

Gradient Descent

How do we train a neural network?

By repeatedly updating the weights and biases in the network.

By carrying out forward and back propagation

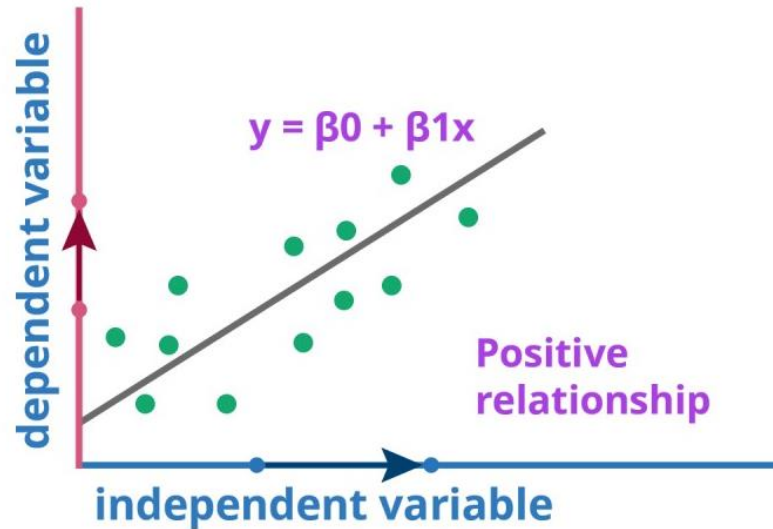
Training a ML model for Heart Health prediction: Example

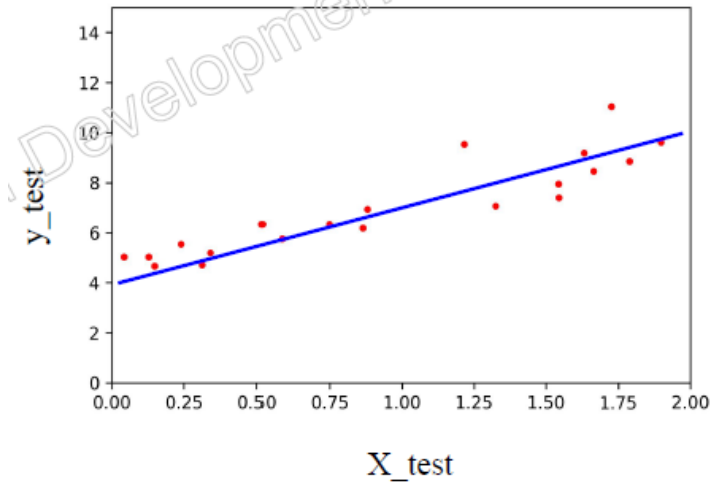
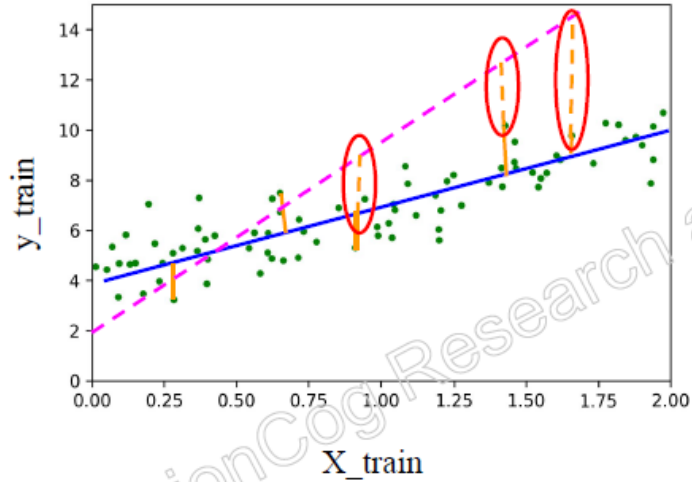
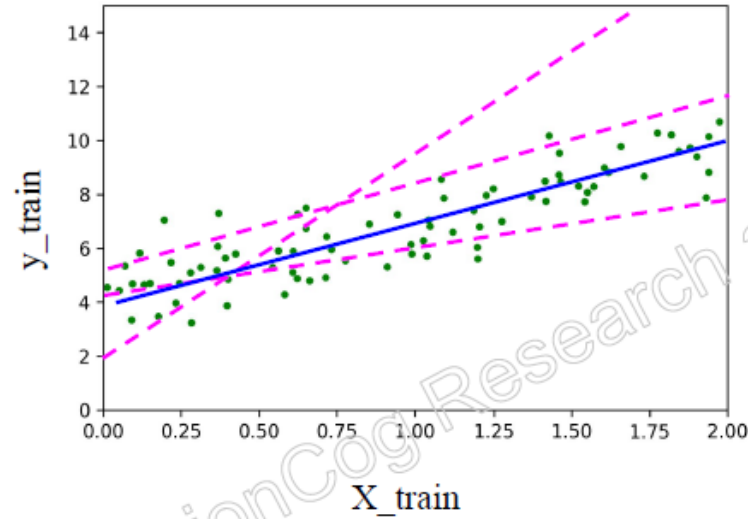
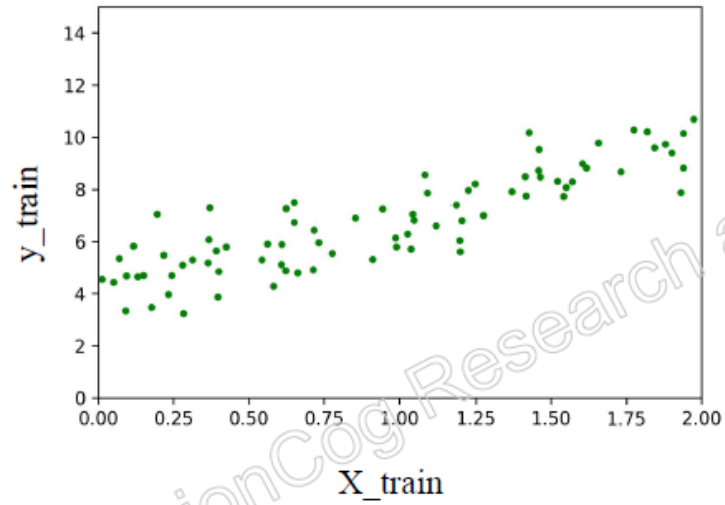
| Gender | Age | Hypertension | Does_smoke | Heart Condition (Y_i) | Predicted Stroke (Y_{ipred}) |
|--------|-----|--------------|------------|---------------------------|----------------------------------|
| 1 | 3 | 0 | 0 | 0 | 0.5 |
| 1 | 58 | 1 | 1 | 0.9 | 0.8 |
| 0 | 8 | 0 | 0 | 0.5 | 0.1 |
| 0 | 70 | 0 | 1 | 1 | 0.9 |
| 1 | 14 | 0 | 0 | 0 | 0 |

