HANDOUT: BACKWARD PROPAGATION ALGORITHM

1) Optimizers

Optimizers are algorithms or methods used to minimize an error function (*loss function*) or to maximize the efficiency. Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights & Biases. Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.

2) Need for Optimizers in Deep Learning

Choosing an appropriate optimizer for a deep learning model is important as it can greatly impact its performance. Optimization algorithms have different strengths and weaknesses and are better suited for certain problems and architectures.

Before proceeding, there are a few terms that you should be familiar with.

- **Epoch** – The number of times the algorithm runs on the whole training dataset.
- **Sample** – A single row of a dataset.
- **Batch** – It denotes the number of samples to be taken to for updating the model parameters.
- **Learning rate** – It is a parameter that provides the model a scale of how much model weights should be updated.
- **Cost Function/Loss Function** – A cost function is used to calculate the cost, which is the difference between the predicted value and the actual value.
- **Weights/ Bias** – The learnable parameters in a model that controls the signal between two neurons.

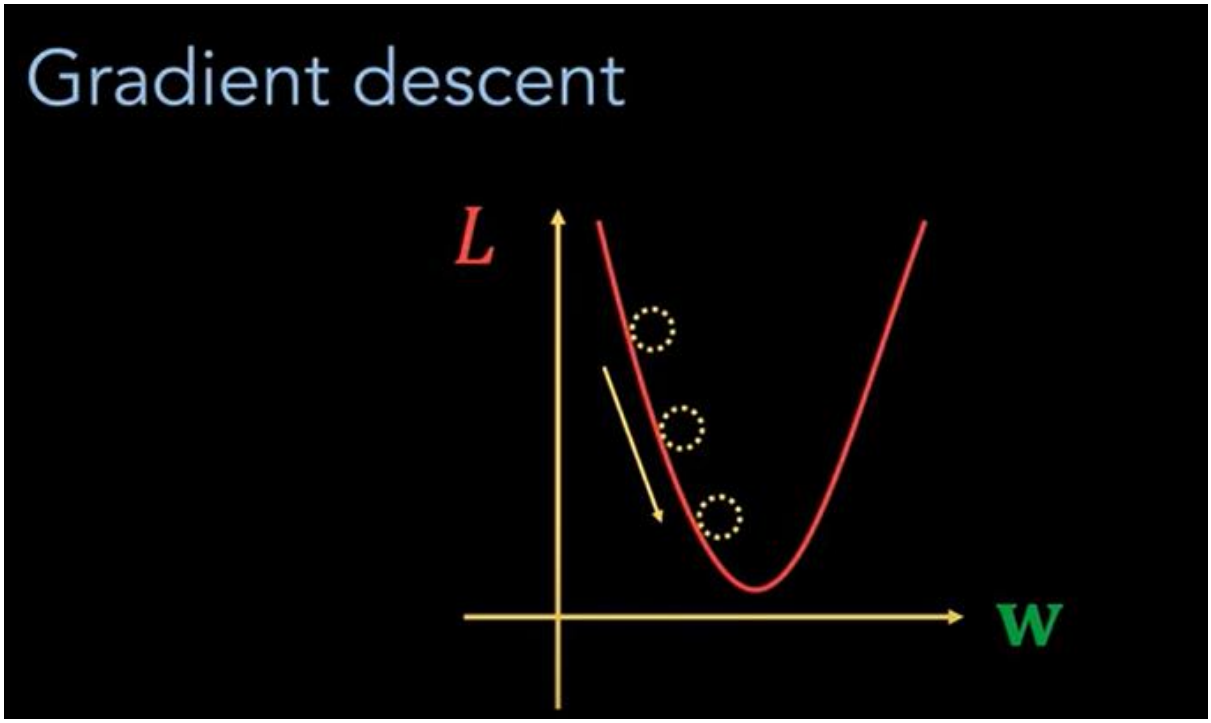
3) Types of optimizers

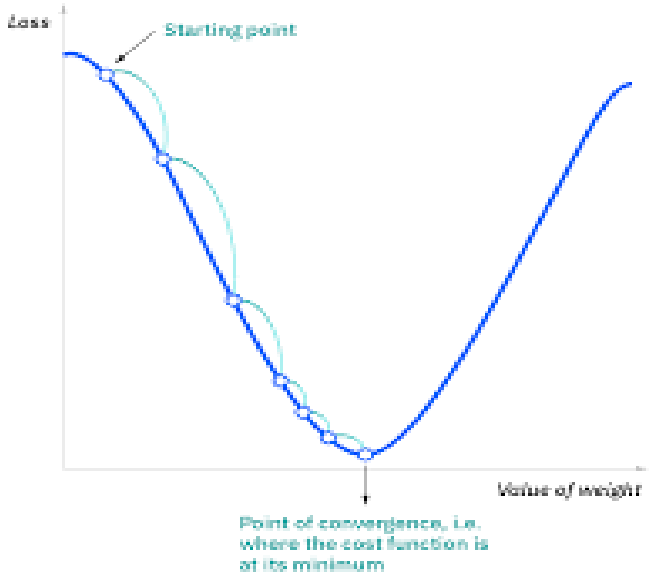
