Neural-Networks-3

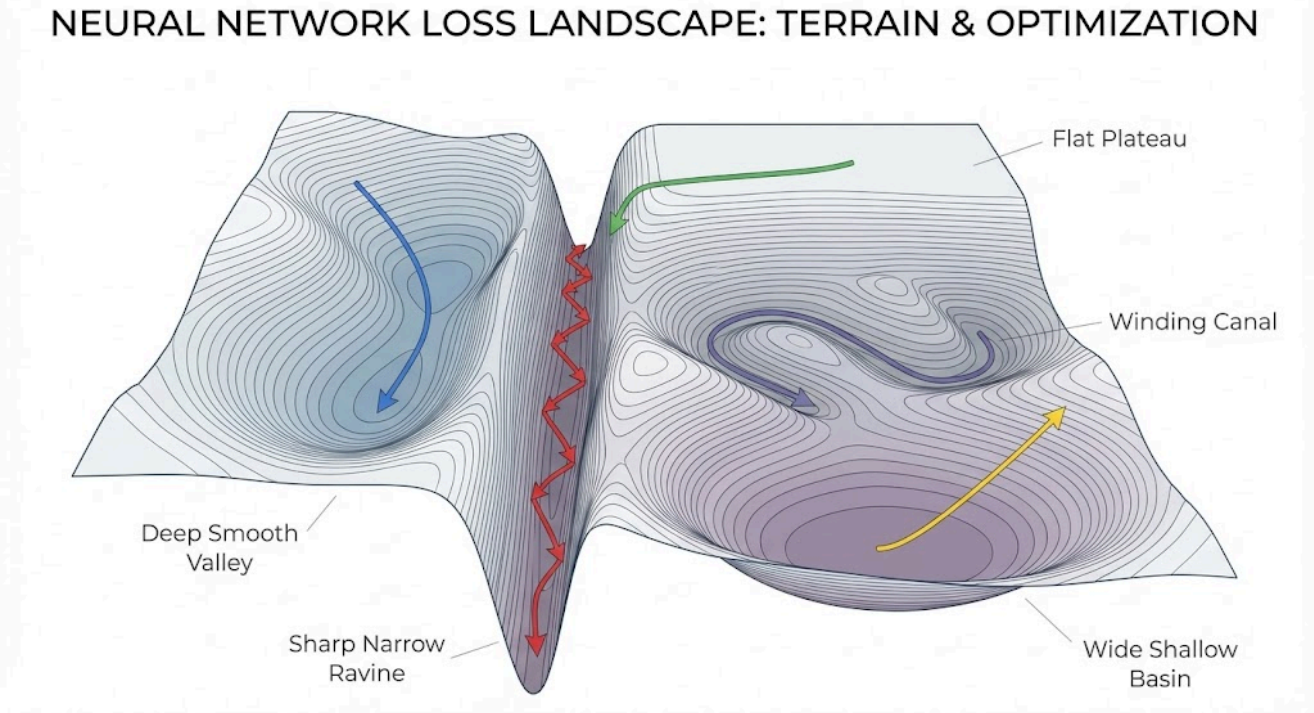
Before moving to main content few more numpy and linear algebra ;)
Colab Solutions

Images that are not cited can be found in the colab file. so do checkout there.

Open this colab files along side the Note.
Colab File : for algorithms code and the comparisons
Good Vizualization blog : to help you vizualize the points written with animation
Playlist suggestion-1 : can refer for more understanding from a good teacher
Playlist suggestion-2 : same as above
Vizualization tool : You can install to vizualize locally in your laptop
Only watch relevant lecture not every. (recall the concept hell issue)

Understand what a few loss function landscape terms

Different types of regions of loss function terrain



1. Valley

A **valley** is a region of the loss surface where:

1. the loss decreases smoothly along one or more directions
2. curvature is **moderate and well-conditioned**
3. gradients are consistent and point toward a minimum
   Formally,

- Hessian eigenvalues are **positive and similar in scale**
- Condition number is relatively low

2. Ravine

A **ravine** is a narrow valley with **highly anisotropic curvature**, meaning:

- extremely steep curvature in one direction
- very flat curvature in the orthogonal direction
  Formally:
- Hessian eigenvalues differ by several orders of magnitude
- Ill-conditioned surface

3. Canal

A **canal** is a curved or winding low-loss path where:

- gradient direction changes continuously
- curvature is moderate but **directionally unstable**
- the optimal descent direction bends over space
  Formally
- Hessian eigenvectors rotate along the trajectory
- Gradient direction is not aligned across steps

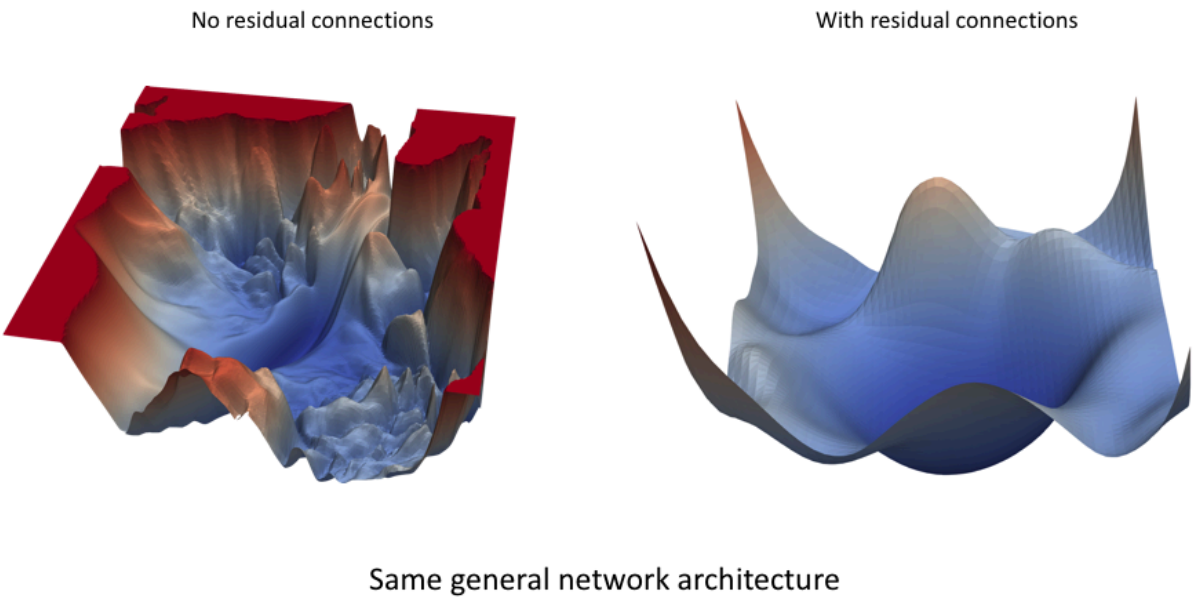
4. Flat region

A **flat region (plateau)** is a region where:

- gradients are near zero in all directions
- curvature is near zero
- loss changes very slowly across space
  Formally:
- Gradient ≈ 0
- Hessian eigenvalues ≈ 0

Homework

#homework



Source: Link
Try to vizualize your self the terrain terms from these images above.

Different types of points in the graph

1. Saddle point

is a point in the loss surface where the **gradient is zero**, but the point is **neither a local minimum nor a local maximum**. In different directions around the point, the function curves upward in some directions and downward in others.
Formally

$$\nabla f = 0$$

Hessian has both positive and negative eigenvalues

Why it is important

High-dimensional loss surfaces are full of saddle points, not local minima.
Optimizers slow down here because gradients vanish even though descent is possible.

2. Minima

1. Local Minima

A **local minimum** is a point where the function value is **lower than all nearby points**, but not necessarily the lowest value overall.
Formally,

$$\begin{aligned} &\nabla f = 0 \\ &f(x) \leq f(x+\epsilon) \qquad \text{for all sufficiently small } \epsilon \end{aligned}$$

and Hessian is positive semi-definite.

2. Global Minima

A **global minimum** is a point where the function value is **lower than or equal to all other points in the entire domain**.
Formally,

$$f(x^*) \leq f(x) \quad \text{for all x in domain}$$

Why it matters ?

Convex problems guarantee a global minimum. Deep learning usually does not, yet often behaves as if it does. That’s the mystery.

2. Maxima

1. Local Maxima

A **local maximum** is a point where the function value is **higher than all nearby points**, but not necessarily the highest value overall.
Formally,

$$\begin{aligned} &\nabla f(x) = 0 \\ &f(x) \geq f(x+\epsilon), \qquad \text{for all sufficiently small } \epsilon \end{aligned}$$

Hessian is negative semi definite

Why it matters ?

Optimizers rarely stay here. If they do, something is broken or learning rate is absurd.

2. Global Maxima

A **global maximum** is a point where the function value is **higher than or equal to all other points in the entire domain**.
Formally,

$$f(x^*) \geq f(x), \qquad \text{for all x in domain}$$

Why it matters ?

Mostly theoretical in loss minimization, but useful for understanding dual problems, energy-based models, and completeness of critical-point analysis.

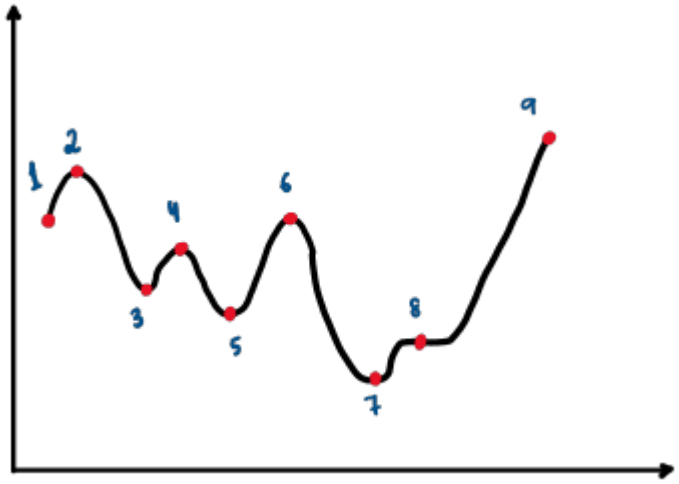
> Single gradient is 0 => it can be minima, maxima or saddle. so zero gradient don't tell us much we have to further investigate to understand it's behavior before any verdict.

> but curvature tells us everything.

Homework

#homework

Correctly identify the points as Saddle point, global minima, local minima, global maxima and local maxima



Optimization Algorithms (Step by Step)

as we have seen few terminologies required for understanding the history of optimization algorithm.

so we will start with

0. Backpropagation

It is a way to **compute gradients efficiently**.
Before backprop, we couldn't even *see* the loss landscape slope in high dimensions.

1. Gradient Descent

There are 3 types of Gradient descent.

1. Batch gradient descent (Vanilla gradient descent)
2. Stochastic gradient descent
3. Min-batch gradient descent

Batch gradient Descent

the algorithm
INPUT:
1. cost function $J(\theta)$,
2. learning rate η,
3. number of epochs N

INITIALIZE:
4. Parameters : θ
COMPUTE:
for i = 0 to N-1 do:
5. Compute the gradient of $J(\theta)$ w.r.t θ . (i.e. $\nabla_\theta J(\theta)$ for all training example)
6. update the parameter θ as

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

code:

> below we used mse as loss function but in the end we will see modular codes to adapt any loss function in optimizers with gradients efficiently.

```
from typing import Tuple
import numpy as np


def batch_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float = 0.01,
    n_iterations: int = 1000,
) -> np.ndarray:
    """
    Perform Batch Gradient Descent for linear regression.

    This function minimizes the Mean Squared Error (MSE) loss:
        J(theta) = (1/m) * ||X theta - y||^2

    using batch gradient descent, where gradients are computed
    using the entire dataset at each iteration.

    Args:
        X (np.ndarray):
            Feature matrix of shape (m, n),
            where m is the number of samples and n is the number of
features.
        y (np.ndarray):
            Target vector of shape (m,) or (m, 1).
        theta (np.ndarray):
            Parameter vector of shape (n,) or (n, 1).
        learning_rate (float, optional):
```

```
            Step size used to update parameters. Defaults to 0.01.
        n_iterations (int, optional):
            Number of gradient descent iterations. Defaults to 1000.

    Returns:
        np.ndarray:
            Optimized parameter vector theta with the same shape as input.
    """

    # Number of training examples
    m: int = X.shape[0]

    # Ensure y is a column vector for consistent matrix operations
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    for _ in range(n_iterations):
        # Step 1: Compute model predictions (Xθ)
        predictions: np.ndarray = X @ theta

        # Step 2: Compute residual errors (Xθ - y)
        errors: np.ndarray = predictions - y

        # Step 3: Compute gradient of MSE loss w.r.t. theta
        gradients: np.ndarray = (2 / m) * (X.T @ errors)

        # Step 4: Update parameters using gradient descent rule
        theta -= learning_rate * gradients

    return theta
```

The problem with BGD is:

1. we need to calculate the gradients for the whole dataset to perform just *one* update
2. batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. (for large datasets)
3. Batch gradient descent also doesn't allow us to update our model *online*, i.e. with new examples on-the-fly. (think about the reason)

But comes up with some good stuffs as well :

1. It guaranteed to converge to global minimum for convex error surface and to a local minima for non-convex surfaces. (see below Concept - 4 in last section for definition of convex error surface and non convex ones)

Stochastic gradient descent

Stochastic gradient descent perform parameter update for each training example.
Here parameter update happens as:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

What it solves from batch gradient descent ?
Batch Gradient Descent

- Computes the gradient using **the entire dataset** before **one** parameter update
- One update per epoch. Slow
- Deterministic update direction. Same data, same step, every time.
- Painfully expensive for large datasets. Memory-heavy, compute-heavy.
  it's like no update parameter before looking at all data

Stochastic Gradient Descent

- Computes gradient using **one training example at a time**.
- **Updates parameters immediately** after each example.
- Many updates per epoch. Chaotic. Impulsive. Somehow effective.
- No need to load the whole dataset into memory.
- Naturally supports **online learning**.
  Why Stochastic gradient is faster ?
- as it starts learning immediately.
- Noise helps it escape flat regions and shallow local minima (how we will see)
  #implementation

algorithm:

INPUT:

- Dataset ($X \in \mathbb{R}^{m \times n}$)
- Targets ($y \in \mathbb{R}^m$)
- Learning rate (η)
  INITIALIZE:
- Parameters ($\theta \in \mathbb{R}^n$)
  COMPUTE:
  Repeat for multiple epochs

1. Shuffle the training data
2. For each training example ((x_i, y_i)):
    1. Predict

$$\hat{y}_i = x_i^\top \theta$$

2. Compute the Loss $J(\theta)$
3. Compute the gradient $J(\theta)$ w.r.t θ (i.e. $\nabla_\theta J(\theta, x_i, y_i)$)
4. Update parameters

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta, x_i, y_i)$$

Code

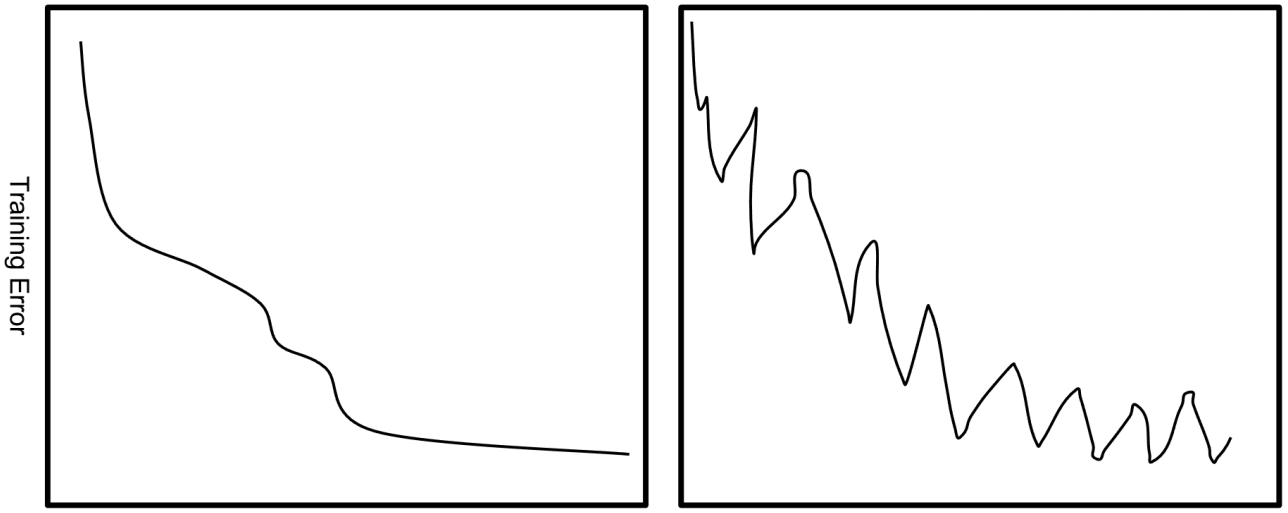
> below we used mse as loss function but in the end we will see modular codes to adapt any loss function in optimizers with gradients efficiently.

```python
import numpy as np
from typing import Tuple


def stochastic_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float = 0.01,
    n_epochs: int = 50,
    shuffle: bool = True,
) -> np.ndarray:
    """
    Perform Stochastic Gradient Descent (SGD) for linear regression
    using Mean Squared Error (MSE) loss.

    Each parameter update is performed using a single training example.

    Args:
        X (np.ndarray):
            Feature matrix of shape (m, n).
        y (np.ndarray):
            Target vector of shape (m,) or (m, 1).
        theta (np.ndarray):
            Parameter vector of shape (n,) or (n, 1).
        learning_rate (float, optional):
            Step size for parameter updates. Defaults to 0.01.
        n_epochs (int, optional):
            Number of passes over the dataset. Defaults to 50.
        shuffle (bool, optional):
            Whether to shuffle data at the beginning of each epoch.
            Defaults to True.
```

```
    Returns:
        np.ndarray:
            Optimized parameter vector theta.
    """

    # Ensure correct shapes
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    m = X.shape[0]

    for _ in range(n_epochs):

        # Optionally shuffle data to avoid cyclic patterns
        if shuffle:
            indices = np.random.permutation(m)
            X = X[indices]
            y = y[indices]

        # Iterate over each training example
        for i in range(m):
            x_i = X[i].reshape(1, -1)
            y_i = y[i]

            # Step 1: Prediction
            y_pred = x_i @ theta

            # Step 2: Error
            error = y_pred - y_i

            # Step 3: Gradient of MSE loss for one sample
            gradient = 2 * x_i.T @ error

            # Step 4: Parameter update
            theta -= learning_rate * gradient

    return theta
```

Aspect	Batch GD	SGD
Gradient	Uses all samples	Uses one sample
Noise	Smooth	Noisy, jittery

Aspect	Batch GD	SGD
Speed per update	Slow	Fast
Convergence	Stable	Oscillatory
Escapes saddle points	Poor	Better



Gradient Descent **Stochastic Gradient Descent**

Let's talk about the zig zag nature of SGD
in stochastic gradient descent the estimate of gradient is noisy, the weights may not move precisely down the gradient at each iteration. but author said this "noise" at each iteration can be advantageous.
in SGD,
Each data point defines its *own* loss surface. it's the approximate gradient of whole dataset. it calculate

$$\nabla l_i(\theta)$$

which is a loss of single data point
So when you do SGD, you are effectively:

- Optimizing a *different* surface at every step
- Jumping between conflicting gradient directions
  Consequence
- Direction flips constantly
- Large variance in updates
- Overshooting in steep directions
  Geometric effect
- You bounce across narrow valleys
- You rarely move straight down the valley
- Path looks jagged and chaotic
  Where as in Batch gradient descent,

it uses whole data each time and give true gradient
here it uses

$$\nabla L(\theta) = \frac{1}{m} \sum_{t=1}^{m} \nabla l_i(\theta)$$

Geometric effect

- Each step points roughly in the same direction as the previous one
- Trajectory follows the valley floor
- No oscillations unless the learning rate is stupidly large

Mini batch gradient descent

it take best of both, it take the advantage of stability from training a batch (small) instead of a single data example and advantage of speed and memory efficient by using mini-batch for training.
it take best of both, it take the advantage of stability from training a batch (small) instead of a single data example and advantage of speed and memory efficient by using mini-batch for training.

compare the stochastic gradient descent parameter update with this

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

where it computes gradient of n-examples at a time instead of a single example.

according to author This way it

1. reduces the variance of the parameter updates, which can lead to more stable convergence #implementation
2. can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.

