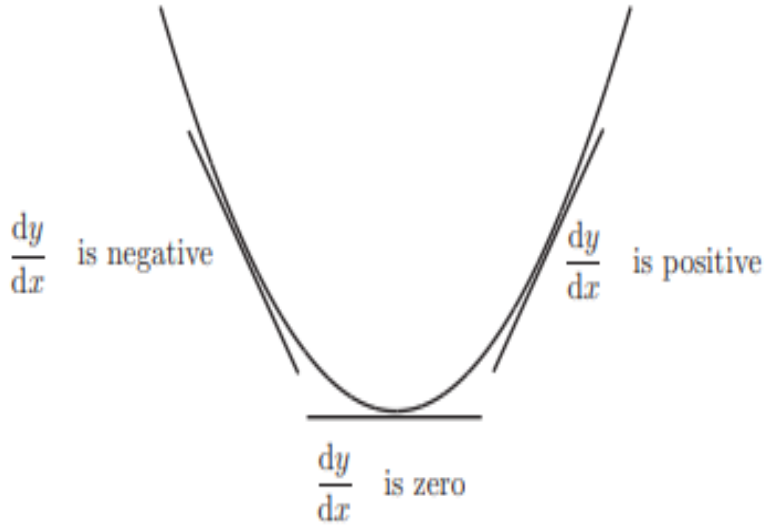
Gradient Descent

- In mathematical terminology, Optimization algorithm refers to the task of minimizing/maximizing an objective function f(x) parameterized by x.
- Similarly, in machine learning, optimization is the task of minimizing the cost/error function parameterized by the model's parameters.
- Gradient Descent is known as one of the most commonly used optimization algorithms to train machine learning and deep learning models by means of minimizing errors between actual and expected results.
- It’s based on a convex function and adjusts its parameters iteratively to minimize a given function to its local minimum.

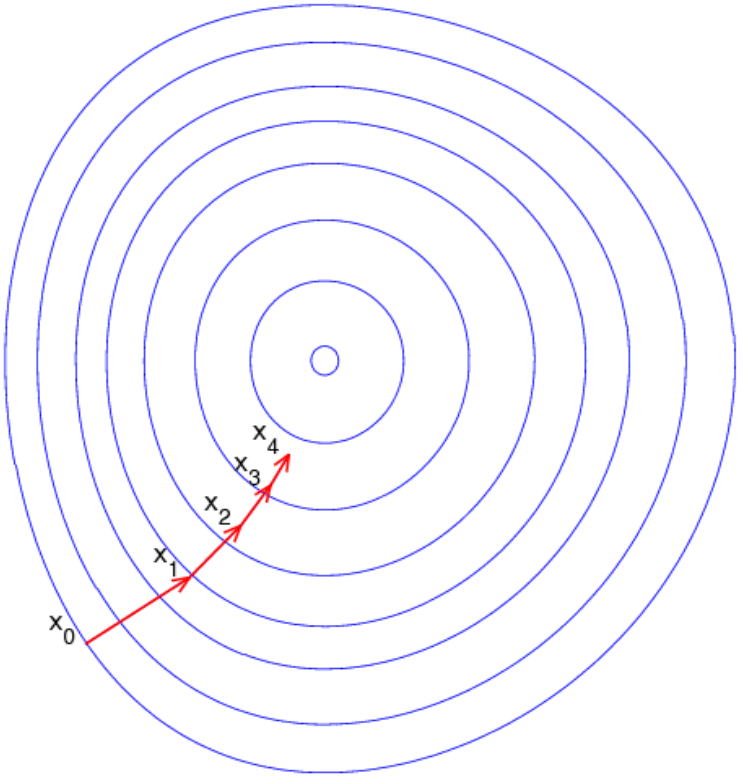
Gradient Descent

- The shape of the curve is same as the hill.
- Let us assume that this is a curve that is of the form **y=f(x).**
- Here we know the slope at any point is the **derivative of y with respect to x** at that particular position.
- Here we find out that slope while coming **downwards decreases and equal to zero at the tip or at the minimal position and increases as we move up again**