Let's learn about different types of optimizers and how they exactly work to minimize the loss function.

Gradient Descent

Gradient descent is an optimization algorithm based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum. Gradient

Descent iteratively reduces a loss function by moving in the direction opposite to that of steepest ascent. It is dependent on the derivatives of the loss function for finding minima. uses the data of the entire training set to calculate the gradient of the cost function to the parameters which requires large amount of memory and slows down the process.





Gradient descent works as follows:

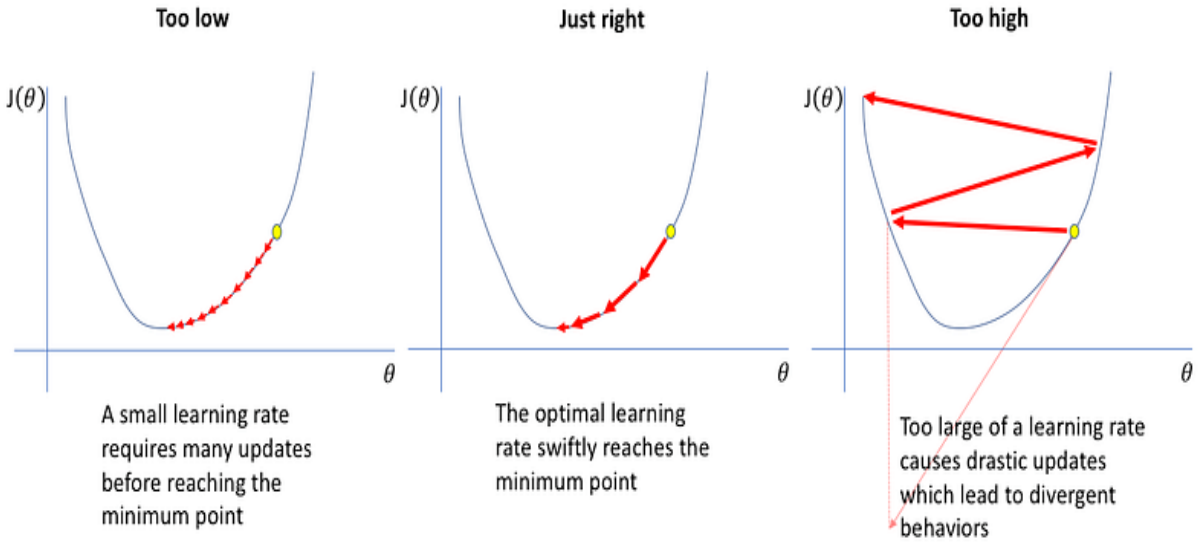
1. **Initialize Coefficients:** Start with initial coefficients.
2. **Evaluate Cost:** Calculate the cost associated with these coefficients.
3. **Search for Lower Cost:** Look for a cost value lower than the current one.

4. **Update Coefficients:** Move towards the lower cost by updating the coefficients' values.
5. **Repeat Process:** Continue this process iteratively.
6. **Reach Local Minimum:** Stop when a local minimum is reached, where further cost reduction is not possible.

$$W_{new} = W_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

Learning Rate

How big/small the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.



