Deep Learning
CSC-Elective

Instructor : Dr. Muhammad Ismail Mangrio

Slides credit Dr. M Asif Khan

ismail@iba-suk.edu.pk

Week 4-5

Contents

Loss functions

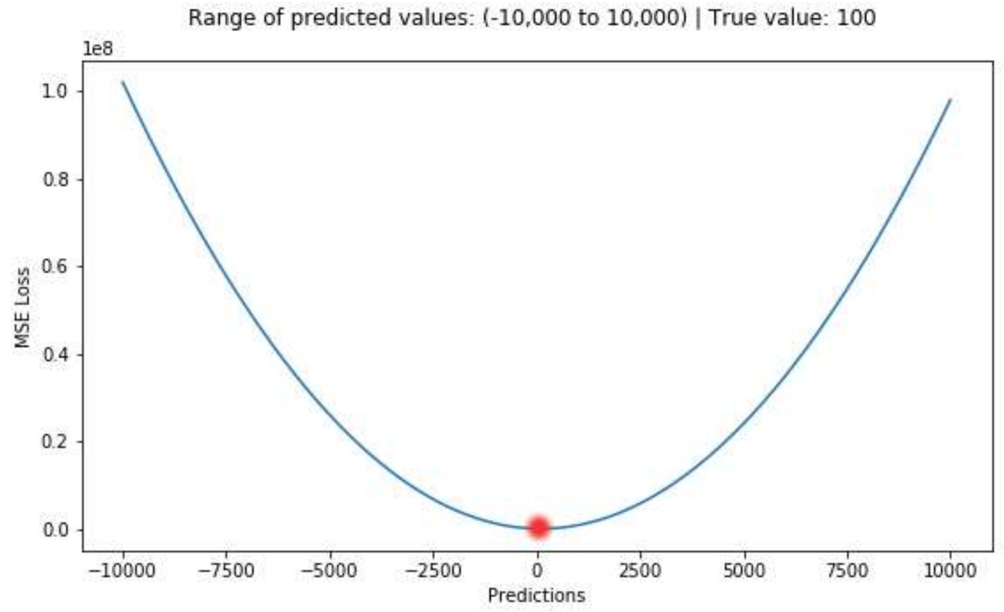
- The loss function is an expression used to measure **how close the predicted value is to the actual value**.
- This expression outputs a value called loss, which tells us the performance of our model.
- By **reducing this loss value** in further **training**, the model can be **optimized** to output values that are **closer to the actual values**.
- There are **two classes** of Loss functions based on the type of learning task
  - **Loss functions for Regression models**: predict continuous values.
  - **Loss functions for Classification models**: predict the output from a set of finite categorical values.

Loss functions (for Regression models)

- In Regression we predict a continuous value.
- Following are the common loss functions for Regression models:
  - **MSE** (Mean Squared Error)
  - **MAE** (Mean Absolute Error)
  - **Huber** Loss function

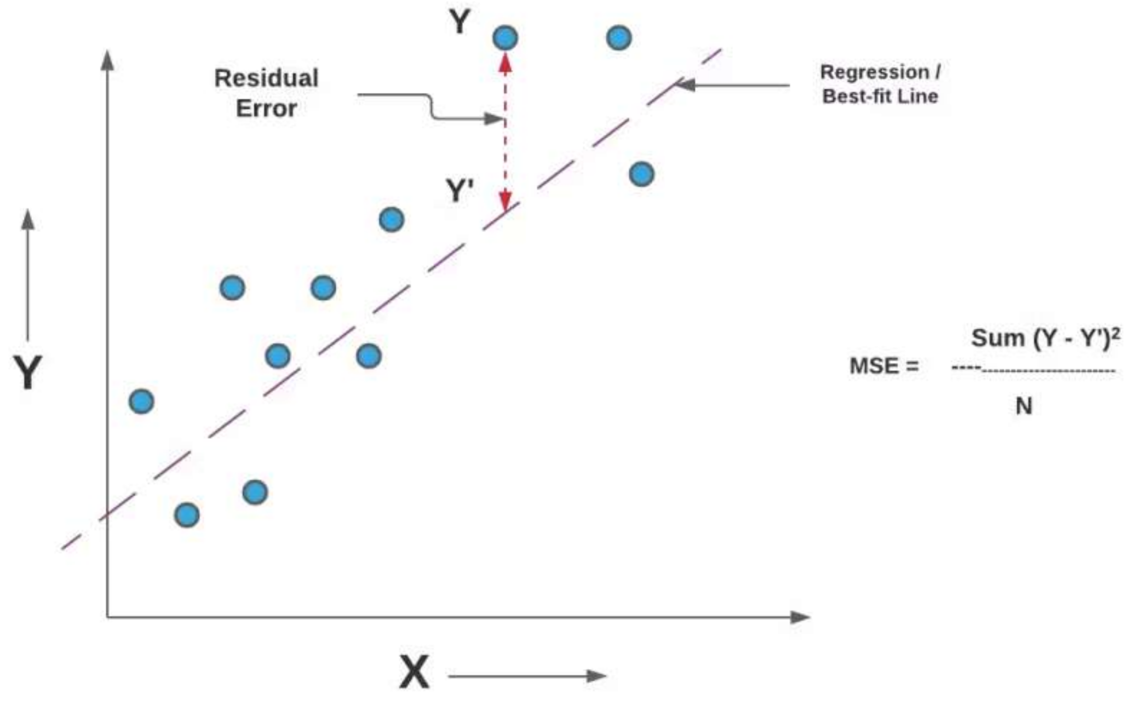
Loss functions (for Regression models, MSE)

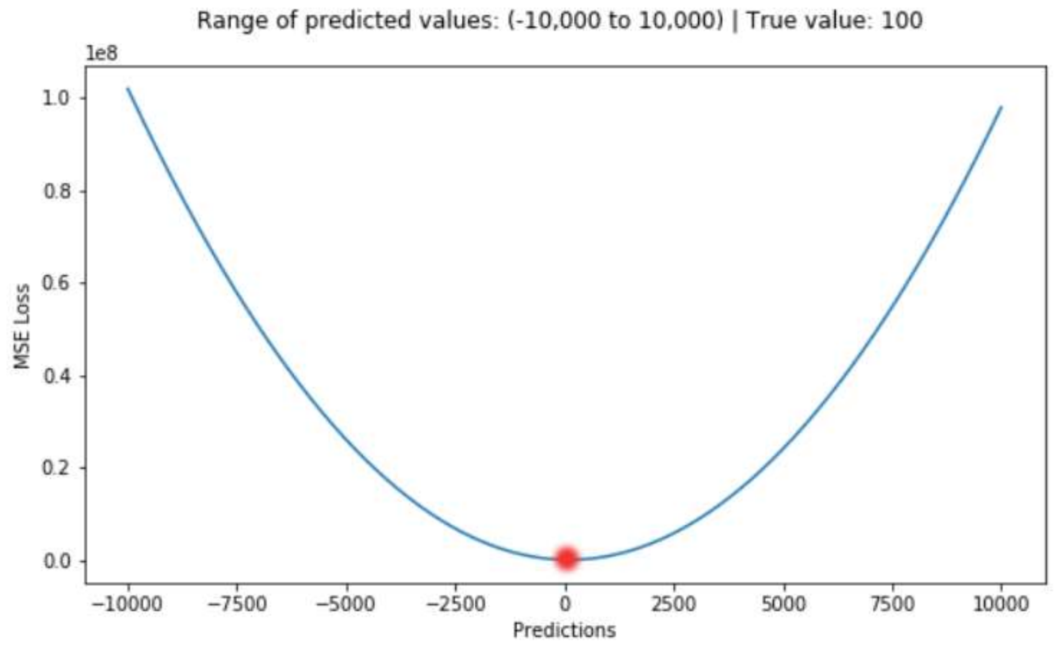
- It is the **Mean of Square of Residuals** for all the data points in the dataset.
- **Residuals is** the difference b/w the actual and the predicted prediction by the model.
- **Squaring of residuals** is done to convert negative values to positive values.
- Squaring also gives **more weightage to larger errors**.
- When the cost function is far away from its minimal value, squaring the error will **penalize the model**
- MSE is **sensitive to outliers**.
- It is also called Quaratic Loss or L2 Loss.
- Good for gradient descent and differentiable



$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Loss functions (for Regression models, MSE)



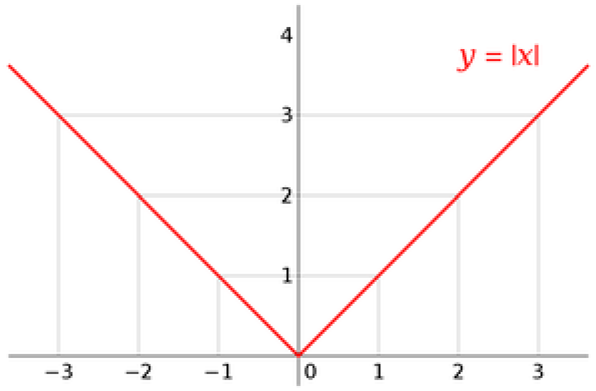


$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Loss functions (for Regression models, MAE)