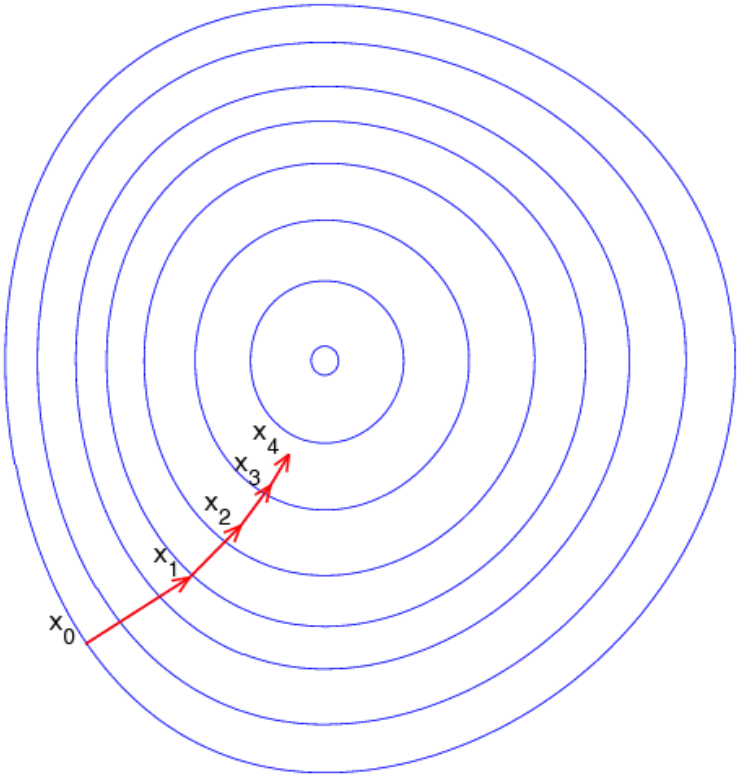
Gradient Descent

- Imagine a blindfolded man who wants to climb to the top of a hill with the fewest steps possible.
- He might start climbing the hill by taking really big steps in the steepest direction.
- But as he comes closer to the top, his steps will get smaller and smaller to avoid overshooting it.
- Imagine the image illustrates our hill from a top-down view and the red arrows are the steps of our climber.
- A gradient in this context is a vector that contains the direction of the steepest step the blindfolded man can take and how long that step should be.



Gradient Descent

- Note that the gradient ranging from X_0 to X_1 is much longer than the one reaching from X_3 to X_4.
- This is because the steepness / slope of the hill, which determines the length of the vector, is less along the top.
- This perfectly represents the example of the hill because the hill is getting less steep the higher it's climbed, so a reduced gradient goes along with a reduced slope and a reduced step size for the hill climber.



How Does Gradient Descent Work?

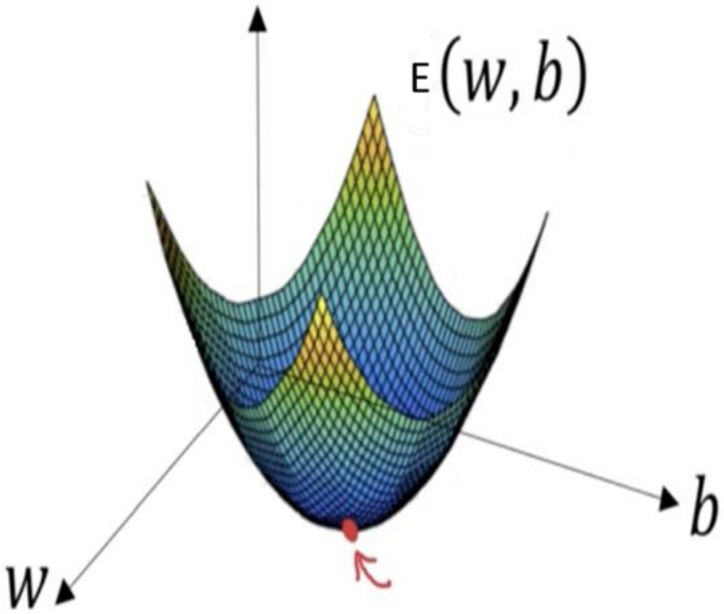
- Instead of climbing up a hill, think of gradient descent as hiking down to the bottom of a valley.
- The equation below describes what the gradient descent algorithm does:

$$b = a - \eta \nabla f(a)$$

- *Here, "*b*"* is the next position of our climber, and "a" represents his current position.
- The minus sign refers to the minimization part of the gradient descent algorithm. The η is a waiting factor and the gradient term ($\nabla f(a)$) is simply the direction of the steepest descent.
- This formula basically tells us the next position we need to go, which is the direction of the steepest descent.

