Optimizers in Neural Network

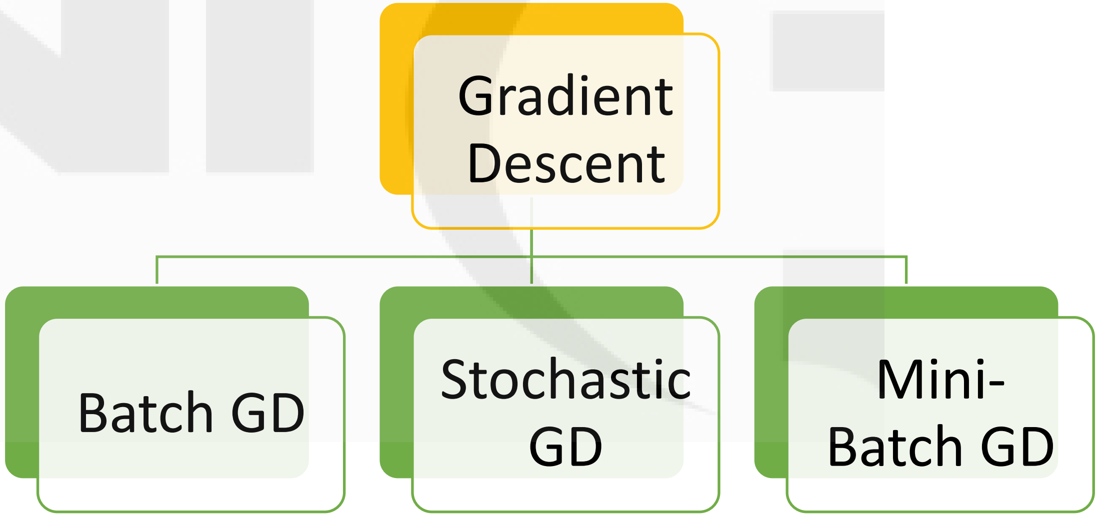
MUKESH KUMAR

AGENDA

- What is Optimizer
- Types of Optimizers
  - Batch Gradient Descent
  - Stochastic Gradient Descent (SGD)
  - Mini-Batch Gradient Descent
  - SGDMomentum
  - AdaGrad
  - RMSprop
  - Adam
- How to choose optimizers

Types of Gradient Descent

- There are 3 types of gradient descent which differ in how much data we use to compute the gradient of the objective function



Batch Gradient Descent

- Prediction is made on all the data points
- For all the prediction loss is calculated
- Only once weights and bias are updated per epoch
- The gradients are computed over the full dataset, and the model parameters (weights and biases) are updated **once per epoch**

Stochastic Gradient Descent

- the model performs a forward pass and computes gradients **one record at a time,** followed by an **immediate update** to the weights and bias

- So. if there are 1000 records and 100 epocs

- Every epoch 1000 times weights and biases will be updated

- Total of 1000* 100 times gradients will be calculated and parameters will be updated

Mini –Batch Gradient Descent

- Mini-Batch is a middle ground between Batch GD and Stochastic GD, best of both worlds.

- Based on the batch size, model updates parameters after every batch.

- So, if there are 1000 records and batch size =200

- Model will make prediction for 200 records at a time
- It computes the **average loss** over those 200 predictions, then finds the gradients and update the parameters.

- So every epoch parameters will be updated 5 times (1000/200)

Comparison

Feature	Batch Gradient Descent	Stochastic Gradient Descent (SGD)	Mini-Batch Gradient Descent
Batch Size	All training data	1 record	Custom size (e.g., 32, 64, 128)
Parameter Update Frequency	Once per epoch	Once per sample	Once per mini-batch
Updates per Epoch	1	*n* (number of records)	*n* / *batch_size*
Speed per Update	Slow	Very fast	Moderate
Convergence Stability	Very stable	Noisy, less stable	Balanced
Memory Usage	High	Low	Medium
Training Time per Epoch	Long	Short	Moderate
When to Use	Small datasets	Very large datasets, online learning	Almost always preferred in practice

- Before moving on to other optimizers we need to understand EWMA

EWMA

- The **Exponentially Weighted Moving Average (EWMA)** is a method to smooth a sequence of data points by giving **more weight to recent observations** and **less weight to older ones**, using an exponential decay.

- The simple idea is that the current values depends on previous values, more on most recent ones and less on older ones

Intuition Behind EWMA:

- Recent points matter more — we're biased toward the present.
- Older points fade out, but never entirely vanish (unlike a simple moving average that forgets old data completely).

EWMA Forumla

$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot x_t$$

Where:

- v_t = EWMA at time t
- x_t = actual value at time t
- $\beta \in [0, 1)$ = smoothing factor (e.g., 0.9 or 0.99)
- v_0 is often initialized as 0 or the first value

EWMA Formula

Vt is the Exponentially weighted moving average at time t

$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot x_t$$

Xt is the data at current time, example : if working with temperature its temp

EWMA at previous time

Beta is constant between 0 and 1

EWMA

- Beta decides how much we value past data points
- High Beta means we are giving more weightage to past data points
- Low beta means we are giving less weightage to past data points

