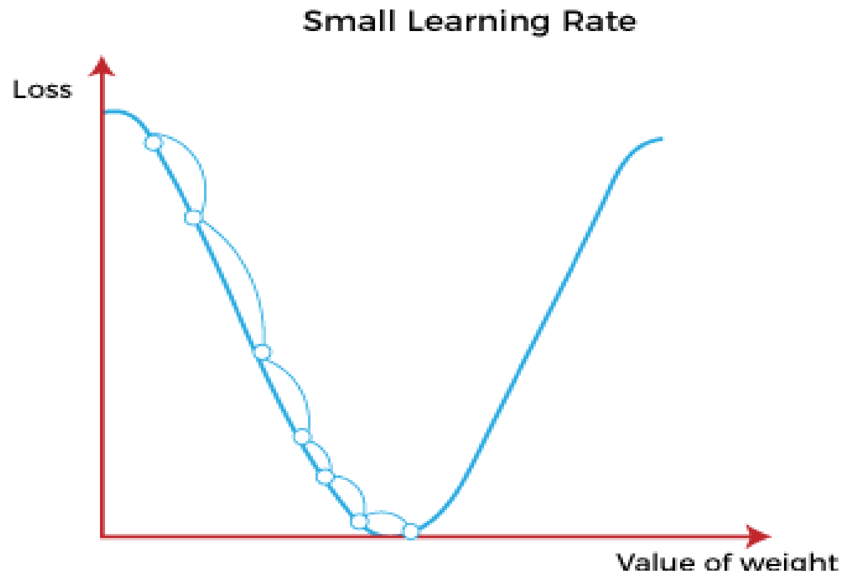
Update the parameters by subtracting a fraction of the gradient from each parameter. This fraction is called the learning rate, which determines the step size of the algorithm.

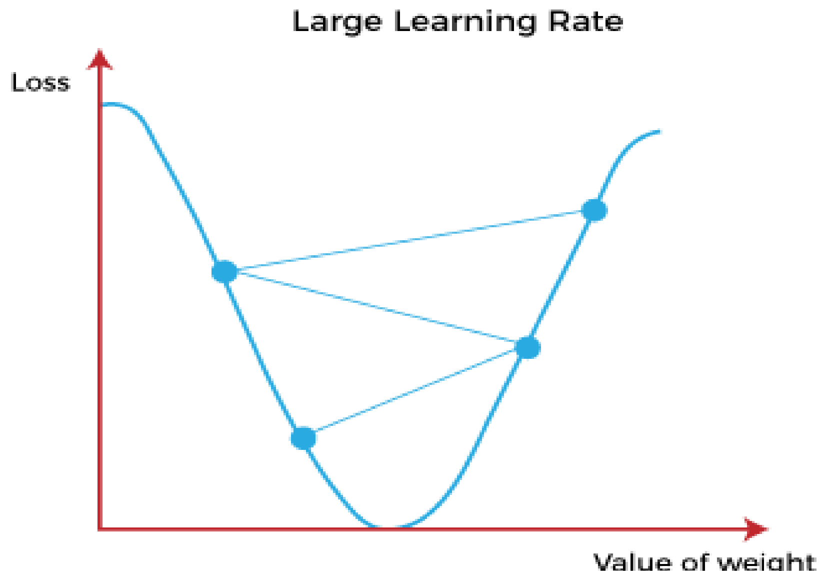
Repeat steps 2 and 3 until convergence, which is achieved when the cost function stops improving or reaches a predetermined threshold

Alpha is called **Learning rate**
– a tuning parameter in the optimization process. It decides the length of the steps.

Learning Rate

- Alpha – The Learning Rate
- ***It must be chosen carefully to end up with local minima.***
- If the learning rate is too high, we might **OVERSHOOT** the minima and keep bouncing, without reaching the minima
- If the learning rate is too small, the training might turn out to be too long





Learning rate

1. Learning rate is ***optimal***, model converges to the minimum
2. Learning rate is ***too small***, it takes more time but converges to the minimum
3. Learning rate is ***higher*** than the optimal value, it overshoots but converges
4. Learning rate is ***very large***, it overshoots and diverges, moves away from the minima, performance decreases on learning

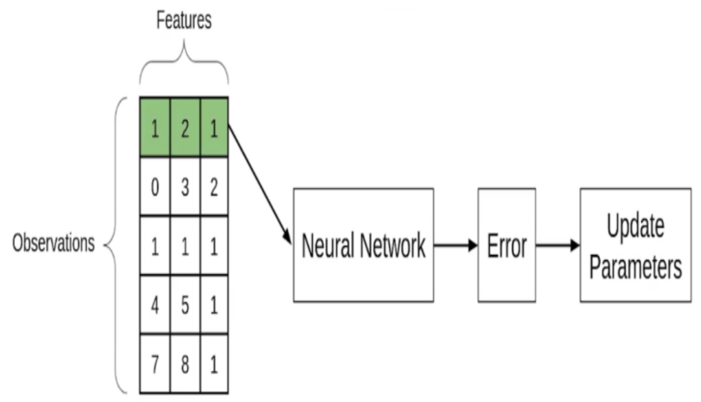
Variants of Gradient Descent

- There are two main variants of gradient descent:

a)Batch gradient descent b) Stochastic gradient descent. C) Mini batch Gradient Descent

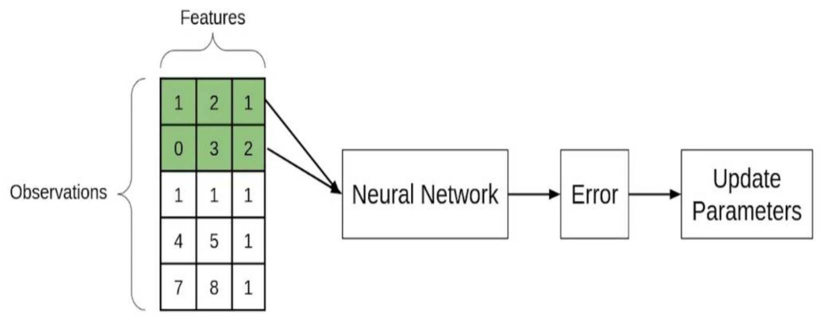
- ***Batch gradient descent*** involves computing the gradient of the cost function with respect to ***all the training examples in the dataset***. This approach can be computationally expensive for large datasets but can lead to more stable convergence.

- ***Stochastic gradient descent*** involves computing the gradient of the cost function with respect to ***only one training example at a time***. This approach can be faster than batch gradient descent but may result in a more noisy convergence.