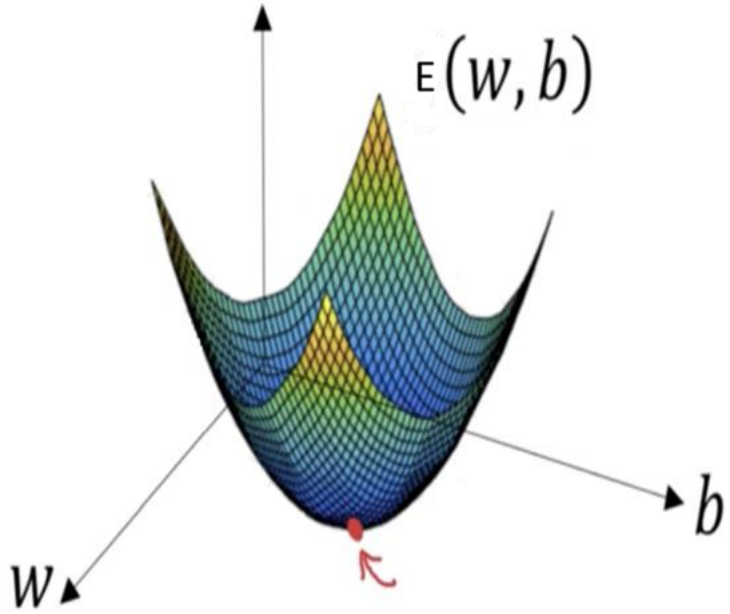
Gradient Descent

- Imagine we want to train our neural network model using our gradient descent algorithm to minimize the cost-function $E(w, b)$ and reach its local minimum by tweaking its parameters (w and b).
- The image shows the horizontal axes representing the parameters (w and b), while the cost function $E(w, b)$ is represented on the vertical axes.



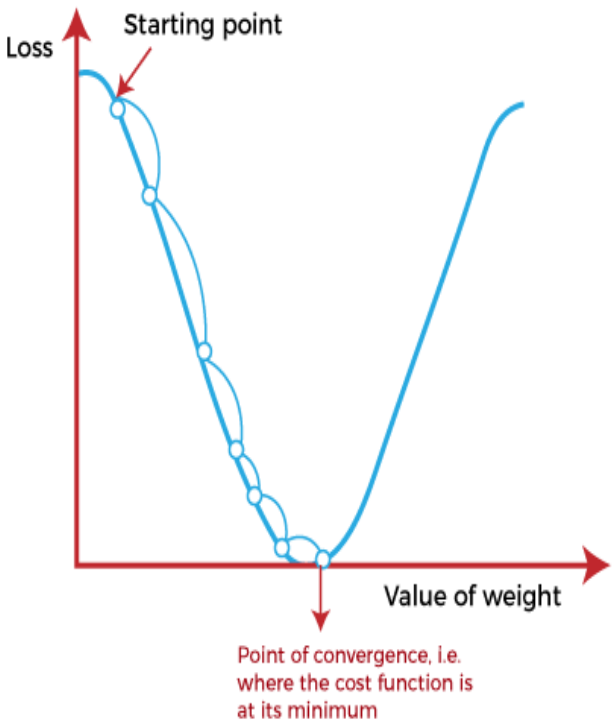
Gradient Descent

- We want to find the values of *w* and *b* that correspond to the minimum of the cost function (marked with the red arrow).
- To start, we initialize w and b with some random numbers.
- Gradient descent then starts at a point somewhere around the top in the curve, and it takes one step after another in the steepest downside direction (i.e., from the top to the bottom) until it reaches the point where the E(w, b) is as small as possible.