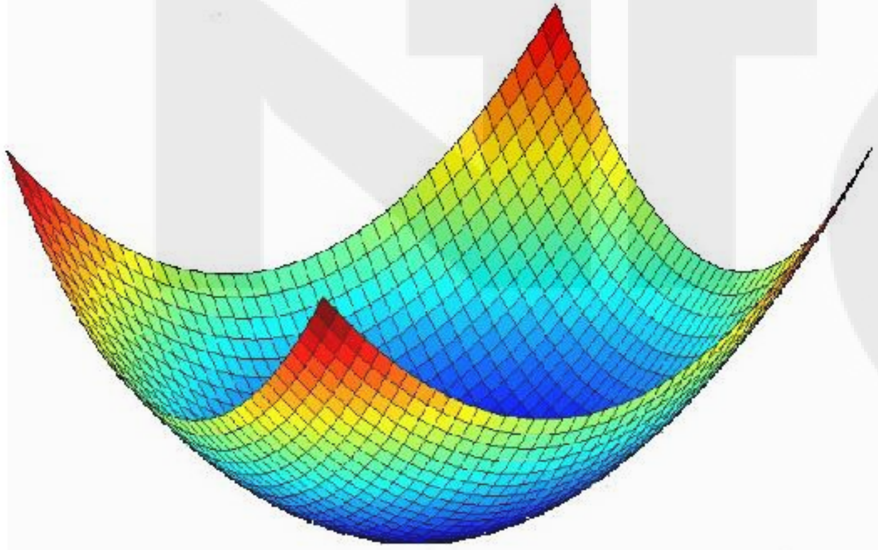
- Demo EWMA in jupyter notebook

More Optimizers

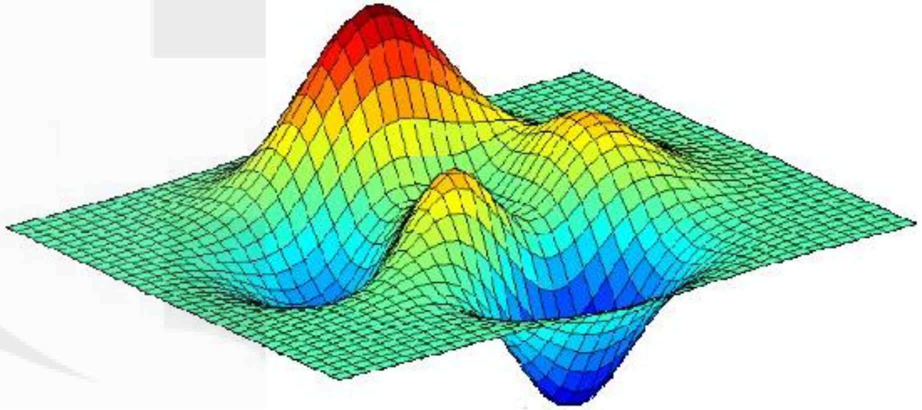
- SGDMomentum
- AdaGrad
- RMSprop
- Adam