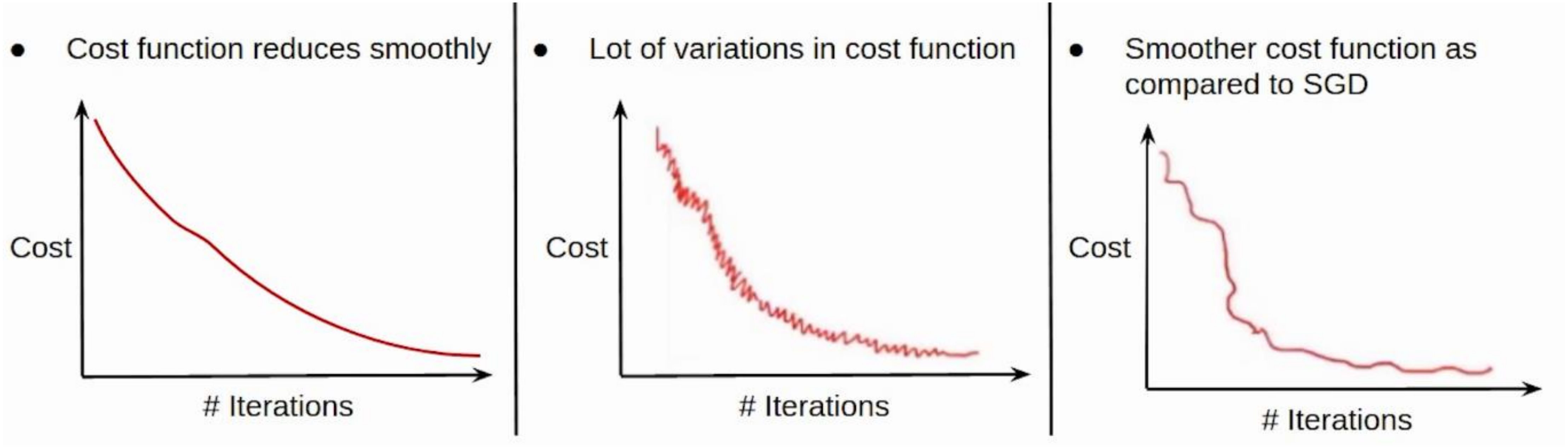


Variants of Gradient Descent

- ***Mini-batch gradient*** descent takes a subset of the entire dataset to calculate the cost function. So if there are ‘m’ observations then the number of observations in each subset or mini-batches will be more than 1 and less than ‘m’.
- In addition to the basic gradient descent, there are also several variants of gradient descent, such as momentum-based methods, Nesterov accelerated gradient, Adagrad, RMSprop, and Adam,



Comparison of Cost Function

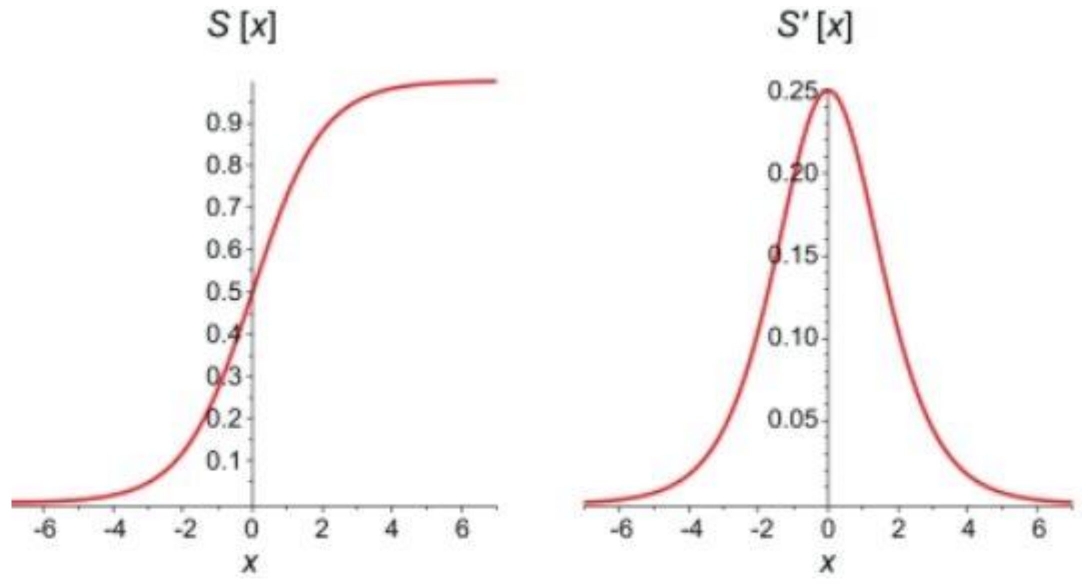


- Cost function reduces smoothly
- Lot of variations in cost function
- Smoother cost function as compared to SGD

Vanishing Gradient Problem

- As the backpropagation algorithm advances 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.
- The sigmoid function is one of the most popular activations functions used for developing deep neural networks.
- The use of sigmoid function restricted the training of deep neural networks because it caused the vanishing gradient problem.
- This caused the neural network to learn at a slower pace or in some cases no learning at all.

- On the graph below you can see a comparison between the sigmoid function itself and its derivative.
- First derivatives of sigmoid functions are bell curves with values ranging from 0 to 0.25.



Vanishing Gradient Problem

- For the nodes with sigmoid activation functions, we know that the partial derivative of the sigmoid function reaches a maximum value of 0.25.
- When there are more layers in the network, the value of the product of derivative decreases until at some point the partial derivative of the loss function approaches a value close to zero, and the partial derivative vanishes.
- We call this the vanishing gradient problem.

Vanishing Gradient Problem

- With shallow networks, sigmoid function can be used as the small value of gradient does not become an issue.
- When it comes to deep networks, the vanishing gradient could have a significant impact on performance.
- The weights of the network remain unchanged as the derivative vanishes.
- During back propagation, a neural network learns by updating its weights and biases to reduce the loss function. In a network with vanishing gradient, the weights cannot be updated, so the network cannot learn.
- The performance of the network will decrease as a result.

Mitigation of Vanishing Gradient problem

- The vanishing gradient problem is caused by the derivative of the activation function used to create the neural network. It can be overcome by