- It is the **Mean of Absolute of Residuals** for all the datapoints in the dataset.
- The **absolute of residuals** is done to convert negative values to positive values.
- Mean is taken to make the loss function independent of number of datapoints in the training set.
- One advantage of MAE is that is **robust to outliers**, as it is not squaring error.
- MAE is generally less preferred over MSE as it is **harder to calculate the derivative of the absolute function** because absolute function is not differentiable at the minima.
- It is also called L1 loss.

$$MAE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}$$



Loss functions (for Regression models, Huber)

- It is the **combination of MSE and MAE**.
- It **takes the good properties of both** loss functions by being less sensitive to outliers and differentiable at minima.
- When the **error is smaller**, the **MSE part** of the Huber is utilized and when the **error is large**, the **MAE part** of Huber loss is used.
- A new hyper-parameter **'δ'** is introduced which **tells** the loss function **where to switch** from MSE to MAE.
- **Additional 'δ'** terms are introduced in the loss function **to smoothen** the transition from MSE to MAE.

$$\text{Loss} = \begin{cases} \frac{1}{2} * (x-y)^2 & \text{if } (|x-y| \leq \delta) \\ \delta * |x-y| - \frac{1}{2} * \delta^2 & \text{otherwise} \end{cases}$$

Loss functions (for Classification models)

- The **Cross Entropy Loss** function is a commonly used loss function for classification problems, both in **binary and multi-class** settings.
- Measures the **difference b/w two probability distributions**: the predicted probability distribution and the true probability distribution (or one-hot encoded labels).
- **Binary Cross Entropy (BCE)** loss function
  - Used for binary classification problems
- **Categorical Cross Entry (CCE)** loss function
  - Used for multi classification problems, used with one-hot encoded vectors
- **Sparse Categorical Cross-Entropy (SCCE)** loss function:
  - Similar to categorical cross-entropy but used when the class labels are integers instead of one-hot encoded vectors.

Loss functions (for Classification models, **BCE**)

- Used for **binary classification tasks**, where each instance is classified into one of two classes (e.g., 0 or 1).
- For a single instance, if the true label is y and the predicted probability of the positive class is p, the **Binary Cross Entropy Loss is**:

$$\mathbf{BCE} = -[\mathbf{y}\,\mathbf{log}(\boldsymbol{p}) + (\mathbf{1} - \boldsymbol{y})\,\mathbf{log}(\mathbf{1} - \mathbf{p})]$$

- y: True label (0 or 1).
- p: Predicted probability of the positive class ($0 \leq p \leq 1$).
- **Example**: Suppose you have: True label (y) = 1
- Predicted probability (p) = 0.8
- BCE = −[1 log(0.8) + (1−1) log(1−0.8)] = −log(0.8) ≈ 0.223

Loss functions (for Classification models, CCE)

- Used **for multi-class classification** tasks, where each instance can be classified into one of multiple classes.
- **Formula**:For a single instance, if the true label is a **one-hot encoded vector** $\boldsymbol{y}$ (with a 1 in the position of the true class and 0 elsewhere), and the predicted probability distribution is p, the Categorical Cross Entropy Loss is:

$$\boldsymbol{CCE = -i \sum y_i \, log(pi)}$$

- y: True label as a one-hot encoded vector.
- p: Predicted probability distribution.

Loss functions (for Classification models, CCE)

$$\boldsymbol{CCE = -i \sum y_i\, log(pi)}$$

- y: True label as a one-hot encoded vector.
- p: Predicted probability distribution.
- **Example:** Suppose you have a classification problem with 3 classes, and for a given instance:
- True label is class 2 (**one-hot encoded as [0, 1, 0]**).
- Predicted probabilities are [0.1, 0.7,0.2].
- CCE = −[0log(0.1)+1log(0.7)+0log(0.2)] = −log(0.7) ≈ 0.357

Loss functions (for Classification models, SCCE)

- **Purpose**: Used for multi-class classification tasks.
- Application: Suitable when **target labels are integers** (**not one-hot encoded**) (e.g., 0, 1, 2 for 3 classes).
- Model Output:
  - Produces a probability distribution over all classes.
  - Example: [0.7 (Cat), 0.2 (Dog), 0.1 (Rabbit)]
- True Label:
  - Represented as an integer.
  - For example, 0 for Cat.
- Loss Calculation:
  - Formula: $\mathbf{-\log(\mathit{ptrue})}$
  - Where $ptrue$ is the predicted probability for the true class.

Example:True label: 0 (Cat)

Predicted probabilities: [0.7, 0.2, 0.1]

Loss = $-\log(0.7) \approx 0.357$

Class Performance test

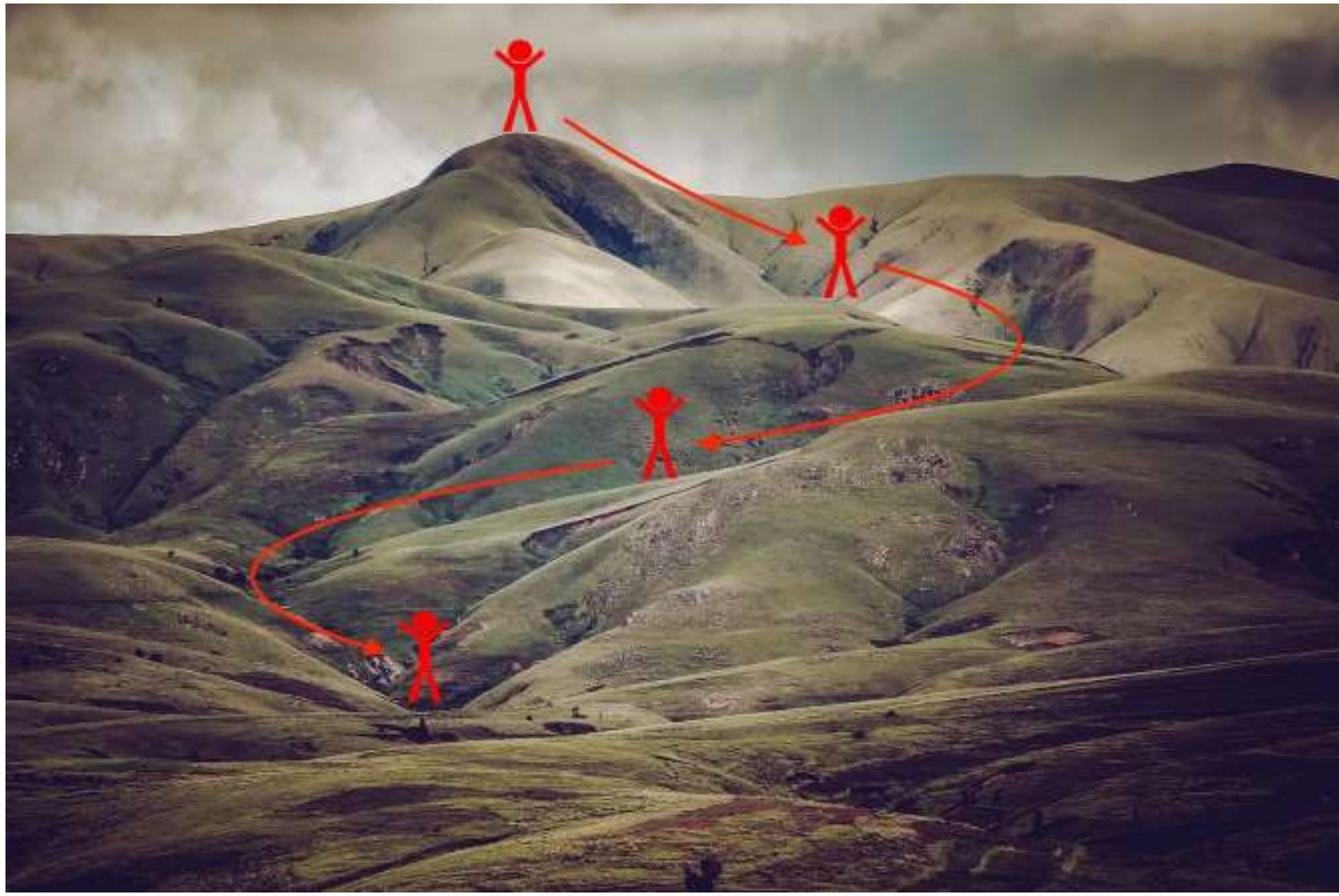
- Answer (type of problem, loss functions name optimizer) for following ANN:
- Input - > ReLU - > softmax()
- Input - > PReLU - > softmax()
- Input - > ReLU - > sigmoid()
- Input - > ReLU - > linear()
- You have 5 minutes to answer