How to Model?

Modeling a relationship between *dependent* and *independent* variables for *prediction*.

Regression Model



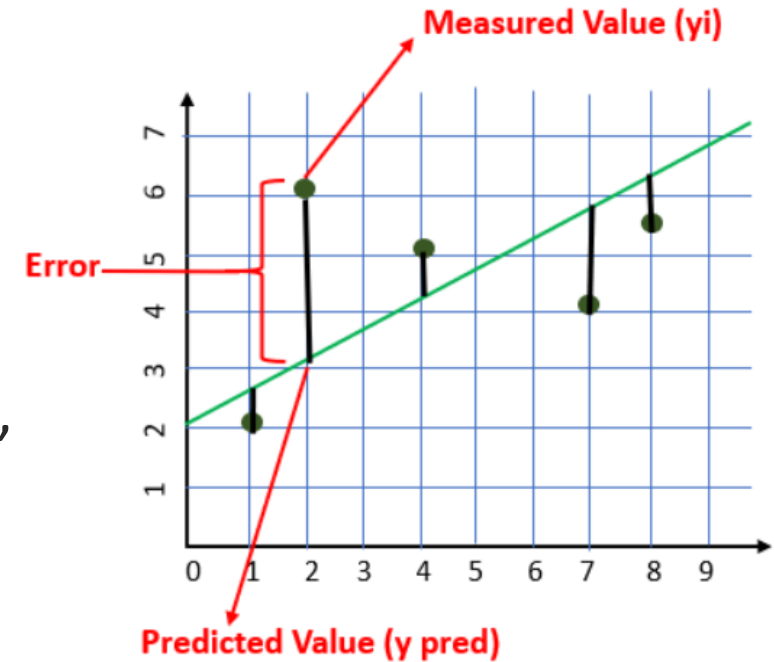


Objective:

- a. Find the best fit line/plane/hyper plane
 - a. Minimizing the error between the actual value and the predicted values
 - a. Minimizing the cost function
 - a. Find the optimal weights associated with independent variables (Features)

How to compute Error?: Cost/Loss Function

- Machines learn by considering loss function.
- It's a method of evaluating how well specific algorithm models the given data.
- Loss functions measure how far an estimated value is from its true value.
- If predictions deviates too much from actual results, loss function will be a very large number.
- Gradually, with the help of some optimization function, algorithm learns to reduce the error in prediction.



Cost Function

- The cost is the error in our predicted value. We will use the Mean Squared Error function to calculate the cost.

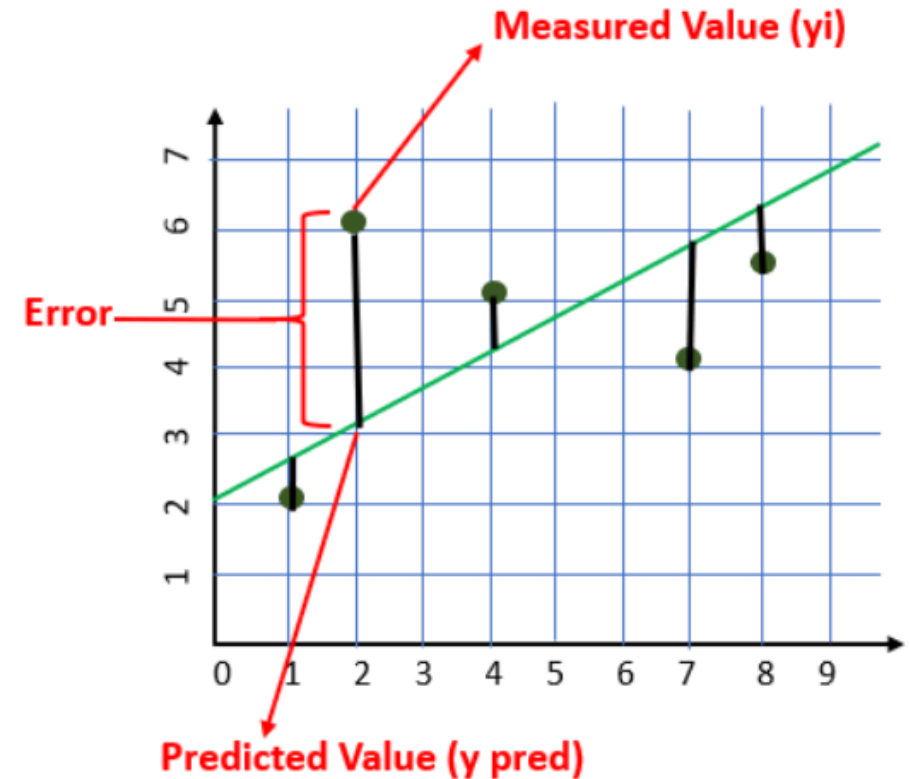
$$\text{Cost Function(MSE)} = \frac{1}{n} \sum_{i=0}^n (y_i - y_{i \text{ pred}})^2$$