a) Using ReLu

b) Batch Normalization

c) Weight Initialization

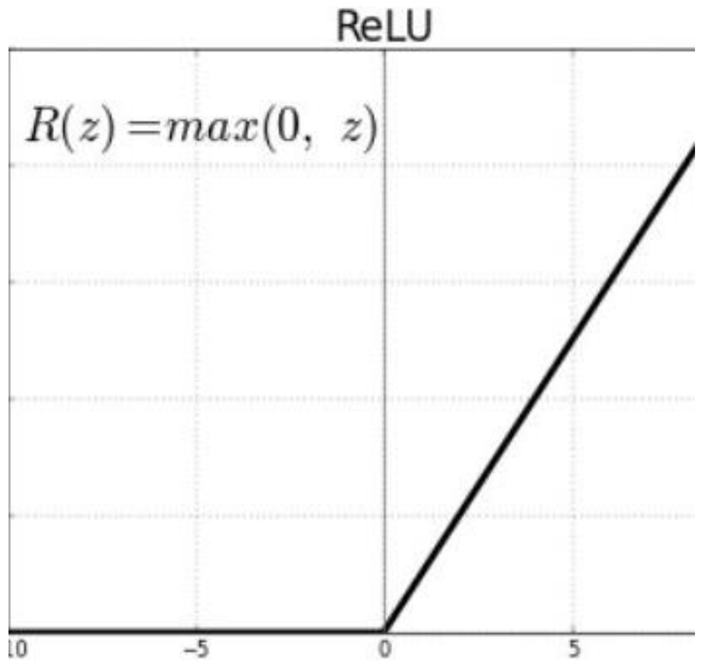
d) Gradient clipping

e) Residual connections

f) Architecture modifications

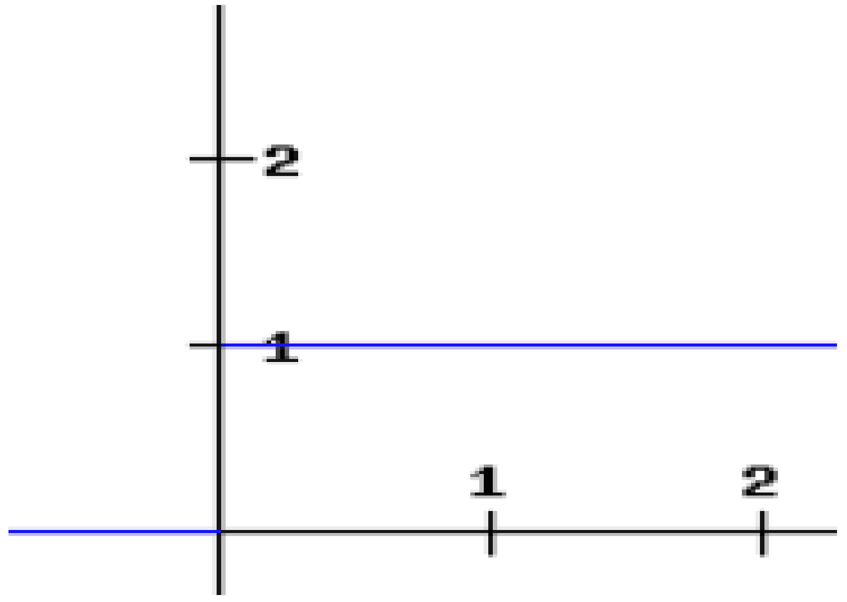
The simplest solution to the problem is to replace the activation function of the network.

- Instead of sigmoid, use an activation function such as ReLU.
- Rectified Linear Units (ReLU) are activation functions that generate a positive linear output when they are applied to positive input values.
- If the input is negative, the function will return zero.



Mitigation of Vanishing Gradient problem- Using ReLu

- The ***derivative of a ReLU function is defined as 1 for inputs that are greater than zero and 0 for inputs that are negative***.
- If the ReLU function is used for activation in a neural network in place of a sigmoid function, the value of the partial derivative of the loss function will be having values of 0 or 1 which ***prevents the gradient from vanishing.***
- The problem with the use of ReLU is ***when the gradient has a value of 0***. In such cases, the node is considered as a ***dead node*** since the old and new values of the weights remain the same.
- This situation can be avoided by the use of ***a leaky ReLU*** function which prevents the gradient from falling to the zero value.



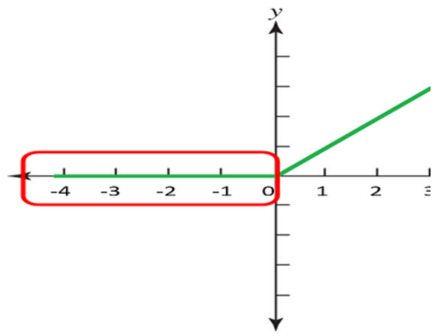
Mitigation of Vanishing Gradient problem

Causes of Dying ReLU Problem

- A dying ReLU always outputs the same value, i.e., 0, on any input value. This condition is known as the **dead state** of ReLU neurons.

(i) High learning rate

$$\underset{\text{New Weight}}{W^*_{ij}} = \underset{\text{Old Weight}}{W_{ij}} - \underset{\text{Learning Rate}}{\alpha}\underset{\text{Derivative of Error with respect to Weight}}{\left(\frac{\partial E}{\partial W_{ij}}\right)}$$



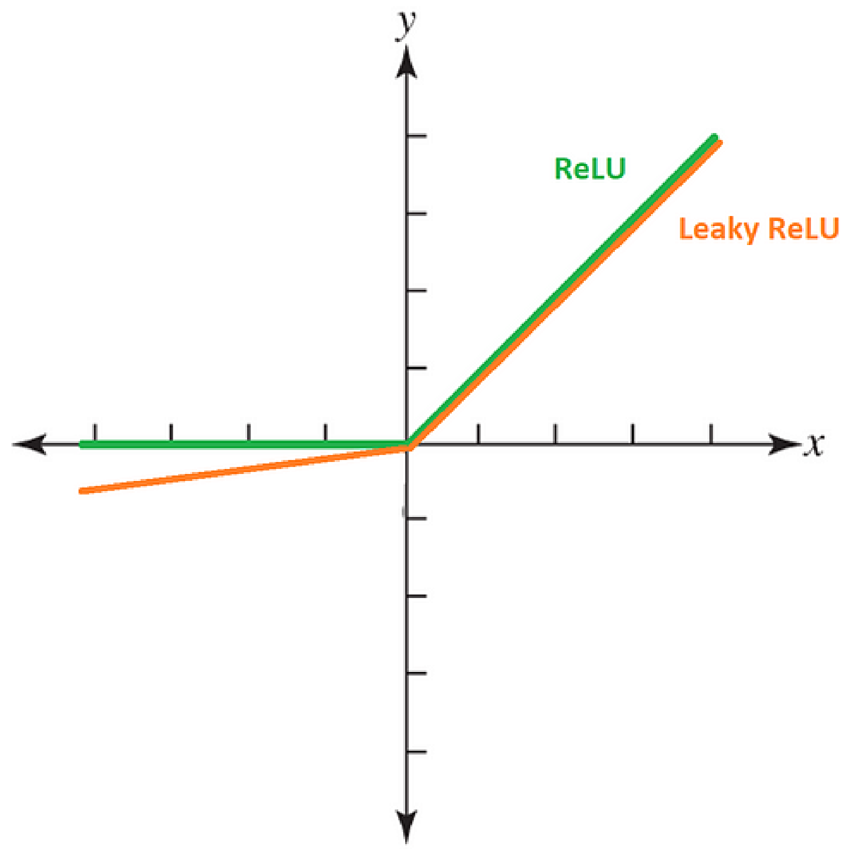
If our learning rate (α) is set too high, there is a significant chance that our new weights will end up in the highly negative value range since our old weights will be subtracted by a large number. These negative weights result in negative inputs for ReLU, thereby causing the dying ReLU problem to happen.

(ii) Large negative bias

A large negative bias term can cause the ReLU activation inputs to become negative. This causes the neurons to consistently output 0, leading to the dying ReLU problem.