Loss Graphs in Machine learning Problems

- Convex vs Non-Convex loss functions

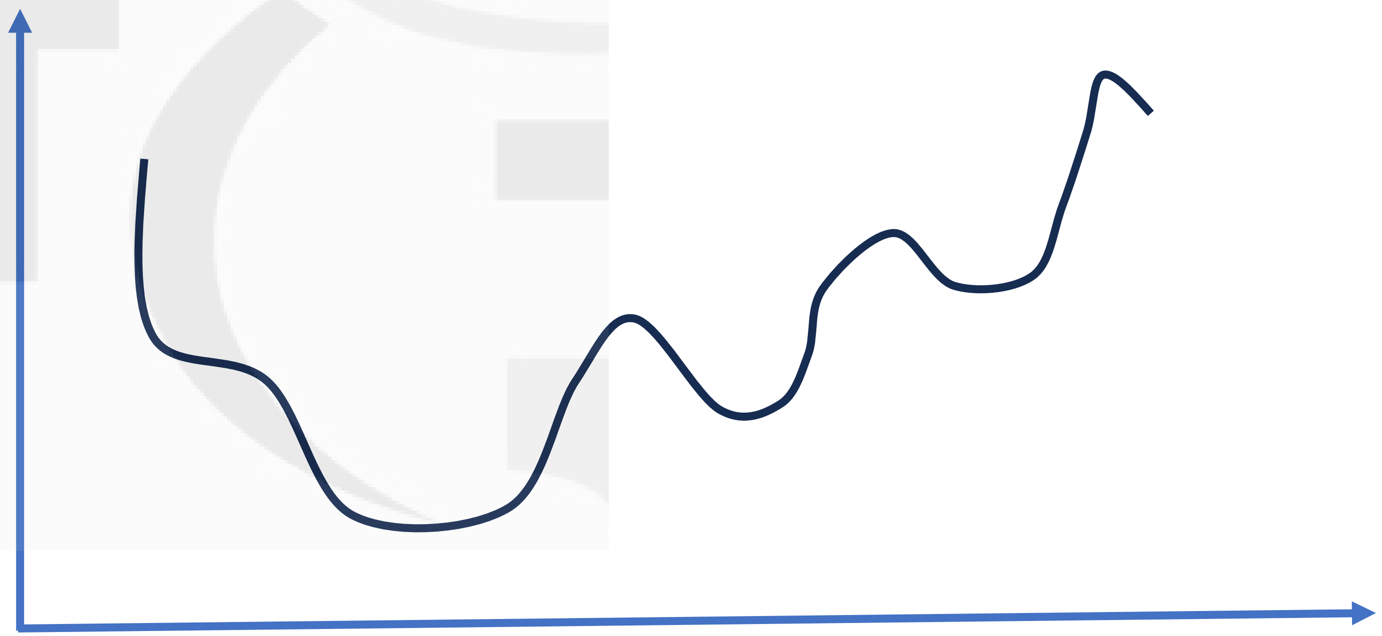


convex function

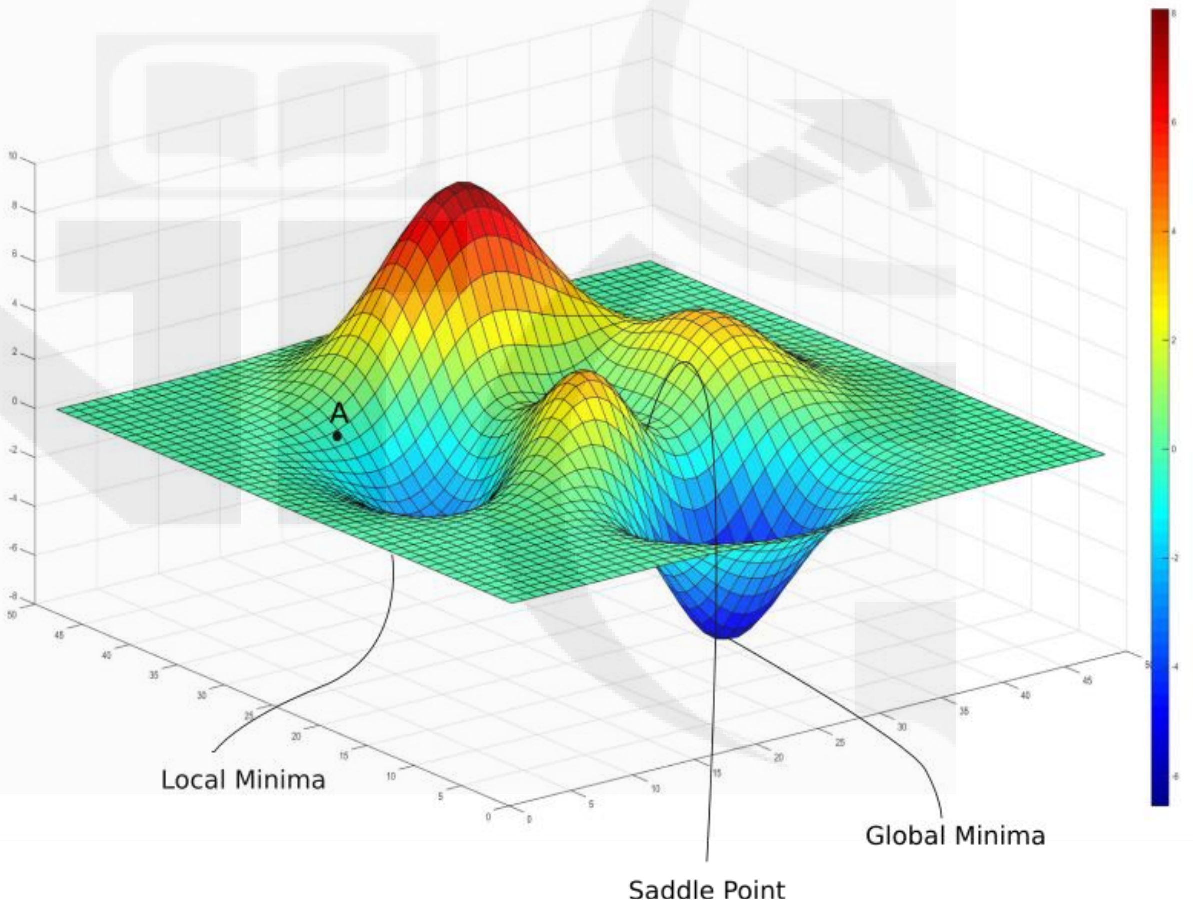


non-convex function

Convex vs Non-Convex loss functions

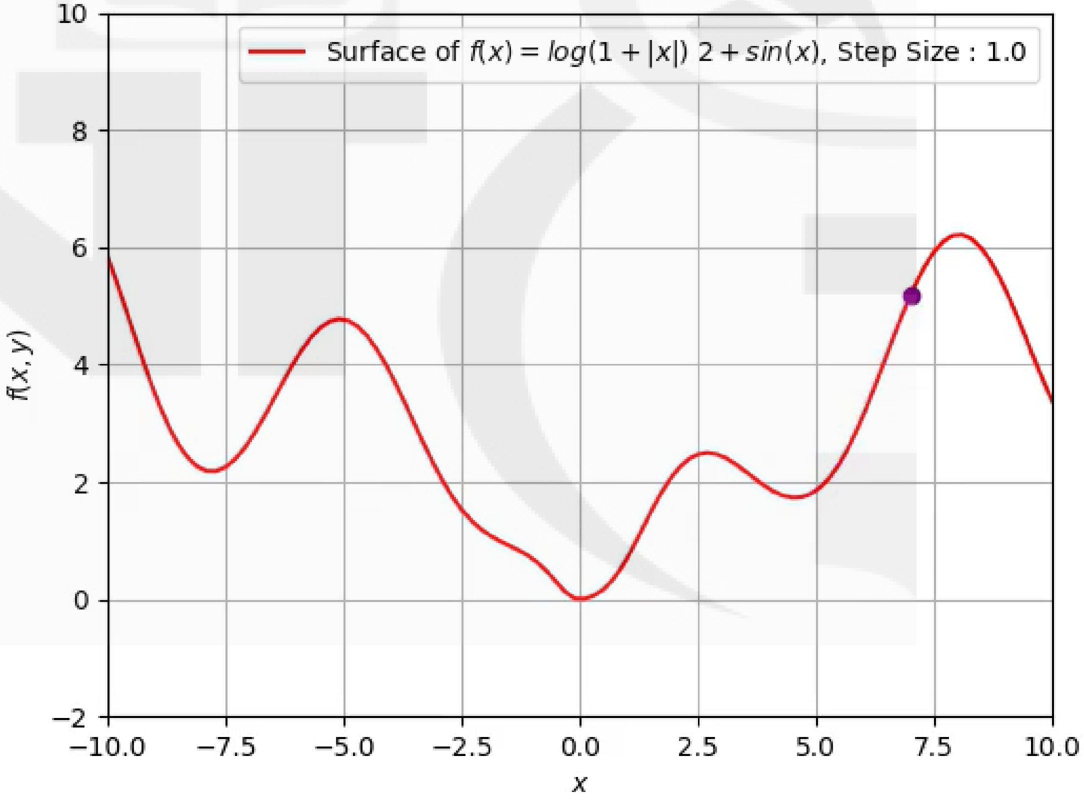


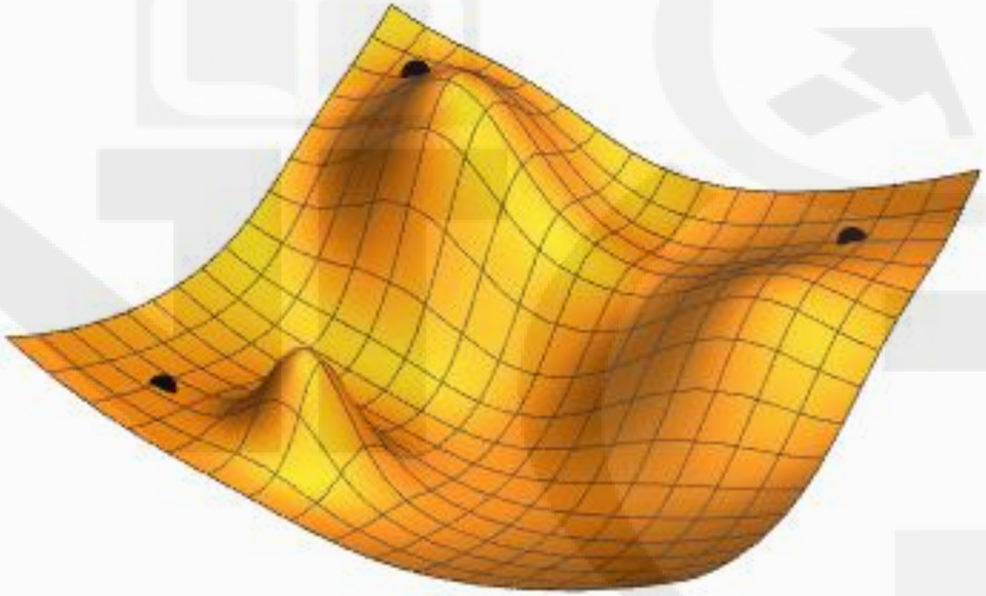
Local Minima Vs Global Minima


A
Local Minima
Saddle Point
Global Minima