> Common mini-batch size varies between 50-256. but can vary for different applications.

Algorithm

INPUT:

- Dataset ($X \in \mathbb{R}^{m \times n}$)
- Targets ($y \in \mathbb{R}^m$)
- Learning rate (η)
- Batch size (**B**)
  INITIALIZE:

- Parameters ($\theta \in \mathbb{R}^n$)
  COMPUTE:
  Repeat for multiple epochs

1. Shuffle the training data
2. Split data into mini-batches of size BBB
3. For each training Batch ((X_b, y_b)):
    1. Predict

$$\hat{y}_b = X_b^\top \theta$$

    2. Compute the error $J(\theta)$.
    3. Compute the gradient $J(\theta)$ w.r.t θ (i.e. $\nabla_\theta J(\theta, x_i, y_i)$)
    4. Update parameters

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta, X_b, y_b)$$

code

```python
import numpy as np


def mini_batch_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float = 0.01,
    n_epochs: int = 50,
    batch_size: int = 32,
    shuffle: bool = True,
) -> np.ndarray:
    """
    Perform Mini-Batch Gradient Descent for linear regression
    using Mean Squared Error (MSE) loss.

    Parameters are updated using gradients computed on small
    subsets (mini-batches) of the training data.

    Args:
        X (np.ndarray):
            Feature matrix of shape (m, n).
        y (np.ndarray):
            Target vector of shape (m,) or (m, 1).
        theta (np.ndarray):
```

```
            Parameter vector of shape (n,) or (n, 1).
        learning_rate (float):
            Step size for parameter updates.
        n_epochs (int):
            Number of passes over the dataset.
        batch_size (int):
            Number of samples per mini-batch.
        shuffle (bool):
            Whether to shuffle data at the start of each epoch.

    Returns:
        np.ndarray:
            Optimized parameter vector theta.
    """

    # Ensure correct shapes
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    m = X.shape[0]

    for _ in range(n_epochs):

        # Shuffle data to prevent learning order bias
        if shuffle:
            indices = np.random.permutation(m)
            X = X[indices]
            y = y[indices]

        # Iterate over mini-batches
        for start_idx in range(0, m, batch_size):
            end_idx = start_idx + batch_size

            X_batch = X[start_idx:end_idx]
            y_batch = y[start_idx:end_idx]

            # Step 1: Forward pass
            predictions = X_batch @ theta

            # Step 2: Compute batch errors
            errors = predictions - y_batch

            # Step 3: Compute gradient of MSE for the batch
```

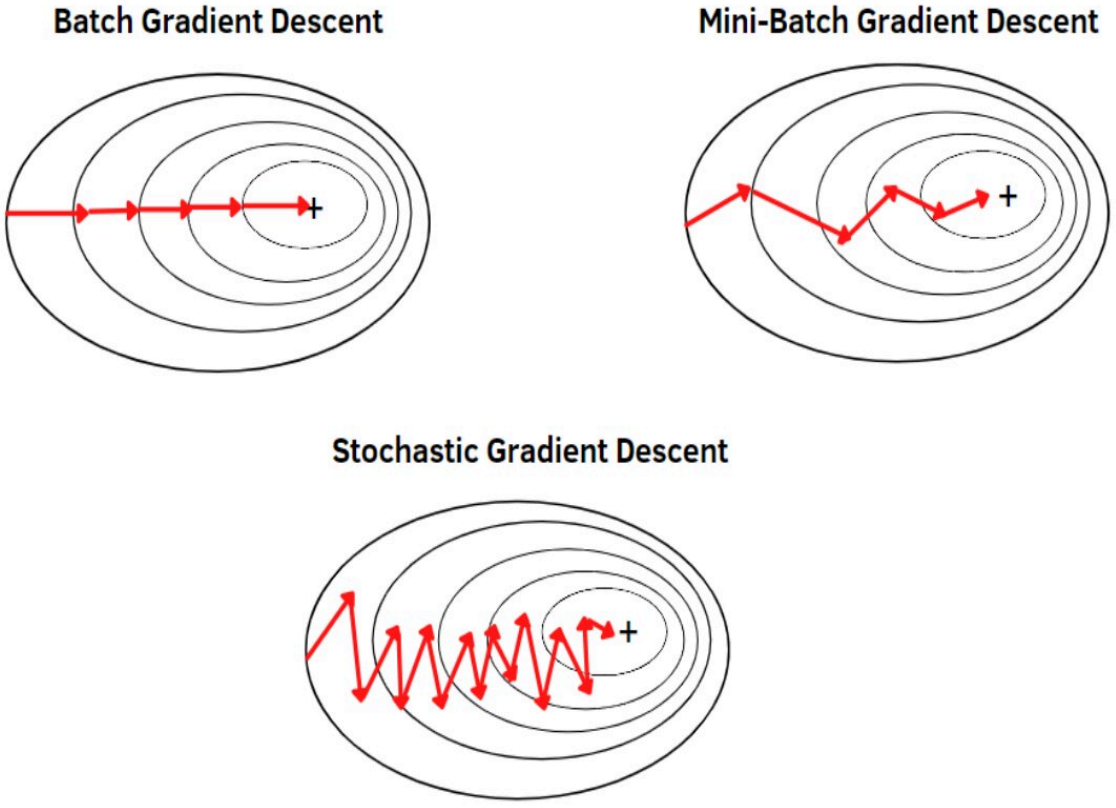
```
            batch_size_actual = X_batch.shape[0]
            gradients = (2 / batch_size_actual) * X_batch.T @ errors

            # Step 4: Parameter update
            theta -= learning_rate * gradients

    return theta
```

but in mini batch,
it is an average of noisy gradients
as it calculates

$$\frac{1}{B} \sum_{i \in batch} \nabla l_i(\theta)$$

Consequence

- Variance is reduced by roughly 1/B
- Direction is more reliable
- Noise is still present, but controlled
  Geometric effect
- Oscillations shrink
- Forward progress improves
- Still enough noise to escape bad curvature



Source: Link

Here we can see it have less zig-zag nature as compare to stochastic gradient descent.

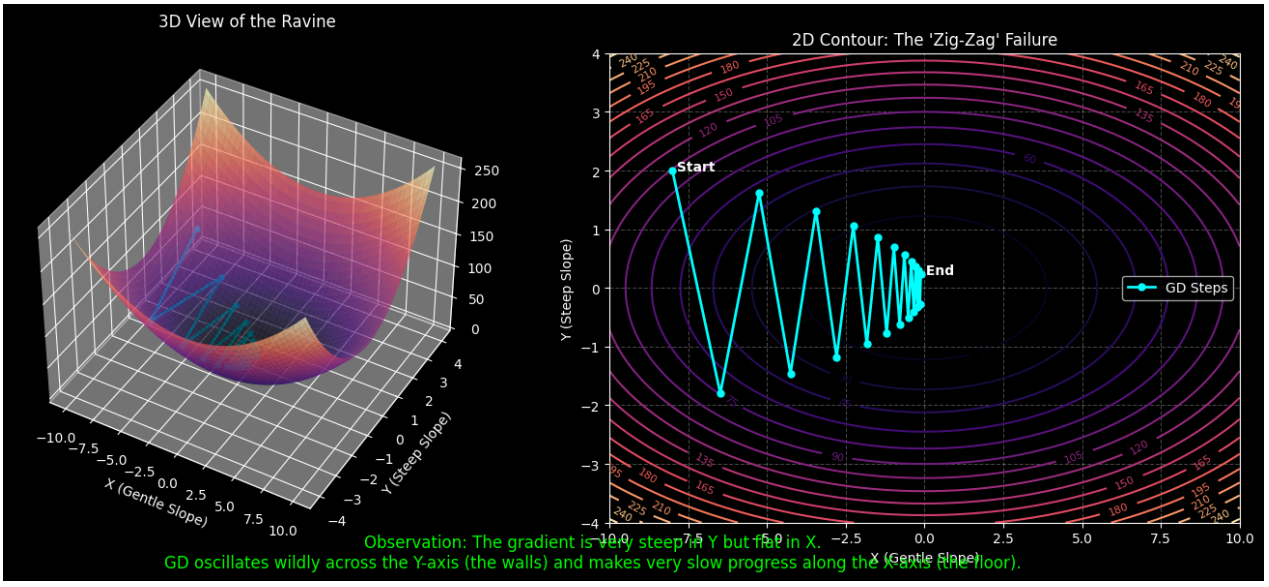
Strength

This will converge to Global minima for the case of convex function and to local minima in the case of non-convex function.

Weakness

we are talking about gradient descent. it means all gradient descent types.

1. Oscillation (Zig-zags violently in ravines)



Left is 3d plot to vizualize the terrain and right to show the zigzag nature

the elipses are not difficult to read (see the concept - 1 for clearly understand how to read such graphs) and the blue line oscillation too much that it stops learning. takes too much steps to learn.

here the function we are using is
$x^2 + 10y^2$ is a simple convex function, but the coefficient `10` creates an elliptical contour (the ravine).

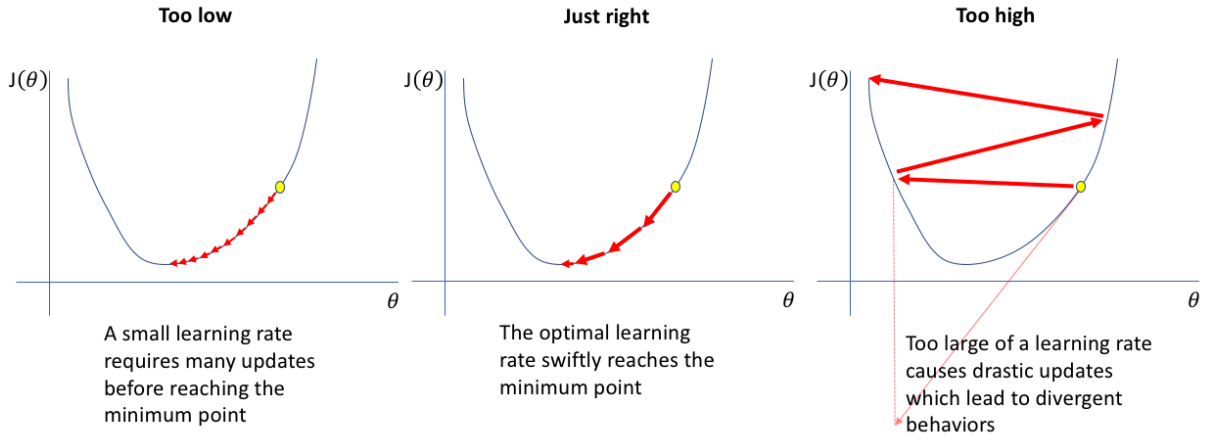
The Learning Rate: I set `lr = 0.095`.

- For the X-axis (coefficient 1), the optimal rate is roughly `1/2 = 0.5`. So `0.095` is **too slow** for X.
- For the Y-axis (coefficient 10), the limit for stability is `1/10 = 0.1`. So `0.095` is **dangerously close to instability** for Y

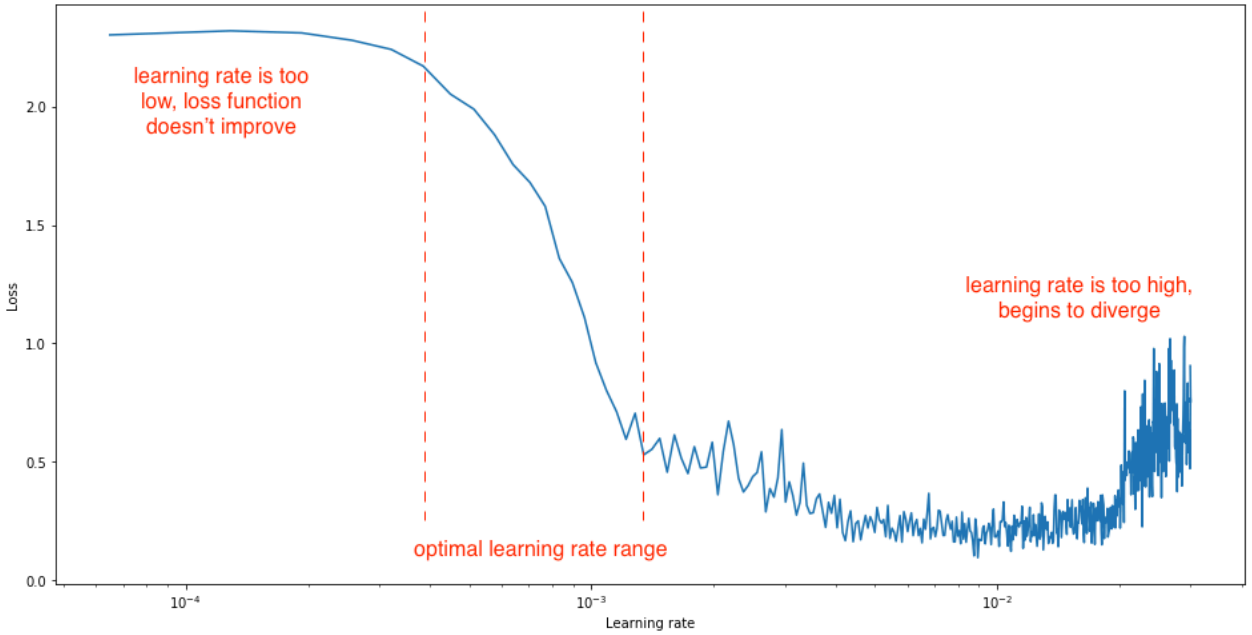
2. Sensitive to learning rate

    - Choosing a proper learning rate can be difficult.
    - Additionally, the same learning rate applies to all parameter updates. if our data is sparse (Data is sparse when most of the entries are zero, missing, or inactive.) and

features have varies frequencies then we might not update all of them to the same extent, but perform a larger update for rarely occurring features.



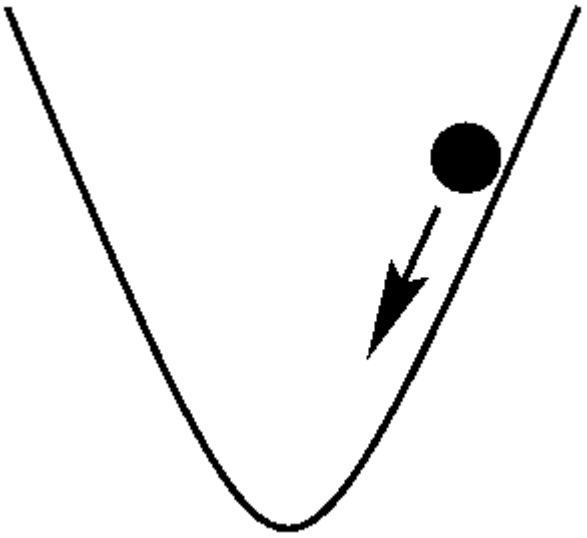
Source: Link



There is a good block for suggesting cyclic learning rate: Link a good read.

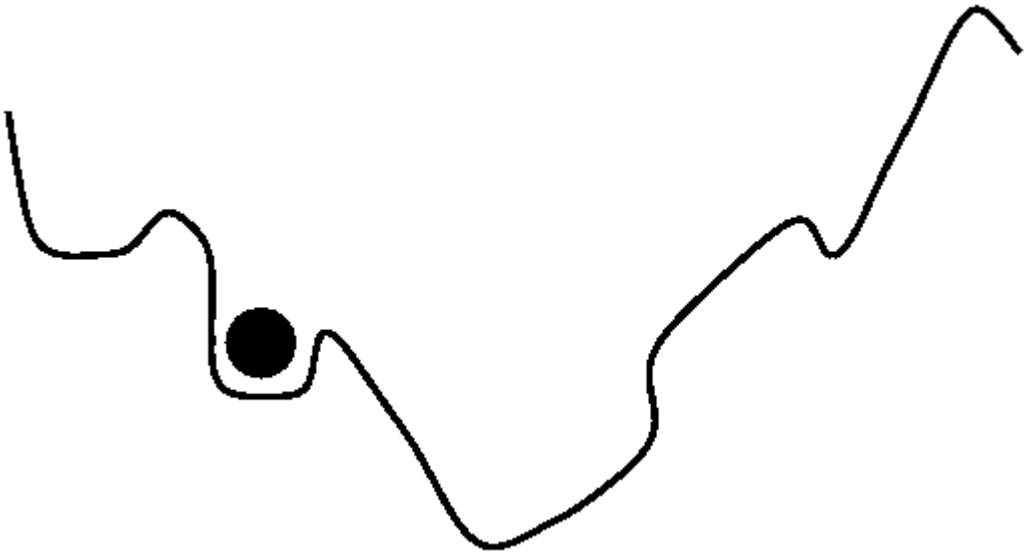
3. minimizing highly non-convex error functions => getting trapped in their numerous suboptimal local minima.