Optimizers

- Optimizers in deep learning are algorithms **used to adjust the parameters (weights and biases)** of a neural network in order **to minimize the loss function** during training.
- Play a crucial role in the training process by determining how the model's parameters are **updated in response to the gradients computed** during **back propagation**.
- Most Common optimizers are:
  - **Gradient Descent**
  - **SGD (Stochastic Gradient Descent)**
  - **Mini-Batch Gradient Descent**
  - **SGD Momentum**
  - **Adagrad**
  - **RMSprop**
  - **Adam (Adaptive Moment Estimation)**

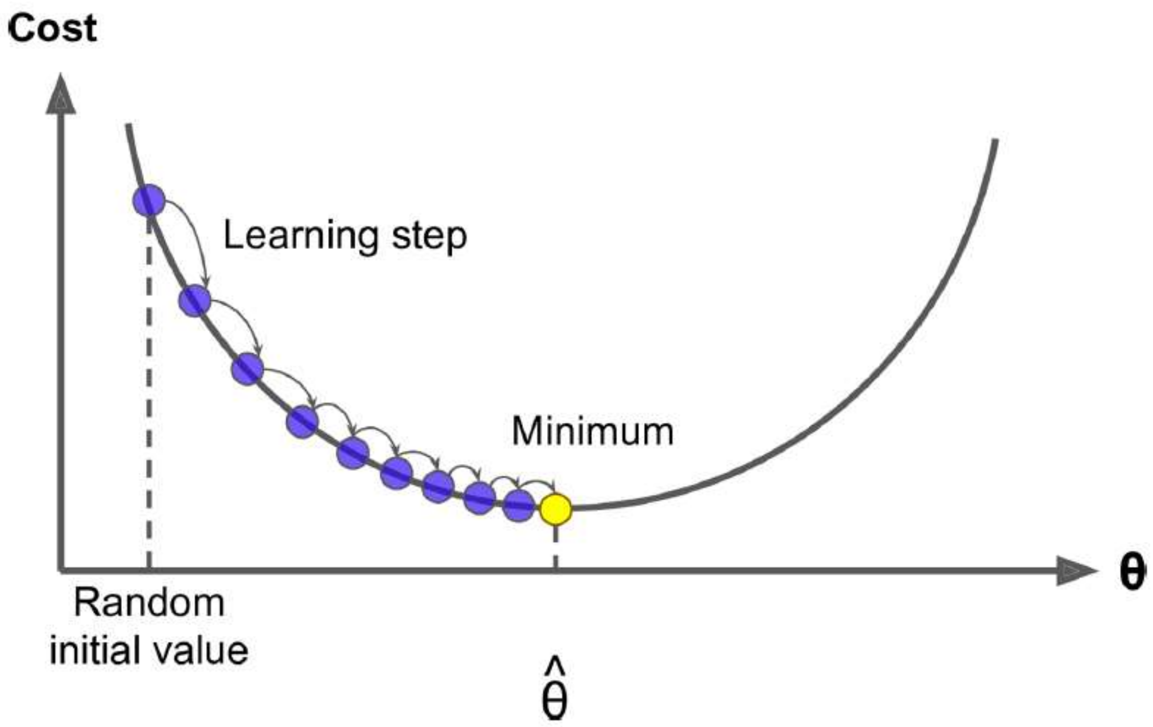
Optimizers (GD-Gradient descent revisited)

- The general idea of Gradient Descent is to **tweak parameters iteratively** in order to **minimize a cost function**.
- Suppose you are **lost in the mountains in a dense fog;** you can only **feel the slope** of the ground below your feet.
- A good strategy to get to the bottom of the valley quickly is to **go downhill in the direction of the steepest slope**. Exactly what Gradient Descent does:



Optimizers (Gradient descent revisited)

- **It measures local gradient of the error function with respect to the parameter vector θ**, and goes in the **direction of descending gradient**.
- Once the **gradient is zero**, you have **reached a minimum!**
- Concretely, **you start by filling θ with random values** (this is called random initialization), and **then you improve it gradually**, taking one baby step at a time, **each step attempting to decrease the cost function** (e.g., the MSE), until the algorithm converges to **a minimum** (see Figure 4-3,see book).



Optimizers (Gradient descent revisited)

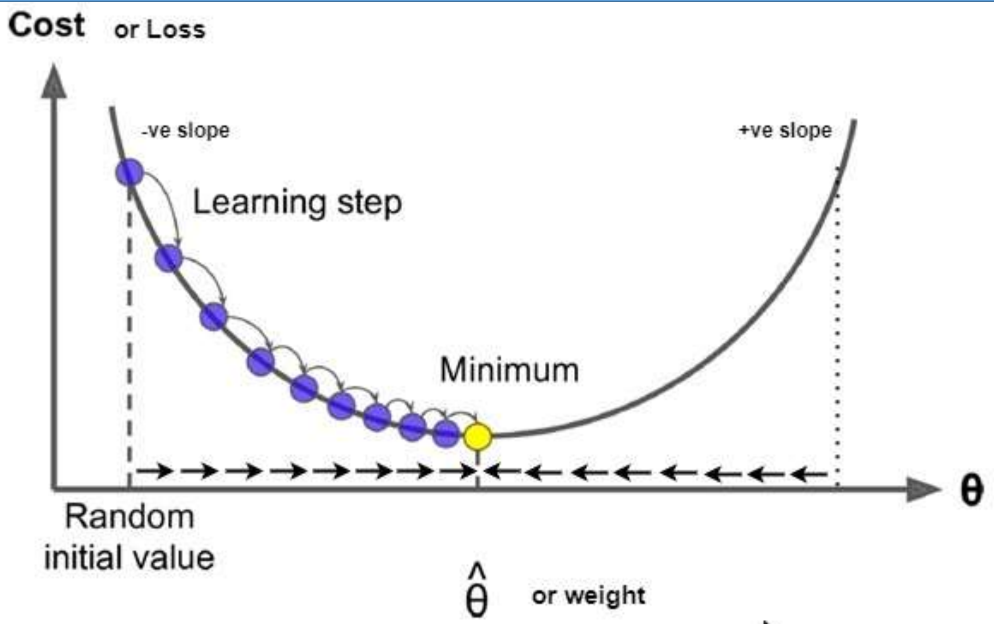
- **Weight update formula**

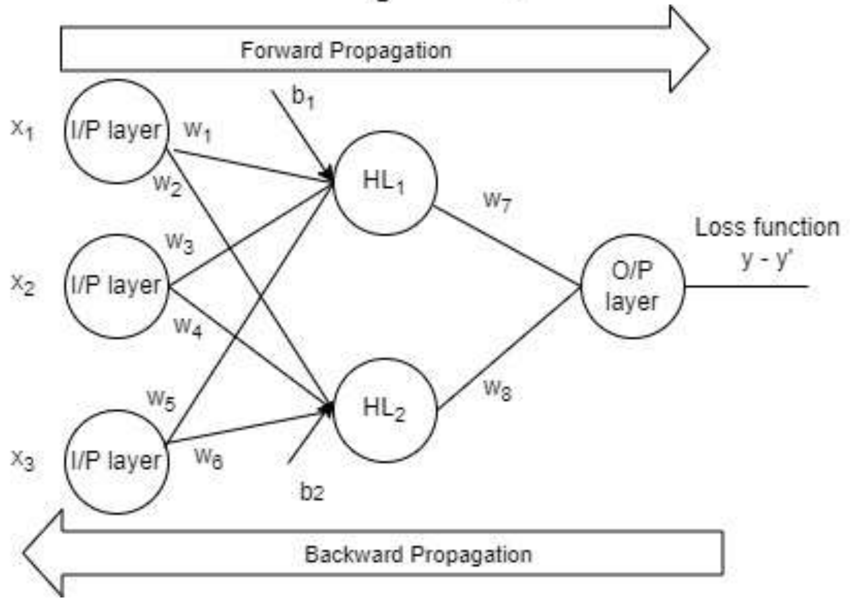
$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

  - Here **η is learning rate**
  - The **partial derivate of loss w.r.t to W_{old}** is the slope of tangent line towards minima
  - For **W_8 to update its weight the formula** would be:

$$W_{8new} = W_{8old} - \eta \frac{\partial L}{\partial W_{8old}}$$

  - The learning rate **η is kept small** to **avoid big jumps**, it is usually **kept 0.001**.