Local Minima Problem/Noisy Gradient

Noisy gradient means there are lots of local minima



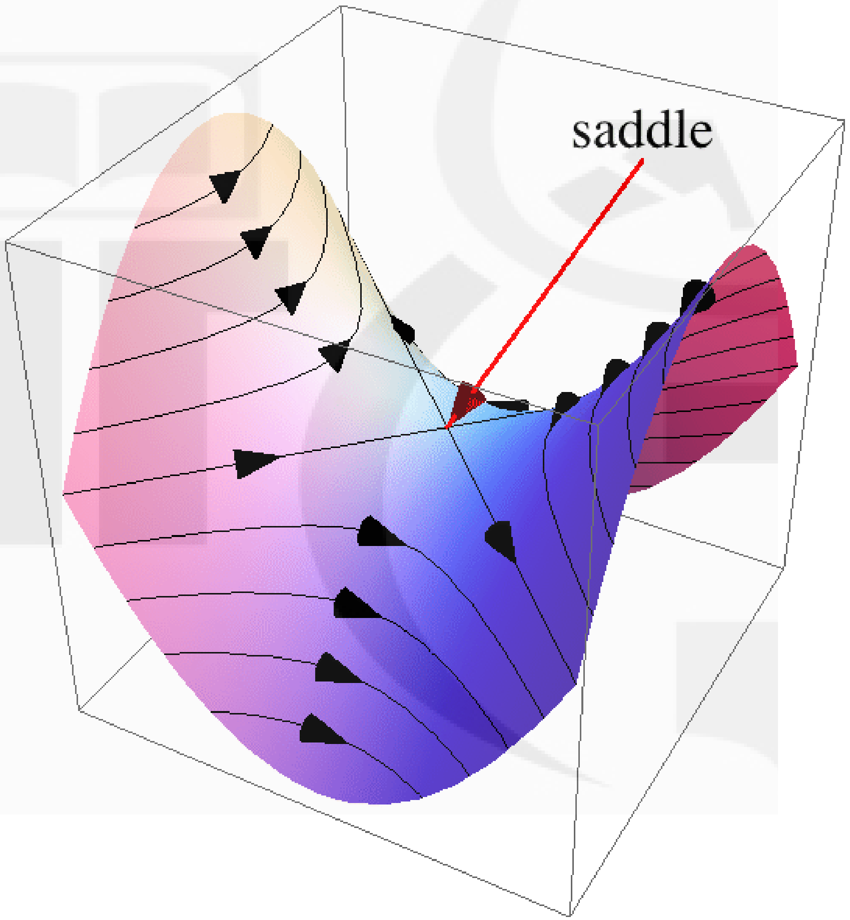


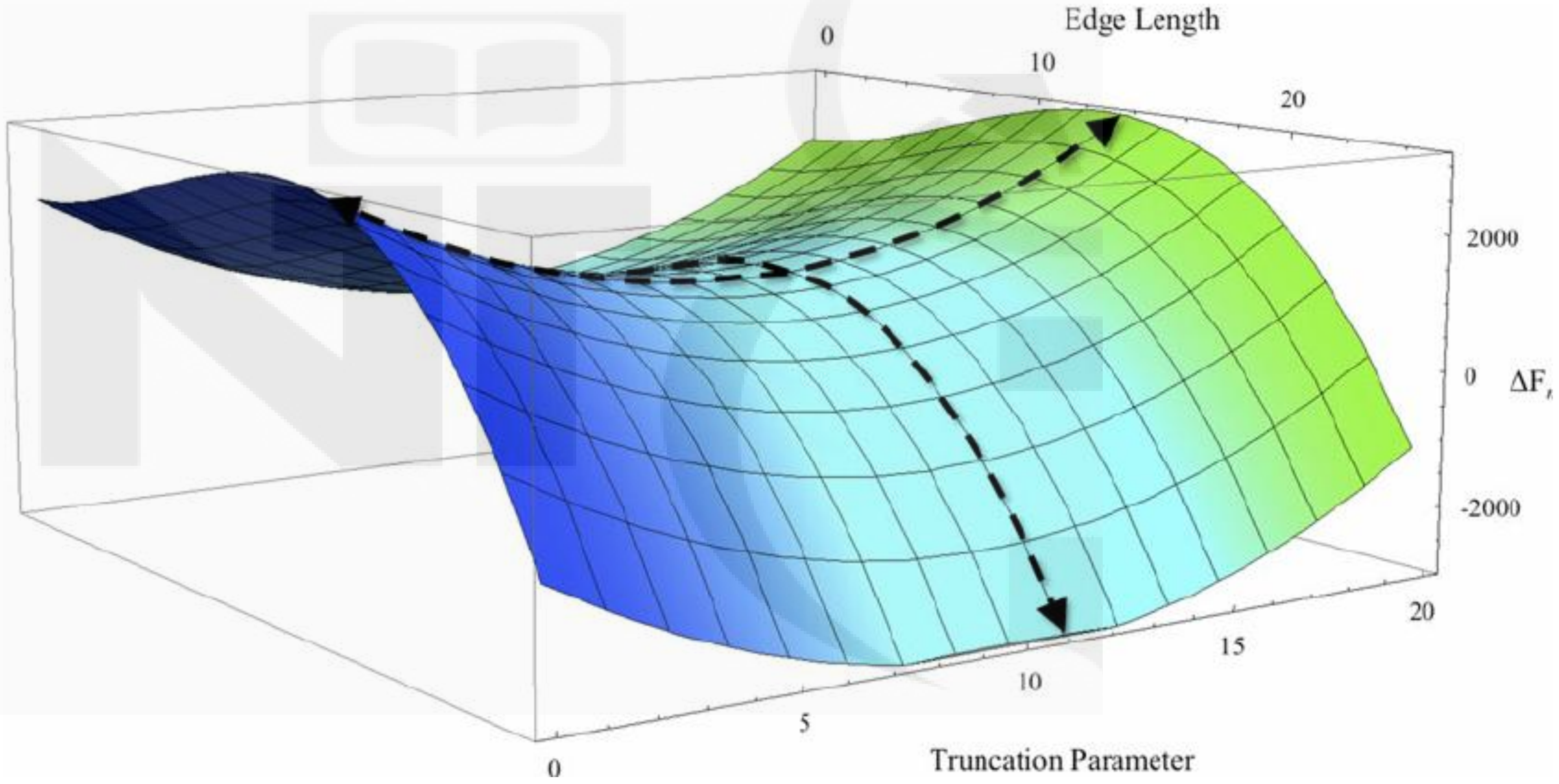
Saddle Point

Imagine a **horse saddle**:

- It curves **upward** along one direction (like left-right),
- And curves **downward** along the other (like front-back).
- So, it’s neither a peak nor a valley — it's in between.