- Convex



source: Link

- Non convex



Source: Link

Solution

for non-convex function we can't stop at local minima right ? we need to converge in global minima. for that we have to somehow take ourselves out from the local minima.

Here comes the momentum. A way to dampen the oscillations and speed up velocity in the correct direction.

2. Momentum (with SGD)

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum

We already saw above in weakness section of gradient descent.

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations.

You love physics. so, here is a concept revisit.
instead of updating with help of gradients we also accounts for previous gradient (or direction) (like a memory)
so formally,
There are 2 implementations , we will see online

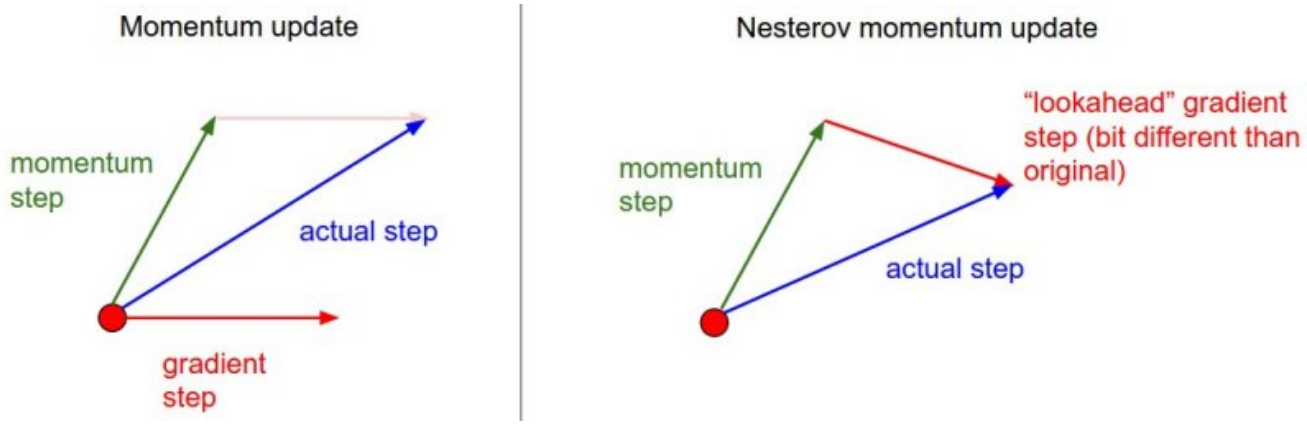
$$v_t = \beta v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta_{t+1} = \theta_t - v_t$$

or

$$v_t = \beta v_{t-1} + (1-\beta) \nabla_\theta J(\theta)$$
$$\theta_{t+1} = \theta_t - \eta v_t$$

instead of
we saw in SGD $\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta)$. we added an extra term which remember the fraction of previous gradients (where fraction is because of β which lies between 0 and 1) but the specific $\beta = 0.9$ or 0.99 according to the author.

we can see the 2 vectors in v_t ,
βv_{t-1} as momentum step vector and
$\eta \nabla_\theta J(\theta)$ as gradient step
and their vector addition is our actual step. we can vizualize that below(left one)



Physics analogy:
Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. γ<1). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

algorithm

INPUTS:

- Dataset ($X \in \mathbb{R}^{m \times n}$)
- Targets ($y \in \mathbb{R}^m$)
- Learning rate (η)
- Momentum Coefficient : (β)
- Number of iteration : (N)
  INITIALIZATION:
  Initial parameter (θ_0)
  Initial velocity :

$$v_0 = 0$$

  COMPUTE:
  For t = 0 to N-1 :

1. Predict

$$\hat{y}_b = X_b^\top \theta$$

2. Compute Loss $J(\theta)$
3. Compute gradient: $\nabla_\theta J(\theta)$
4. Update velocity:

$$v_t = \beta v_{t-1} + \eta \nabla_\theta J(\theta)$$

5. Update parameters:

$$\theta_{t+1} = \theta_t - v_t$$

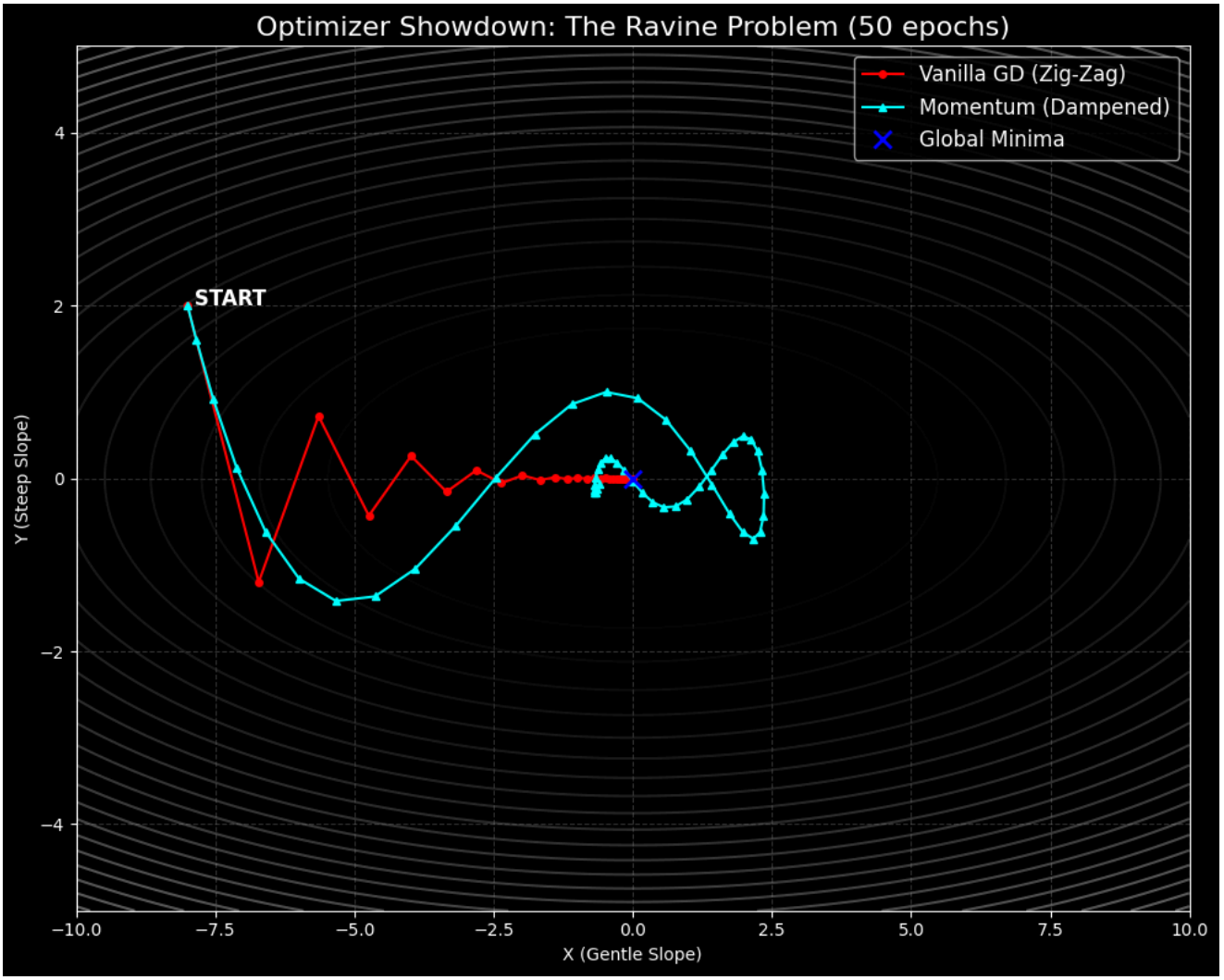
Code

```python
import numpy as np

def momentum_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float,
    beta: float,
    n_iterations: int,
    batch_size: int,
) -> np.ndarray:
    """
    Gradient Descent with Momentum for Mean Squared Error (MSE).
```

```
    Args:
        X (np.ndarray):
            Input data of shape (m, n).
        y (np.ndarray):
            Targets of shape (m,) or (m, 1).
        theta (np.ndarray):
            Initial parameters of shape (n,) or (n, 1).
        learning_rate (float):
            Learning rate (η).
        beta (float):
            Momentum coefficient (β).
        n_iterations (int):
            Number of iterations.
        batch_size (int):
            Mini-batch size.

    Returns:
        np.ndarray:
            Optimized parameters.
    """

    m = X.shape[0]

    # Ensure column vectors
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    # Initialize velocity
    velocity = np.zeros_like(theta)

    for _ in range(n_iterations):

        # ---- 1. Sample mini-batch ----
        indices = np.random.choice(m, batch_size, replace=False)
        X_b = X[indices]
        y_b = y[indices]

        # ---- 2. Predict ----
        y_hat = X_b @ theta

        # ---- 3. Compute gradient of MSE ----
        gradient = (2 / batch_size) * X_b.T @ (y_hat - y_b)
```

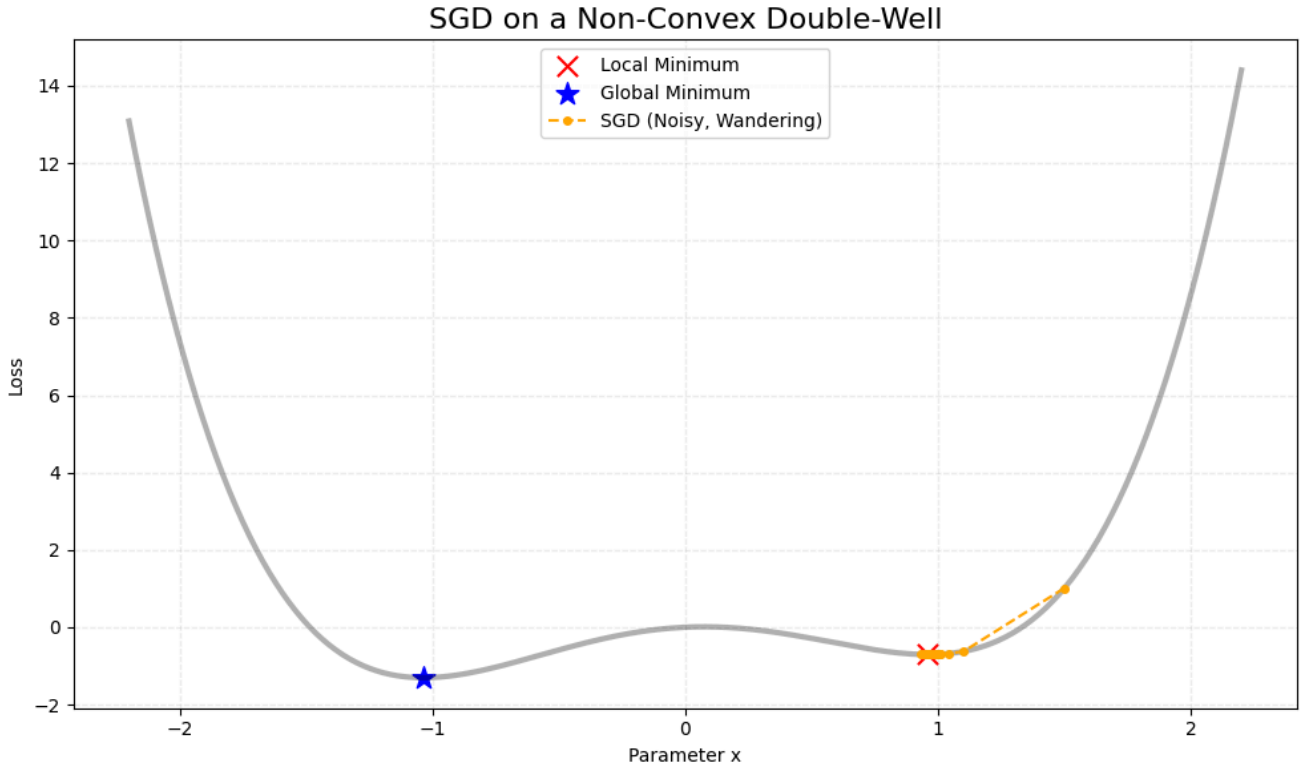
```
        # ---- 4. Update velocity ----
        velocity = beta * velocity + learning_rate * gradient

        # ---- 5. Update parameters ----
        theta = theta - velocity

    return theta
```

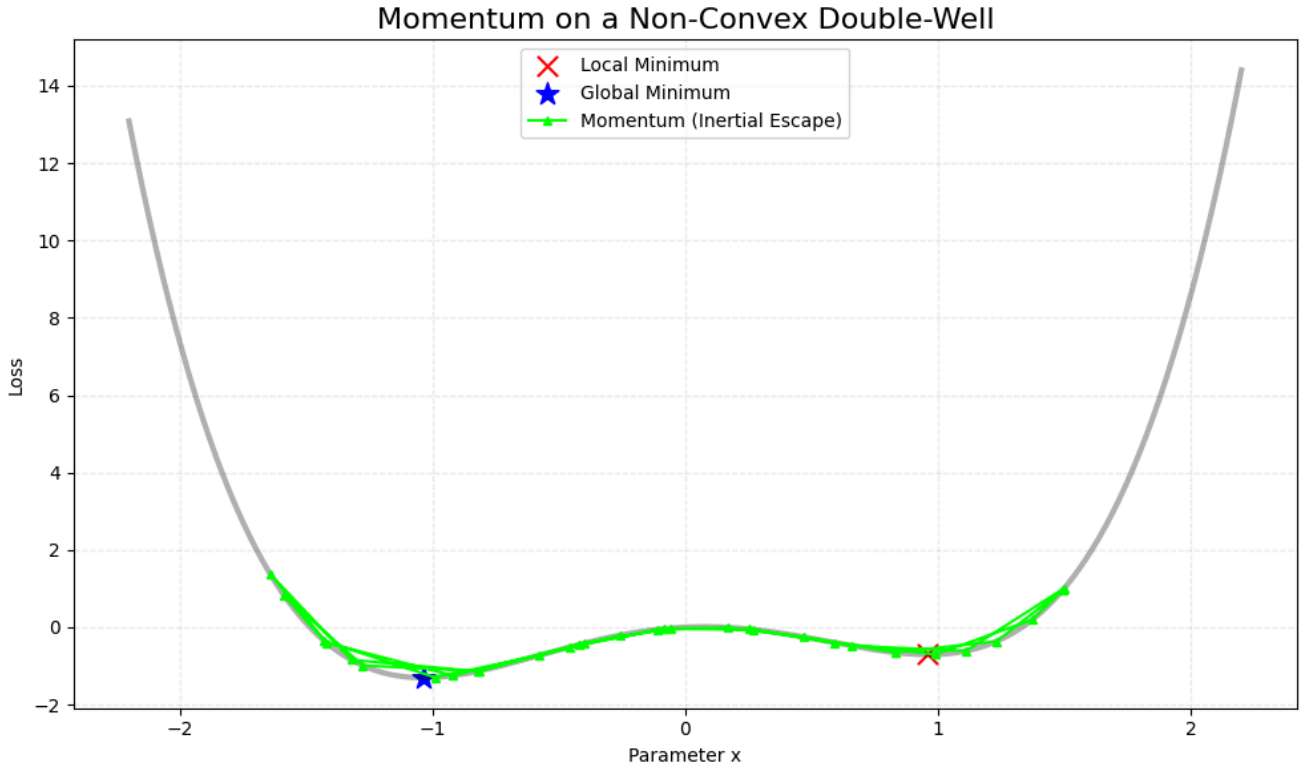
“Vanilla GD oscillates because gradients flip sign in steep directions, while momentum accumulates velocity along consistent directions, reducing wasted updates.”



Solving the Oscillation issue.

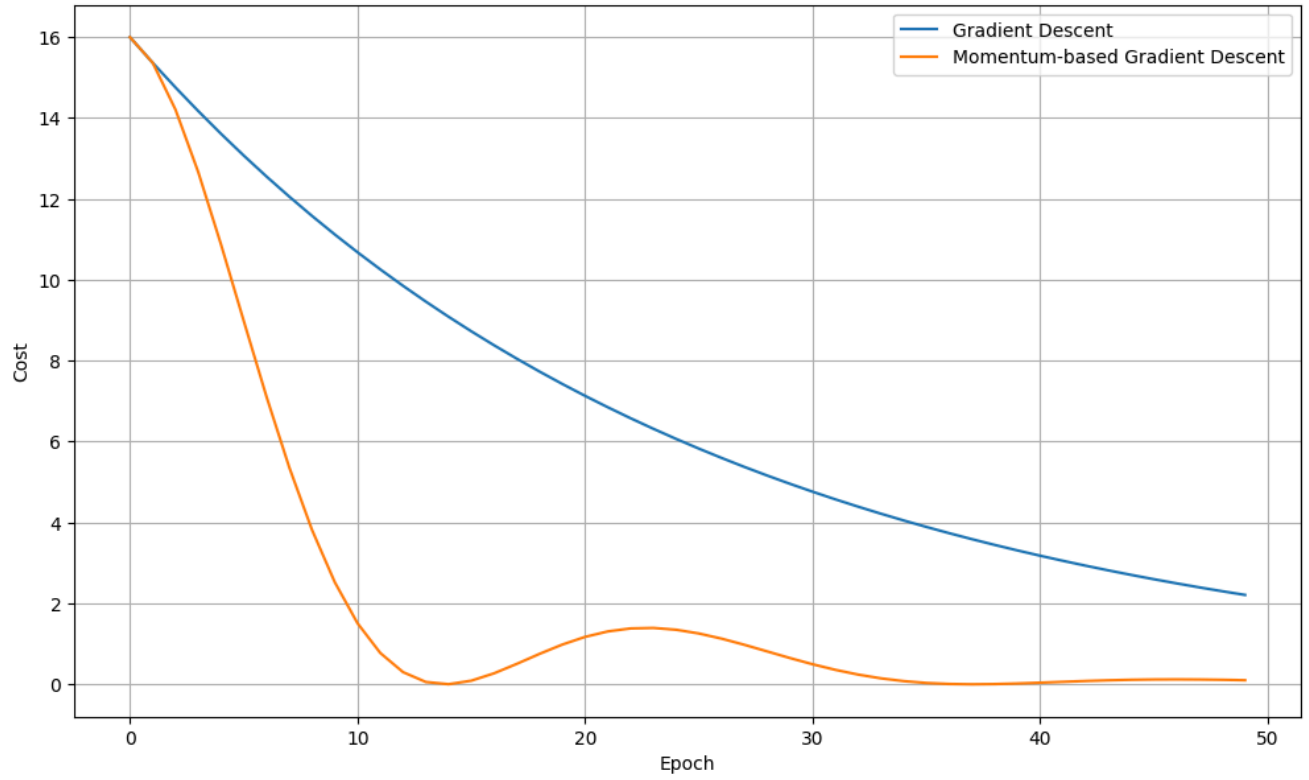
Solving the local minima issue





Here momentum converges to global minima overcoming the issue of local minima. Like in physics the "intertial" kept it move in the same direction from start point : 1.5 in x axis. the "intertia" helped to overcome short valley.

the efficiency and faster convergence can be seen here



code can be found in colab for all of these.