Replace $y_{i \text{ pred}}$ with $mx_i + c$

$$\text{Cost Function(MSE)} = \frac{1}{n} \sum_{i=0}^n (y_i - (mx_i + c))^2$$

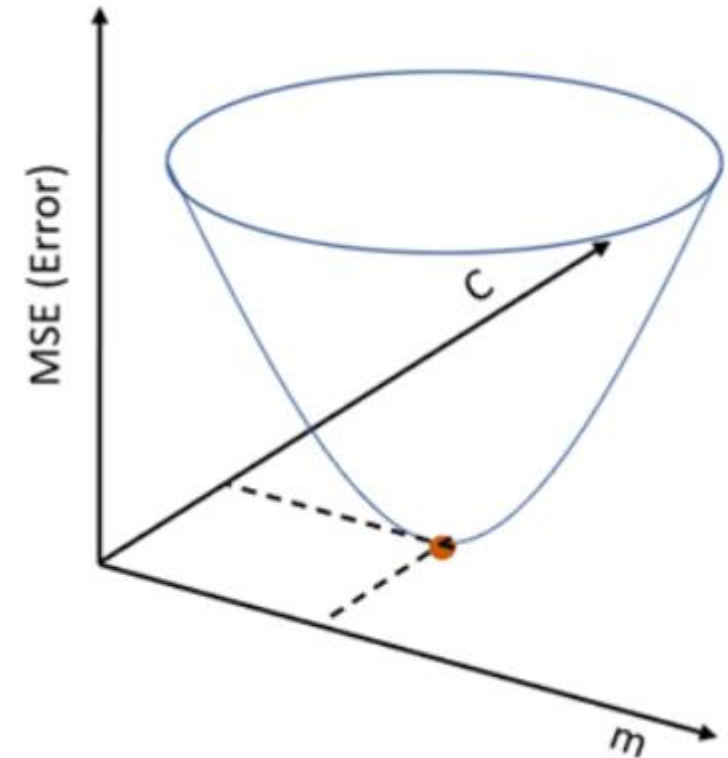
Our goal is to minimize the cost as much as possible in order to find the best fit line.

Try all the permutation and combination of m and c (inefficient way) to find the best-fit line?.



Gradient Descent Algorithm

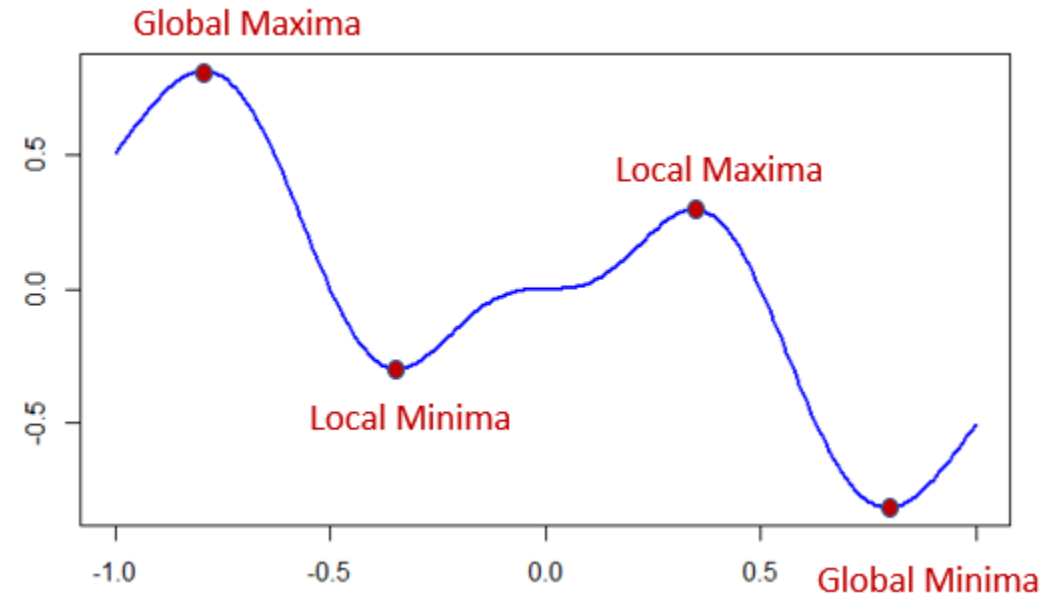
- Gradient descent is an optimization algorithm used to minimize a loss function by iteratively adjusting the model's parameters (weights and biases) in the direction of the steepest descent.
- It is commonly used in machine learning and deep learning to find the optimal parameters (like weights in a neural network) that minimize the error (or loss).
- It helps to find the best-fit line (plane/hyperplane) for a given training dataset in a smaller number of iterations.
- Consider, the case of one dependent variable and one independent variable
 - Then, our objective is to find the equation of a best fit line ($y_{\text{pred}} = mx + c$)
 - If we plot m and c against MSE, it will acquire a bowl shape
 - For some combination of m and c , we will get the least Error (MSE).
 - That combination of m and c will give us our best fit line.