Recovering from dying ReLU

- Different techniques are used to solve the dying ReLU problem.
- **i)Lowering the learning rates and negative bias**
- Lowering the learning rates and using a positive bias can mitigate the chance of dying ReLU.
- This pushes the ReLU activation inputs to the non-negative side(+ve). This technique helps activate the neurons with the flow of gradient.
- ii) **Using Leaky ReLU**
- solves a dying ReLU problem **by** adding a slight slope in the negative range. This modifies the function to generate small negative outputs when input is less than 0.

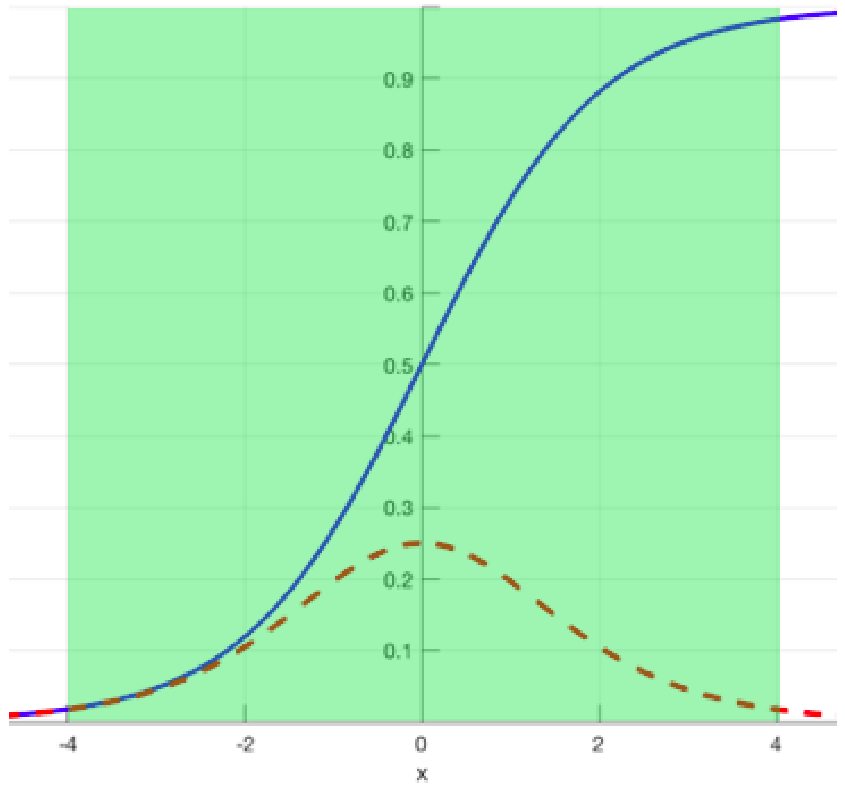


Mitigation of Vanishing Gradient problem

B) **Batch normalization**

The Vanishing Gradient problem arises when a large input space is mapped to a small one, causing the derivatives to disappear.

- Batch normalization reduces this problem by simply normalizing the input so |x| doesn't reach the outer edges of the sigmoid function.
- As seen in fig, it normalizes the input so that most of it falls in the green region, where the derivative isn't too small.



Mitigation of Vanishing Gradient problem

C) Weight Initialization

This is the process of assigning initial values to the weights in the neural network so that during back propagation, the weights never vanish.

D)Gradient clipping: Limiting the magnitude of the gradients can prevent them from becoming too small.

Eg:
```
# inside the optimizer we are doing clipping
optimizer=tf.keras.optimizers.SGD(clipvalue=0.5)
```

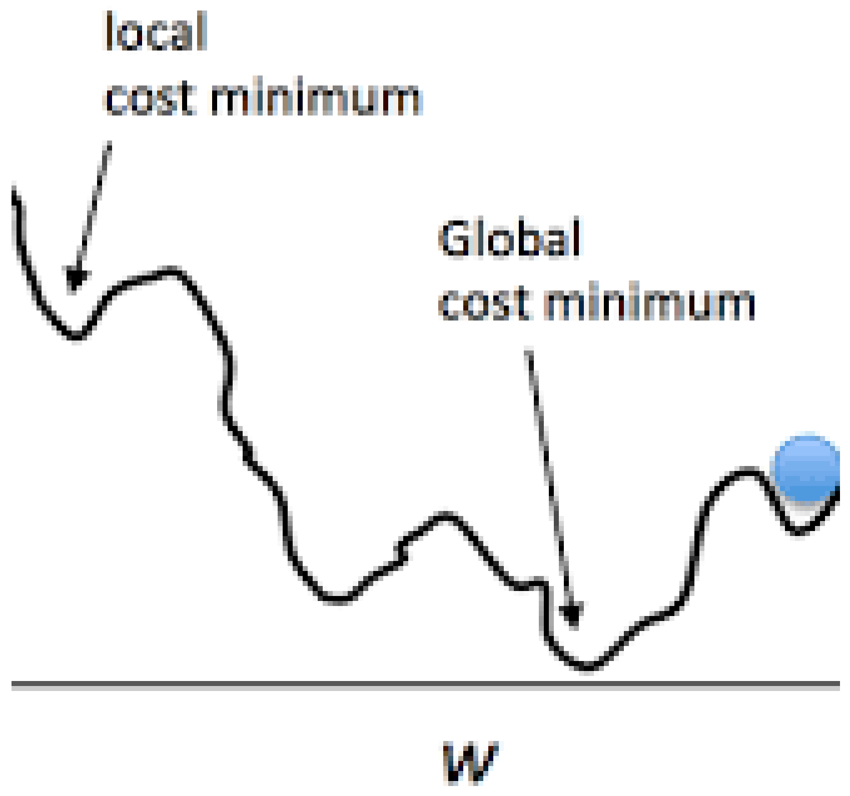
E)Residual connections: Residual connections allow the gradients to flow directly through the network, which can help to prevent them from becoming too small.

F**)Architecture modifications**: Changing the architecture of the network, such as using skip connections or recurrent connections, can help to prevent the gradients from vanishing.

By using these techniques, the vanishing gradient problem can be mitigated, allowing for more effective training of deep neural networks.

What is Local Minima

- A local minimum problem in neural networks refers to the phenomenon of a neural network getting stuck in a suboptimal solution while training, due to the shape of the error function.
- This can occur when the error function has multiple local minima (i.e., points of low error) in addition to a global minimum (i.e., the point of lowest error).
- If the neural network gets stuck in one of the local minima, it may not be able to find the global minimum and may achieve suboptimal performance.



RelU Heuristics for Avoiding Bad Local Minima

- ReLU (Rectified Linear Unit) is a popular activation function used in deep learning neural networks.
- It is a simple, non-linear function that is computationally efficient and has been shown to work well in a wide range of applications.
- One potential issue with ReLU is that it can result in bad local minima during training.
- This occurs when the gradient of the loss function becomes zero or close to zero, and the network is unable to improve its performance.