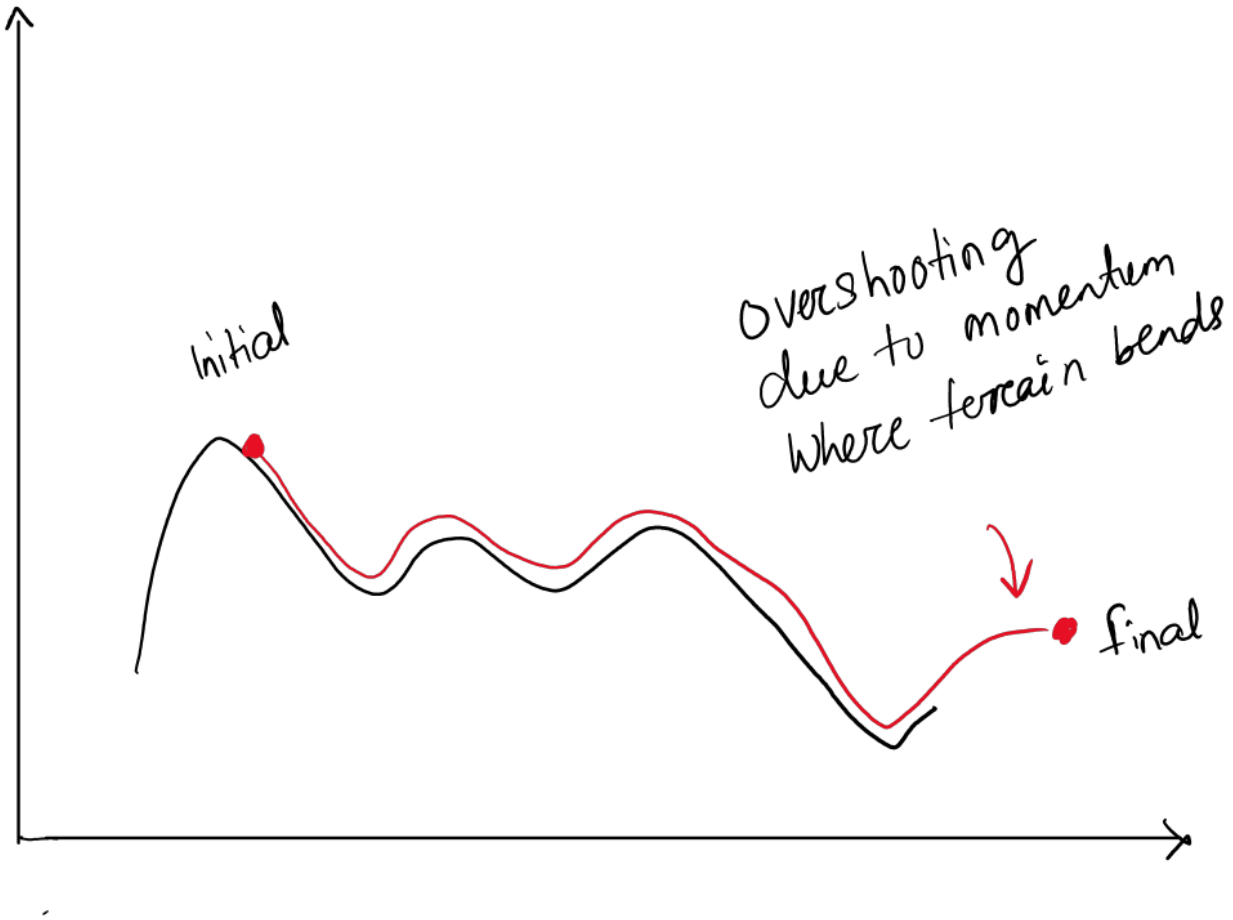
Strength

1. Overcome Ravines with repeated oscillations (unlike the SGD)
2. Powers through flat areas (saddle points) and reduces oscillation.
3. Consistent gradient direction
4. Accelerates convergence along shallow axis

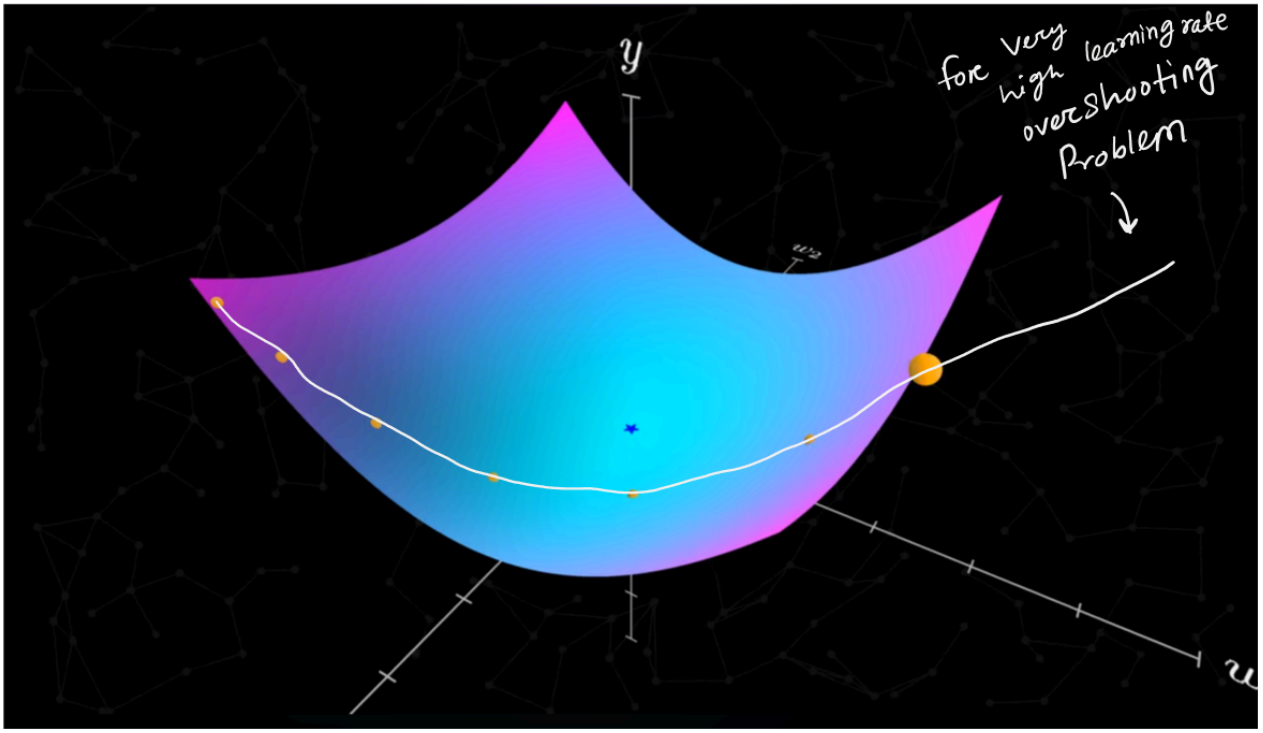
Weakness

- Still one global learning rate (see the concept-5 for problems with single global learning rate)
- Still blind to per-parameter scale

- Overshoots when terrain bends



- Struggles with uneven feature scales
- Overshooting minima
- Poor handling of sparse gradients
- if learning rate is too high then still overshoot



Visualize here : [at timestamp]

Let's talk a little more about Momentum before moving to next

1. **Noise averaging**
   Plain GD reacts to *every* gradient like it's gospel. Momentum has a memory and therefore some skepticism.

- In mini-batch SGD, gradients are noisy.
- GD zigzags randomly.
- Momentum acts like a low-pass filter over gradients.

Mathematically, the velocity is an **exponential moving average of gradients**. That means:

- random noise cancels out
- consistent directions accumulate
  Scene: Flat-ish region with noisy gradient arrows. #implementation
  Strength: more stable convergence under stochasticity, not just faster.

2. **Directional persistence**
   in narrow valley, curved ravines, banana shaped loss counter : Momentum build directional confidence, continues moving even when the gradient temporarily weakens for non-zero velocity.
   Strength: Momentum approximates second-order behavior *without computing Hessians*.
3. **Faster traversal of plateaus and flats**
   In high-dimensional NN loss surfaces:

- many regions have near-zero gradients
- not minima, just *uninformative flats*
  GD:
- gradient ≈ 0
- progress ≈ dead

Momentum:

- carries velocity from previous informative regions
- coasts through flats

4. Enables higher effective learning rates
   With momentum:
   - you can use a higher learning rate without divergence
   - GD would explode

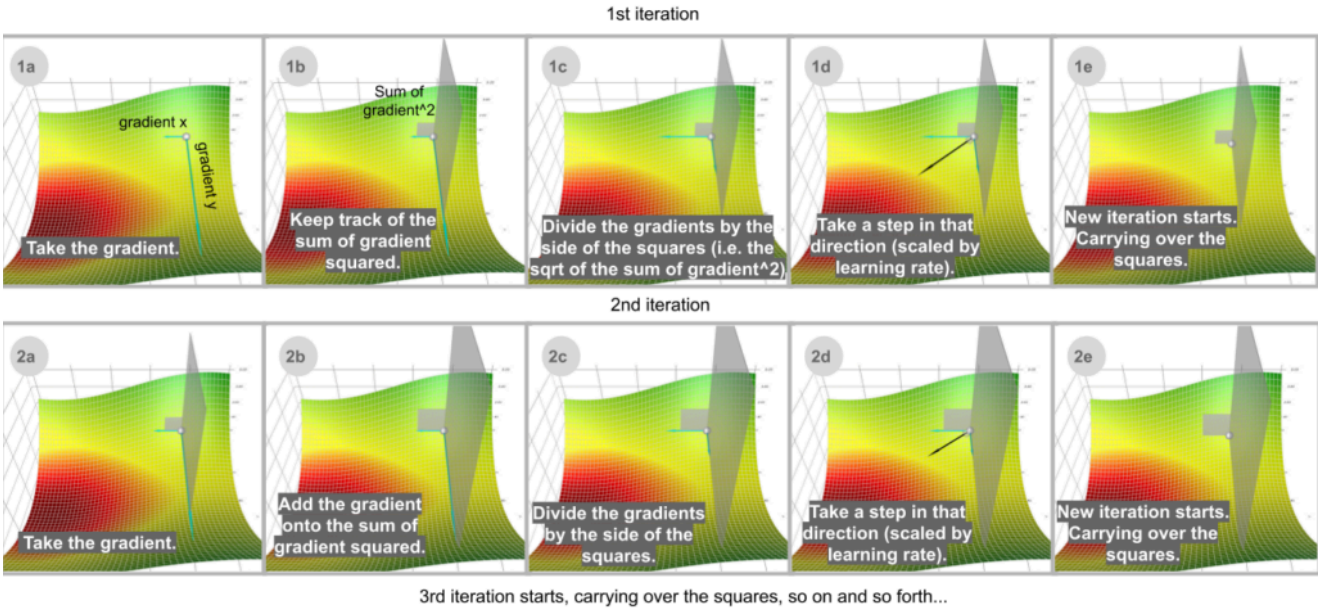
Solution

Single global learning rate, poor handling of sparse features solved by adagrad by implementing Per-parameter learning rates where Step size ∝ 1 / sqrt(sum of past

gradients²).

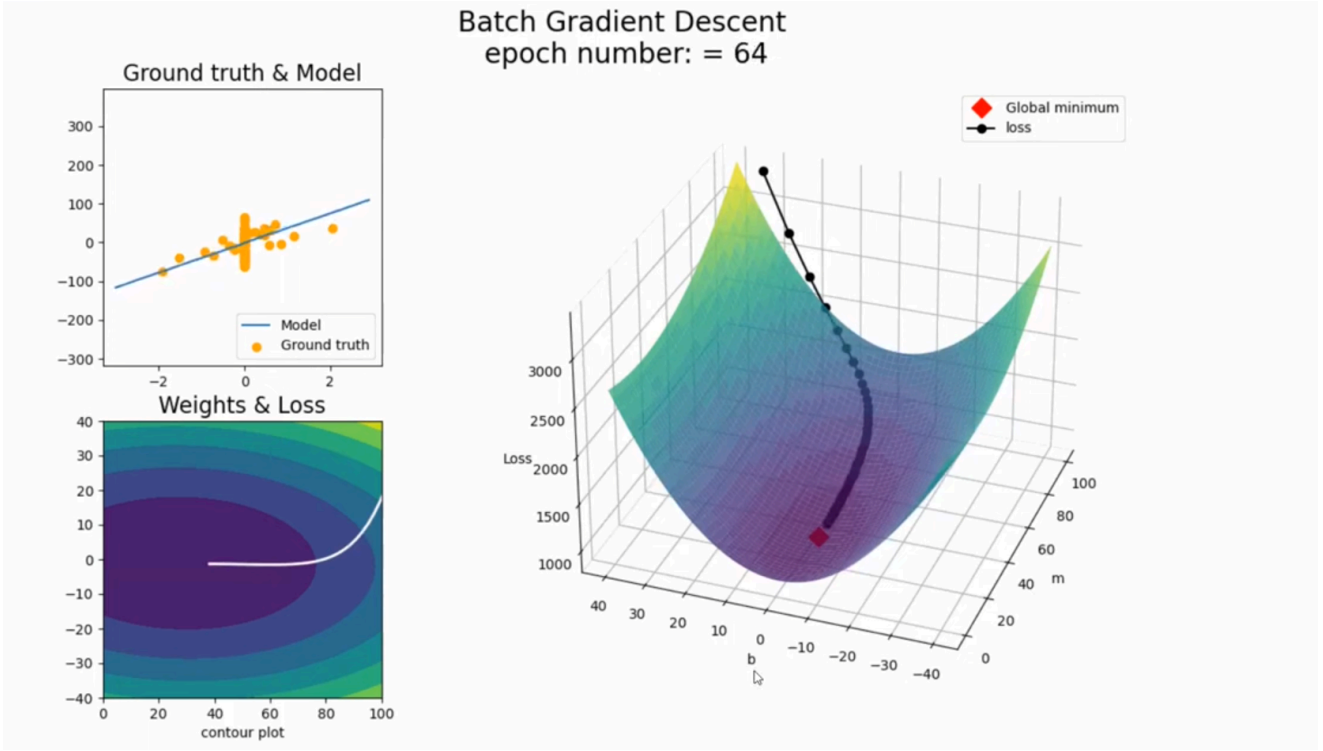
3. AdaGrad

adaptive gradient



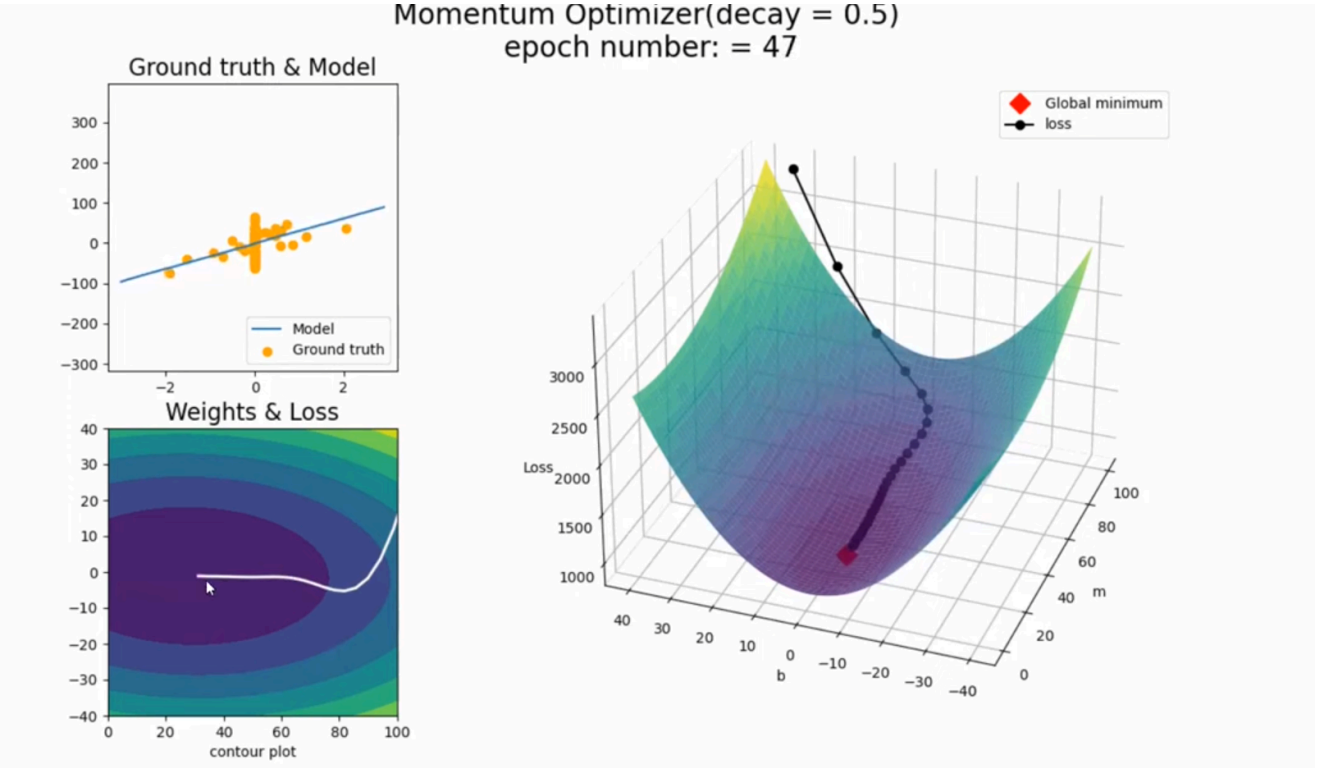
Let's visualize what it is solving.
imagine a case where there are 2 features one get frequent updates and another rarely update. so we can say ultimately
feature-1 (here b) : dense feature
feature-2 (here m) : rare feature or sparse feature