Understanding Gradient Descent

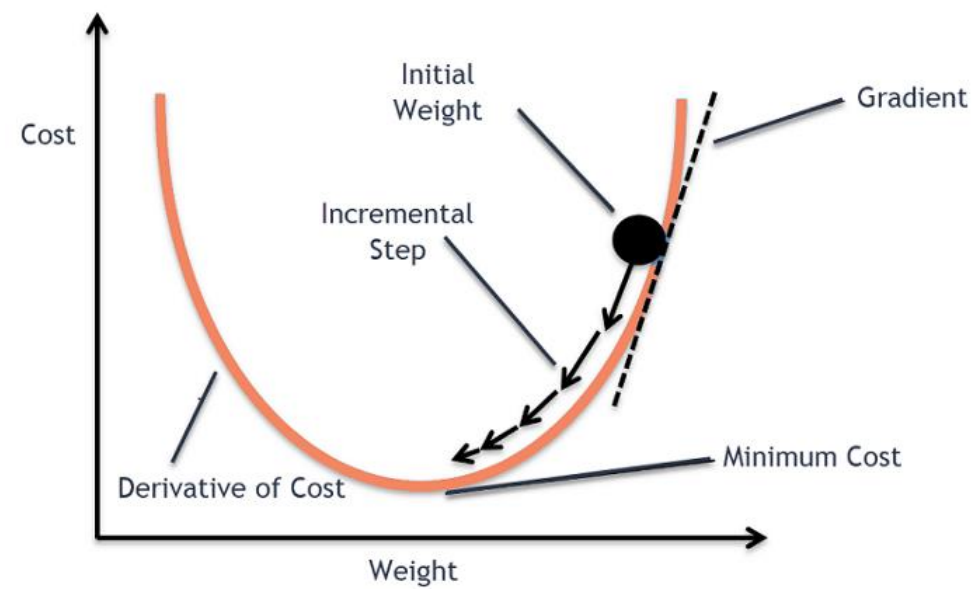
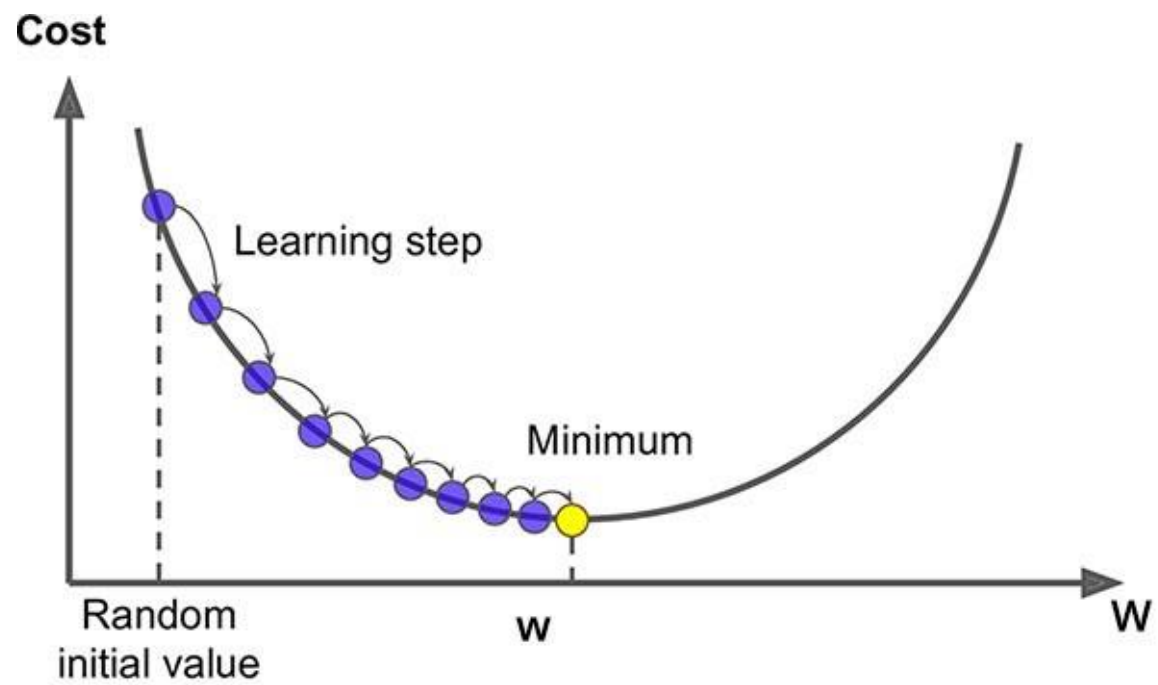
An analogy for understanding gradient descent

- The basic intuition behind gradient descent can be illustrated by a hypothetical scenario. A person is stuck in the mountains and is trying to get down (i.e., trying to find the global minimum). There is heavy fog such that visibility is extremely low. Therefore, the path down the mountain is not visible, so they must use local information to find the minimum. They can use the method of gradient descent, which involves looking at the steepness of the hill at their current position, then proceeding in the direction with the steepest descent (i.e., downhill). If they were trying to find the top of the mountain (i.e., the maximum), then they would proceed in the direction of steepest ascent (i.e., uphill).
- Using this method, they would eventually find their way down the mountain or possibly get stuck in some hole (i.e., local minimum or [saddle point](#)), like a mountain lake. However, assume also that the steepness of the hill is not immediately obvious with simple observation, but rather it requires a sophisticated instrument to measure, which the person happens to have at the moment. It takes quite some time to measure the steepness of the hill with the instrument, thus they should minimize their use of the instrument if they wanted to get down the mountain before sunset. The difficulty then is choosing the frequency at which they should measure the steepness of the hill so not to go off track.



- In this analogy, the person represents the algorithm, and the path taken down the mountain represents the sequence of parameter settings that the algorithm will explore. The steepness of the hill represents the slope of the error surface at that point. The instrument used to measure steepness is differentiation (the slope of the error surface can be calculated by taking the derivative of the squared error function at that point). The direction they choose to travel in aligns with the gradient of the error surface at that point. The amount of time they travel before taking another measurement is the step size.

- In summary, Gradient Descent is a first-order iterative optimization algorithm for finding the local minimum of a differentiable function. To get the value of the parameter that will minimize our objective function we iteratively move in the opposite direction of the gradient of that function or in simple terms in each iteration we move a step in direction of steepest descent. The size of each step is determined by a parameter which is called Learning Rate. Gradient Descent is the first-order algorithm because it uses the first-order derivative of the loss function to find minima. Gradient Descent works in space of any number of dimensions.