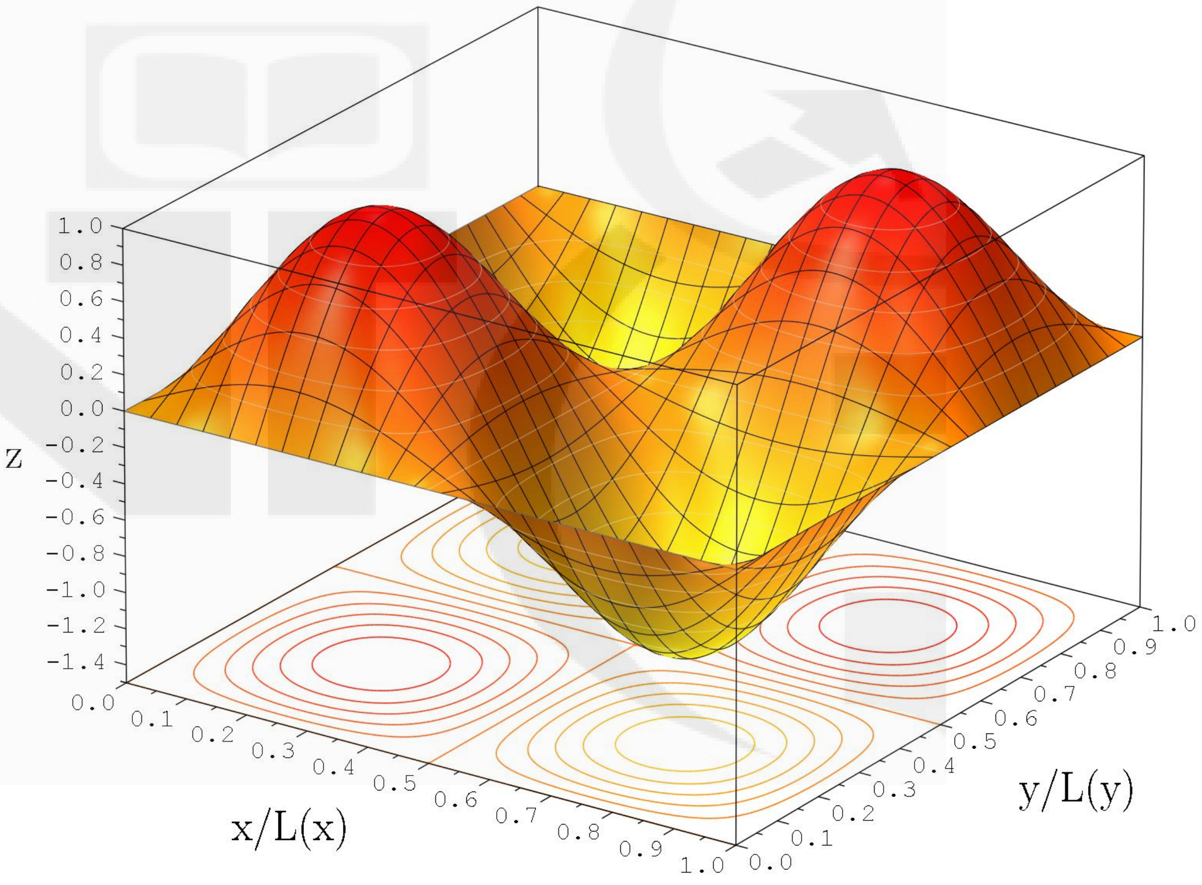
- **Because all the optimizers work based on slopes , saddle point is difficult to handle because slope is almost zero**