SUKKUR IBA UNIVERSITY COMPUTER SCIENCE

Optimizers (Gradient descent revisited)

- Weight update formula

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

  - From **left side** of the curve the gradient/slope is **–ve**. Therefore:

$$W_{new} = W_{old} - \eta \{-ve\ slope\}$$

$$W_{new} >> W_{old}$$

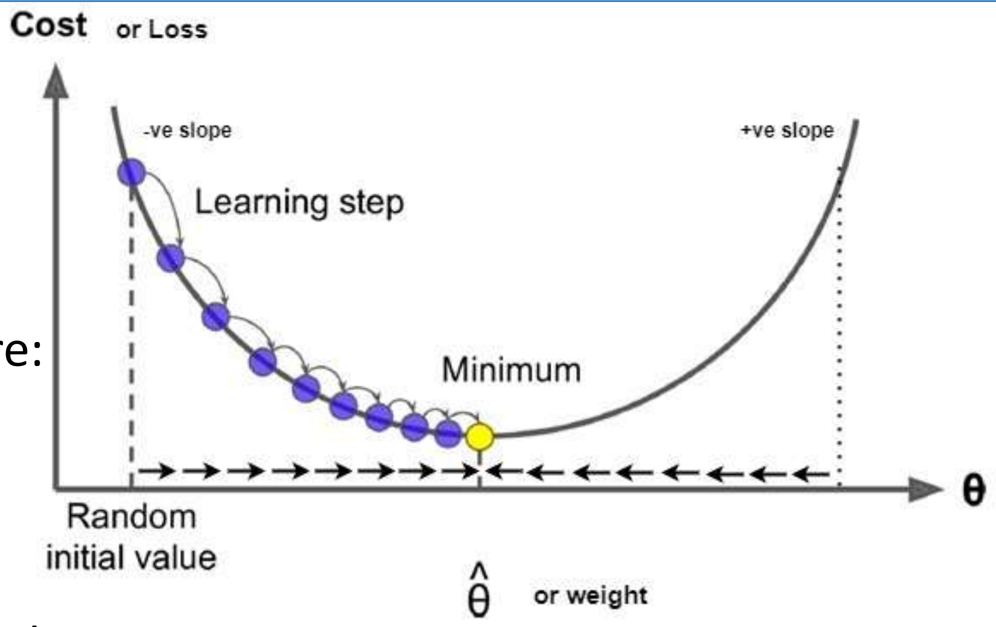
  - On-contrary gradient/slope from the **right side** curve is **+ve** and:

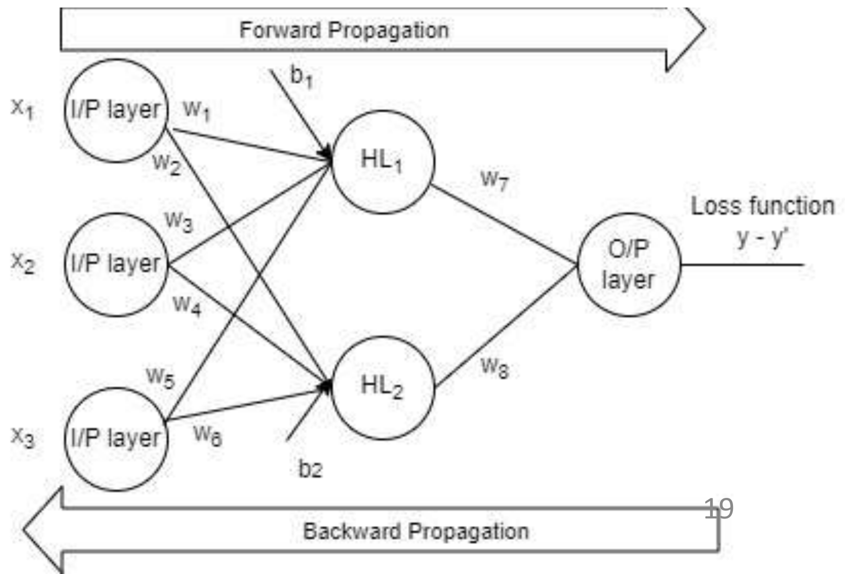
$$W_{new} = W_{old} - \eta \{+ve\ slope\}$$

$$W_{new} << W_{old}$$

  - When updated **W reaches global minima**:

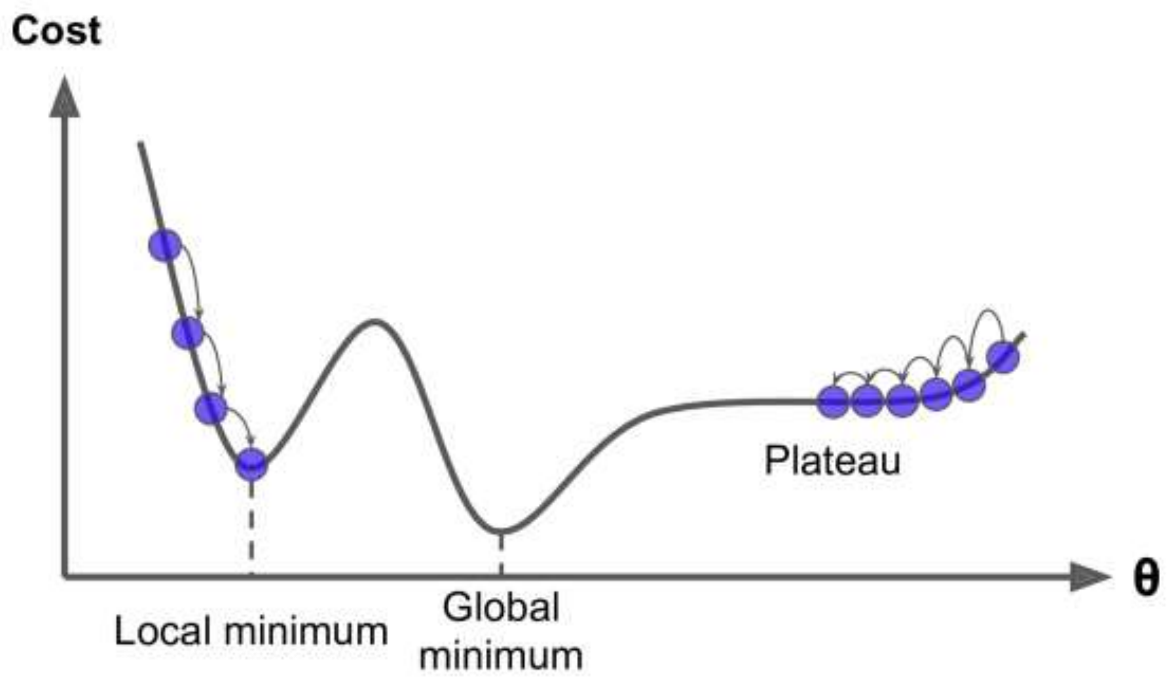
$$W_{new} = W_{old}$$





Optimizers (Gradient descent revisited)

- **Advantage**
  - It will converge
- **Disadvantage**
  - Needs loads of resources (e.g., RAM, CPU power etc).
  - As it **needs to pass entre dataset** a once and then calculate average loss function and average gradient,
  - then back propagation is applied to adjust weights. This process take **huge time**.
  - It can **get stucked in local minima**



Epoch

- An epoch in ANN refers to one **complete pass through the entire training dataset**.
- It is **one full iteration over the entire training dataset**.
- **Iteration**: A single update of the model's weights, based on a batch of training data.
- **Batch**: A subset of the training data.
- Each epoch represents an opportunity for the model to l**earn from the entire dataset and update its parameters** based on the computed gradients.
- Here’s an example:
  - An **epoch in GD means** the model has processed every sample in the training dataset once, and the parameters have been updated based on the aggregate gradients computed from the entire dataset.
- **Training typically involves multiple epochs** to allow the model to learn better from the data

Epoch

- **Example**:Imagine you have a dataset with 100 samples, and you are training a model with a batch size of 10.
- **Training Process for One Epoch**:
  - **Initialization:**
    - Dataset: 100 samples.
    - Batch Size: 10 samples.
  - **Epoch 1:**
    - Iteration 1: Process samples 1 to 10.
    - Iteration 2: Process samples 11 to 20.
    - Iteration 3: Process samples 21 to 30.
    - Iteration 4: Process samples 31 to 40.
    - Iteration 5: Process samples 41 to 50.
    - Iteration 6: Process samples 51 to 60.
    - Iteration 7: Process samples 61 to 70.

Iteration 8: Process samples 71 to 80.
Iteration 9: Process samples 81 to 90.
Iteration 10: Process samples 91 to 100.
After Iteration 10,
all 100 samples have been processed once,
completing Epoch 1.

Epoch (How many Epochs?)