- The algorithm starts with some random value of m and c .
- We calculate MSE (cost) at point $m=0$, $c=0$. Let say the MSE (cost) at $m=0$, $c=0$ is 100.
- Then we reduce the value of m and c by some amount (Learning Step).
- We will notice a decrease in MSE (cost). We will continue doing the same until our loss function is a very small value or ideally 0 (which means 0 error or 100% accuracy).

Cost Function(Remember...)

- The cost is the error in our predicted value. We will use the Mean Squared Error function to calculate the cost.

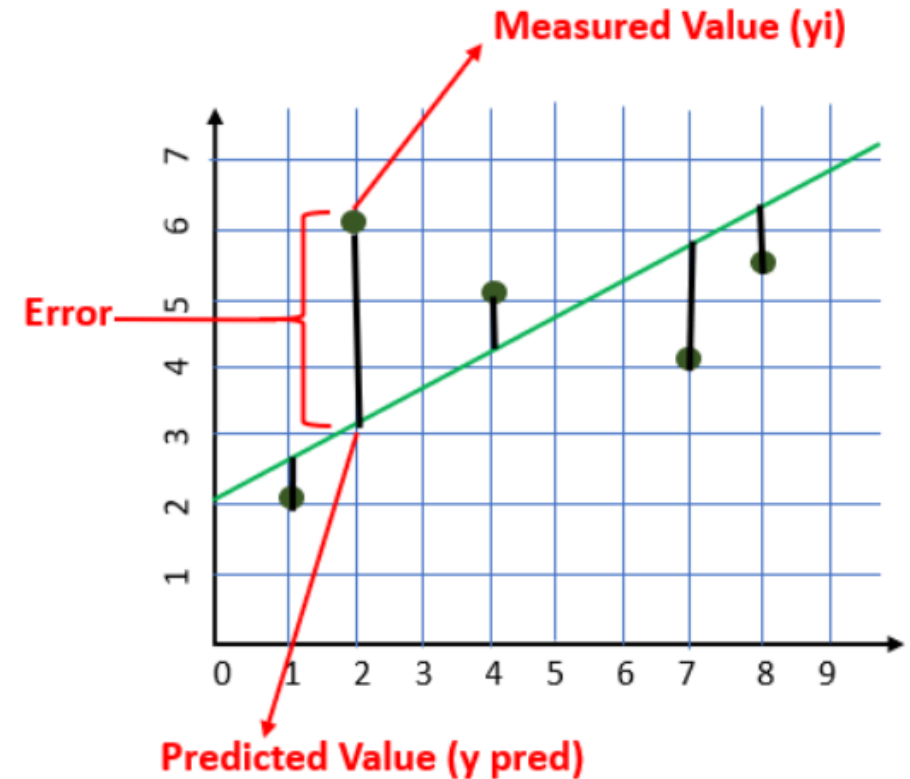
$$\text{Cost Function(MSE)} = \frac{1}{n} \sum_{i=0}^n (y_i - y_{i \text{ pred}})^2$$

Replace $y_{i \text{ pred}}$ with $mx_i + c$

$$\text{Cost Function(MSE)} = \frac{1}{n} \sum_{i=0}^n (y_i - (mx_i + c))^2$$

Our goal is to minimize the cost as much as possible in order to find the best fit line.

Try all the permutation and combination of m and c (inefficient way) to find the best-fit line?.



Step by Step Algorithm:

1. Let m and c be initialized with some random values. Let L be our learning rate. It could be a small value like 0.01 for good accuracy.

Learning rate gives the rate of speed where the gradient moves during gradient descent. Setting it too high would make your path instable, too low would make convergence slow. Put it to zero means your model isn't learning anything from the gradients.

2. Calculate the partial derivative of the Cost function with respect to m . Let partial derivative of the Cost function with respect to m be D_m (With little change in m how much Cost function changes).

$$\begin{aligned} D_m &= \frac{\partial(\text{Cost Function})}{\partial m} = \frac{\partial}{\partial m} \left(\frac{1}{n} \sum_{i=0}^n (y_i - y_{i \text{ pred}})^2 \right) \\ &= \frac{1}{n} \frac{\partial}{\partial m} \left(\sum_{i=0}^n (y_i - (mx_i + c))^2 \right) \\ &= \frac{1}{n} \frac{\partial}{\partial m} \left(\sum_{i=0}^n (y_i^2 + m^2 x_i^2 + c^2 + 2mx_i c - 2y_i m x_i - 2y_i c) \right) \\ &= \frac{-2}{n} \sum_{i=0}^n x_i (y_i - (mx_i + c)) \\ &= \frac{-2}{n} \sum_{i=0}^n x_i (y_i - y_{i \text{ pred}}) \end{aligned}$$

Similarly, find the partial derivative with respect to c. Let partial derivative of the Cost function with respect to c be D_c (With little change in c how much Cost function changes).

$$\begin{aligned} D_c &= \frac{\partial(\text{Cost Function})}{\partial c} = \frac{\partial}{\partial c} \left(\frac{1}{n} \sum_{i=0}^n (y_i - y_{i \text{ pred}})^2 \right) \\ &= \frac{1}{n} \frac{\partial}{\partial c} \left(\sum_{i=0}^n (y_i - (mx_i + c))^2 \right) \\ &= \frac{1}{n} \frac{\partial}{\partial c} \left(\sum_{i=0}^n (y_i^2 + m^2 x_i^2 + c^2 + 2mx_i c - 2y_i mx_i - 2y_i c) \right) \\ &= \frac{-2}{n} \sum_{i=0}^n (y_i - (mx_i + c)) \\ &= \frac{-2}{n} \sum_{i=0}^n (y_i - y_{i \text{ pred}}) \end{aligned}$$