Where see how Batch gradient descent case

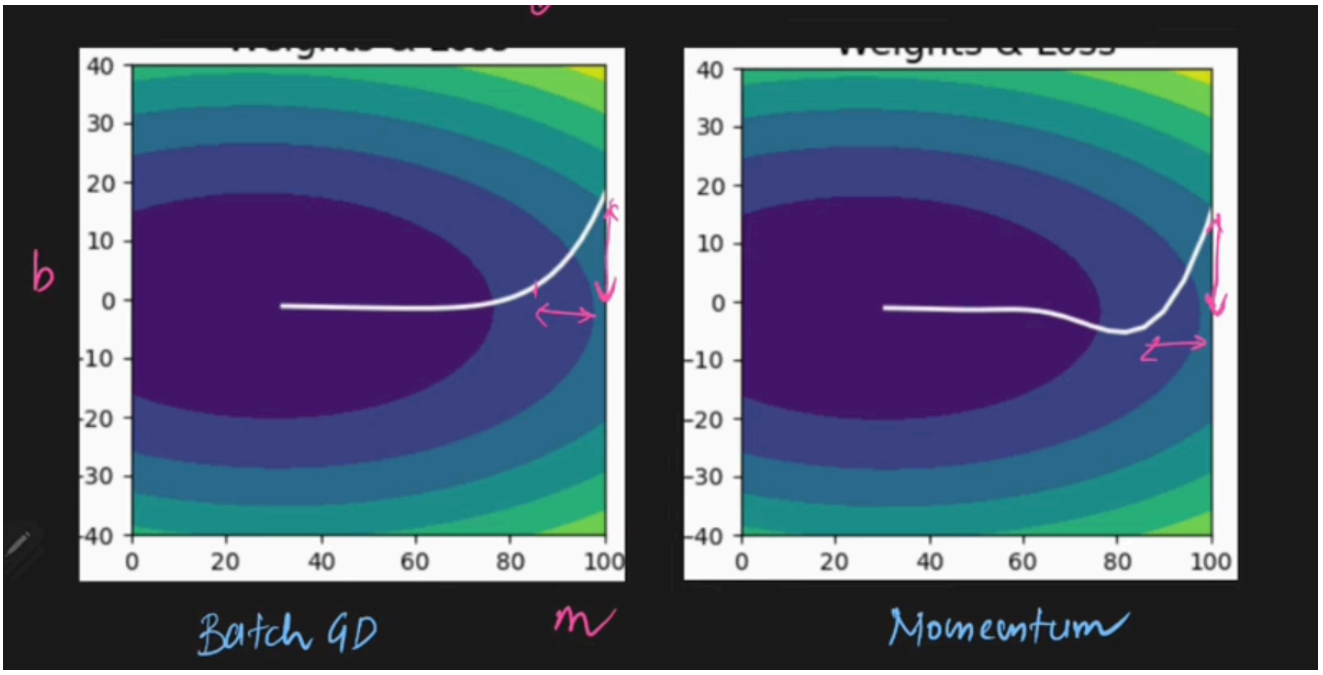


Source: Link

and the momentum performs like this

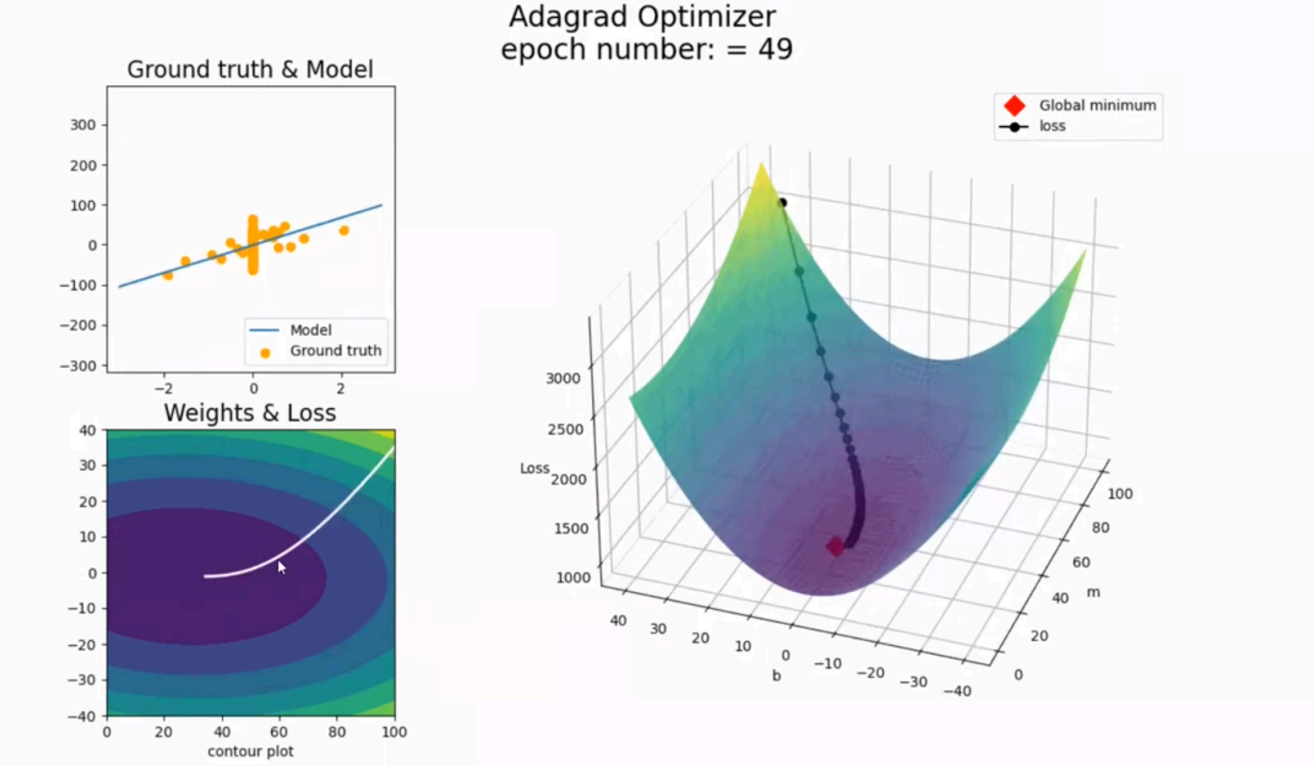


both in counter plot



Where we can see it moved rapid changes in b first and then changes in m.

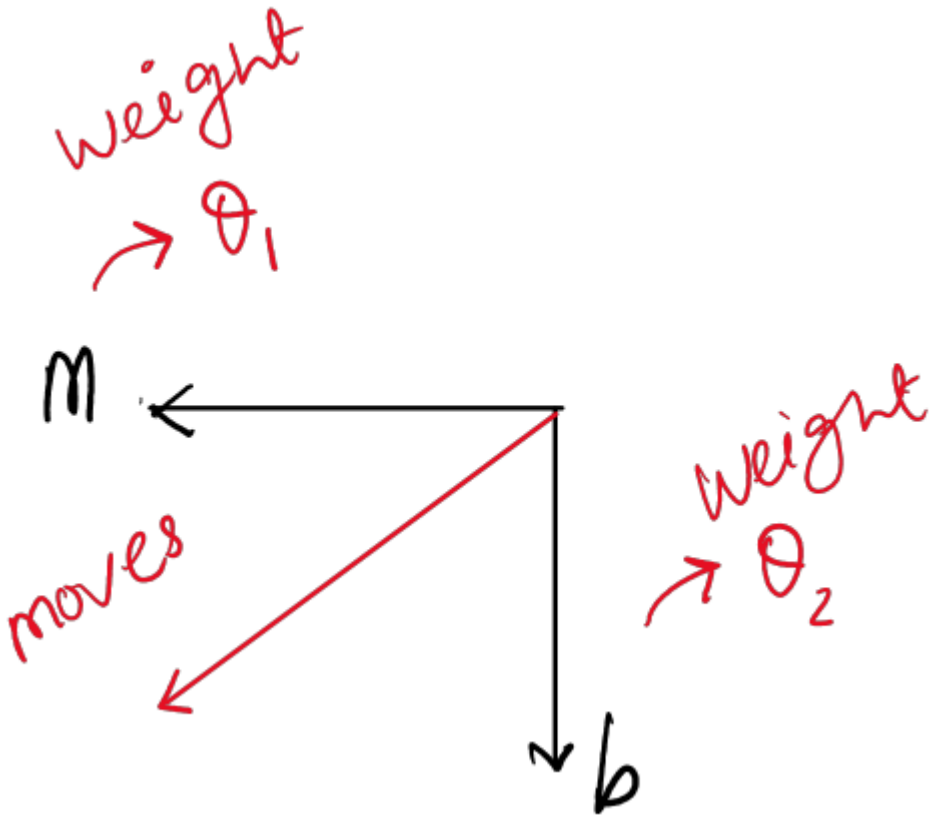
Where what it should look like can be seen with adagrad plot



Source: Link
where it converges faster.

Why it happens ?

1. as we know it has 2 direction in this case



if m is sparse then update in direction b will dominate. that's why we can see the dominance of moving in direction b in BGD and momentum.

where as in adagrad make sure the both features learned or update in same rate. what it does is have different learning rates for different features. so they are updating similarly so we can save many steps and computation. and it can converges faster here as compare to gradient descent and momentum.

Algorithm

INPUTS:

- Dataset:$X \in \mathbb{R}^{m \times n}$
- Targets: $y \in \mathbb{R}^m$
- Learning rate: η
- Small constant for numerical stability: ϵ
- Number of iterations: N

INITIALIZATION:
Parameter θ_0
Accumulated squared gradients: $G_0 = 0$
COMPUTE:
For t=0 to N−1:

1. **Predict**

$$\hat{y}_b = X_b^\top \theta$$

2. **Compute loss**: $J(\theta)$
3. **Compute gradient**: $g_t = \nabla_\theta J(\theta)$
4. **Accumulate squared gradients**

$$G_t = G_{t-1} + g_t^2$$

5. **Adaptive parameter update**

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t} + \epsilon} \odot g_t$$

   You will ask that "Hey where are we having different learning rate for different parameters. ?".
   the answer is here in $\frac{\eta}{\sqrt{G_t}+\epsilon}$.
   AdaGrad turns a scalar learning rate into a **per-parameter adaptive learning rate vector**.

- ϵ **prevents division by zero**
- It also stabilizes early updates numerically
- It is **not** responsible for adaptivity
  AdaGrad achieves different learning rates per parameter by scaling the gradient with the

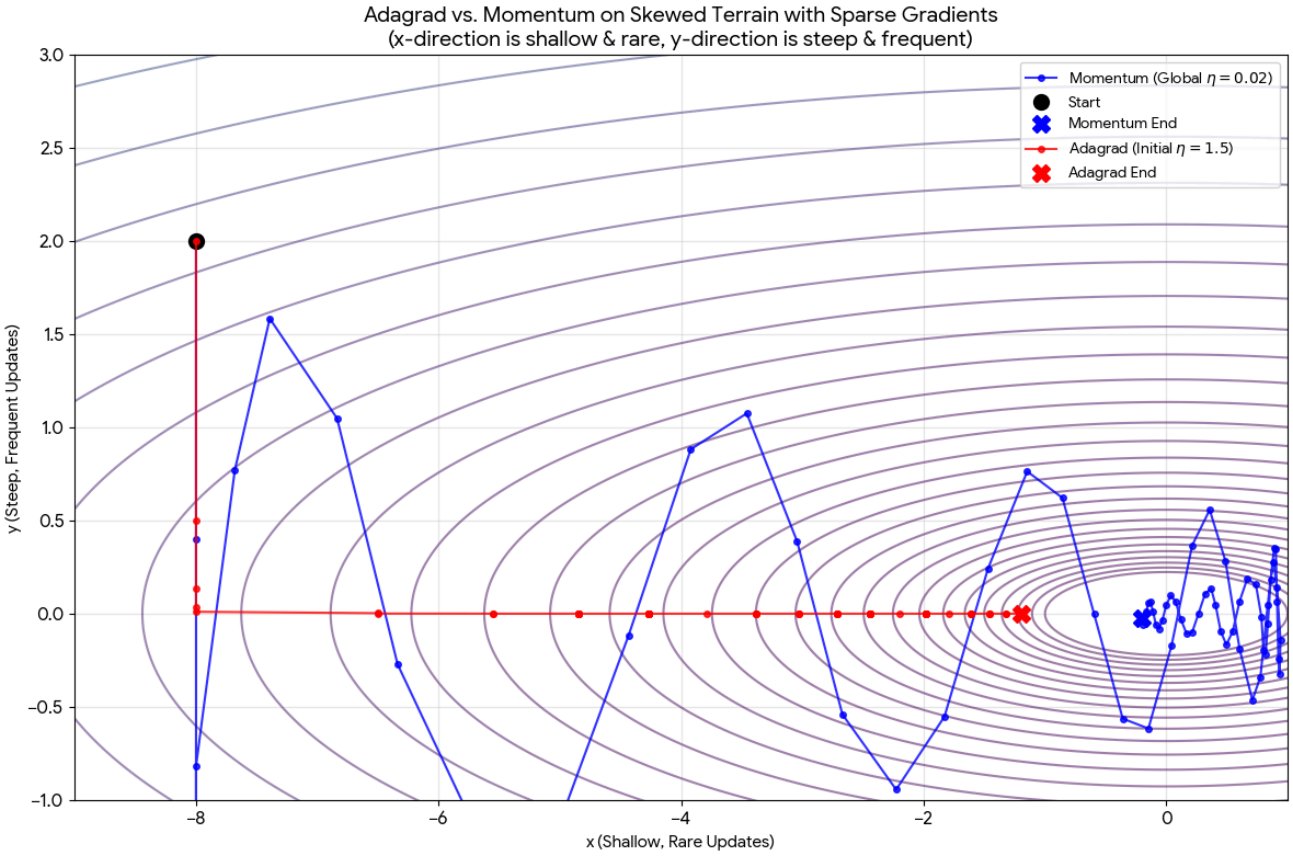
inverse square root of the accumulated squared gradients, making the effective learning rate a vector rather than a scalar.

Code
same here i used mse but we will write a modular code support all type of loss function. a bit latter in this note series

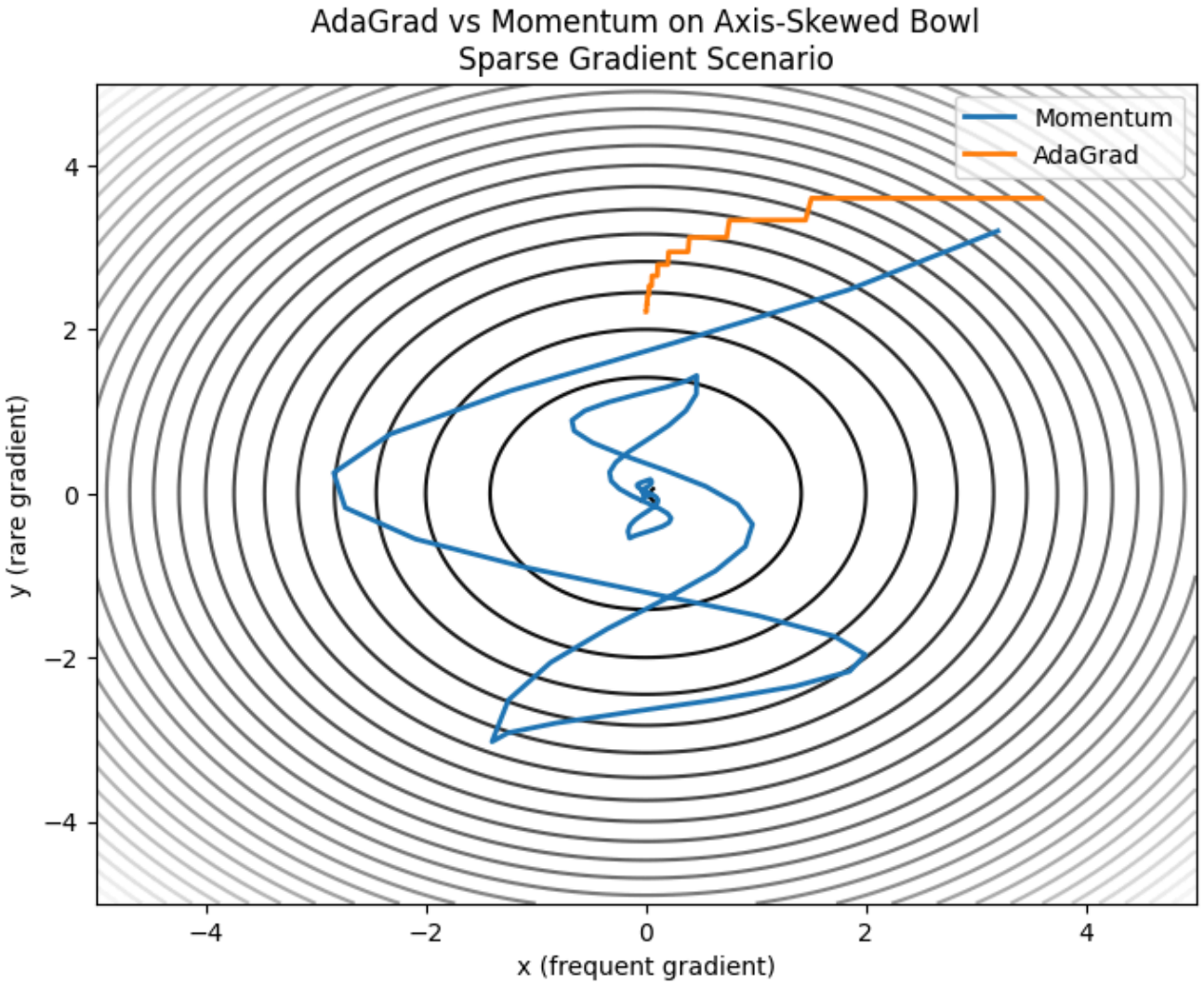
```python
import numpy as np

def adagrad_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float,
    n_iterations: int,
    batch_size: int,
    epsilon: float = 1e-8,
) -> np.ndarray:
    """
    AdaGrad for Mean Squared Error (MSE).

    Args:
        X (np.ndarray):
            Input data of shape (m, n).
        y (np.ndarray):
            Targets of shape (m,) or (m, 1).
        theta (np.ndarray):
            Initial parameters of shape (n,) or (n, 1).
        learning_rate (float):
            Base learning rate (η).
        n_iterations (int):
            Number of iterations.
        batch_size (int):
            Mini-batch size.
        epsilon (float):
            Small constant to avoid division by zero.

    Returns:
        np.ndarray:
            Optimized parameters.
    """

    m = X.shape[0]
```

```
    # Ensure column vectors
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    # Accumulator for squared gradients
    grad_accumulator = np.zeros_like(theta)

    for _ in range(n_iterations):

        # ---- 1. Sample mini-batch ----
        indices = np.random.choice(m, batch_size, replace=False)
        X_b = X[indices]
        y_b = y[indices]

        # ---- 2. Predict ----
        y_hat = X_b @ theta

        # ---- 3. Compute gradient of MSE ----
        gradient = (2 / batch_size) * X_b.T @ (y_hat - y_b)

        # ---- 4. Accumulate squared gradients ----
        grad_accumulator += gradient ** 2

        # ---- 5. Adaptive parameter update ----
        adjusted_lr = learning_rate / (np.sqrt(grad_accumulator) + epsilon)
        theta = theta - adjusted_lr * gradient

    return theta
```

but the reality is here it may be good but never converge in real scenereos like this



here momentum converges where x is the shallow features. adagrad was good but never able to converge to the minima.
Here is another



it was stable as compare to momentum but never able to converge.

but but but
it tried to solve a major problem with

1. per-parameter scaling
2. different learning rates

Strength

- **Great for Sparse Data:** AdaGrad works well for features with many zeros because it adapts the learning rate for each feature based on its gradient history.
- **No Manual Tuning of Learning Rates:** The adaptive learning rate means you don't have to adjust the learning rate manually for each parameter.
- great for NLP-style problems
- Parameters with very different update frequencies

Weakness

- Learning rate decays forever

    - **Converges Slowly for Complex Problems:** Over time, the accumulated gradients make the learning rate so small that the algorithm stops making meaningful updates.
    - **Limited Use in Neural Networks:** AdaGrad is rarely used in deep neural networks because it slows down too much due to its shrinking learning rate.
- Eventually stops learning
- Mostly never converges so never used in real work
- Training freezes on plateaus

Solution

adadelta and rms prop

4. Adadelta

Adadelta was designed specifically to fix the two biggest flaws of **Adagrad**:

1. Learning rate collapses

$$\sum g^2 \uparrow \Rightarrow \eta_{eff} \downarrow \Rightarrow 0$$

2. Still needs a global η

What it does is
Instead of fixing AdaGrad's learning rate decay, AdaDelta **removes** η **entirely**.