Gradient Descent

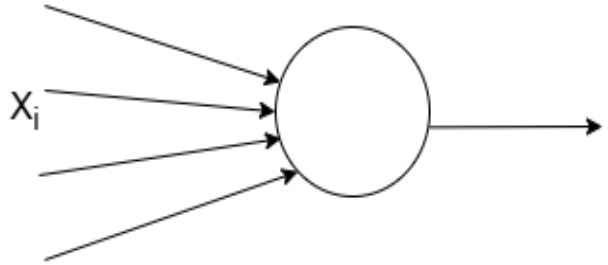
- A gradient is the slope of a function. In mathematical terms, a gradient is a partial derivative with respect to its inputs.
- The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning.
- Gradient descent is a numerical optimization algorithm that aims to find the optimal parameters—weights and biases—of a neural network by minimizing a defined cost function of a neural network model during training.
- It works by iteratively adjusting the weights or parameters of the model in the opposite direction of the gradient of the cost function until the minimum of the cost function is reached.



Gradient Descent

- The gradient descent algorithm is based on a convex function.
- Let there are N samples of the form (X_i, y_i) and neuron predicts the target class as y_i-hat instead of y_i. Thus the error function E(W) is the sum of square error.

$$\hat{y}_i = (W^t X_i)$$



$$E = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 = \frac{1}{2}\sum_{i=1}^{N}(W^t X_i - y_i)^2$$

$$\nabla_E W = \sum_{i=1}^{N}(\hat{y}_i - y_i)X_i$$

Gradient descent algorithm:

$$W = W - \eta(\nabla_E W)$$

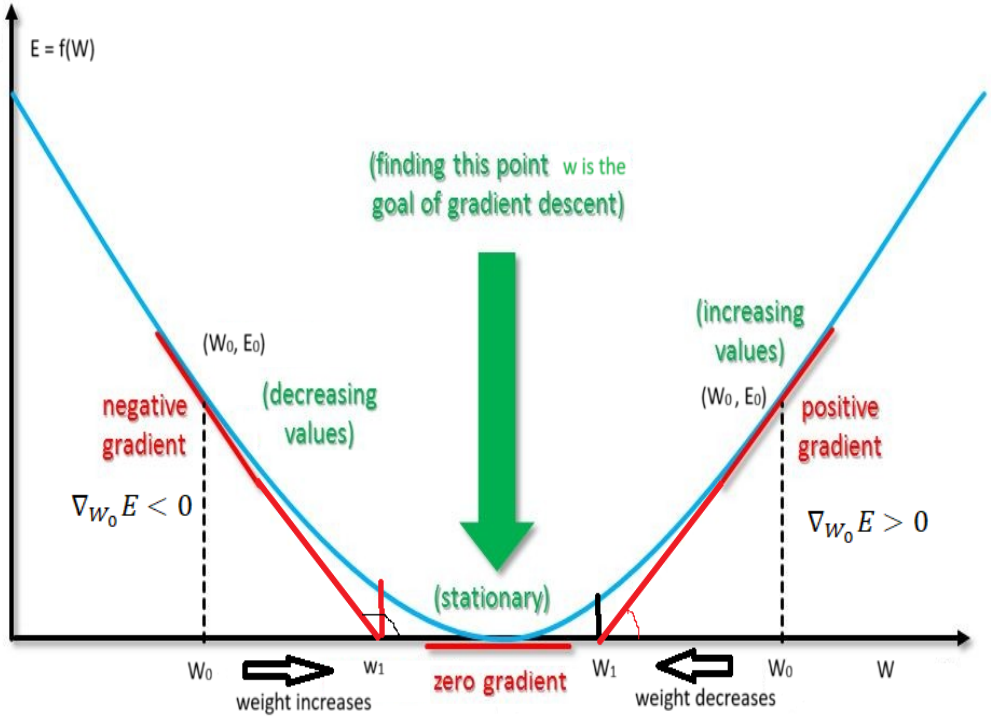
Rate of convergence factor or learning rate

- This is called Weight updation rule or Delta Rule

Gradient Descent

• If the gradient is positive at any point, then the weight parameter decreases.
•If the gradient is negative at any point, then the weight parameter increases.