Types of Gradient Descent

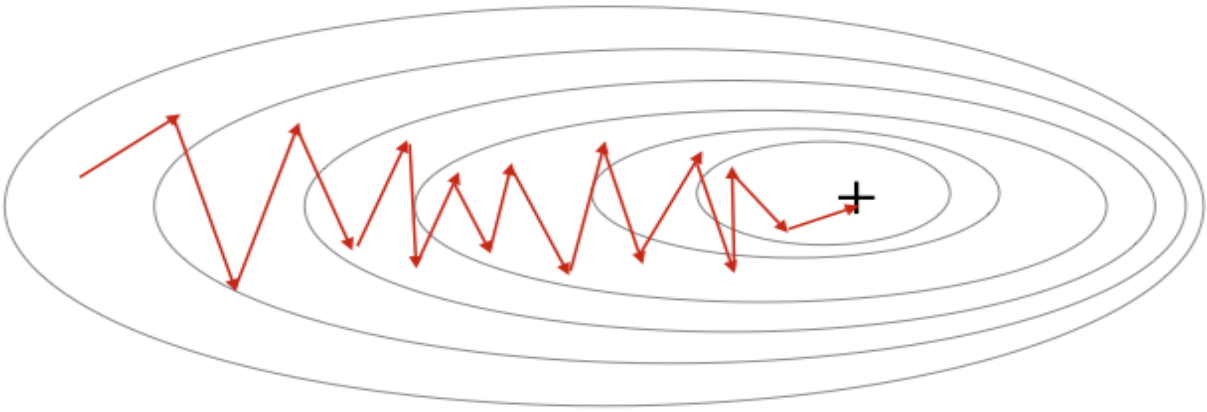
- **Batch** → sees whole dataset each step (accurate but slow).
- **Stochastic (SGD)** → one sample each step (fast but noisy).
- **Mini-batch** → balance (used in practice).

Example:

- Batch = reading the whole textbook before exam.
- Stochastic = studying one page, then testing.
- Mini-batch = studying 1–2 chapters at a time.

Stochastic Gradient Descent

It is a variant of Gradient Descent. It update the model parameters one by one. If the model has 10K dataset SGD will update the model parameters 10k times.



Advantages of Stochastic Gradient Descent

1. Frequent updates of model parameter
2. Requires less Memory.
3. Allows the use of large data sets as it has to update only one example at a time.

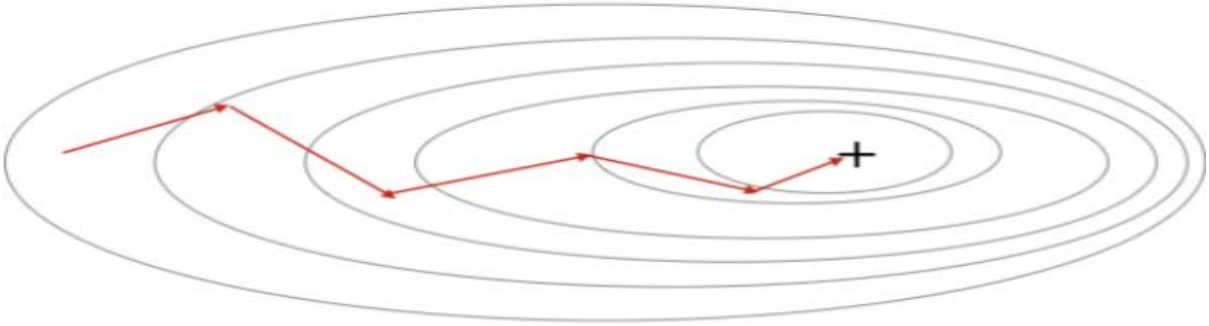
Disadvantages of Stochastic Gradient Descent

1. The frequent can also result in noisy gradients which may cause the error to increase instead of decreasing it.
2. High Variance.
3. Frequent updates are computationally expensive.

Mini-Batch Gradient Descent

It is a combination of the concepts of SGD and batch gradient descent. It simply splits the training dataset into small batches and performs an update for each of those batches. This creates a balance between the robustness of stochastic gradient descent and the efficiency of

batch gradient descent. it can reduce the variance when the parameters are updated, and the convergence is more stable. It splits the data set in batches in between 50 to 256 examples, chosen at random.



Advantages of Mini Batch Gradient Descent:

1. It leads to more stable convergence.
2. more efficient gradient calculations.
3. Requires less amount of memory.

Disadvantages of Mini Batch Gradient Descent

1. Mini-batch gradient descent does not guarantee good convergence,
2. If the learning rate is too small, the convergence rate will be slow. If it is too large, the loss function will oscillate or even deviate at the minimum value.

Other Optimizers

1. Momentum

Problem with GD: Slow, zigzags in valleys.

Imagine you are riding a bicycle uphill and downhill:

- At first, you pedal hard (like plain gradient descent).
- Each push only moves you a little.
- But once you gain speed, the bicycle keeps rolling forward even if you stop pedaling for a moment.

That forward carry = momentum.