RelU Heuristics for Avoiding Bad Local Minima

- To avoid this problem, there are some heuristics that can be applied when using ReLU:

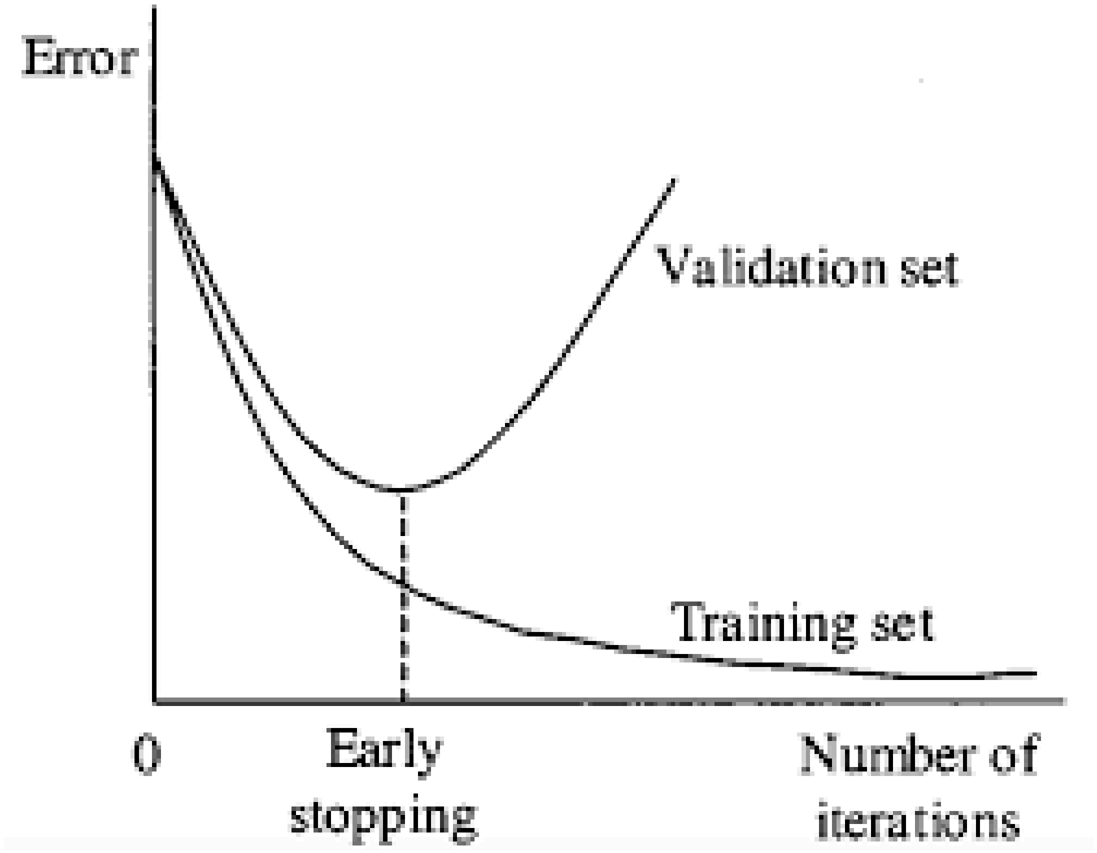
1. **Initialization:** Proper initialization of the weights can help to avoid bad local minima. A common technique is to initialize the weights with small random values.
2. **Learning rate scheduling**: Adjusting the learning rate during training can help to avoid bad local minima. Gradually decreasing the learning rate can allow the network to explore the parameter space more effectively.
3. **Adding noise:** Adding noise to the input data or the weights can help to avoid bad local minima. This can help the network to escape from the local minima and find better solutions.
4. **Regularization:** Regularization techniques, such as L1 or L2 regularization, can help to avoid overfitting and improve the generalization of the network. This can help to prevent the network from getting stuck in bad local minima.
5. **Ensemble learning:** Training multiple networks with different initializations and combining their outputs can help to avoid bad local minima. This can improve the performance and generalization of the network.

Heuristics for Faster Training

- There are several heuristics that can be applied to speed up the training process of deep learning models:
- A) **Mini-batch training**
- B) **Early stopping**
- **C) Learning rate scheduling**
- **D) Batch normalization**
- **E) Transfer learning**
- **F)Parallelization**

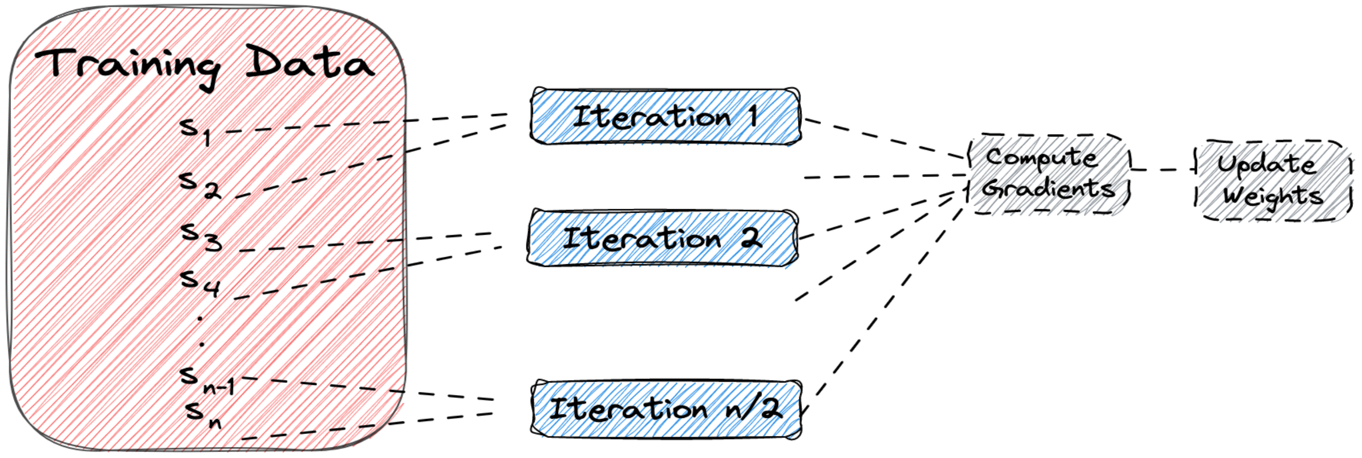
RelU Heuristics for Avoiding Bad Local Minima

1. **Early stopping**: Early stopping is a kind of cross-validation strategy where we keep one part of the training set as the validation set . When we see that the performance on the validation set is getting worse, we immediately stop the training on the model. This is known as early stopping.