•Gradient descent aims to minimize the cost function.
•It calculates the gradient (first-order derivative) of the cost function.
•It moves in the direction opposite to the gradient to reduce the cost function, with the step size controlled by the learning rate.



$$W_1 = W_0 - \eta \nabla_{W_0} E$$

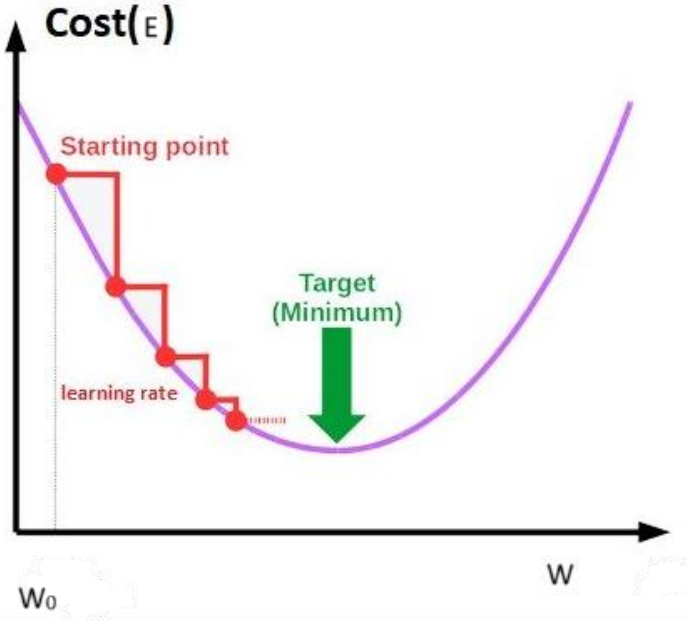
$$W_1 = W_0 + (+ve)quantity$$

$$W_1 = W_0 - \eta \nabla_{W_0} E$$

$$W_1 = W_0 - (+ve)quantity$$

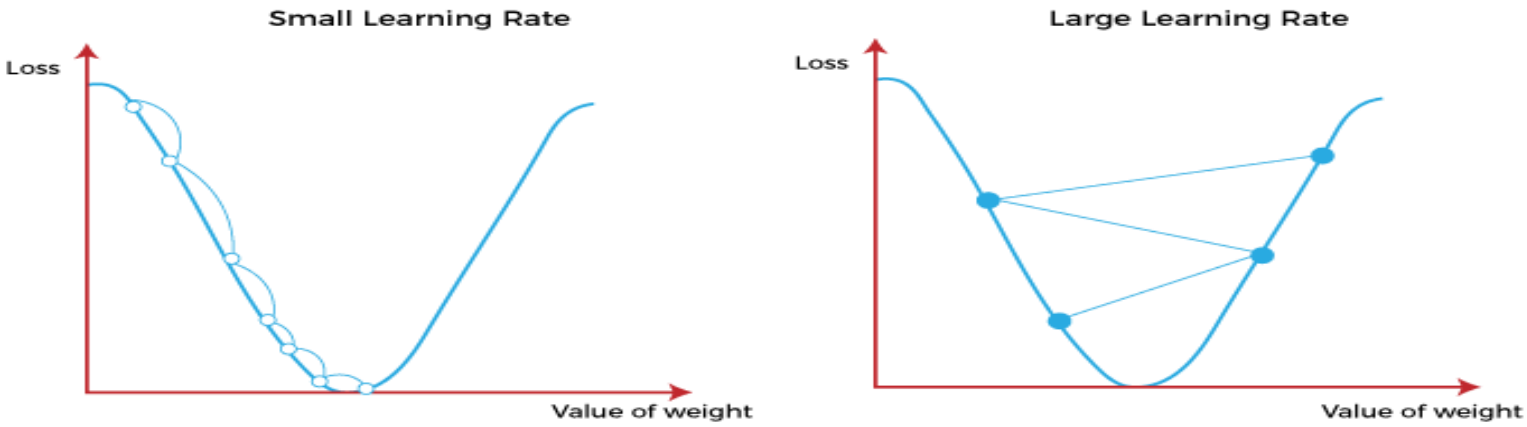
Gradient Descent

- The starting point is used to evaluate the performance as it is considered just as an arbitrary point.
- At this point, we will derive the gradient to calculate the steepness of this slope.
- Use weight update rule to update the parameters (weights and bias).
- The slope becomes steeper at the starting point or arbitrary point.
- For new parameters (or new points), the steepness gradually reduces, and
- At the lowest point, the gradient approaches to zero, which is called **a point of convergence.**



$$W_1 = W_0 - \eta \nabla_{W_0} E$$

Learning Rate



- Learning rate is defined as the step size taken to reach the minimum or lowest point.
- This is typically a small value that is evaluated and updated based on the behavior of the cost function.
- If the learning rate is high, it results in larger steps but also leads to risks of overshooting the minimum. This can cause the algorithm to miss the optimal point and repeatedly move across it, failing to converge to the desired minimum.
- At the same time, a low learning rate shows the small step sizes, which compromises overall efficiency but gives the advantage of more precision.