3. Now update the current values of m and c using the following equation:

$$m = m - LD_m$$

$$c = c - LD_c$$

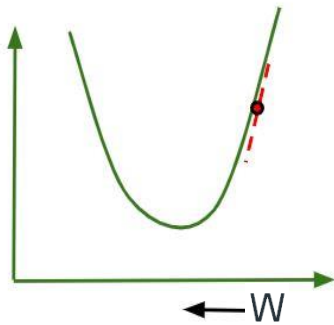
Where, L is the Learning Rate. L is also denoted as α

4. We will repeat this process until our Cost function is very small (ideally 0).

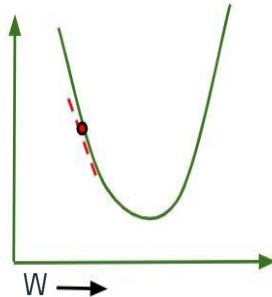
Gradient Descent Algorithm gives optimum values of m and c of the regression equation (in this case). With these values of m and c, we will get the equation of the best-fit line and ready to make predictions.

The effect of Learning rate

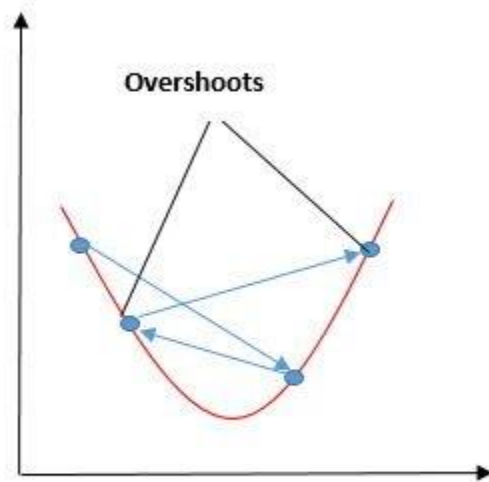
- Gradient Descent step-downs the cost function in the direction of the steepest descent. The size of each step is determined by parameter α or L known as **Learning Rate**.
In the Gradient Descent algorithm, one can infer two points :
- **If slope is +ve** : $W_j = W_j - (+ve \text{ value})$. Hence value of W_j decreases.



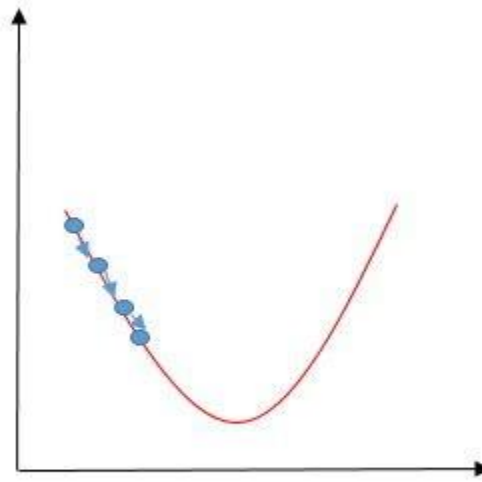
- **If slope is -ve** : $W_j = W_j - (-ve \text{ value})$. Hence value of W_j increases.



- The choice of correct learning rate is very important as it ensures that Gradient Descent converges in a reasonable time. :
- If we choose L or α to be **very large**, Gradient Descent can overshoot the minimum. It may fail to converge or even diverge.
- If we choose α to be very small, Gradient Descent will take small steps to reach local minima and will take a longer time to reach minima.



High learning rate



Low learning rate

Summary: How Gradient Descent Works

- **1. Initialize Parameters**
 - Start with random initial values for the model parameters (weights and biases).
- **2. Calculate Predictions**
 - Use the current model parameters to compute the predictions for the training data. This involves forward propagation in the case of neural networks.
- **3. Compute Loss**
 - The loss function measures how far the model's predictions are from the actual target values. Common loss functions include:
 - **Mean Squared Error (MSE)** for regression problems.
 - **Categorical Cross entropy** for multi-class classification (used in your code).
- **4. Calculate Gradients**
 - Compute the gradient of the loss function with respect to each parameter (partial derivatives). This tells us how the loss changes as each parameter changes.
- **5. Update Parameters**
 - Update each parameter using the gradient:
 - $W_i := W_i + \Delta w_i$, where:
 - w_i : Current parameter (weight or bias).
 - $\Delta W_i = -\alpha \frac{\partial L}{\partial W_i}$, where L is the loss function.
 - α : Learning rate, a small positive number that controls the step size.
 - The negative sign indicates moving in the opposite direction of the gradient (steepest descent).
- **6. Repeat Until Convergence**
 - Repeat steps 2–5 for a fixed number of iterations (epochs) or until the loss stops decreasing.

Variants of Gradient Descent

When we carry out forward and back propagation using gradient descent, we can use basically one of three methods:

1. **Stochastic** gradient descent
2. **Mini-batch** gradient descent
3. **Batch** gradient descent

- Consider the dataset:

Training examples

| Student | X1 Science grade [%] | X2 Chemistry grade [%] | X3 # hours studied [hrs] | Y _{act} Final math grade[%] |
|---------|----------------------------|------------------------------|--------------------------------|--|
| 1 | 60 | 80 | 5 | 82 |
| 2 | 70 | 75 | 7 | 94 |
| 3 | 50 | 55 | 10 | 45 |
| 4 | 40 | 56 | 7 | 43 |