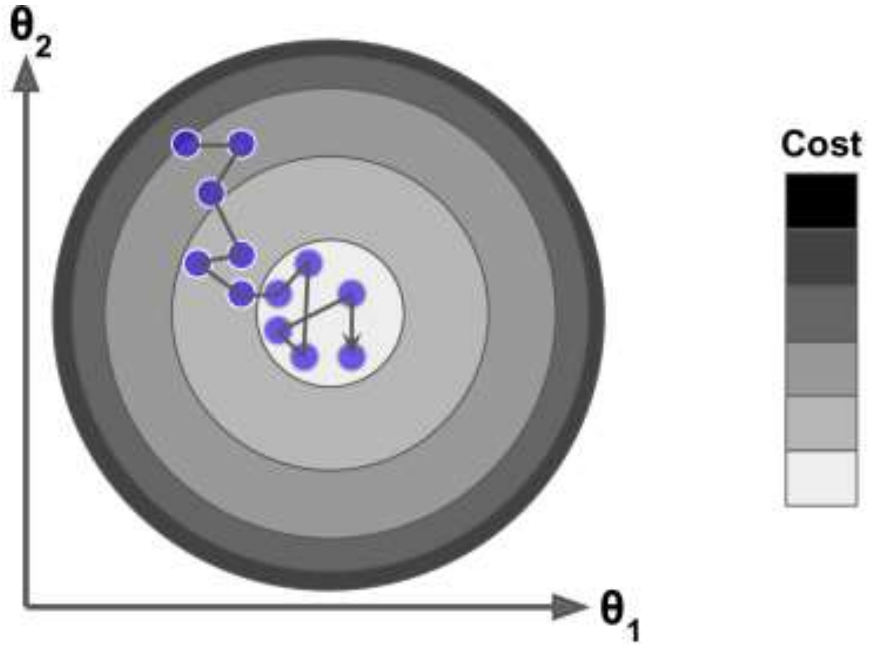
- Why **Multiple Epochs**?
- **Learning:** The model **might not learn enough** from a **single pass.**
- Multiple epochs **help the model learn patterns and reduce error**.
- **Overfitting Risk**: Too many epochs can **lead to overfitting**.
- It's essential to **monitor validation** performance.

Optimizers (SGD)

- It’s a **variation of the Gradient Descent algorithm**.
- In Gradient Descent, we analyze the entire dataset in each step, which may not be efficient when dealing with very large datasets.
- To address **issue in GD**, we have **Stochastic Gradient Descent (SGD)**.
- In SGD, we **process just one example at a time** to perform a single step.
- So, if the dataset contains 10000 rows, SGD will **update the model parameters 10000 times** in a single cycle through the dataset, as opposed to just once in the case of Gradient Descent.

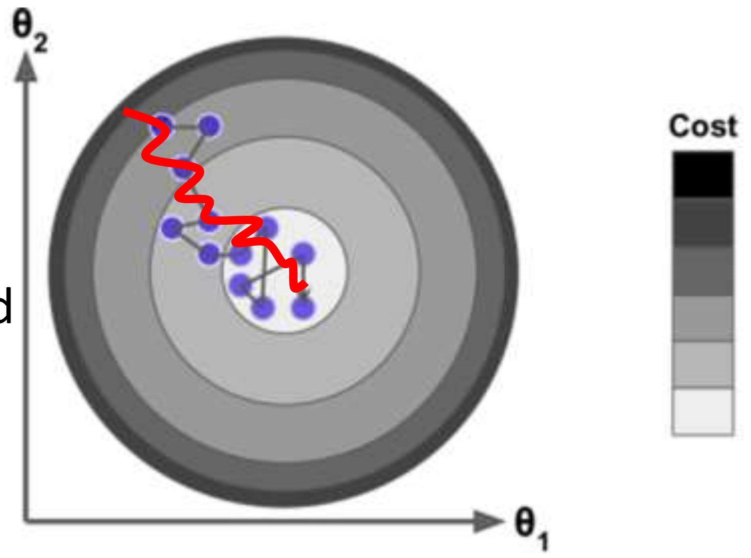
Optimizers (SGD)

- We **don’t need large RAM** for SGD
- Here the **convergence will be slower** as if we had a dataset with 1 million records then weights will be updated 1 million times.
- As result it **takes more time**.
- Also, the **gradient will be noisy**.



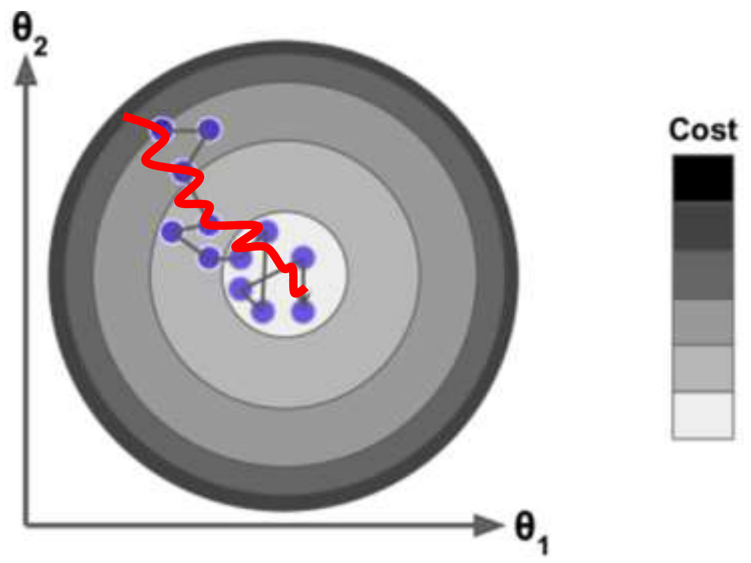
Optimizers (mini-batch SGD)

- Mini-batch stochastic gradient descent, consists of **a predetermined number of training examples**, smaller than the full dataset.
- This approach **combines the advantages GD and SGD**.
- **In one epoch**, following the creation of fixed-size mini-batches with following steps:
  - Select a mini-batch.
  - Compute the mean gradient of the mini-batch.
  - Apply the mean gradient obtained in step 2 to update the model's weights.
  - Repeat steps 1 to 2 for all the mini-batches that have been created



Optimizers (SGD with Momentum)

- In Stochastic Gradient Descent, we don't calculate the precise derivative of our loss function.
- Instead, we estimate it using a small batch. This results in "noisy" derivatives, which implies that we don't always move in the optimal direction.
- To address this issue, **Momentum was introduced to mitigate the noise in SGD.**
- It speeds up convergence towards the relevant direction and diminishes fluctuations in irrelevant directions.
- The concept behind Momentum involves **denoising the derivatives by employing an exponential weighting average** by assigning more weight to recent updates compared to previous ones



Optimizers (SGD with Momentum)