Types of Gradient Descent

- Based on the error in various training models, the Gradient Descent learning algorithm can be divided into **Batch gradient descent, stochastic gradient descent, and mini-batch gradient descent.**
- Gradient Descent aims to minimize a loss function (cost function) by iteratively adjusting the model parameters in the opposite direction of the gradient (steepest descent).

(1) Batch Gradient Descent:

- Batch gradient descent (BGD) computes the gradient using the entire dataset, i.e., using all point in the training set and update the weights.
- This procedure is known as the training epoch.

Advantages/disadvantages of Batch gradient descent:

- It produces less noise in comparison to other gradient descent.
- It produces stable gradient descent convergence.
- It is computationally less efficient as all resources are used for all training samples.

Stochastic gradient descent

- In SGD, instead of computing the gradient of the cost function using the whole dataset, it updates the model parameters for each training sample one at a time.
- 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.
- It is relatively fast to compute than batch gradient descent.
- It is more efficient for large datasets.
- As it requires only one training example at a time, hence it is easier to store in allocated memory.
- 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.

Mini Batch 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 moderate computational efficiency and noisy gradient descent.
- **Advantages of Mini Batch gradient descent:**
- It is easier to fit in allocated memory.
- It is computationally efficient compare to batch GD.
- It produces stable gradient descent convergence.

Challenges with the Gradient Descent

- Although we know Gradient Descent is one of the most popular methods for optimization problems, it still also has some challenges. There are a few challenges as follows:

1. Local Minima and Saddle Point:

- For convex problems, gradient descent can find the global minimum easily, while for non-convex problems, it is sometimes difficult to find the global minimum, where the machine learning models achieve the best results.

Challenges with the Gradient Descent

- Whenever the slope of the cost function is at zero or just close to zero, this model stops learning further.
- Apart from the global minimum, there occur some scenarios that can show this slop, which is saddle point and local minimum.
- Local minima generate the shape similar to the global minimum, where the slope of the cost function increases on both sides of the current points.
- A **saddle point** is a point on the surface of a function where the gradient (slope) is zero, but the point is neither a local minimum nor a local maximum.
- Instead, a saddle point has characteristics of both: in some directions, the function appears to increase (like a local minimum), and in other directions, it decreases (like a local maximum).

Key Characteristics of a Saddle Point

(i) Zero Gradient:

- At a saddle point, the gradient (first derivative) of the function is zero, meaning there is no immediate slope or direction of steepest descent/ascent.
- However, unlike minima or maxima, the second derivative (or curvature) of the function varies in different directions.

Key Characteristics of a Saddle Point

(ii) Mixed Curvature:

- A saddle point has both positive and negative curvatures depending on the direction. For example:
  - If you move in one direction (e.g., along the x-axis), the function might behave like a local minimum, increasing in value as you move away from the saddle point.
  - But if you move in another direction (e.g., along the y-axis), the function behaves like a local maximum, decreasing as you move away from the saddle point.