- **Replace cumulative sum with exponential moving average** (like RMSProp)
- **Scale updates by past update magnitudes, not a learning rate**
  This makes the update:
- unit-consistent
- self-scaled
- memory-limited

instead of

$$G_t = G_{t-1} + g_t^2$$

which is in adagrad.
Where as what it does is

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

Where it only remember fraction of previously collected gradient squares $E[g^2]_{t-1}$ and fraction of current gradient g_t^2 . instead of all previous gradient. (Crazy right !!)

so now we can compare
In SGD, Now,

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

In AdaGrad,

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

In Adadelta,

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

but wait we also have to remove the learning rate from the scene.
before that
what is the another problem of adagrad as unit less ?

- Gradient units: $[g] = \text{loss}/\text{parameter}$
- Update should have units of parameter.
  AdaGrad / RMSProp:

$$\frac{g}{\sqrt{g^2}} \Rightarrow \text{unitless} \quad (\text{needs } \eta \text{ to fix units})$$

Coming to the removal of learning rate in adadelta
we have to do dimensional analysis.
Let's write **units**, not symbols

- Parameters: $[\theta]$
- Loss: $[L]$
- Gradient:

$$[g] = \frac{[L]}{[\theta]}$$

Now look at RMSProp's core term:

$$\frac{g_t}{\sqrt{E[g^2]_t}}$$

Units:

$$\frac{[L]/[\theta]}{\sqrt{[L]^2/[\theta]^2}} = \frac{[L]/[\theta]}{[L]/[\theta]} = 1$$

So the whole thing is **unitless**.

That's why you need η\etaη to reintroduce parameter units:

$$[\Delta\theta] = \eta \times (\text{unitless})$$

Past parameter updates.
Those already have correct units:

$$[\Delta\theta] = [\theta]$$

So define:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)(\Delta\theta_t)^2$$

Then take RMS:

$$\text{RMS}[\Delta\theta]_{t-1} = \sqrt{E[\Delta\theta^2]_{t-1}}$$

This is **exactly the scale we want**.

Now our equation becomes

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

And there is no learning rate term in it.

AdaDelta fixes AdaGrad by replacing the cumulative sum of squared gradients with exponential moving averages and scaling updates using past update magnitudes instead of a global learning rate.

Algorithm

INPUTS

- Dataset: $X \in \mathbb{R}^{m \times n}$
- Targets: $y \in \mathbb{R}^m$
- Decay rate: $\gamma \in (0, 1)$
- Small constant for numerical stability: ϵ
- Number of iterations: N

INITIALIZATION

- Initial parameters: θ_0
- Exponential moving average of squared gradients:

$$E[g^2]_0 = 0$$

- Exponential moving average of squared parameter updates:

$$E[\Delta\theta^2]_0 = 0$$

COMPUTE

For t=0 to N−1:

1. **Predict**

$$\hat{y}_b = X_b^\top \theta$$

2. **Compute loss**: $J(\theta)$
3. **Compute gradient**: $g_t = \nabla_\theta J(\theta)$
4. **Accumulate squared gradients (exponential moving average)**

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

5. Compute parameter update

$$\Delta\theta_t = -\frac{\sqrt{E[\Delta\theta^2]_{t-1}}}{\sqrt{E[g^2]_t + \epsilon}} \odot g_t$$

6. Accumulate squared updates

$$E[\Delta\theta^2]_t = \rho E[\Delta\theta^2]_{t-1} + (1-\rho)(\Delta\theta_t)^2$$

7. Update parameters

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Code

```
import numpy as np

def adadelta_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    rho: float,
    n_iterations: int,
    batch_size: int,
    epsilon: float = 1e-6,
) -> np.ndarray:
    """
    AdaDelta for Mean Squared Error (MSE).

    Args:
        X (np.ndarray):
            Input data of shape (m, n).
```

```
        y (np.ndarray):
            Targets of shape (m,) or (m, 1).
        theta (np.ndarray):
            Initial parameters of shape (n,) or (n, 1).
        rho (float):
            Decay rate for moving averages (ρ).
        n_iterations (int):
            Number of iterations.
        batch_size (int):
            Mini-batch size.
        epsilon (float):
            Small constant for numerical stability.

    Returns:
        np.ndarray:
            Optimized parameters.
    """

    m = X.shape[0]

    # Ensure column vectors
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    # Moving averages
    Eg2 = np.zeros_like(theta)        # E[g^2]
    Edelta2 = np.zeros_like(theta)    # E[Δθ^2]

    for _ in range(n_iterations):

        # ---- 1. Sample mini-batch ----
        indices = np.random.choice(m, batch_size, replace=False)
        X_b = X[indices]
        y_b = y[indices]

        # ---- 2. Predict ----
        y_hat = X_b @ theta

        # ---- 3. Gradient of MSE ----
        gradient = (2 / batch_size) * X_b.T @ (y_hat - y_b)

        # ---- 4. Accumulate gradient RMS ----
        Eg2 = rho * Eg2 + (1 - rho) * gradient**2
```

```
        # ---- 5. Compute update ----
        delta = - (np.sqrt(Edelta2 + epsilon) /
                   np.sqrt(Eg2 + epsilon)) * gradient

        # ---- 6. Accumulate update RMS ----
        Edelta2 = rho * Edelta2 + (1 - rho) * delta**2

        # ---- 7. Update parameters ----
        theta = theta + delta

    return theta
```

Strength

1. Continues learning after initial phase
2. Prevents learning rate from going to zero
3. Unit consistency
4. Per-parameter adaptivity

Weakness

- No momentum
- Reactive, not anticipatory (Less responsive to sudden curvature changes)
- Slow adaptation to new terrain
  Let's Discuss more problems of adadelta

1. Update scale depends on history of updates (feedback loop)
   AdaDelta scales updates by:

   $$RMS[\Delta\theta]_{t-1}$$

   Which means

- If updates become small early → future updates stay small
- If early updates are noisy → scale propagates

This creates a **self-referential feedback loop**.
Once it slows down, it tends to *stay* slow.

2. No explicit control over step size
3. Slower practical convergence than RMSProp

AdaDelta is stable, but conservative. In modern training, conservative often means “wasting GPU hours.”

Solution

bring back the learning rate but keep the **Exponential Moving Average**.-> RMSprop

5. RMS Prop

Root Mean Square Propagation
it's an adaptive learning rate method proposed by Geoff Hinton in his lecture according to author of the blog.

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates.

so it's same to adadelta till the first update vector update we discussed above

$$
\begin{aligned}
E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\
\Delta\theta_t &= -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}g_t \\
\theta_{t+1} &= \theta_t + \Delta\theta_t
\end{aligned}
$$

But here the the $\gamma = 0.9$ and the $(1-\gamma) = 0.1$ in $E[g^2]_t$. but author suggested " while a good default value for the learning rate η is 0.001."

Algorithm

INPUTS

- Dataset: $X \in \mathbb{R}^{m \times n}$
- Targets: $y \in \mathbb{R}^m$
- Learning rate: η
- Decay rate: $\gamma \in (0, 1)$
- Small constant for numerical stability: ϵ
- Number of iterations: N

INITIALIZATION

- Initial parameters: θ_0
- Exponential moving average of squared gradients:

$$E[g^2]_0 = 0$$

- Exponential moving average of squared parameter updates:

$$E[\Delta\theta^2]_0 = 0$$

COMPUTE

For t=0 to N−1:

1. **Predict**

$$\hat{y}_b = X_b^\top \theta$$

2. **Compute loss**: $J(\theta)$
3. **Compute gradient**: $g_t = \nabla_\theta J(\theta)$
4. **Accumulate squared gradients (exponential moving average)**

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

5. Compute parameter update

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot g_t$$

6. Update parameters

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Code

```python
import numpy as np

def rmsprop_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float,
    rho: float,
    n_iterations: int,
    batch_size: int,
    epsilon: float = 1e-8,
) -> np.ndarray:
    """
    RMSProp for Mean Squared Error (MSE).

    Args:
        X (np.ndarray):
            Input data of shape (m, n).
        y (np.ndarray):
            Targets of shape (m,) or (m, 1).
        theta (np.ndarray):
```

```
            Initial parameters of shape (n,) or (n, 1).
        learning_rate (float):
            Base learning rate (η).
        rho (float):
            Decay rate for squared gradients (ρ).
        n_iterations (int):
            Number of iterations.
        batch_size (int):
            Mini-batch size.
        epsilon (float):
            Small constant for numerical stability.

    Returns:
        np.ndarray:
            Optimized parameters.
    """

    m = X.shape[0]

    # Ensure column vectors
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    # Moving average of squared gradients
    Eg2 = np.zeros_like(theta)

    for _ in range(n_iterations):

        # ---- 1. Sample mini-batch ----
        indices = np.random.choice(m, batch_size, replace=False)
        X_b = X[indices]
        y_b = y[indices]

        # ---- 2. Predict ----
        y_hat = X_b @ theta

        # ---- 3. Gradient of MSE ----
        gradient = (2 / batch_size) * X_b.T @ (y_hat - y_b)

        # ---- 4. Accumulate squared gradients ----
        Eg2 = rho * Eg2 + (1 - rho) * gradient**2

        # ---- 5. Adaptive parameter update ----
```

```
        theta = theta - (learning_rate / (np.sqrt(Eg2) + epsilon)) *
gradient

    return theta
```

6. Nesterov Momentum

Solves "Momentum overshoots **curved valleys**."

it is far below of momentum but think of it's parallel continuation to adagrad.

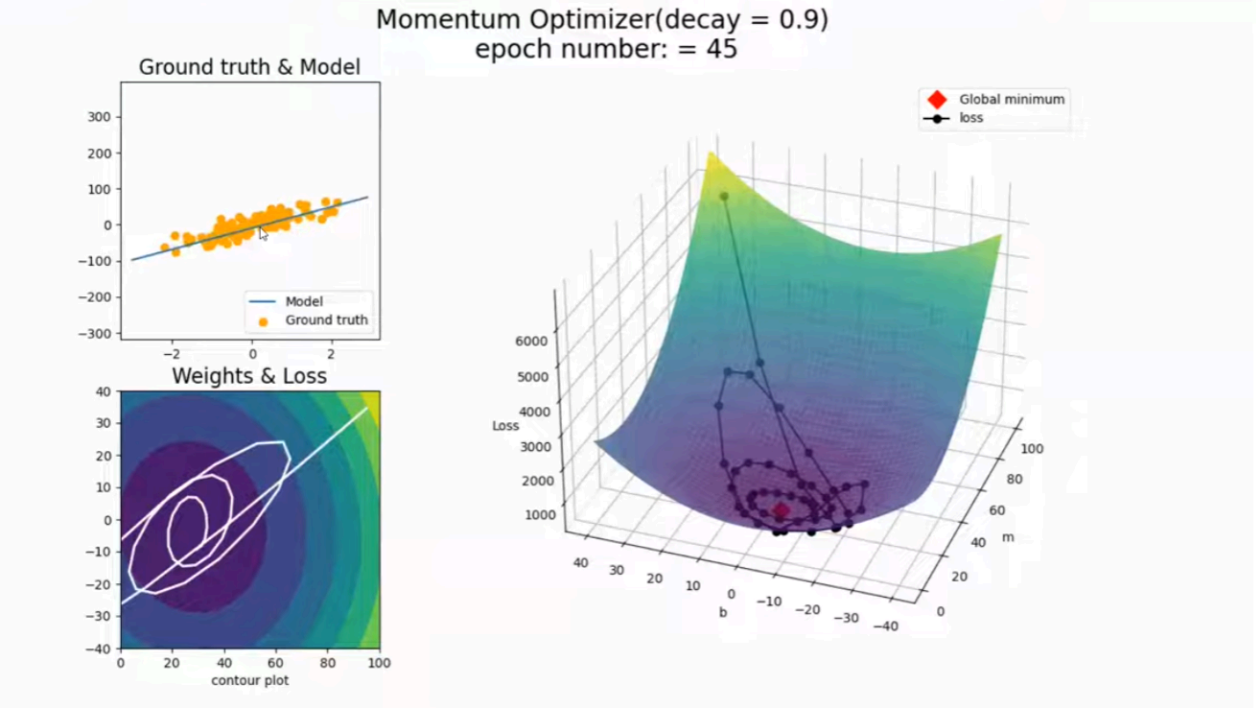
Author's Words
"However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again."

let's feel the problem
Standard momentum updates velocity **after** seeing the gradient at the current position:

$$v_t = \beta v_{t-1} + \eta g(\theta_t)$$

Problem:

- You might already be moving fast
- Gradient doesn't know where you're *about to be*
- Causes overshooting in curved valleys

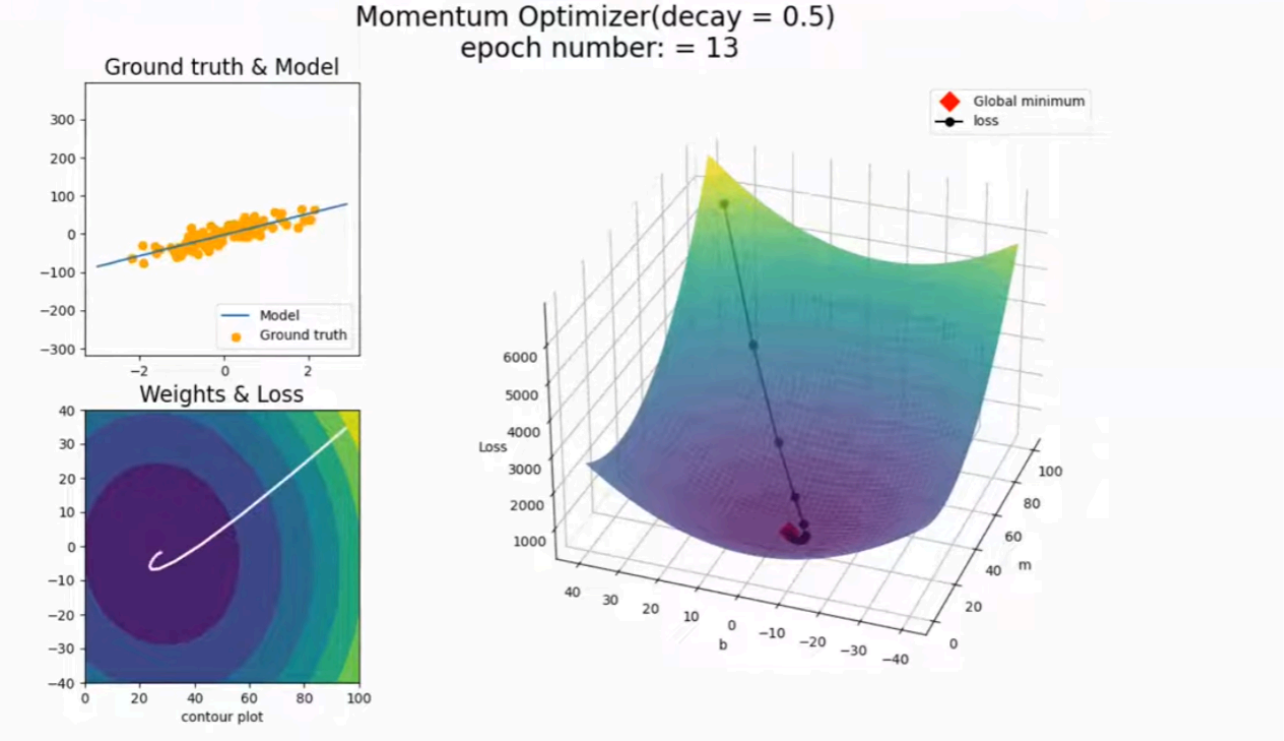


source: Link

Here it continues it's oscillation around minima but after many steps it gets stable and able to converges. but it cover initial path very fast.

but here the decay factor is 0.9

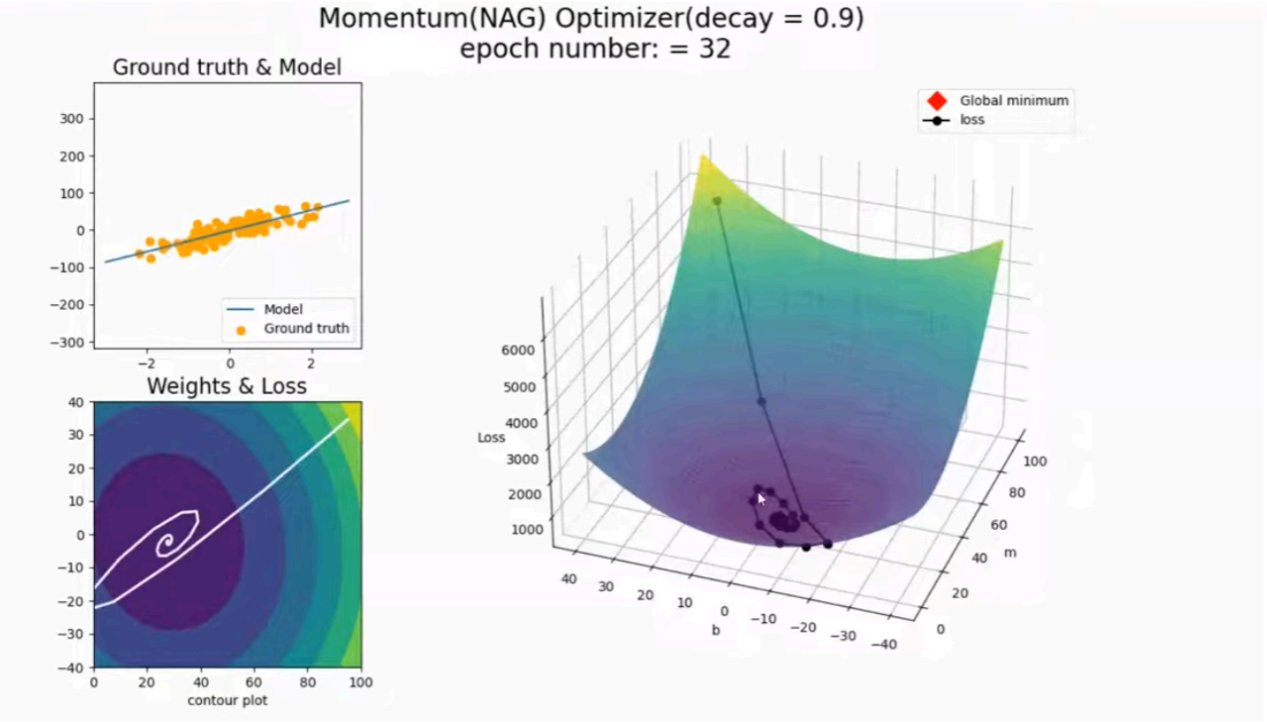
but when it is less like it is 0.5 then



it able to converge in less epochs or steps.

in non-convex loss function we get into issue in momentum.

but in nag we get to converge efficiently even in the same decay factor = 0.9



Source : Link

as it dampens the oscillation.