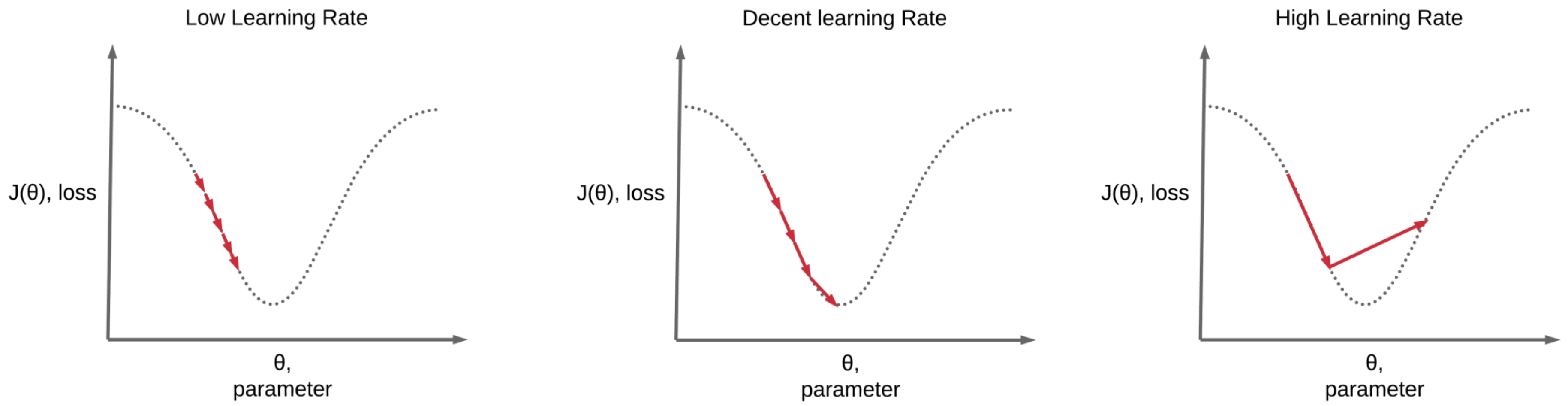
Heuristics for Faster Training

- **Mini-batch training**: Instead of updating the weights after each individual training example, mini-batch training updates the weights after processing a small batch of examples. This can lead to faster convergence of the loss function and can make more efficient use of computational resources.



Heuristics for Faster Training

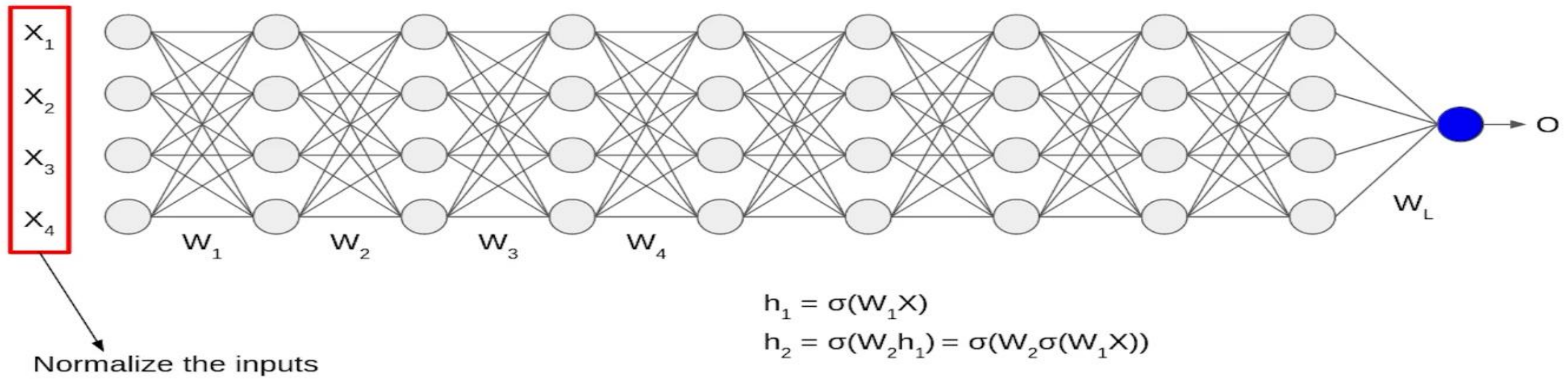
2.Learning rate scheduling: Adjusting the learning rate during training can help to speed up the training process. Initially, a high learning rate can be used to quickly converge to a reasonable solution, and then the learning rate can be gradually decreased to fine-tune the model.



Heuristics for Faster Training

4.Batch normalization: Batch normalization is a process to make neural networks faster and more stable through adding extra layers in a deep neural network.

The new layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer.



$h_1 = \sigma(W_1X)$

$h_2 = \sigma(W_2h_1) = \sigma(W_2\sigma(W_1X))$

$O = \sigma(W_Lh_{L-1})$

Heuristics for Faster Training

- **Batch normalization- works in two steps**

a) Normalization of the Input

b)Offsetting

a) Normalization of the Input

Data is transformed with a mean 0 and Standard Deviation1.

$$\mu = \frac{1}{m} \sum h_i$$

- *i* is the ith hidden layer
- *m* is the number of neurons at layer h

Standard Deviation of the hidden activations is

$$\sigma = \left[\frac{1}{m} \sum (h_i - \mu)^2 \right]^{1/2}$$

Heuristics for Faster Training

- **Batch normalization**
- We will normalize the hidden activations using mean and SD values. For this, we will subtract the mean from each input and divide the whole value with the sum of standard deviation and the smoothing term (ε).
- The smoothing term(ε) assures numerical stability within the operation by stopping a division by a zero value.

$$h_{i(norm)} = \frac{(h_i - \mu)}{\sigma + \varepsilon}$$

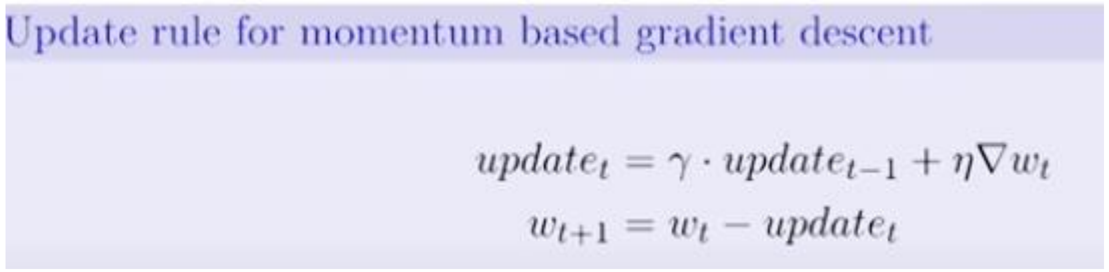
5.**Transfer learning:** Transfer learning is a technique that can speed up the training process by reusing pre-trained models. The pre-trained model can be used as a starting point for a new model, which can help to reduce the number of epochs needed to train the model.

6.**Parallelization:** Parallelization is a technique that can speed up the training process by distributing the computation across multiple devices or nodes. This can help to reduce the time needed to train the model, especially for very large datasets or models.

By applying these heuristics, it is possible to speed up the training process of deep learning models and reduce the time and resources needed to train high-quality models.