Example

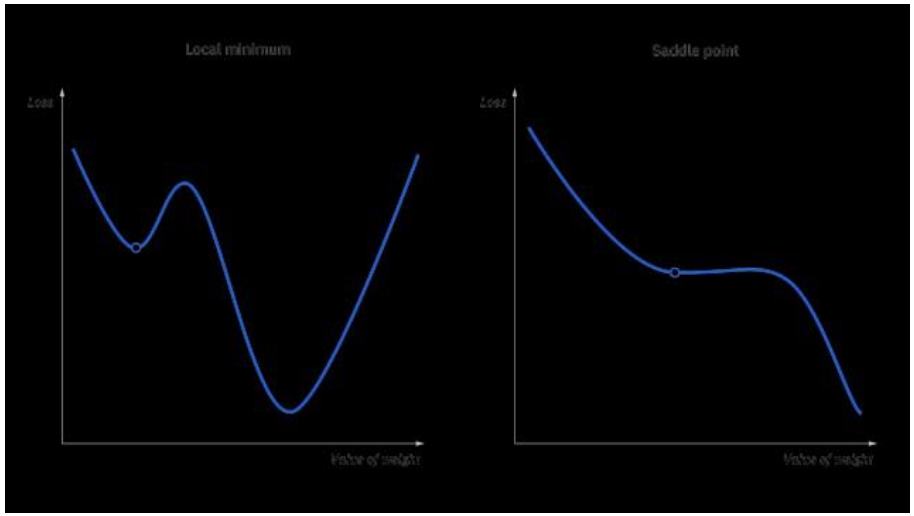
- A classic example of a saddle point is the shape of a horse saddle or the surface of a hyperbolic paraboloid, described by the function:
- $f(x,y)=x^2-y^2$. In this case, the point (0,0) is a saddle point because:
  - Along the x-axis (where y=0), the function behaves like a parabola that curves upwards, indicating a local minimum.
  - Along the y-axis (where x=0), the function behaves like a parabola that curves downwards, indicating a local maximum.
  - This mixed behavior makes it a saddle point, as it's neither a strict minimum nor maximum but exhibits properties of both.

Saddle Points in Optimization

- In optimization problems, especially in high-dimensional spaces like neural network training, saddle points are common.
- They can slow down gradient-based optimization algorithms (like gradient descent), as the algorithm may get "stuck" around these points, thinking it has reached an optimal solution because the gradient is zero.
- Unlike local minima or maxima, saddle points are less desirable because they do not represent an optimal solution but rather a point where the optimization process might get confused.

Saddle Point vs. Local Minima/Maxima:

- **Local Minimum:** The function value is lower than its neighboring points in all directions.
- **Local Maximum:** The function value is higher than its neighboring points in all directions.
- **Saddle Point:** The function has mixed behavior, acting like a minimum in some directions and a maximum in others.



The name of local minima is because the value of the loss function is minimum at that point in a local region. In contrast, the name of the global minima is given so because the value of the loss function is minimum there, globally across the entire domain the loss function.

Challenges with the Gradient Descent

2. Vanishing and Exploding Gradient

- In a deep neural network, if the model is trained with gradient descent and back propagation, there can occur two more issues other than local minima and saddle point.

i) Vanishing Gradients:

ii) Exploding Gradient:

Vanishing Gradient

- The **vanishing gradient problem** is a common issue in training deep neural networks, particularly those with many layers, like Recurrent Neural Networks (RNNs) and deep feed forward networks.
- It occurs when the gradients used in backpropagation become extremely small as they are propagated backward through the network.
- This can lead to very slow or stalled learning, because the weights in the earlier layers of the network are updated very little, if at all.

How it happens?

- In deep networks, during backpropagation, the gradients of the loss function are calculated and propagated backward from the output layer to the input layer, layer by layer.
- This involves applying the chain rule of calculus to compute gradients. In each layer, the gradient is multiplied by the derivative of the activation function.
- If the activation function has small derivatives (e.g., in the case of the sigmoid or hyperbolic tangent (tanh) functions), then the gradient can become very small as it is propagated through multiple layers.
- The result is that, in earlier layers of the network, the gradient becomes almost zero. This small gradient means the weights in these layers are updated very slowly, which significantly hampers the learning process.

Key factors contributing to the vanishing gradient problem

- **Activation Functions**: Functions like sigmoid and tanh can squish the input into a small range (between 0 and 1 for sigmoid, and -1 to 1 for tanh). Their derivatives are small in these ranges, leading to small gradients.
- **Deep Networks**: The more layers the network has, the more the gradients get multiplied together, leading to an exponential decrease in gradient magnitude.

Effects of vanishing gradients:

- **Slow learning in early layers**: The weights of early layers receive very small updates, meaning they learn much more slowly compared to the later layers.
- **Poor performance**: The network may fail to learn effectively, especially in the earlier layers, which are crucial for extracting important features from the input data.
- **Training may stop**: In extreme cases, the gradient can become so small that the network essentially stops learning.