$t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \; ---- \; t_n$

$b_1 \quad \boxed{b_2} \quad b_3 \quad b_4 \quad b_5 \; --- \; b_n$

$t=1 \quad \boxed{V_{t_1} = b_1}$

$0 \leq \gamma \leq 1$

0.5

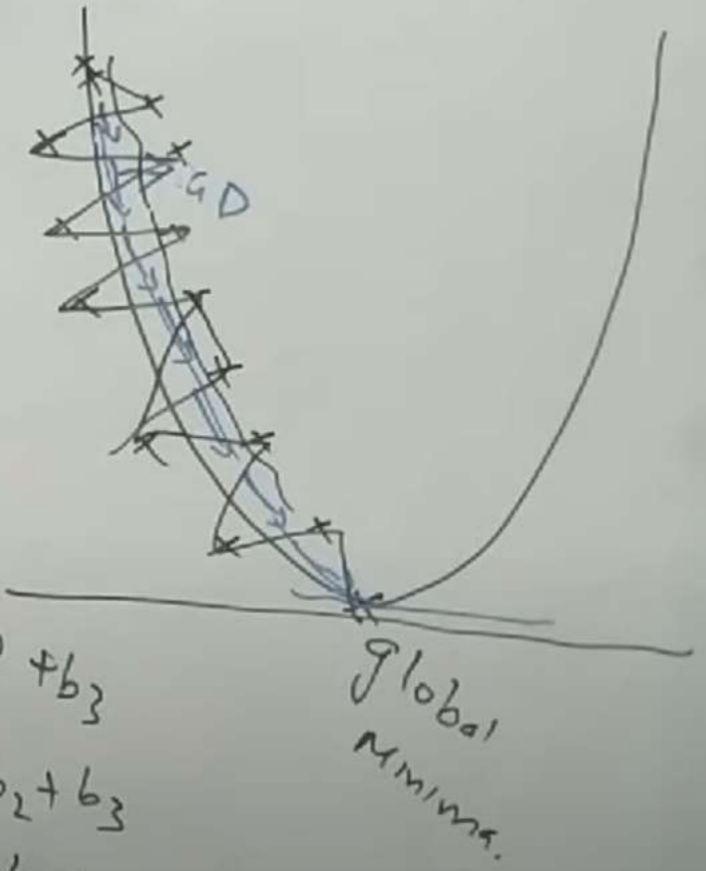
$V_2 = [\gamma \times V_1 + b_2]$

$= 0.5 \times \underline{\underline{b_1}} + \boxed{b_2}$

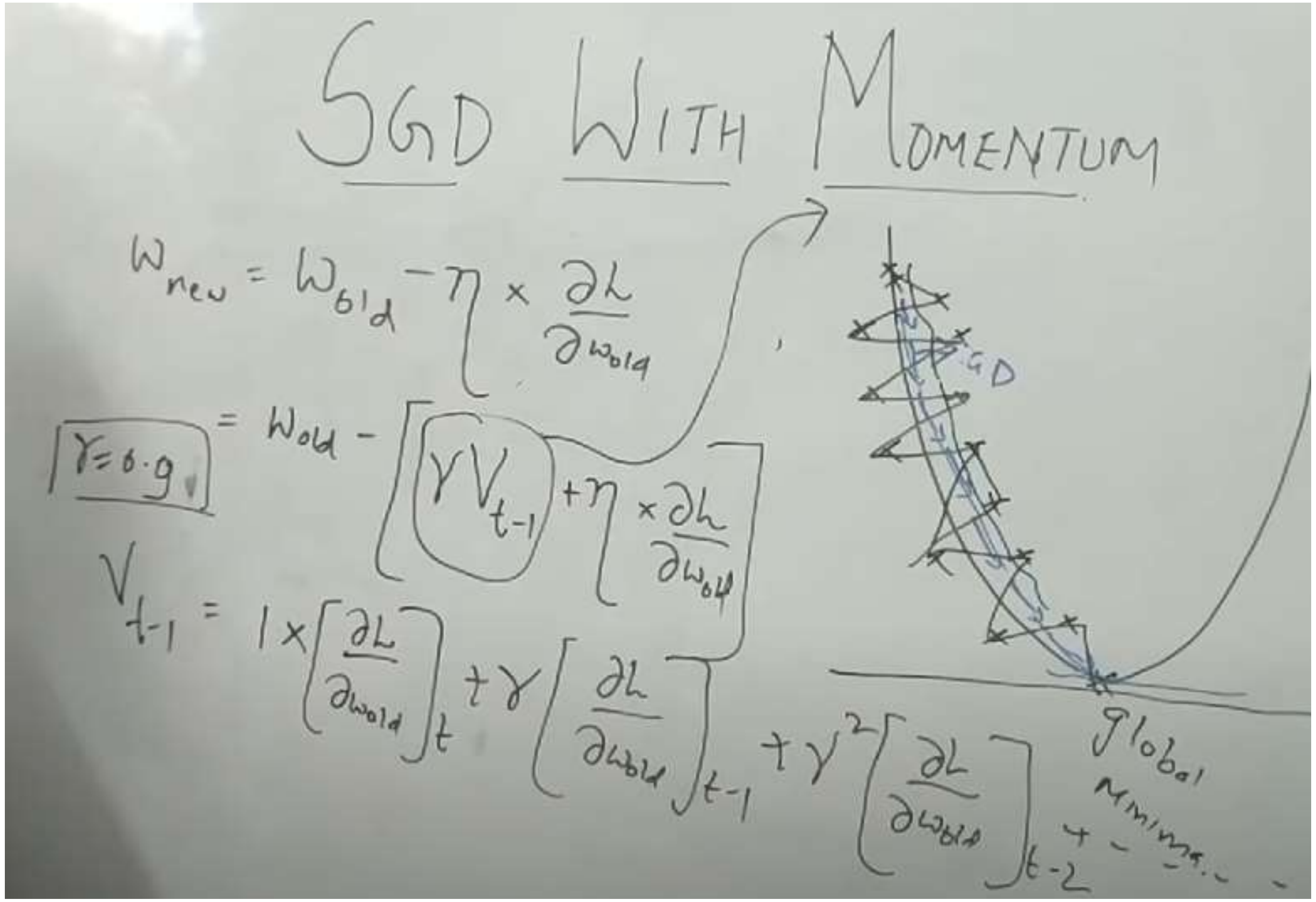
$V_3 = \gamma \times V_2 + b_3 = \gamma(\gamma \times v_1 + b_2) + b_3$

$= \gamma^2 \times b_1 + \gamma b_2 + b_3$

$= 0.25 b_1 + 0.5 b_2 + 1 \times b_3$



Optimizers (SGD with Momentum)



SGD WITH MOMENTUM

$$w_{new} = w_{old} - \eta \times \frac{\partial L}{\partial w_{old}}$$

$$\boxed{\gamma = 0.9} \quad = w_{old} - \left[\boxed{\gamma V_{t-1}} + \eta \times \frac{\partial L}{\partial w_{old}}\right]$$

$$V_{t-1} = 1 \times \left[\frac{\partial L}{\partial w_{old}}\right]_t + \gamma \left[\frac{\partial L}{\partial w_{old}}\right]_{t-1} + \gamma^2 \left[\frac{\partial L}{\partial w_{old}}\right]_{t-2} + \cdots$$

SGD
global minima

Optimizers (SGD with Momentum)

- Momentum works by **maintaining a velocity vector** that is updated with a combination of the previous velocity and the current gradient. The velocity vector is used to update the model parameters.

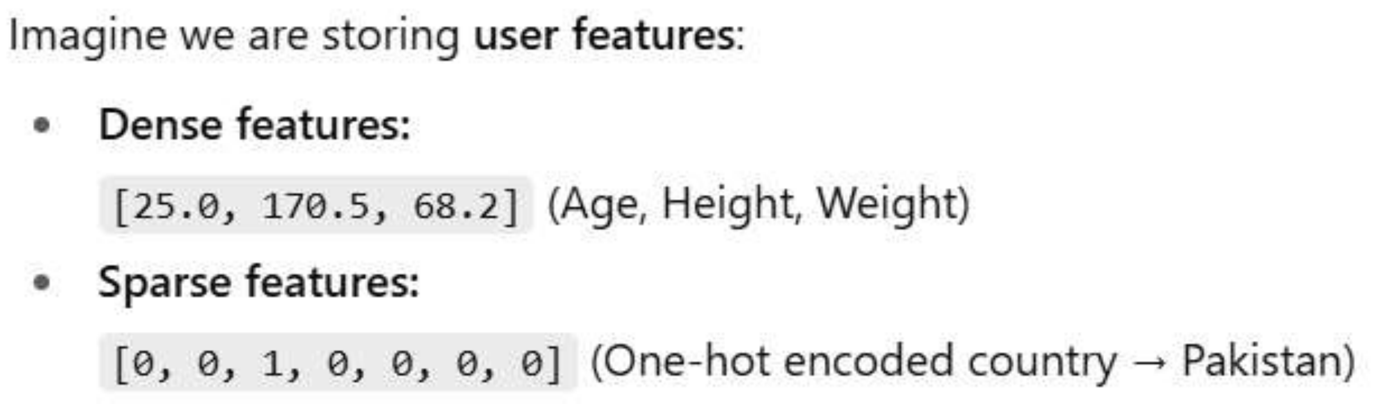
Exponential Weighted Average $v_t = \beta \times vt_{-1} + (1 - \beta) \times \partial L / \partial wold$

$$W_{new} = W_{old} - \eta \, v_{dw} \qquad W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

- v_t (v_{dw}) is the velocity at time t.
- β is the momentum coefficient (typically set between 0.8 and 0.99).
- v_{t-1} is the previous velocity.
- $\nabla L(\theta)$ is the gradient of the loss function with respect to the model parameters.
- We will apply v_t instead derivative of L w.r.t to old weight to update to weights

Optimizers (Adagrad)

- Dense vs Sparse Features



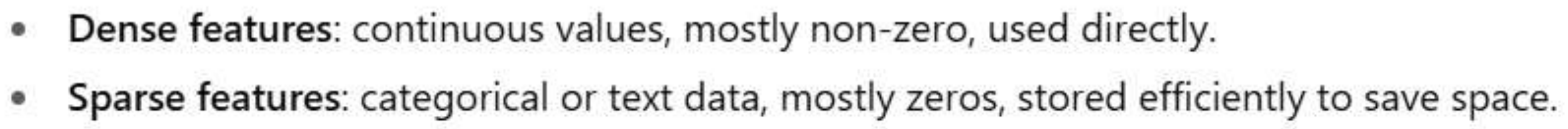
Imagine we are storing **user features**:

- **Dense features:**

  `[25.0, 170.5, 68.2]` (Age, Height, Weight)

- **Sparse features:**

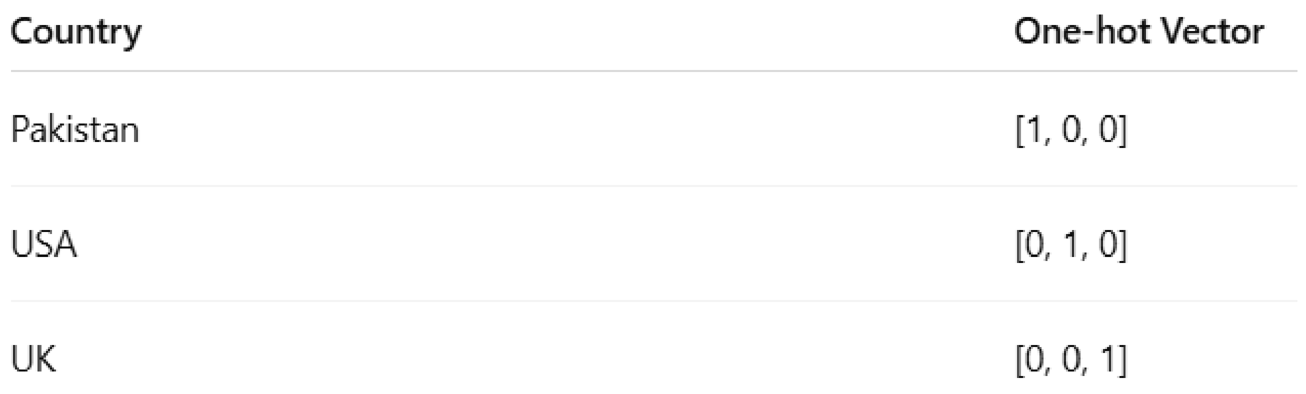
  `[0, 0, 1, 0, 0, 0, 0]` (One-hot encoded country → Pakistan)



- **Dense features**: continuous values, mostly non-zero, used directly.
- **Sparse features**: categorical or text data, mostly zeros, stored efficiently to save space.