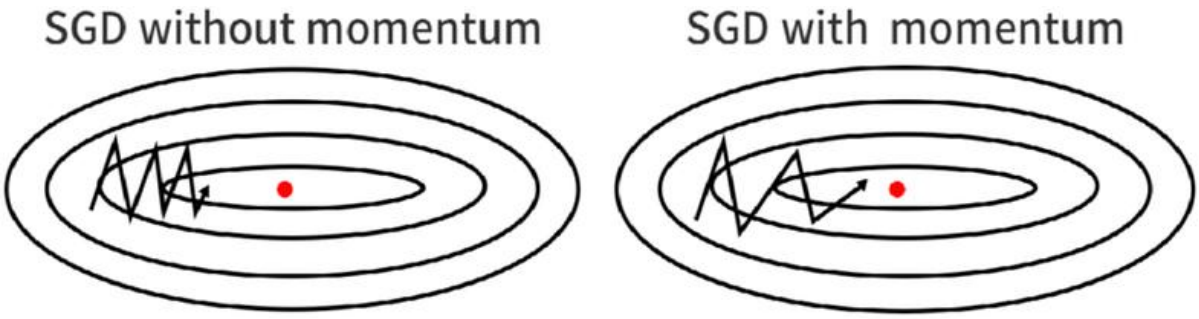
- Typing on your phone – when you swipe type, your finger doesn't stop at each letter.
- It carries forward smoothly with speed → that's momentum helping you move faster.

In training:

Momentum means the optimizer doesn't restart at every step; it uses the energy from previous steps to go faster and smoother.

1) SGD with Momentum

SGD with Momentum is a stochastic optimization method that adds a momentum term to regular stochastic gradient descent. Momentum simulates the inertia of an object when it is moving, that is, the direction of the previous update is retained to a certain extent during the update, while the current update gradient is used to fine-tune the final update direction. In this way, you can increase the stability to a certain extent, so that you can learn faster, and also have the ability to get rid of local optimization.



$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

Momentum Formula

Advantages of SGD with momentum

1. Momentum helps to reduce the noise.
2. Exponential Weighted Average is used to smoothen the curve.

Disadvantage of SGD with momentum

1. Extra hyperparameter is added.

2) AdaGrad(Adaptive Gradient Descent)

In all the algorithms that we discussed previously the learning rate remains constant. The intuition behind AdaGrad is can we use different Learning Rates for each and every neuron for each and every hidden layer based on different iterations.

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

Advantages of AdaGrad

1. Learning Rate changes adaptively with iterations.
2. It is able to train sparse data as well.

Disadvantage of AdaGrad

1. If the neural network is deep the learning rate becomes very small number which will cause dead neuron problem.

AdaGrad gives each parameter its **own learning rate**.

- Frequently updated weights → smaller steps.
- Rarely updated weights → bigger steps.

Example: A student preparing for exams – spends more time on weak subjects and less on strong ones.

Benefit: Great for sparse data (like text, embeddings).
Limitation: Learning rate keeps shrinking → may stop learning.

3) RMS-Prop (Root Mean Square Propagation)

RMS-Prop is a special version of Adagrad in which the learning rate is an exponential average of the gradients instead of the cumulative sum of squared gradients. RMS-Prop basically combines momentum with AdaGrad.

$$cache_{new} = \gamma * cache_{old} + (1 - \gamma) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

Advantages of RMS-Prop

1. In RMS-Prop learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.

Disadvantages of RMS-Prop

1. Slow Learning

RMSProp fixes it by **forgetting very old updates**.
It balances learning rate, so the model keeps learning.

- **Example**: While jogging, you don't carry yesterday's tiredness forever; you focus on recent energy levels.
- **Benefit**: Works well for RNNs and deep networks.

4) AdaDelta

Adadelta is an extension of Adagrad and it also tries to reduce Adagrad's aggressive, monotonically reducing the learning rate and remove decaying learning rate problem. In Adadelta we do not need to set the default learning rate as we take the ratio of the running average of the previous time steps to the current gradient.

Advantages of Adadelta

1. The main advantage of AdaDelta is that we do not need to set a default learning rate.

Disadvantages of Adadelta

1. Computationally expensive

5) Adam(Adaptive Moment Estimation)

Adam optimizer is one of the most popular and famous gradient descent optimization algorithms. It is a method that computes adaptive learning rates for each parameter. It stores both the decaying average of the past gradients , similar to momentum and also the decaying average of the past squared gradients , similar to RMS-Prop and Adadelta. Thus, it combines the advantages of both the methods.