Solutions to the vanishing gradient problem:

- **ReLU activation function**: The Rectified Linear Unit (ReLU) and its variants (e.g., Leaky ReLU) have gradients of 1 for positive inputs, which helps avoid the gradient shrinking to zero.
- **Batch Normalization**: Normalizing the input of each layer can help maintain a healthy gradient flow by preventing extreme values that lead to vanishing or exploding gradients.
- **Residual Networks (ResNets)**: Adding skip connections between layers in deep networks allows gradients to flow more directly through the network, which helps mitigate the vanishing gradient problem.
- **Weight initialization techniques**: Proper weight initialization (e.g., Xavier or He initialization) can help ensure that the gradients neither vanish nor explode in the early stages of training.
- **Gradient clipping**: Clipping gradients during backpropagation can prevent extremely small (or large) updates to the weights.

Where Batch Normalization is Applied:

- Batch normalization is typically applied after the linear transformation of each layer (i.e., after the matrix multiplication but before the activation function).
- For example, in a typical neural network layer, it would follow this order:

  Linear transformation (WX + b)→Batch Normalization→Activation Function (e.g., ReLU)

Exploding gradient problem

- The **exploding gradient problem** is a common issue that occurs during the training of deep neural networks, particularly in very deep networks and recurrent neural networks (RNNs).
- It happens when the gradients (the partial derivatives of the loss function with respect to the model parameters) become excessively large during backpropagation.
- This leads to unstable updates to the network's weights, causing the model's parameters to grow uncontrollably and ultimately causing the model to diverge or fail to learn properly.

How the Exploding Gradient Problem Happens:

- During the backpropagation process, the gradients are calculated layer by layer, from the output layer back to the input layer.
- In deep networks, each layer's gradient is computed by applying the chain rule, which involves multiplying gradients across many layers.
- If any of these gradients is large, then as you propagate backward, the gradient can exponentially increase, leading to extremely large weight updates in earlier layers.
- This problem is particularly prevalent in **deep networks** and **RNNs**, where gradients are propagated through many time steps or layers.
- The key cause is similar to the vanishing gradient problem but in the opposite direction—here, instead of gradients shrinking, they grow exponentially.

Effects of Exploding Gradients:

- **Unstable training**: The learning process becomes unstable because the updates to the weights are too large, which can cause the model to overshoot the optimal weights.
- **Divergence**: The model may fail to converge, and the loss function might fluctuate or increase indefinitely as the network tries to minimize it.
- **Weight overflow**: In extreme cases, the weights can grow so large that they result in numerical overflow, causing the network to break.

Causes of Exploding Gradients:

- **Deep Networks**: The more layers (or time steps in RNNs), the more products of gradients you have, which can lead to rapid growth of gradients if any gradient is large.
- **Poor Weight Initialization**: If weights are not initialized properly, it can contribute to large gradient values.
- **Inappropriate Learning Rates**: If the learning rate is too high, it can exacerbate the effect of exploding gradients.

Solutions to the Exploding Gradient Problem:

Several techniques are used to prevent or mitigate the exploding gradient problem:

1. **Gradient Clipping**: This is one of the most common techniques.

- When gradients become too large, they are "clipped" or scaled back to a smaller value within a predefined range.
- This prevents gradients from becoming excessively large while still allowing learning to continue.

Solutions to the Exploding Gradient Problem:

2. **Weight Regularization (L2 regularization)**: Applying L2 regularization can help constrain the growth of the weights, which in turn can prevent the gradients from becoming too large.
3. **Proper Weight Initialization**: Initializing weights carefully can prevent the gradients from growing too quickly. For example, **Xavier initialization** or **He initialization** is often used to ensure the gradients are not too large or too small at the start of training.
4. **Smaller Learning Rates**: Using a smaller learning rate can reduce the effect of large gradients, helping to prevent weights from changing drastically during updates.

Solutions to the Exploding Gradient Problem:

5. **Batch Normalization**: Batch normalization helps keep activations and gradients in check by normalizing the inputs to each layer.
- This can help mitigate both vanishing and exploding gradients.
6. **Resilient Architectures**: Architectures like **Residual Networks (ResNets)**, which introduce skip connections, help ensure that gradients can flow more easily through the network without becoming too large or too small.

Thank You!