Optimizers (Adagrad)

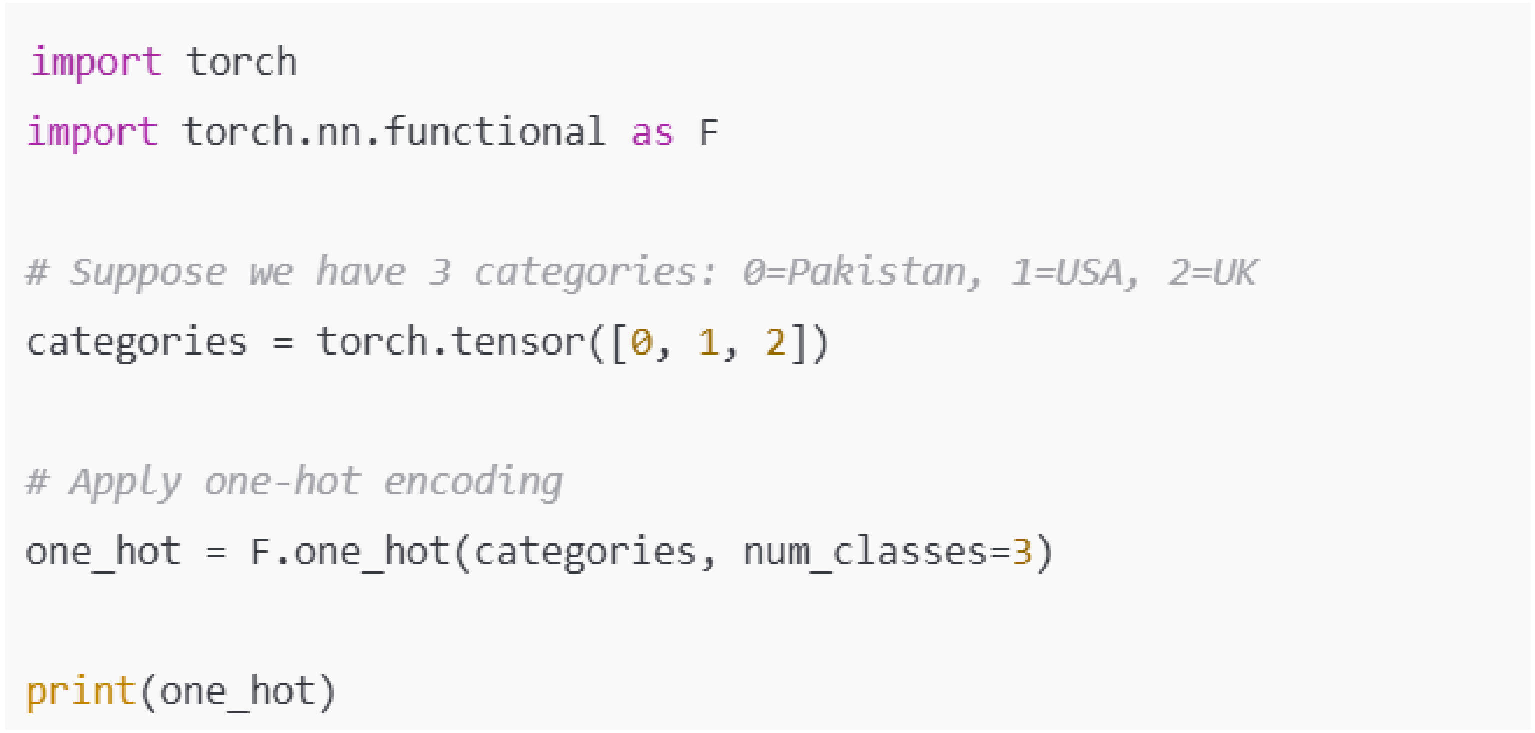
- One Hot Encoding
  - A method to represent **categorical data (labels, categories, classes)** as numbers so that a computer can understand them.
  - Instead of giving a category a single number (like Pakistan = 1, USA = 2, UK = 3), which could confuse the model (it may think 3 > 1),
  - We use a **binary vector** where **only one position is "hot" (1)**, and all others are 0.



Country	One-hot Vector
Pakistan	[1, 0, 0]
USA	[0, 1, 0]
UK	[0, 0, 1]

Optimizers (Adagrad)