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{S_{dw_t} - \varepsilon}} * V_{dw_t}$$

$$b_t = b_{t-1} - \frac{\eta}{\sqrt{S_{db_t} - \varepsilon}} * V_{db_t}$$

Advantages of Adam

1. Easy to implement
2. Computationally efficient.

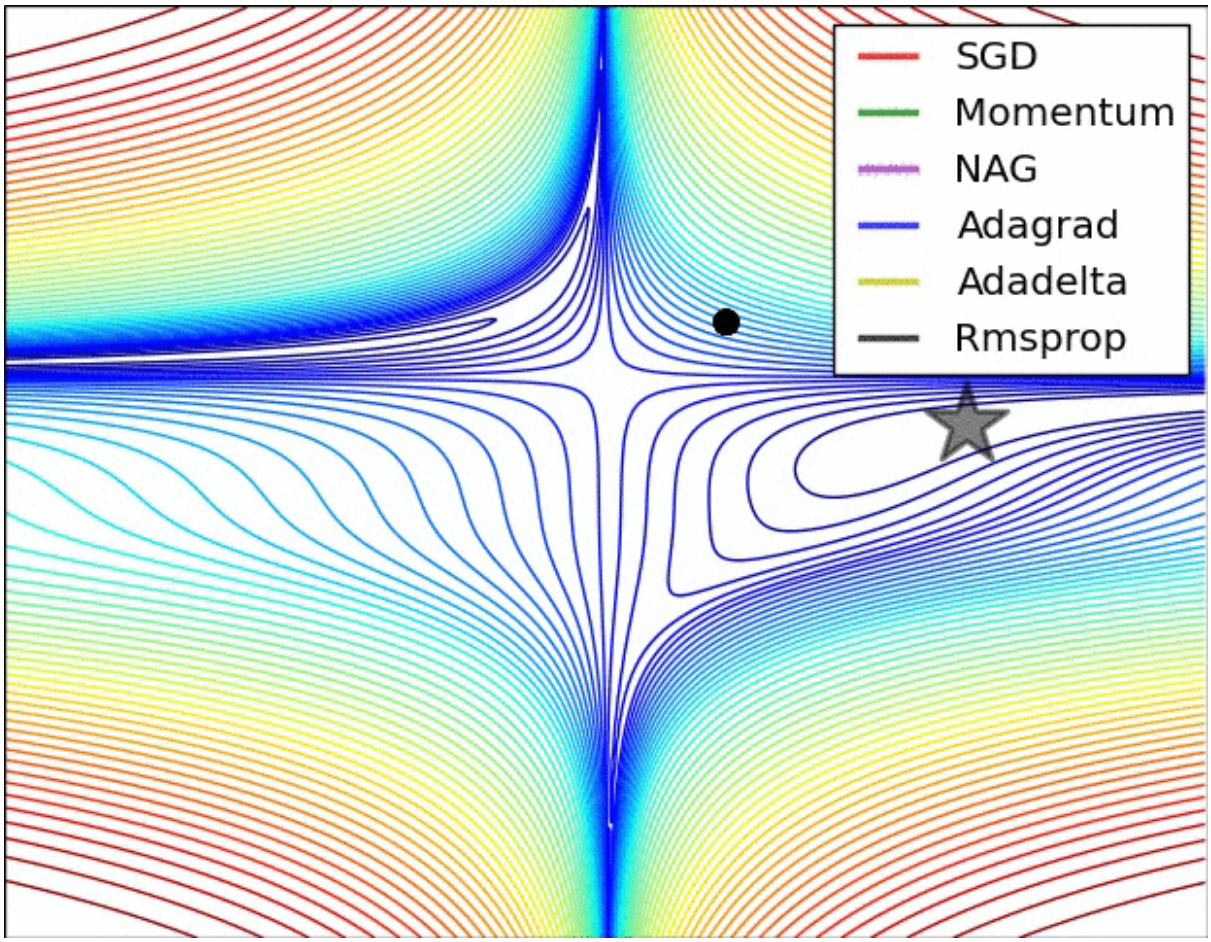
3. Little memory requirements.