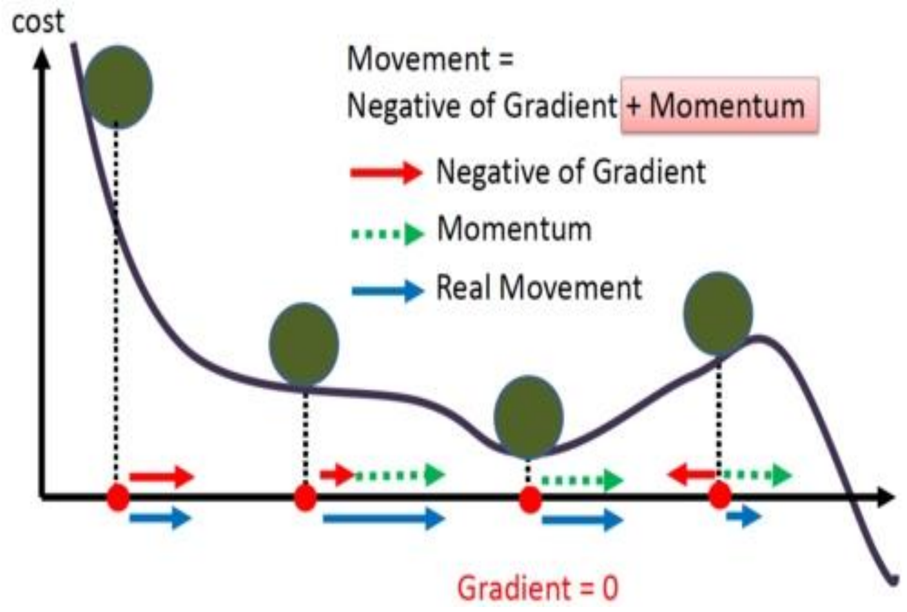
Gradient descent with momentum

- The issue discussed above can be solved by including the previous gradients in our calculation.
- The intuition behind this is if we are repeatedly asked to go in a particular direction, we can take bigger steps towards that direction.
- The weighted average of all the previous gradients is added to our equation, and it acts as momentum to our step.



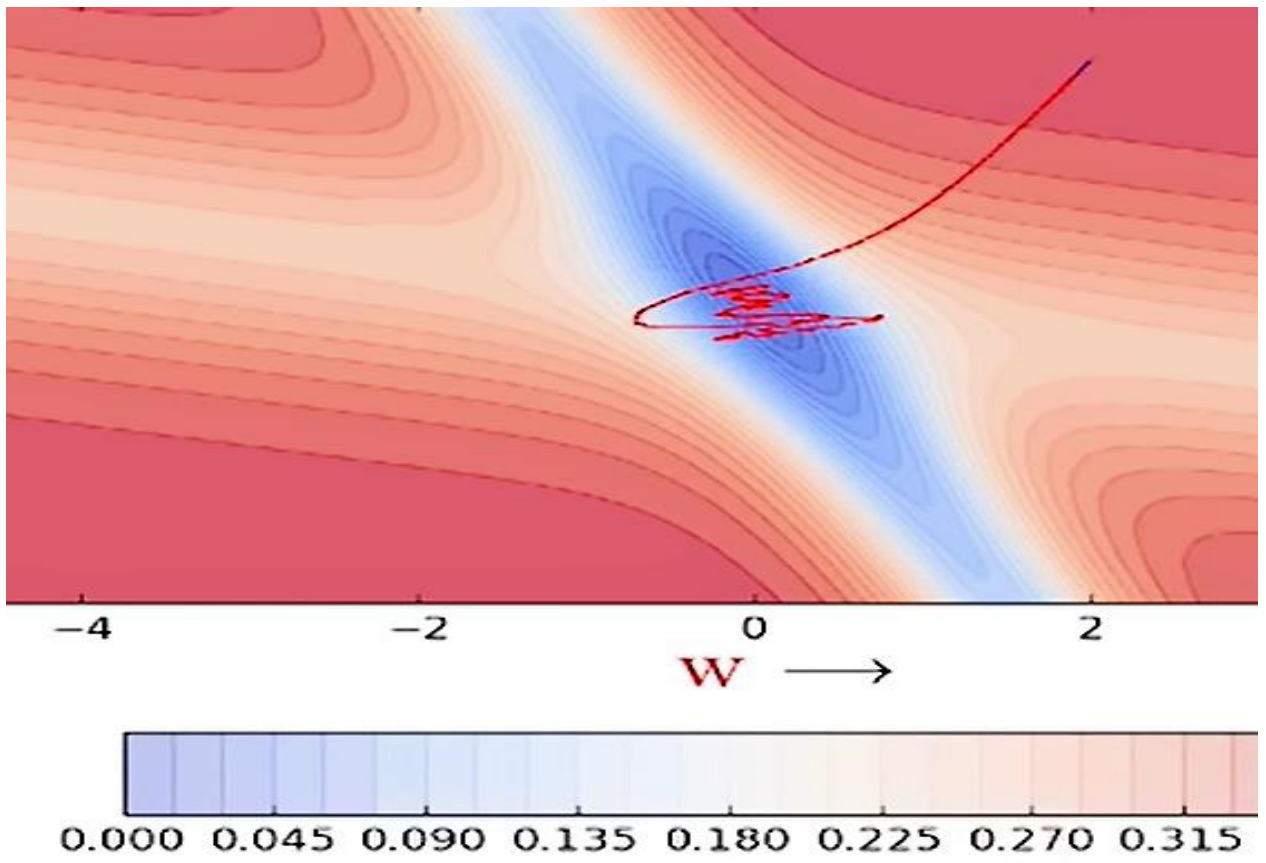
Update rule for momentum based gradient descent

$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$



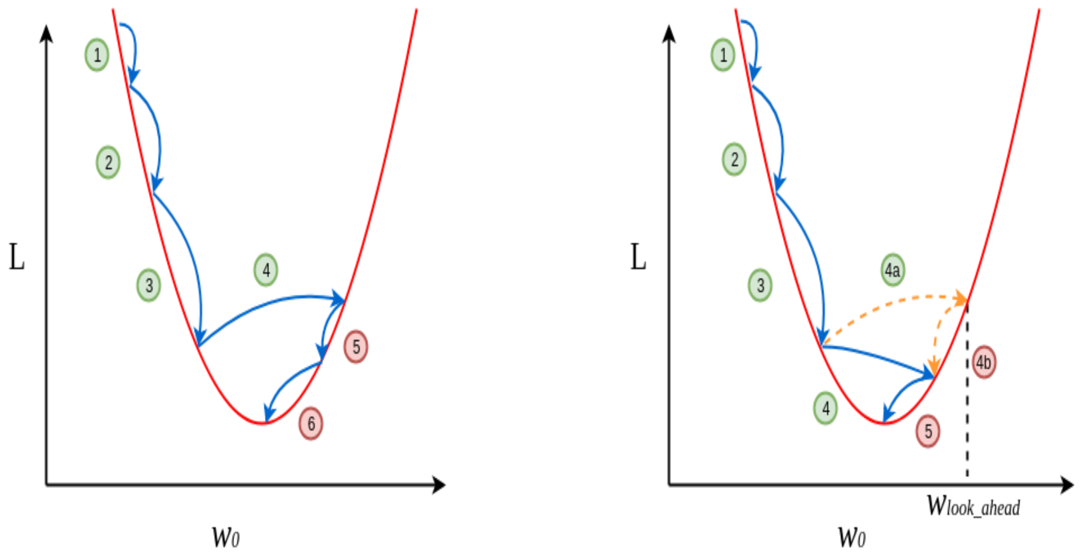
Gradient descent with momentum

- We can understand gradient descent with momentum from the above image. As we start to descend, the momentum increases, and even at gentle slopes where the gradient is minimal, the actual movement is large due to the added momentum.
- But this added momentum causes a different type of problem.
- We actually cross the minimum point and have to take a U-turn to get to the minimum point. Momentum-based gradient descent oscillates around the minimum point, and we have to take a lot of U-turns to reach the desired point.
- Despite these oscillations, momentum-based gradient descent is faster than conventional gradient descent.
- To reduce these oscillations, we can use Nesterov Accelerated Gradient.