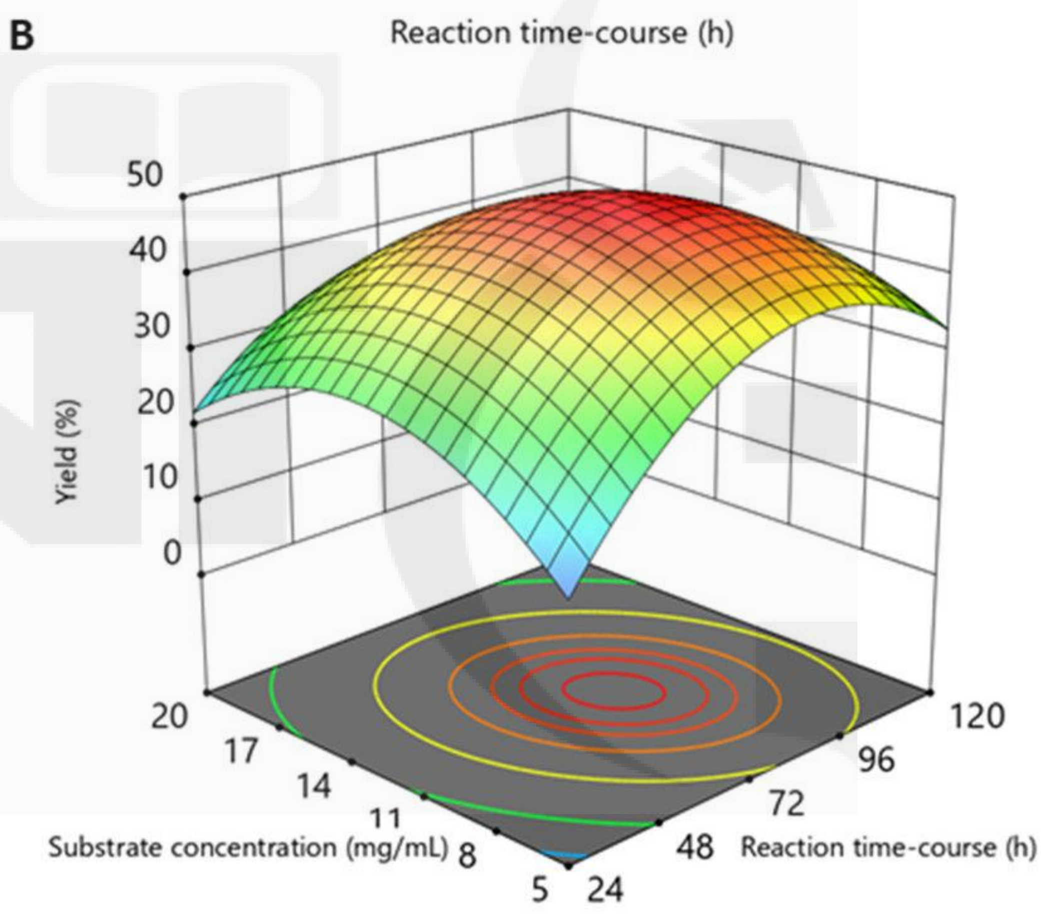

saddle

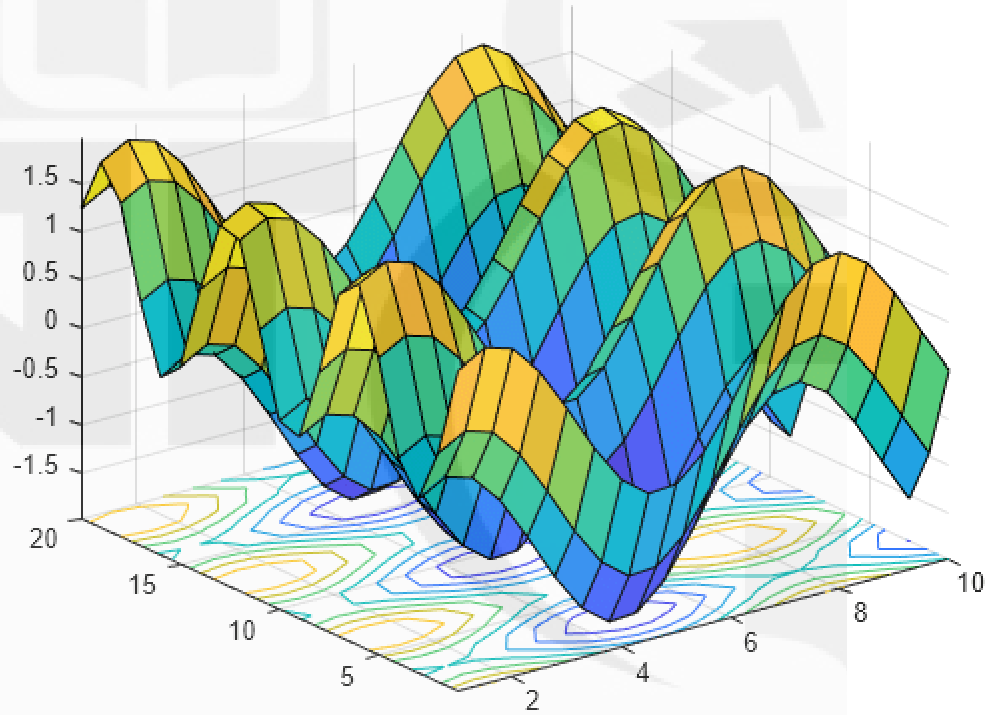

Edge Length
0
10
20
2000
0
-2000
ΔF
0
5
10
15
20
Truncation Parameter

Contour Plots

Projection from top view


z
1.0
0.8
0.6
0.4
0.2
0.0
-0.2
-0.4
-0.6
-0.8
-1.0
-1.2
-1.4
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
x/L(x)
y/L(y)


B
Reaction time-course (h)
Yield (%)
50
40
30
20
10
0
20
17
14
11
8
5
Substrate concentration (mg/mL)
24
48
72
96
120
Reaction time-course (h)


1.5
1
0.5
0
-0.5
-1
-1.5
20
15
10
5
2
4
6
8
10

GD with Momentum

- GD fails to handle these problems:
  - Local minima
  - Saddle point

- **What momentum solves?**
  - Due to non-convex surfaces the GD doesn't work well (small slope, local minima)
  - Momentum works well in above situation

How Momentum works

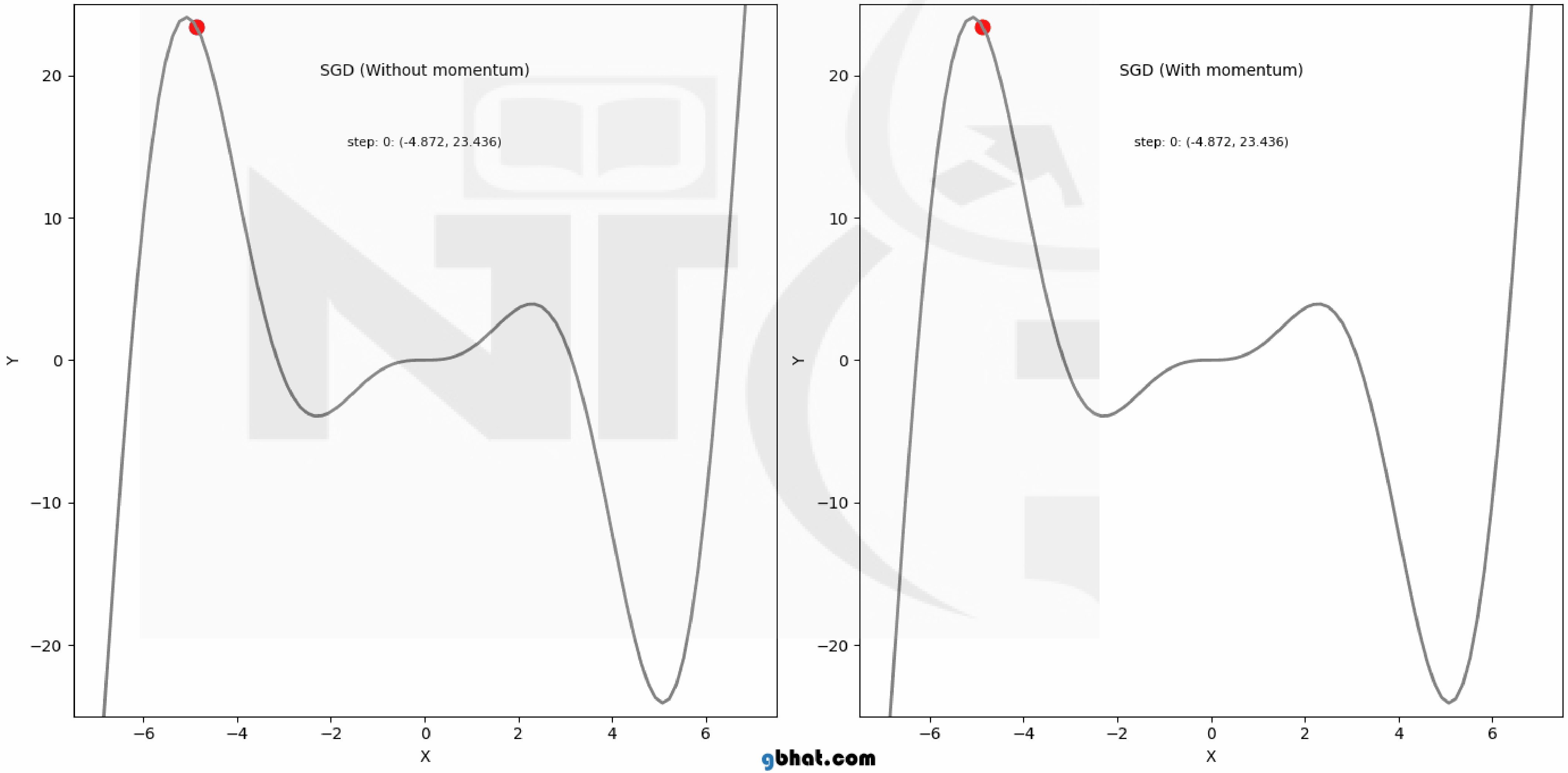
- Its like rolling a ball on a slope , as the ball goes down due to its velocity momentum goes on increasing as it goes further down the slope
- We rely on velocity
- We look the history of velocity and based on it we build momentum
- Momentum biggest advantage is speed compared to GD