- **1. Stochastic Gradient Descend**

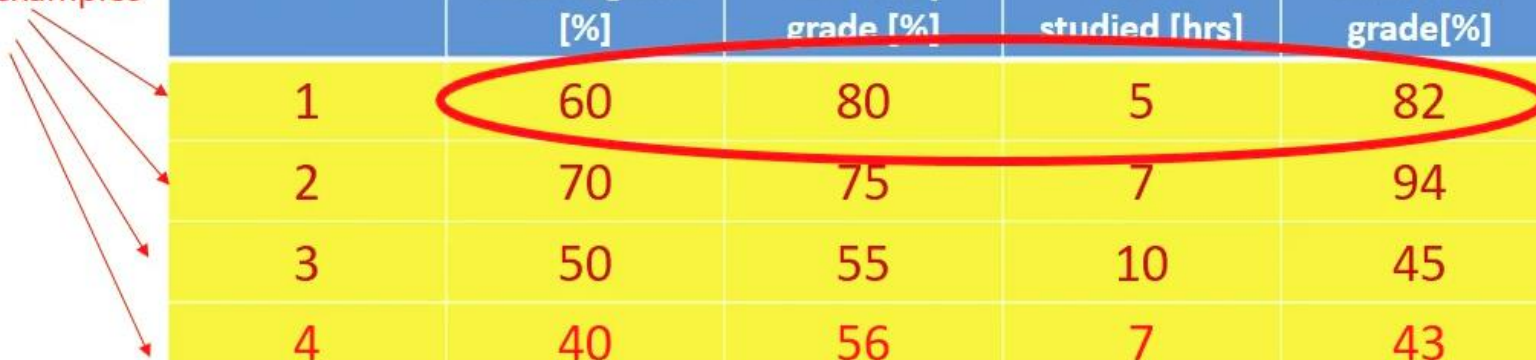
If we input **one training example** at a time

And carry out forward and back propagation to update the weights

That is called **stochastic gradient descent**

Training examples

| Student | X1 Science grade [%] | X2 Chemistry grade [%] | X3 # hours studied [hrs] | Y_{act} Final math grade[%] |
|---------|----------------------------|------------------------------|--------------------------------|-------------------------------------|
| 1 | 60 | 80 | 5 | 82 |
| 2 | 70 | 75 | 7 | 94 |
| 3 | 50 | 55 | 10 | 45 |
| 4 | 40 | 56 | 7 | 43 |

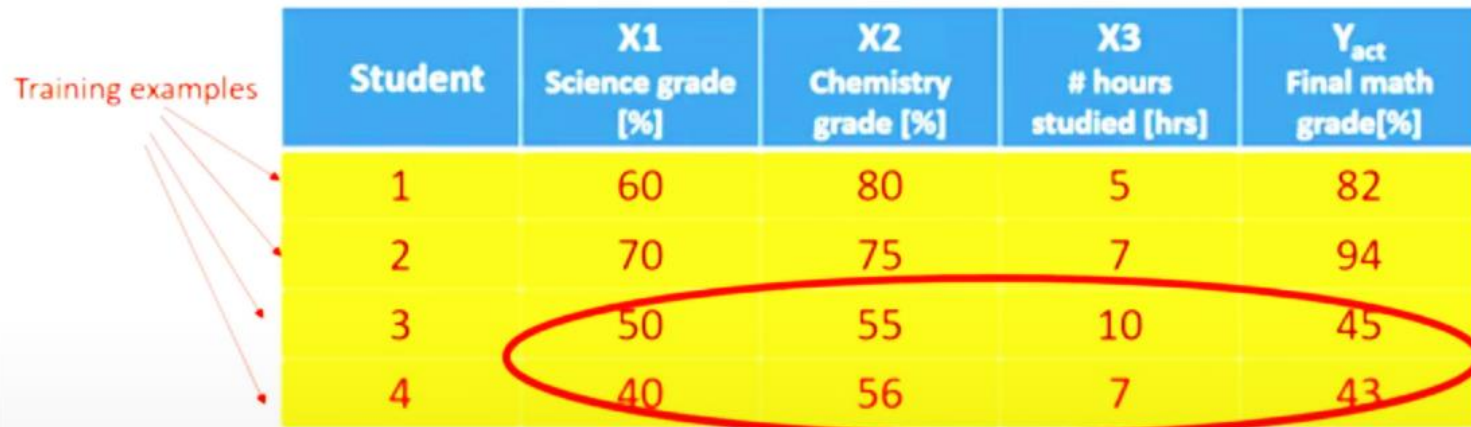


- **Stochastic gradient descent (SGD)** changes the parameters for each training sample one at a time for each training example in the dataset.
- Depending on the task, this can make SGD faster than batch gradient descent.
- One benefit is that the regular updates give us a fairly accurate idea of the rate of improvement.
- However, the frequent updates can also produce noisy gradients, which could cause the error rate to fluctuate rather than gradually go down.
- Computationally expensive

2. MINI-BATCH GRADIENT DESCENT:

If we input multiple training samples at a time, (but less than the entire training dataset) it is called mini batch gradient descent

Mini-batch



Training examples

| Student | X1 Science grade [%] | X2 Chemistry grade [%] | X3 # hours studied [hrs] | Y _{act} Final math grade[%] |
|---------|----------------------------|------------------------------|--------------------------------|--|
| 1 | 60 | 80 | 5 | 82 |
| 2 | 70 | 75 | 7 | 94 |
| 3 | 50 | 55 | 10 | 45 |
| 4 | 40 | 56 | 7 | 43 |

The diagram shows a table with 5 columns: Student, X1 Science grade [%], X2 Chemistry grade [%], X3 # hours studied [hrs], and Y_{act} Final math grade[%]. The table contains 4 rows of data. A red oval highlights the last two rows (Student 3 and 4), indicating a mini-batch. Red arrows point from the text 'Training examples' to the first three rows of the table.

- 2. MINI-BATCH GRADIENT DESCENT:

- It divides the training dataset into manageable groups and updates each separately. This strikes a balance between batch gradient descent's effectiveness and stochastic gradient descent's durability.
- Since mini-batch gradient descent combines the ideas of batch gradient descent with SGD, it is the preferred technique.
 - Mini-batch sizes typically range from 50 to 256, although, like with other machine learning techniques, there is no set standard because it depends on the application. The most popular kind in deep learning, this method is used when training a neural network.