Nesterov Accelerated Gradient (NAG)

- NAG resolves this problem by ad
  equation. The intuition behind N
  before you leap'. Let us try to un



(a) Momentum-Based Gradient Descent (b) Nesterov Accelerated Gradient Descent

🟢 $\Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)}$ 🔴 $\Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$

- As can see, in the momentum-based gradient, the steps become larger and larger due to the accumulated momentum, and then we overshoot at the 4th step.
- We then have to take steps in the opposite direction to reach the minimum point.
- However, the update in NAG happens in two steps.
- First, a partial step to reach the look-ahead point, and then the final update. We calculate the gradient at the look-ahead point and then use it to calculate the final update.
- If the gradient at the look-ahead point is negative, our final update will be smaller than that of a regular momentum-based gradient.
- Like in the above example, the updates of NAG are similar to that of the momentum-based gradient for the first three steps because the gradient at that point and the look-ahead point are positive. But at step 4, the gradient of the look-ahead point is negative.

- In NAG, the first partial update 4a will be used to go to the look-ahead point and then the gradient will be calculated at that point without updating the parameters. Since the gradient at step 4b is negative, the overall update will be smaller than the momentum-based gradient descent.
- We can see in the above example that the momentum-based gradient descent takes six steps to reach the minimum point, while NAG takes only five steps.
- This looking ahead helps NAG to converge to the minimum points in fewer steps and reduce the chances of overshooting.

How NAG actually works?

- We saw how NAG solves the problem of overshooting by ‘looking ahead’. Let us see how this is calculated and the actual math behind it.
- **Update rule for gradient descent:**
  $w_{t+1} = w_t - \eta \nabla wt$
  In this equation, the weight (W) is updated in each iteration. η is the learning rate, and ∇wt is the gradient.
- **Update rule for momentum-based gradient descent:**
  In this, momentum is added to the conventional gradient descent equation. The update equation is
  $w_{t+1} = w_t - update_t$
- **update$_t$ is calculated by:**
  $update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$
-

NAG

$$
\begin{aligned}
update_0 &= 0 \\
update_1 &= \gamma \cdot update_0 + \eta \nabla w_1 = \eta \nabla w_1 \\
update_2 &= \gamma \cdot update_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2 \\
update_3 &= \gamma \cdot update_2 + \eta \nabla w_3 = \gamma(\gamma \cdot \eta \nabla w_1 + \eta \nabla w_2) + \eta \nabla w_3 \\
&= \gamma \cdot update_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3 \\
update_4 &= \gamma \cdot update_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4 \\
&\vdots \\
update_t &= \gamma \cdot update_{t-1} + \eta \nabla w_t = \gamma^{t-1} \cdot \eta \nabla w_1 + \gamma^{t-2} \cdot \eta \nabla w_1 + ... + \eta \nabla w_t
\end{aligned}
$$

This is how the gradient of all the previous updates is added to the current update.

NAG

- **Update rule for NAG:**
  $w_{t+1} = w_t - \text{update}_t$
  While calculating the update_t, We will include the look ahead gradient ($\nabla w_{\text{look\_ahead}}$).
  $\text{update}_t = \gamma \cdot \text{update}_{t-1} + \eta \nabla w_{\text{look\_ahead}}$
- **$\nabla w_{\text{look\_ahead}}$ is calculated by:**
  $w_{\text{look\_ahead}} = w_t - \gamma \cdot \text{update}_{t-1}$
- This look-ahead gradient will be used in our update and will prevent overshooting

Regularization

- Regularization is a technique used in deep learning to prevent overfitting and improve the generalization performance of a neural network. Overfitting occurs when a model learns to fit the training data too closely, resulting in poor performance on new, unseen data.
- There are several types of regularization techniques that can be used in deep learning:
- L1 and L2 regularization: L1 and L2 regularization are techniques that add a penalty term to the loss function of the network. The penalty term is proportional to the L1 or L2 norm of the weights, respectively, and helps to prevent overfitting by encouraging the network to learn simpler, more generalizable models.
- Dropout: Dropout is a technique that randomly drops out (sets to zero) a fraction of the neurons in a layer during training. This can help to prevent overfitting by forcing the network to learn more robust features that are not dependent on the presence of specific neurons.
- Early stopping: Early stopping is a technique that stops training the model when the performance on a validation set stops improving. This can help to prevent overfitting by preventing the model from continuing to learn the training data too closely.

- Data augmentation: Data augmentation is a technique that artificially increases the size of the training dataset by applying random transformations to the data, such as rotation, translation, and scaling. This can help to prevent overfitting by providing the network with more diverse examples to learn from.
- Batch normalization: Batch normalization is a technique that can help to prevent overfitting by reducing internal covariate shift. It normalizes the input to each layer of the network, which can help to stabilize the training process and reduce the risk of overfitting.
- By applying these regularization techniques, it is possible to improve the generalization performance of deep learning models and prevent overfitting. However, it is important to carefully choose which regularization techniques to use and tune the hyperparameters appropriately, as different techniques may be more effective for different models and datasets.

Dropout

- Dropout is a regularization technique used in deep learning to prevent overfitting and improve the generalization performance of a neural network. It works by randomly dropping out (setting to zero) a fraction of the neurons in a layer during training.
- When a neuron is dropped out, it is as if it does not exist, so its input and output connections are removed from the network. This has the effect of forcing the network to learn more robust features that are not dependent on the presence of specific neurons.

Dropout

- During testing, the entire network is used, but the outputs of the dropped out neurons are scaled by the dropout rate to compensate for the fact that more neurons are active during testing than during training.
- The dropout rate is a hyperparameter that determines the fraction of neurons to be dropped out during training. A typical dropout rate is between 0.2 and 0.5, but the optimal value may depend on the specific dataset and network architecture.
- Dropout is a widely used regularization technique in deep learning, as it is simple to implement, computationally efficient, and has been shown to be effective in preventing overfitting in a variety of applications. However, it may not be suitable for all models and datasets, and its effectiveness may depend on the specific hyperparameters chosen.