To understand NAG, let’s explore how it differs from simple momentum:

1. **Momentum Update**: Traditional momentum calculates an update based on the current gradient and a weighted sum of past gradients, creating a “velocity” term that guides the update direction.
2. **NAG Update**: Instead of calculating the gradient at the current point (i.e. $\nabla_\theta J(\theta)$, NAG takes a look-ahead step using the momentum term. It then computes the gradient at this new look-ahead position and adjusts the update based on this gradient. This approach makes it easier to predict when the optimizer is likely to overshoot, allowing for more controlled updates.

We know that we will use our momentum term γv_{t-1} to move the parameters θ. in momentum implementation where

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

but here we will compute $\theta - \gamma v_{t-1}$ which give us approximation of the next position of the parameters (the gradient is missing for the full update).

So what happens here is

My quote:
""“Momentum already tells us where we’re going, so instead of reacting after we overshoot, let’s peek at that future position and correct earlier.”"

at time t, we already have
current parameter : θ_t
previous velocity: v_t
From momentum alone, you *expect* to move approximately to:

$$\theta_t - \gamma v_{t-1}$$

This is the **lookahead position**. (it's the future estimate)
(analogy the eyes in the slide in source Link)
This is a **rough prediction** of where the parameters will land if momentum keeps pushing.

we take gradient here (i.e. $\nabla_\theta J(\theta - \gamma v_{t-1})$)
we don't update the parameter yet.

This gradient tells you:

- whether momentum is pushing you *too far*
- whether you should slow down or correct direction

Now we update the velocity term (v_t) to correct the earliest

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

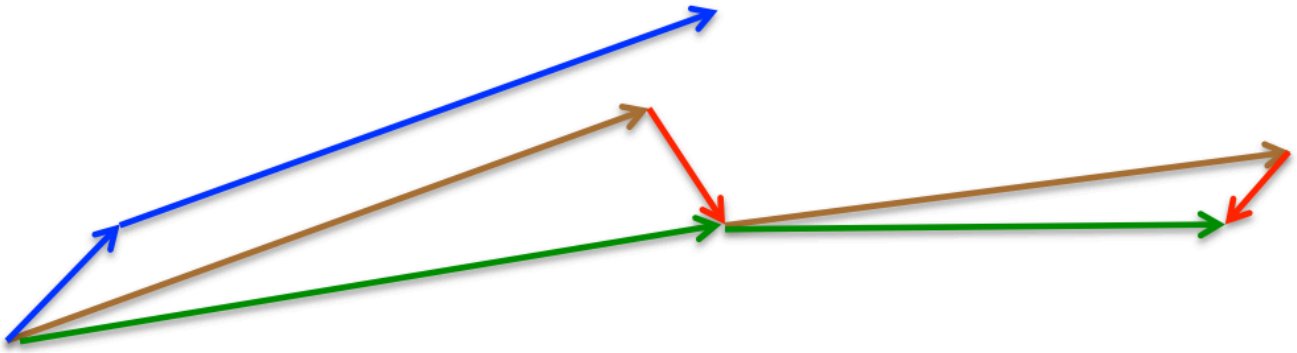
and then we update the parameters

$$\theta_{t+1} = \theta_t - v_t$$

Let's try again to have a better intuition
"In Nesterov momentum, we first form an approximate estimate of where the parameters will move due to momentum. Instead of computing the gradient at the current parameters, we compute it at this lookahead position($\theta - \gamma v_{t-1}$). This allows the optimizer to anticipate future overshooting and correct the velocity before the parameters are actually updated."

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta_{t+1} = \theta_t - v_t$$

Here the momentum term is γ , we set this to 0.9 or 0.99 as recommended.
To see this in action. Here is plot from a famous lecture of G. Hinton.



Source: Nesterov update (Source: G. Hinton's lecture 6c)

Here is little more intiution

1. Momentum (Blue)
   While Momentum first computes the current gradient (small blue vector in Image 4) and then takes a big jump in the direction of the updated accumulated gradient (big blue vector),
2. NAG
   NAG first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (green vector).
   the green vector is the move that is better than the momentum move which is the brown line in 2nd image.

This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs on a number of tasks

Physics intiution (Ha Ha)
"Imagine rolling a ball down a hill. With simple momentum, you only know the direction you're moving in and rely on past speed to guide you. But with NAG, it's like you're peeking a few steps ahead to see if you're about to overshoot. This helps you adjust the speed to avoid missing the target."

"Momentum updates velocity using the current gradient; Nesterov momentum updates velocity using the gradient at the predicted future position."

Why this is actually better than vanilla momentum

Vanilla momentum:

- reacts **after** overshooting

Nesterov momentum:

- corrects **before** overshooting

Same cost. Same memory. Better geometry.

Crazy.. Huuh

Algorithm

INPUTS:

- Dataset ($X \in \mathbb{R}^{m \times n}$)
- Targets ($y \in \mathbb{R}^{m}$)
- Learning rate (η)
- Momentum Coefficient : (γ)
- Number of iteration : (N)
  INITIALIZATION:
  Initial parameter (θ_0)
  Initial velocity:

$$v_0 = 0$$

  COMPUTE:
  For t = 0 to N-1 :

1. Calculate Lookahead position

$$\tilde{\theta_t} = \theta - \gamma v_{t-1}$$

2. Predict

$$\hat{y}_b = X_b^\top \tilde{\theta}$$

3. Compute Loss $J(\tilde{\theta})$
4. Compute gradient: $g_t = \nabla_\theta J(\tilde{\theta})$
5. Update velocity:

$$v_t = \gamma v_{t-1} + \eta g_t$$

6. Update parameters:

$$\theta_{t+1} = \theta_t - v_t$$

Code

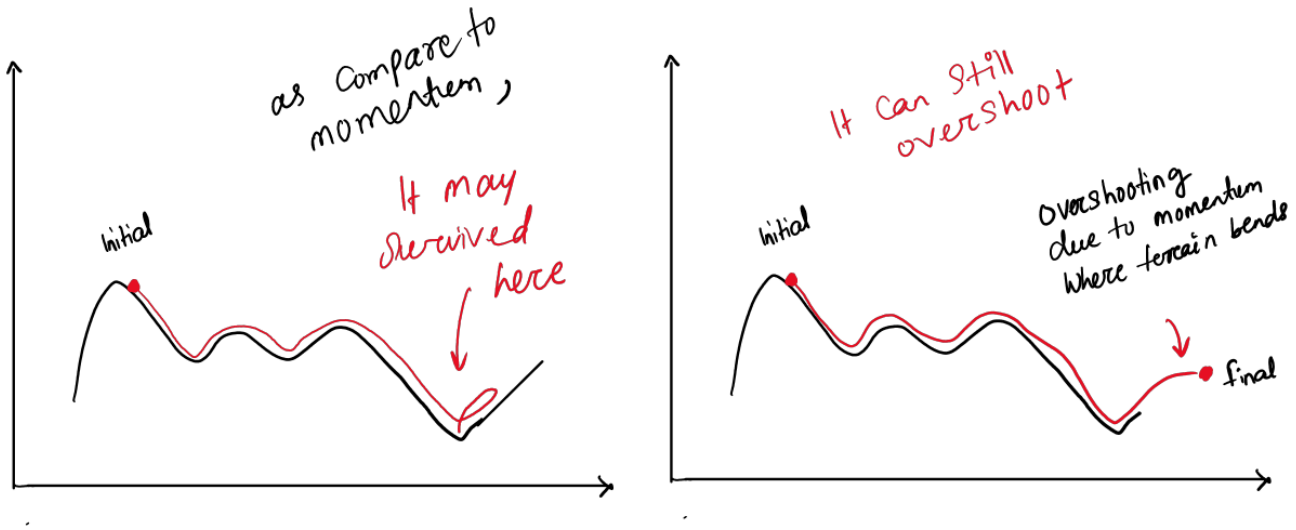
```python
import numpy as np

def nesterov_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float,
    gamma: float,
    n_iterations: int,
    batch_size: int,
) -> np.ndarray:
    """
    Nesterov Accelerated Gradient (NAG) for Mean Squared Error (MSE).

    Args:
        X (np.ndarray):
            Input data of shape (m, n).
        y (np.ndarray):
            Targets of shape (m,) or (m, 1).
        theta (np.ndarray):
            Initial parameters of shape (n,) or (n, 1).
        learning_rate (float):
            Learning rate (η).
        gamma (float):
            Momentum coefficient (γ).
        n_iterations (int):
            Number of iterations.
        batch_size (int):
            Mini-batch size.

    Returns:
        np.ndarray:
            Optimized parameters.
    """

    m = X.shape[0]

    # Ensure column vectors
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)
```

```
    # Initialize velocity
    velocity = np.zeros_like(theta)

    for _ in range(n_iterations):

        # ---- 1. Sample mini-batch ----
        indices = np.random.choice(m, batch_size, replace=False)
        X_b = X[indices]
        y_b = y[indices]

        # ---- 2. Lookahead position ----
        theta_lookahead = theta - gamma * velocity

        # ---- 3. Predict ----
        y_hat = X_b @ theta_lookahead

        # ---- 4. Gradient at lookahead ----
        gradient = (2 / batch_size) * X_b.T @ (y_hat - y_b)

        # ---- 5. Velocity update ----
        velocity = gamma * velocity + learning_rate * gradient

        # ---- 6. Parameter update ----
        theta = theta - velocity

    return theta
```

Strength

1. **Reduced Oscillations**: By peeking ahead, NAG reduces oscillations around the minimum, which can be especially useful in error surfaces with multiple peaks and valleys.
2. **Faster Convergence**: NAG often converges faster than simple momentum-based methods because it reduces unnecessary steps around the optimum.

Weakness



Solution

7. Adam

Adaptive Moment Estimation
it's a method that computes adaptive learning rates for each parameter.
What Adam combines

- Momentum (1st moment)
- RMSProp-style adaptation (2nd moment)
- Bias correction
  What is bias correction ?
  Because EMAs start at zero:
- Early values of mtm_tmt and vtv_tvt are **biased toward zero**
- Not because gradients are small
- But because the average hasn't "warmed up" yet

Why this actually matters

Adam update (without correction):

$$\theta_t = -\frac{\eta}{\sqrt{v_t} + \epsilon} m_t$$

If:

- m_t is too small
- v_t is too small
  Then:
- Steps are **artificially tiny**
- Early learning slows down
- Training looks "stuck" at the beginning

Bias correction compensates for the initialization bias of exponential moving averages by rescaling them so they are unbiased estimates during early iterations.

This is **initialization bias**, not statistical bias in the philosophical sense.
In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum.

okay, so What is different ? between v_t and m_t ?
before that few things we have to know
an analogy: " we can see m_t as momentum, Whereas momentum can be seen as a ball running down a slope, Adam is a ball whose **mass (momentum)** comes from m_t, and whose **friction coefficient changes per direction** based on v_t"

"Adam stabilizes training by combining momentum with per-parameter adaptive scaling, but it does not inherently bias optimization toward flat minima."

So the final analogy is
"In Adam, m_t represents a momentum term as an exponential moving average of past gradients, providing directional smoothing and inertia. The term v_t represents an exponential moving average of squared gradients and controls the adaptive step size per parameter. While m_t determines *where* to move, v_t determines *how much* to move. Adam combines momentum with adaptive scaling, acting like a system with directionally varying friction rather than a simple heavy ball."

We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

where the analogies are as follows
m_t: estimates of the first moment (the mean) of the gradient
→ EMA of the **gradient**
→ Estimates the **first moment** (mean)
v_t: estimates of the second moment (the uncentered variance) of the gradient
→ EMA of the **squared gradient**
→ Estimates the **second moment** (uncentered variance / energy)

m_t: direction + inertia

- Smooths noisy gradients
- Accumulates consistent direction over time
- Prevents zig-zagging
- This **is momentum**, mathematically and conceptually

v_t : scale + trust control

- Measures how large gradients usually are
- Shrinks steps where gradients are consistently large
- Allows larger steps where gradients are small
- Acts **per-parameter**
  so it's a per-dimension step-size regulator.

m_t decides the direction of motion, v_t decides the step size.

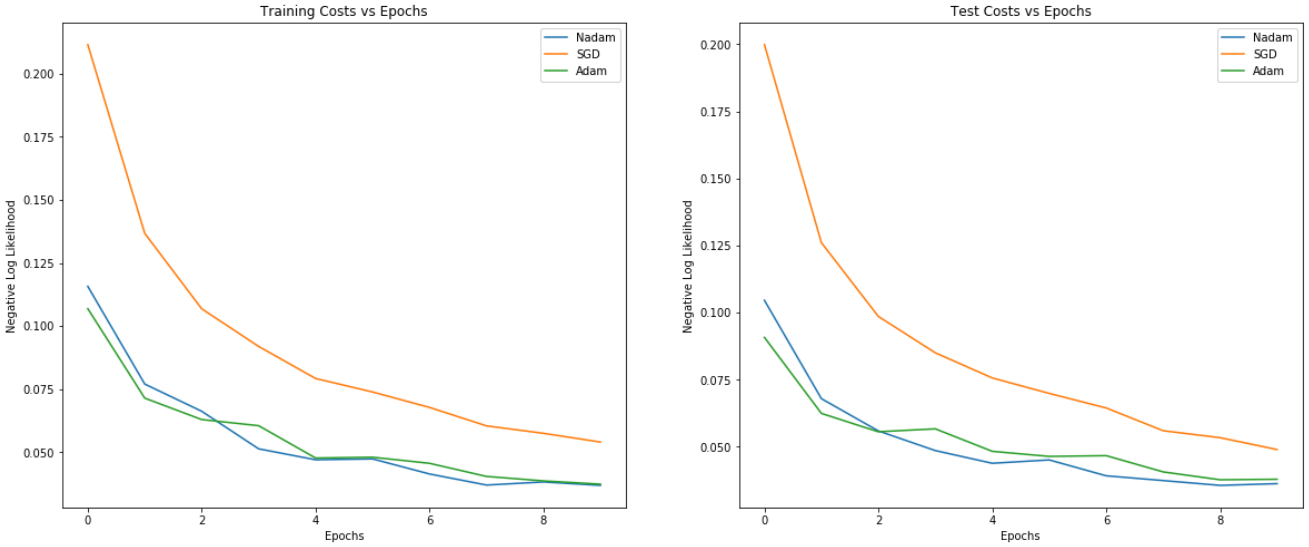
They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

then we use these to update the parameters just as we have seen adadelta and RMSprop, so the adam update rule become:

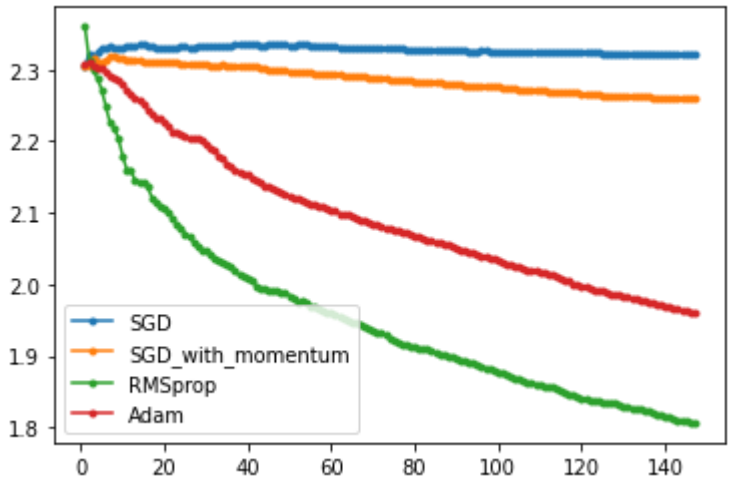
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ.