3. BATCH GRADIENT DESCENT

If the entire training samples are considered at a time, it is called mini batch gradient descent

Entire Dataset - Batch

Training examples

| Student | X1 Science grade [%] | X2 Chemistry grade [%] | X3 # hours studied [hrs] | Y _{act} Final math grade[%] |
|---------|----------------------------|------------------------------|--------------------------------|--|
| 1 | 60 | 80 | 5 | 82 |
| 2 | 70 | 75 | 7 | 94 |
| 3 | 50 | 55 | 10 | 45 |
| 4 | 40 | 56 | 7 | 43 |

- **BATCH GRADIENT DESCENT:**

- Batch gradient descent, also known as vanilla gradient descent, calculates the error for each example within the training dataset.
- However, the model is not changed (ie., the weights are not updated) until every training sample has been assessed.
- The entire procedure is referred to as a cycle and a training epoch.
- Benefits:
 - Produces a stable error gradient and a stable convergence.
- Drawbacks
 - The stable error gradient can sometimes result in a state of convergence that isn't the best the model can achieve.
 - It also requires the entire training dataset to be in memory and available to the algorithm.
 - Computationally expensive, particularly if dataset is very large

| Variant | Speed | Stability | Recommended For |
|---------------|----------|-------------|-------------------------------------|
| Batch GD | Slow | Very stable | Small datasets |
| SGD | Fast | Unstable | Large datasets, online learning |
| Mini-Batch GD | Moderate | Stable | General-purpose |
| Momentum | Fast | Stable | Deep valleys or oscillating regions |
| NAG | Faster | Stable | Faster convergence |
| Adagrad | Moderate | Stable | Sparse data |
| RMSProp | Fast | Very stable | Non-stationary data |
| Adam | Fast | Very stable | General-purpose deep learning |

Vanishing and Exploding Gradient Problem

- The **vanishing gradient problem** is a common issue in deep neural networks, particularly when using activation functions like sigmoid or tanh.
- It occurs during backpropagation when gradients of the loss function with respect to earlier layers become very small, leading to slow or no learning in those layers.

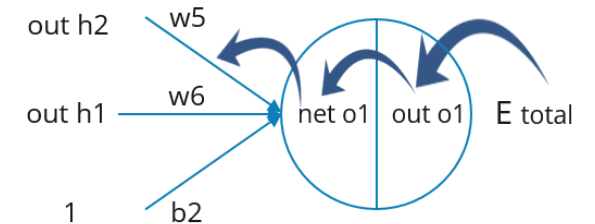
Backpropagation updates the weights using **gradient descent**, where the weight update for layer l is:

$$W^{(l)} = W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}}$$

where:

- $W^{(l)}$ are the weights at layer l ,
- η is the learning rate,
- L is the loss function.

$$\frac{\delta E_{total}}{\delta w_5} = \frac{\delta E_{total}}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w_5}$$



The gradient of the loss with respect to weights is computed using the **chain rule**:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial a^{(L)}} \cdot \prod_{k=l}^L \frac{\partial a^{(k)}}{\partial a^{(k-1)}} \cdot \frac{\partial a^{(l)}}{\partial W^{(l)}}$$

where:

- $a^{(k)}$ is the activation at layer k ,
- L is the number of layers.

If the activation function has a small derivative (like sigmoid or tanh), then:

$$\frac{\partial a^{(k)}}{\partial a^{(k-1)}} \approx \text{small value}$$

For a deep network, this multiplication happens many times, leading to:

$$\prod_{k=l}^L \frac{\partial a^{(k)}}{\partial a^{(k-1)}} \rightarrow 0$$

which results in **very small gradients**. When gradients vanish, weight updates become negligible, preventing earlier layers from learning effectively.

Example with Sigmoid Activation

The **sigmoid function** is:

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

Its derivative:

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

For large or small x , $\sigma'(x)$ becomes very small (< 0.25), causing gradients to vanish when propagated back through multiple layers.

- Imagine you have the following:
 - Output Layer Gradient: 0.1
 - Gradient for Layer 2: $0.1 \times 0.1 = 0.01$
 - Gradient for Layer 1: $0.01 \times 0.1 = 0.001$
- By the time the gradients reach the first few layers, they are so small (close to zero) that these layers do not get significant updates, making it impossible for them to learn effectively. This leads to a network that **learns slowly** or becomes unable to capture important features in the input, especially in early layers.

- **Exploding gradient problem:** In the exploding gradient problem, the gradients become too large as they propagate backward through the network, causing huge updates to the weights. This results in unstable training, with the loss function either oscillating wildly or becoming too large to compute, causing the network to diverge.
- Example: Let's assume you are training a simple feedforward neural network with several layers. Assume that we have gradients larger than 1, such as using the ReLU activation function in poorly initialized networks:
 - **During backpropagation, gradients are multiplied by numbers greater than 1.**
 - **As these values are multiplied through the layers, the gradients become exponentially larger.**
 - **Imagine you have the following:**
 - Output Layer Gradient: 5
 - Gradient for Layer 2: $5 \times 5 = 25$
 - Gradient for Layer 1: $25 \times 5 = 125$

- As gradients explode, the weight updates become extremely large. This leads to:
 - **Instability in learning:** The weights change drastically from one iteration to the next, preventing the model from converging to a solution.
 - **Oscillating or Diverging Loss:** The loss function might increase instead of decreasing, or oscillate wildly, making it impossible to achieve good performance.

Epoch

- An **epoch** in machine learning and deep learning refers to **one complete pass through the entire training dataset** by the learning algorithm.
 - In our example, in stochastic gradient descent, we would have carried out 4 passes before completing an epoch
 - In mini batch gradient descent (with n batches), we would have carried out n passes before completing an epoch. In the example $n=2$
 - In batch gradient descent, we carry out 1 pass for completing an epoch
- Note: It will take many epochs for the model to be fully trained