Intuitive Analogy:

Think of momentum like a **ball rolling down a hill**:

- Even if the slope becomes flatter, the ball keeps moving due to **inertia** (accumulated momentum).
- Without momentum, you'd stop quickly on flat or gentle slopes.

GD with Momentum



gbhat.com

GD with Momentum Formula

- This method helps accelerate gradient descent in relevant directions and dampens oscillations

$$v_t = \beta v_{t-1} + (1 - \beta) g_t$$

$$w_{t+1} = w_t - \eta v_t$$

Think of Vt as velocity at time t

0<beta<1, usually 0.9

Higher Beta give more weightage to the first part i.e past velocities/gradients

And lower beta give more weightage to second part i.e. current gradient

- w_t: weights at iteration t
- $g_t = \nabla J(w_t)$: gradient of the loss function at iteration t
- v_t: velocity (momentum term)
- β: momentum coefficient (commonly set to 0.9)
- η: learning rate

Momentum in a Nutshell

When you're **going downhill** (i.e., descending the slope of the loss function), and:

The slope (gradient) is reducing:

- It means you're approaching a minimum.
- Normally, plain gradient descent would **slow down** as gradients become small.

- **Momentum helps counter this** by **accumulating velocity from previous steps.**
  - So even if the current gradient is small, the **velocity term vt** still carries forward motion from earlier, **pushing the weights further** in the right direction.

Disadvantages of Momentum in Gradient Descent

1. Overshooting the minimum

- Since momentum accumulates velocity, it can **overshoot the optimal point** — especially when the learning rate or momentum coefficient is too high.
- This can cause **oscillations** or **divergence** if not tuned well.

2. Sensitive to hyperparameters

- Requires careful tuning of:
  - **Learning rate (η)**
  - **Momentum coefficient (β)** — usually 0.9, but can vary
- Poor tuning can slow convergence or destabilize training.

Disadvantages of Momentum in Gradient Descent

Not adaptive

- Momentum uses the **same learning rate for all parameters**.
- It doesn't adjust learning rates dynamically like **Adam** or **RMSProp,** which can perform better in sparse or noisy settings.