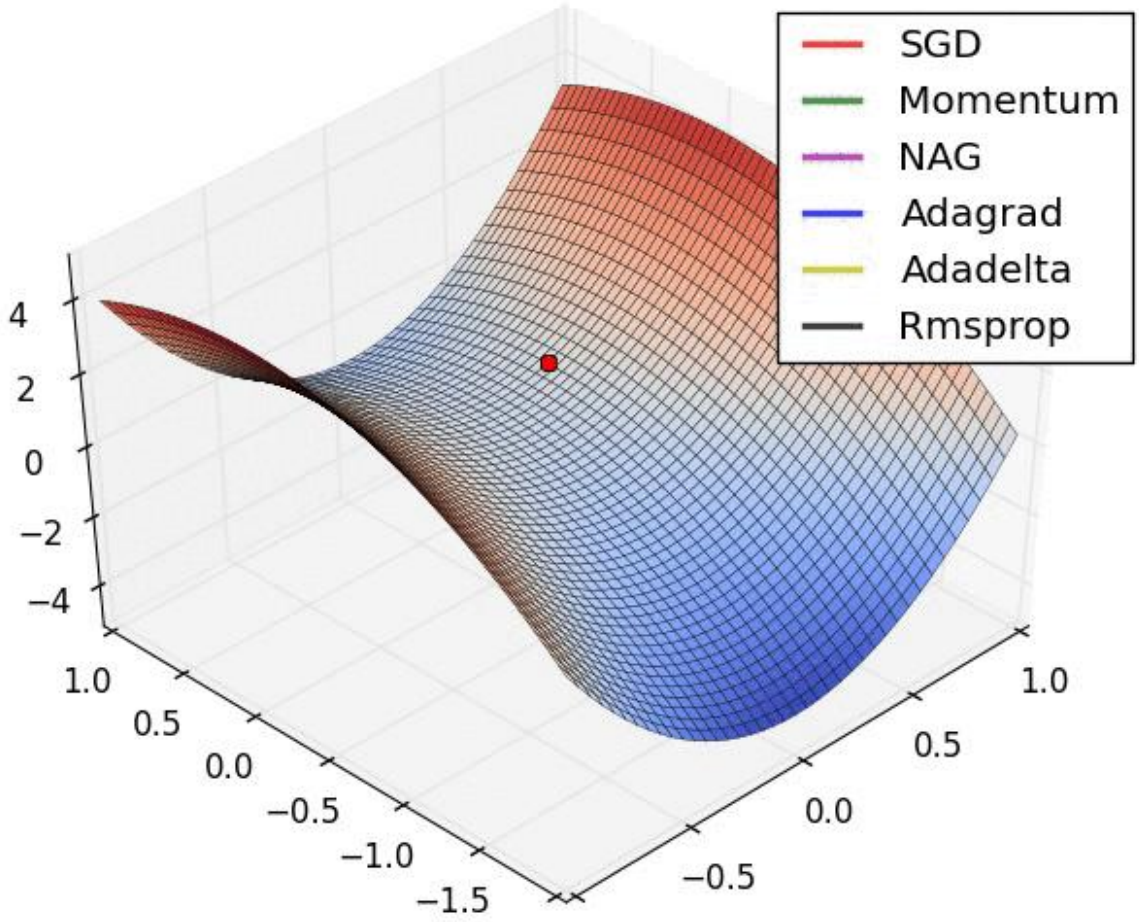
Adam = **Momentum + RMSProp combined**.

- Uses speed from momentum.
- Uses adaptiveness from RMSProp.

Example: A smart student → studies regularly (momentum) but also changes strategy depending on which subjects need more attention (adaptive).

Benefit: Fast, stable, works well for most problems → default optimizer in practice.





4) choosing an optimizer

stochastic gradient descent is a simple and efficient optimizer that is widely used, but it may need help to converge on problems with complex, non-convex loss functions. On the other hand, Adam is a more sophisticated optimizer that combines the ideas of momentum and adaptive learning rates and is often considered one of the most effective optimizers in deep learning.

Here are a few pointers to keep in mind when choosing an optimizer:

- Understand the problem and model architecture, as this will help you determine which optimizer is most suitable
- Experiment with different optimizers in deep learning to see which one works best for your problem
- Adjust the hyperparameters of the optimizer, such as the learning rate, to see if it improves performance
- Remember that the optimizer's choice is not the only factor affecting model performance.

- Other important factors include the choice of architecture, the quality of the data, and the amount of data available.

Summary:

- If the data is sparse, use the self-applicable methods, namely Adagrad, Adadelta, RMSprop, Adam.
- RMSprop, Adadelta, Adam have similar effects in many cases.
- Adam just added bias-correction and momentum on the basis of RMSprop,
- As the gradient becomes sparse, Adam will perform better than RMSprop.