- One Hot Encoding

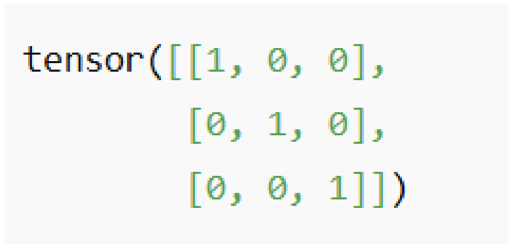


```python
import torch
import torch.nn.functional as F

# Suppose we have 3 categories: 0=Pakistan, 1=USA, 2=UK
categories = torch.tensor([0, 1, 2])

# Apply one-hot encoding
one_hot = F.one_hot(categories, num_classes=3)

print(one_hot)
```



```
tensor([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

Optimizers (Adagrad)

- Imagine you are learning to walk on different terrains:
  - On a flat road (easy), you quickly learn → so you take smaller steps later.
  - On a rocky road (hard), you need to keep taking bigger steps to adjust.
- AdaGrad does exactly this for parameters!

- Each parameter gets its own adaptive learning rate, based on how often it has been updated.

Optimizers (Adagrad)

- AdaGrad, short for adaptive gradient, signifies that the learning rates are adjusted or adapted over time based on previous gradients.
- A limitation of the previously discussed optimizers (SGD, Minibatch SGD) is the use of a **fixed learning rate** for all parameters throughout each cycle.
- This can **hinder the training features** which often exhibit **small average gradients** causing them to **train at a slower pace**.
- A **solution is to set different learning rates for each feature**, this can become complex .
- AdaGrad addresses this issue by implementing the concept that **the more a feature has been updated in the past**, the **less it will be updated in the future**.
- This provides an opportunity for other features, such as **sparse** features, to catch up.
- AdaGrad, as an optimizer, **dynamically adjusts the learning rate** for each parameter at every time step ‘t’.

Optimizers (Adagrad)

- ϵ is a small positive value number to avoid divide by zero error if in case alpha(t) becomes 0
- gi is derivative of loss w.r.t weight and g_i^2 will always be +ve since its a square term, this implies that alpha(t) >= alpha(t-1)
- One main **disadvantage** of Adagrad optimizer is that **alpha(t) can become large** as the number of iterations will increase and due to this **η_t' will decrease at the larger rate**.
- This will make the old weight almost equal to the new weight which may lead to slow convergence.

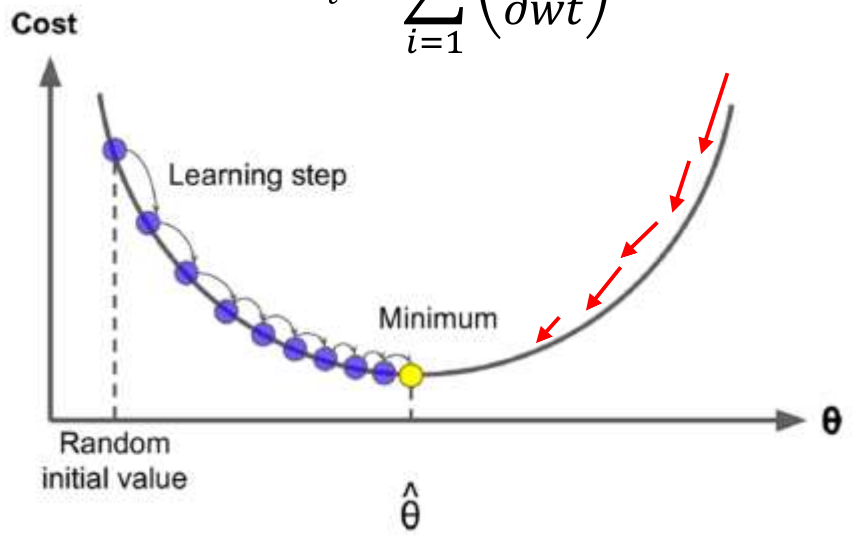
$$(\mathrm{w})_{\text{new}} = (\mathrm{w})_{\text{old}} - \eta \frac{\partial \mathrm{L}}{\mathrm{dw}(\text{ old })}$$

$$w_t = w_{t-1} - \eta \frac{\partial L}{\partial w(t-1)}$$

$$\mathrm{w}_t = \mathrm{w}_{t-1} - \eta_t' \frac{\partial \mathrm{L}}{\partial \mathrm{w}(\mathrm{t}-1)}$$

$$\eta_t' = \frac{\eta}{\mathrm{sqrt}(\alpha_t + \epsilon)}$$

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial wt} \right)^2$$



Optimizers (RMSProp)

- RMSprop stands for **Root Mean Square Propagation**.
- RMSprop optimizer **doesn’t let gradients accumulate for momentum** instead only accumulates gradients in a particular fixed window (avoid alpha to reach big value).
- It can be considered as an **updated version of AdaGrad** with few improvements.

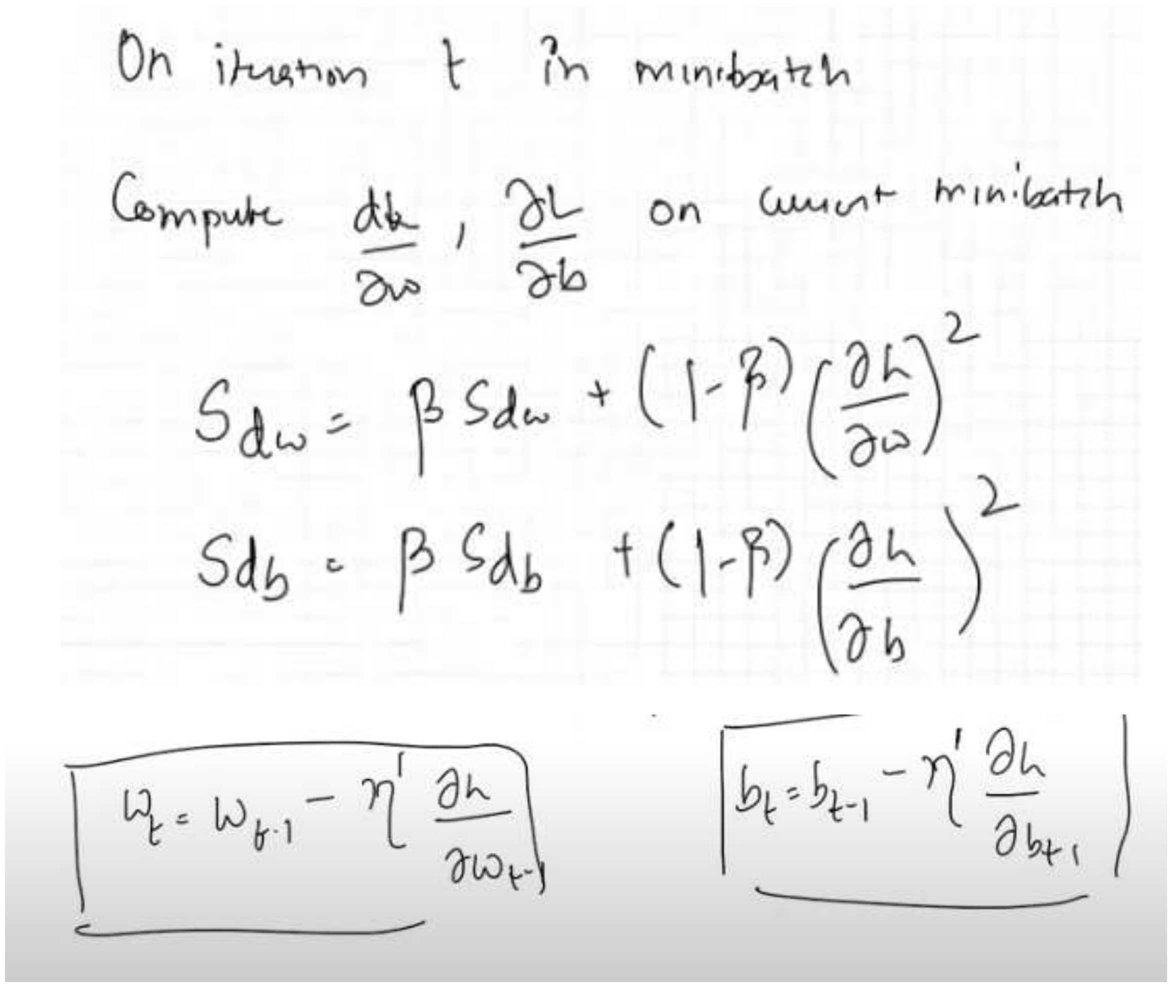
$$w_t = w_{t-1} - \eta'_t \frac{\partial L}{\partial w(t-1)} \qquad \eta' = \frac{\eta}{\sqrt{\boldsymbol{Sdw} + \boldsymbol{\epsilon}}}$$

$$\mathbf{Sdw_t} = \boldsymbol{\beta} \times \boldsymbol{Sdwt}\_{\,1} + (\mathbf{1} - \boldsymbol{\beta}) \times \left(\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial wt}}\right)^{\mathbf{2}}$$

- Sdw value at initial time t will be 0
- The drawback of AdaGrad is learning rate some times can become very slow due to high α_t value.
- Here α_t is replace by Sdw that slow the derivate of L w.r.t weight by multiplying it with $(1 - \beta)$.
- Let say $beta$ value is 0.9 then value multiplying derivative will be 0.1.
- However, still there is the smoothening is not achieved as in AdaGrad we have vdw instead of derivative.

Optimizers (RMSProp)

- Implementation Details:



On iteration t in minibatch

Compute $\frac{dL}{\partial w}$, $\frac{\partial L}{\partial b}$ on current minibatch

$$S_{dw} = \beta S_{dw} + (1-\beta)\left(\frac{\partial h}{\partial w}\right)^2$$

$$S_{db} = \beta S_{db} + (1-\beta)\left(\frac{\partial h}{\partial b}\right)^2$$

$$\boxed{w_t = w_{t-1} - \eta' \frac{\partial h}{\partial w_{t-1}}} \qquad \boxed{b_t = b_{t-1} - \eta' \frac{\partial h}{\partial b_{t-1}}}$$

Optimizers (Adam)

- Imagine you are walking down a mountain:
    - Momentum → keeps track of your past direction, so you don’t zig-zag too much.
    - Adaptive learning rate (RMSProp) → if the slope is steep, take small steps; if it’s flat, take bigger steps.
- Adam combines both → smooth + adaptive.

Optimizers (Adam)

- If we need smoothening of gradient as **AdaGrad** and controlled learning rate like **RMSProp** then we can **combine both** to achieve this and such optimizer is called Adam optimizer.
- **Adaptive Moment Estimation** (Adam)
- Here, we used both Sdw and Vdw

$$w_t = w_{t-1} - \eta'_t \frac{\partial L}{\partial w(t-1)} \xrightarrow{\text{becomes}} \boldsymbol{w_t = wt \_ {}_{1} - \eta' V dw}$$

$$\eta' = \frac{\eta}{\sqrt{Sdw + \epsilon}}$$

$$vdw_t = \beta \times vdwt \_ {}_{1} + (1 - \beta) \times \partial L / \partial wold$$

$$\mathrm{Sdw_t} = \beta \times Sdwt \_ {}_{1} + (1 - \beta) \times \left(\frac{\partial L}{\partial wt}\right)^2$$

- **Adaptive Learning rate + smoothening**
- **One of the best optimizer of today with various variants!**

Optimizers (Adam)

Compute $\frac{\partial h}{\partial w}$, $\frac{\partial h}{\partial b}$ using current mini batch

$$\left.\begin{aligned} V_{dw} &= \beta_1 V_{dw} + (1-\beta)\frac{dh}{\partial w} \\ V_{db} &= \beta_1 V_{dw} + (1-\beta)\frac{\partial h}{\partial b} \end{aligned}\right\} \text{Momentum}$$

$$\left.\begin{aligned} S_{dw} &= \beta_2 S_{dw} + (1-\beta)\left(\frac{\partial h}{\partial w}\right)^2 \\ S_{db} &= \beta_2 S_{db} + (1-\beta)\left(\frac{\partial h}{\partial b}\right)^2 \end{aligned}\right\} \text{Rms Prop}$$

initial LR

$$\boxed{\begin{aligned} W_t &= W_{t-1} - \frac{\eta * V_{dw}}{\sqrt{S_{dw} + \epsilon}} \\ b_t &= b_{t-1} - \frac{\eta * V_{db}}{\sqrt{S_{db} + \epsilon}} \end{aligned}}$$

Adam optimizer

Summary

- Discussed about loss calculations and its variants
- Discussed about various optimizers
- Discussed about hierarchy of optimizers