SGD with Momentum Code

```
from tensorflow.keras.optimizers import SGD

optimizer = SGD(learning_rate=0.01, momentum=0.9)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

ADAGRAD

- Short for Adaptive Gradient
- Learning rate is not fixed
- When ADAGRAD work best:
  - if the input features scale is very different, usually we normalize so it won't apply.
  - Another scenario is when your data is sparse (i.e. most of the values in col are 0)

Prob with Sparse feature

- Elongate bowl problem
- Loss is elongated in the direction of sparse column direction
- Due to elongated axis the slope is constant in one ddirection , due to which there is one slope in direction and no gradient in other direction

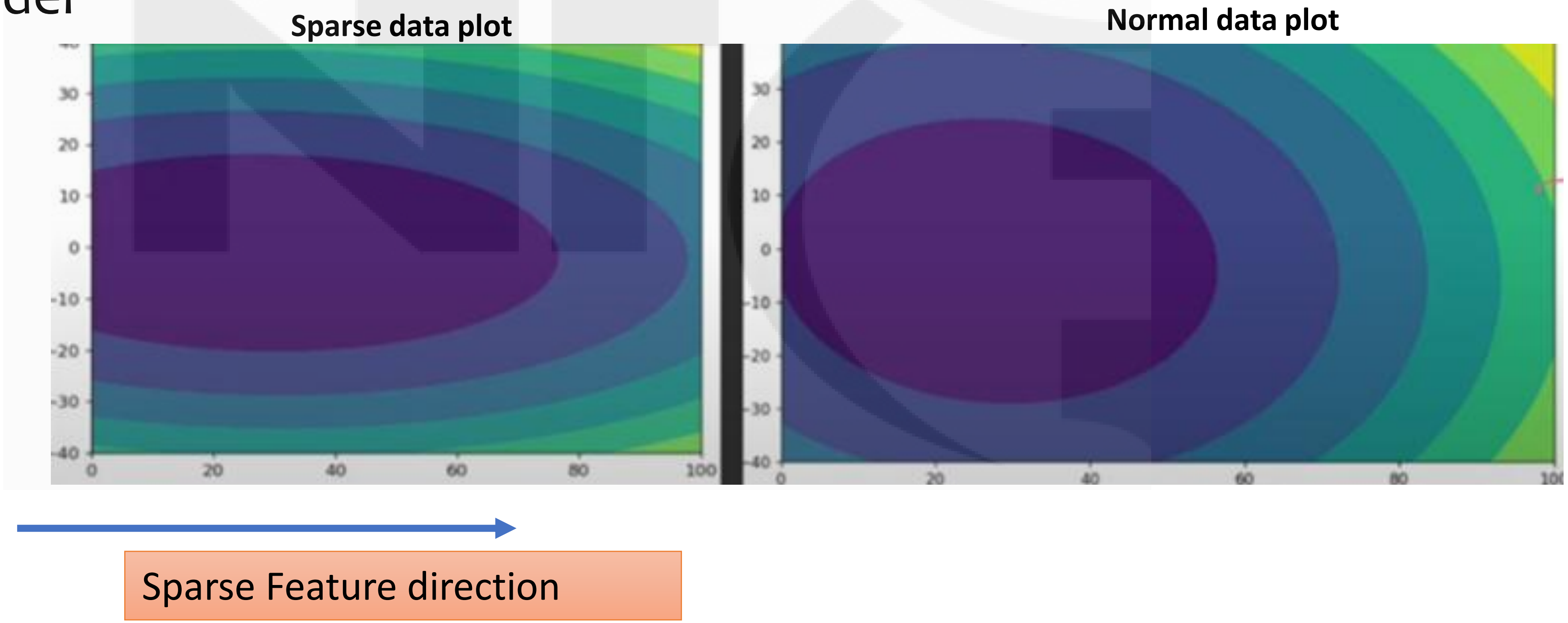
- https://www.desmos.com/3d

Sample Sparse Data

Row	x1 (dense)	x2 (sparse)
1	2.5	0
2	3.1	0
3	4	0
4	1.8	1
5	2.2	0
6	3.3	0
7	2.7	0
8	4.5	0
9	3.9	0
10	2.1	0
11	3	2
12	2.6	0
13	3.7	0
14	2.4	0
15	4.1	0

Sparse data

- When a dimension with sparse data is involved in a contour plot, it can "stretch" the contour, making the slope appear relatively flat or shallow. This happens because the lack of data points in that dimension means that the model



$$w_t = w_{t-1} - \boxed{\frac{\eta}{\sqrt{G_t + \epsilon}}} \cdot \nabla_w J(w)$$

Gt is the sum squared of all previous gradients

- w_t are the **weights** at time step t,
- η is the **learning rate**,
- G_t is the **accumulated squared gradient** at time step t, calculated as:

$$G_t = \sum_{i=1}^{t} \nabla_w J(w_i)^2$$

- $\nabla_w J(w)$ is the **gradient of the loss** with respect to the weights w,
- ϵ is a small value (typically 10^{-8}) to avoid division by zero.

For the parameters with high gradient learning rate component will be low and vice versa

- All the optimizer we learnt so far have fixed learning rates for all the parameter but in adaGrad its different for each parameter

- Reduce the learning rate of the Parmeter for which the gradient is high and vise versa so the optimization happens with all direction equally

Animation showing diff optimizers vs Adagrad

- https://github.com/lilipads/gradient_descent_viz

- C:\Users\MUKESH\Downloads\gradient_descent_viz-master\gradient_descent_viz-master\gradient_descent_viz_windows64bit\gradient_descent_viz_windows64bit

Disadvantages

- We never use ADAGRAD in Neural network
- Because we are reducing the learning rate by dividing it by past gradients
- Past gradient increase over epochs so denominator increases so the overall learning rate becomes very low causing very small updates towards the end
- This is the reason why adagrad never reaches the optimized solution , it stops before that, it never converges
- We are learning it because this is reused in upcoming optimizers

AdaGrad Code

```
from tensorflow.keras.optimizers import Adagrad

optimizer = Adagrad(learning_rate=0.01)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

RMSProp

- Short for Root Mean Square Propagation
- Its an improvement over Adagrad

1. **Update the squared gradient moving average:**

$$V_t = \beta \cdot V_{t-1} + (1-\beta) \cdot g_t^2$$

2. **Update the parameter:**

$$w_{t+1} = w_t - \eta \cdot \frac{g_t}{\sqrt{V_t} + \epsilon}$$

- w_t = parameter at time step t
- $g_t = \nabla_w L(w_t)$ = gradient of the loss with respect to w_t
- $\beta \in [0, 1)$ = decay rate (usually 0.9)
- η = learning rate
- ϵ = small value for numerical stability (e.g., 10^{-8})
- V_t = running average of squared gradients

How its better than AdaGrad

1. Update the squared gradient moving average:

$$V_t = \beta \cdot V_{t-1} + (1 - \beta) \cdot g_t^2$$

2. Update the parameter:

$$w_{t+1} = w_t - \eta \cdot \frac{g_t}{\sqrt{V_t} + \epsilon}$$

- Usually beta is 0.95

This term gtsquare is the square of all the previous gradient , now its impact is reduced as it gets multiplied with a very small number

Due to this the denominator term in the new weight equation will reduce >> over impact would be higher learning rate over time allowing the model to learn and converge

Disadvantages

- None
- RMSProp is one of the best optimizer technique however Adam is slightly better

RMSProp Code

```
from tensorflow.keras.optimizers import RMSprop

optimizer = RMSprop(learning_rate=0.001, rho=0.9)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

ADAM

- Adaptive Moment Estimation
- It combines Momentum and ADAGRAD
- It takes velocity component from the Momentum and Adaptive learning rate from ADAGRAD

Formula

1. First moment estimate (gradient mean):

$$M_t = \beta_1 M_{t-1} + (1 - \beta_1) g_t$$

2. Second moment estimate (squared gradient average):

$$V_t = \beta_2 V_{t-1} + (1 - \beta_2) g_t^2$$

3. Parameter update (no bias correction):

$$w_{t+1} = w_t - \eta \cdot \frac{M_t}{\sqrt{V_t} + \epsilon}$$

M_t — Momentum-like behavior

V_t — Adaptive learning rate

- g_t: Gradient of the loss function at time step t
- β_1: Decay rate for the **first moment** (typically 0.9)
- β_2: Decay rate for the **second moment** (typically 0.999)
- M_t: First moment estimate (mean of gradients)
- V_t: Second moment estimate (uncentered variance of gradients)
- η: Learning rate (step size)
- ϵ: A small constant to avoid division by zero (e.g., 10^{-8})
- w_t: Parameter (weights) at time t

Adam Code

```
from tensorflow.keras.optimizers import Adam

optimizer = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

More Optimizers

- https://keras.io/api/optimizers/