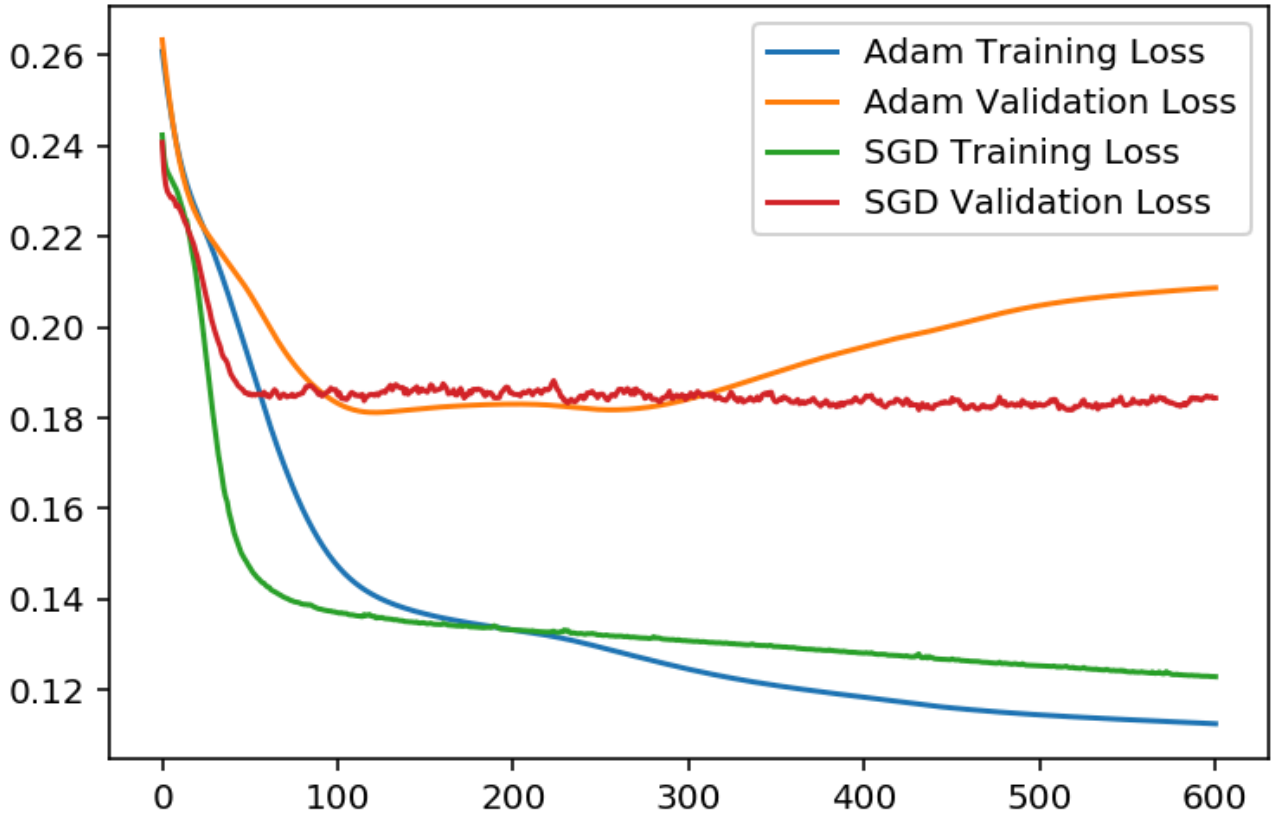
Source: Link

Nadam (blue) and Adam (green) demonstrate significantly faster convergence rates than SGD (red). Nadam appears to lag behind Adam in the first two epochs on both the training and test sets before pulling very slightly ahead of Adam for the remaining epochs. However, the margin is small enough that we cannot conclude either way whether Adam or Nadam performs better



Source: Link

This is to test different optimizers in imagenette dataset
We can see `Adam` is performing the best out of the three and is able to reduce the loss at a much faster pace.



SourceLink

From the plots, we can observed that using Adam, weights of the neural network are more smoothly adjusted to reduce the training loss.

The benefits of using Adam are not so obvious as the size of the data is very small and increasing training epochs tend to lead to overfitting and early-stopping is required. It is recommended to set the epochs for Adam to around 200 for the above hyperparameters configuration, as the training and validation loss starts diverging. However, the author kept the epochs for both networks the same for plotting.

Algorithm

INPUTS

- Dataset: $X \in \mathbb{R}^{m \times n}$
- Targets: $y \in \mathbb{R}^m$
- Learning rate: η
- Exponential decay rate for first moment : $\beta_1 \in (0, 1)$
- Exponential decay rate for second moment : $\beta_2 \in (0, 1)$
- Small constant for numerical stability: ϵ
- Number of iterations: N
  INITIALIZATION
- Initial parameters: θ_0
- First moment (mean of gradients):

$$m_0$$

- Second moment (mean of squared gradients):

$$v_0$$

- Time step:

$$t = 0$$

  COMPUTE

For t=0 to N−1:

1. **Predict**

$$\hat{y}_b = X_b^\top \theta$$

2. **Compute loss**: $J(\theta)$
3. **Compute gradient**: $g_t = \nabla_\theta J(\theta)$
4. Update biased first moment estimate (momentum)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

5. Update biased second moment estimate (RMSProp)

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

6. Bias correction

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

   This step matters. Without it, early updates are wrong.

7. Compute parameter update

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot \hat{m}_t$$

8. Update parameters

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Code

```python
import numpy as np

def adam_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float,
    beta1: float,
    beta2: float,
    n_iterations: int,
    batch_size: int,
    epsilon: float = 1e-8,
) -> np.ndarray:
    """
    Adam optimizer for Mean Squared Error (MSE).

    Args:
        X (np.ndarray):
            Input data of shape (m, n).
        y (np.ndarray):
            Targets of shape (m,) or (m, 1).
        theta (np.ndarray):
            Initial parameters of shape (n,) or (n, 1).
        learning_rate (float):
            Learning rate (η).
        beta1 (float):
            Decay rate for first moment (β1).
        beta2 (float):
            Decay rate for second moment (β2).
        n_iterations (int):
            Number of iterations.
        batch_size (int):
            Mini-batch size.
```

```
        epsilon (float):
            Small constant for numerical stability.

    Returns:
        np.ndarray:
            Optimized parameters.
    """

    m = X.shape[0]

    # Ensure column vectors
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    # Moment estimates
    m_t = np.zeros_like(theta)
    v_t = np.zeros_like(theta)

    t = 0

    for _ in range(n_iterations):
        t += 1

        # ---- 1. Sample mini-batch ----
        indices = np.random.choice(m, batch_size, replace=False)
        X_b = X[indices]
        y_b = y[indices]

        # ---- 2. Predict ----
        y_hat = X_b @ theta

        # ---- 3. Gradient of MSE ----
        gradient = (2 / batch_size) * X_b.T @ (y_hat - y_b)

        # ---- 4. First moment update ----
        m_t = beta1 * m_t + (1 - beta1) * gradient

        # ---- 5. Second moment update ----
        v_t = beta2 * v_t + (1 - beta2) * (gradient ** 2)

        # ---- 6. Bias correction ----
        m_hat = m_t / (1 - beta1 ** t)
        v_hat = v_t / (1 - beta2 ** t)
```

```
        # ---- 7. Parameter update ----
        theta = theta - learning_rate * m_hat / (np.sqrt(v_hat) + epsilon)

    return theta
```

Weakness

- Tends toward sharp minima
- Improper weight decay
- Generalization issues

8. AdaMax

Adamax scales momentum-smoothed gradients by an exponentially decayed infinity norm of past gradients, making it robust to large or sparse gradient values.

v_t in adam update rule scales the gradient inversely proportional to the l2 norm of the the past gradient (via v_{t-1} term) and the current gradient $|g_t^2|$.
Okay so it implies $\Delta\theta_t$ is inversely proportional to past gradients and current gradient. (ovio)

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

How inversely proportional ?
let's go step by step
adam keeps second moment (variance) of gradients as we can see from the above equation of v_t.
the update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

where
v_t : EMA of squared gradient. (Guess the EMA ? it's Exponential moving average)
$\sqrt{v_t}$: RMS or L2 norm of past gradient
if gradients are large, implies v_t will grow which will make denominator large => updates will be smaller.

Okay so let's generalize the L2 norm to L_p norm , so the the equation becomes

$$v_t = \beta_2^p v_{t-1} + (1-\beta_2^p) g_t^p$$

Norms for large p values generally become numerically unstable, so we don't see much l_3, or l_{99} norm instead we see the popular l_1 and l_2 in practice.

However l_∞ generally exhibits stable behavior. so the author of adamax (Kingma and Ba, 2015) proposes v_t with l_∞ converges to more stable values.

Author used another term instead of v_t , a new term u_t.

$$\begin{aligned} u_t &= \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty)|g_t|^\infty \\ &= max(\beta_2 \cdot v_{t-1}, |g_t|) \end{aligned}$$

so we can plug this into adam update equation for adamax

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t + \epsilon} \odot \hat{m}_t$$

Note that as u_t relies on the max operation, it is not as suggestible to bias towards zero as m_t and v_t in Adam, which is why we do not need to compute a bias correction for u_t.

Adam uses:

$$\sqrt{E[g^2]}$$

This:

- Can be unstable with spikes
- Can overreact to outliers

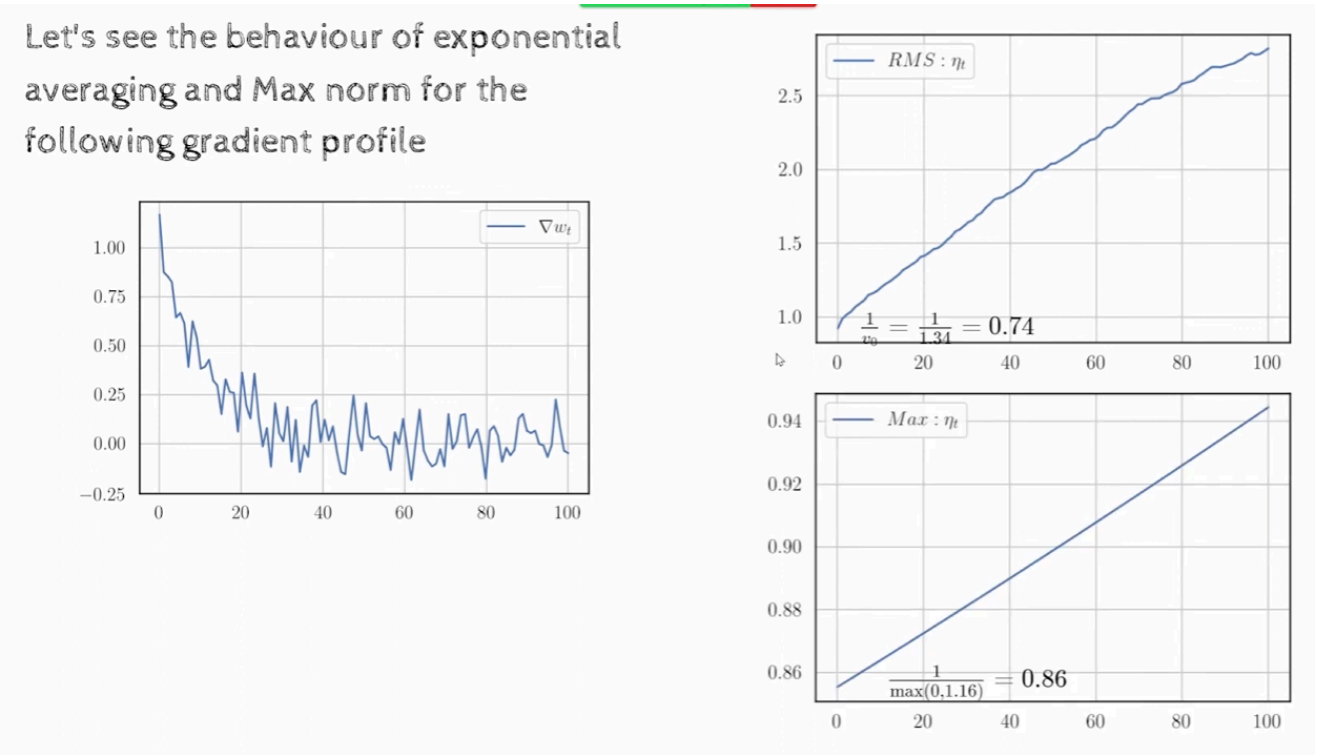
Adamax uses:

$$\|g\|_\infty$$

Which:

- Is robust to gradient spikes
- Works better with sparse features
- Has simpler numerical behavior

Adamax is Adam under the L_∞ norm**. That’s literally where the name comes from.



Source: Link

when the data is sparse and you don't want that affect your learning by that then adamax is useful. as adamax don't change the learning rate too much for the sparse features.

Algorithm

INPUT:

- Dataset: $X \in \mathbb{R}^{m \times n}$
- Targets: $y \in \mathbb{R}^m$
- Learning rate: η
- Exponential decay rate for first moment : $\beta_1 \in (0, 1)$
- Exponential decay rate for second moment : $\beta_2 \in (0, 1)$
- Small constant for numerical stability: ϵ
- Number of iterations: N
  INITIALIZE:
- Initial parameters: θ_0
- First moment (mean of gradients):

$$m_0 = 0$$

- Second moment (mean of squared gradients):

$$v_0 = 0$$

- Time step:

$$t = 0$$

COMPUTE:
For t=0 to N−1:

1. **Predict**

$$\hat{y}_b = X_b^\top \theta$$

2. **Compute loss**: $J(\theta)$
3. **Compute gradient**: $g_t = \nabla_\theta J(\theta)$
4. Update biased first moment estimate (momentum)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

5. Update biased second moment estimate (only change from adam)

$$\begin{aligned} u_t &= \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty) |g_t|^\infty \\ &= max(\beta_2 \cdot v_{t-1}, |g_t|) \end{aligned}$$

   This is **element-wise max**, not a scalar max.
6. Bias correction

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

This step matters. Without it, early updates are wrong.
and There is **no bias correction for u_t.
The max operation makes it unnecessary.

7. Compute parameter update

$$\Delta\theta_t = -\frac{\eta}{u_t + \epsilon} \odot \hat{m}_t$$

8. Update parameters

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Code

```
import numpy as np

def adamax_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float,
    beta1: float,
    beta2: float,
```

```
    n_iterations: int,
    batch_size: int,
    epsilon: float = 1e-8,
) -> np.ndarray:
    """
    Adamax optimizer for Mean Squared Error (MSE).

    Args:
        X (np.ndarray):
            Input data of shape (m, n).
        y (np.ndarray):
            Targets of shape (m,) or (m, 1).
        theta (np.ndarray):
            Initial parameters of shape (n,) or (n, 1).
        learning_rate (float):
            Learning rate (η).
        beta1 (float):
            Decay rate for first moment (β1).
        beta2 (float):
            Decay rate for infinity norm (β2).
        n_iterations (int):
            Number of iterations.
        batch_size (int):
            Mini-batch size.
        epsilon (float):
            Small constant for numerical stability.

    Returns:
        np.ndarray:
            Optimized parameters.
    """

    m = X.shape[0]

    # Ensure column vectors
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    # Moment estimates
    m_t = np.zeros_like(theta)
    u_t = np.zeros_like(theta)

    t = 0
```

```
    for _ in range(n_iterations):
        t += 1

        # ---- 1. Sample mini-batch ----
        indices = np.random.choice(m, batch_size, replace=False)
        X_b = X[indices]
        y_b = y[indices]

        # ---- 2. Predict ----
        y_hat = X_b @ theta

        # ---- 3. Gradient of MSE ----
        gradient = (2 / batch_size) * X_b.T @ (y_hat - y_b)

        # ---- 4. First moment update ----
        m_t = beta1 * m_t + (1 - beta1) * gradient

        # ---- 5. Infinity norm update ----
        u_t = np.maximum(beta2 * u_t, np.abs(gradient))

        # ---- 6. Bias correction (first moment only) ----
        m_hat = m_t / (1 - beta1 ** t)

        # ---- 7. Parameter update ----
        theta = theta - learning_rate * m_hat / (u_t + epsilon)

    return theta
```

9. Nadam

simply
Adam = RMSprop + momentum
Author beautifully explained that for adam "RMSprop contributes the exponentially decaying average of past squared gradients, v_t and Momentum accounts for the exponentially decaying average of past gradients m_t ".

and also stated "NAG is better than momentum" (as it correct itself earliest to avoid overshooting)

So,
Nadam = NAG + Adam

In order to incorporate NAG into Adam, we need to modify its momentum term m_t.

we will go little bit above in the note to the momentum update rules :

$$\begin{aligned} g_t &= \nabla_{\theta_t} J(\theta_t) \\ m_t &= \gamma m_{t-1} + g_t \\ \theta_{t+1} &= \theta_t - \eta m_t \end{aligned}$$

Here we used m_t where as earlier we used v_t in momentum portion. and a separate formula as well (i told both of the representation)
like

$$\begin{aligned} v_t &= \beta v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned}$$

$$\text{or}$$

$$\begin{aligned} v_t &= \beta v_{t-1} + (1-\beta) \nabla_\theta J(\theta) \\ \theta_{t+1} &= \theta_t - \eta v_t \end{aligned}$$

so here it's the 2nd one.

m_{t-1} in equation of expanded momentum (the 2nd one), we can use current moment (m_t) to look ahead in θ_{t+1}.

as like
In classical NAG:

$$\nabla J(\theta_t - \gamma v_{t-1})$$

In Adam-style methods, we **cannot explicitly evaluate**

$$\nabla J(\theta_t - \gamma v_{t-1})$$

because Adam works with moment estimates, not raw velocities.
So NAdam does the next best thing:

> It **algebraically approximates** the lookahead gradient using moment estimates.

But in order to add Nesterov Momentum to Adam, we can thus similarly replace previous momentum vector with the current momentum vector.

recalling adam updating rules:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \end{aligned}$$

we are expanding little into the 3rd equation with the help of second equation and it would look like

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \frac{\beta_1 m_{t-1} + (1-\beta_1) g_t}{1-\beta_1^t}$$
$$= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} (\frac{\beta_1 m_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1) g_t}{1-\beta_1^t})$$

Where $\frac{\beta_1 m_{t-1}}{1-\beta_1^t}$ is a bias correction estimate of moment vector of previous time step. replacing it with $\hat{m}_{t-1}$. so now the equation become

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} (\beta_1 \hat{m}_{t-1} + \frac{(1-\beta_1) g_t}{1-\beta_1^t})$$

where the

$$\frac{\beta_1 m_{t-1}}{1-\beta_1^{t-1}} = \beta_1 \hat{m}_{t-1}$$

but wait the denominator in $\frac{\beta_1 m_{t-1}}{1-\beta_1^t}$ have t not t-1 in for β_1. but author clearly mentioned that "Note that for simplicity, we ignore that the denominator is $1-\beta_1^t$ and not $1-\beta_1^{t-1}$ as we will replace the denominator in the next step anyway."

than It's **controlled approximation**.

The difference is negligible (mathematically)

Compare the two denominators:

$$1-\beta_1^t \quad \text{vs} \quad 1-\beta_1^{t-1}$$

Relation:

$$1-\beta_1^t = 1-\beta_1^{t-1}\beta_1 = (1-\beta_1^{t-1}) + \beta_1^{t-1}(1-\beta_1)$$

For typical values:

- $\beta_1 = 0.9$
- $t \geq 10$

Then:

- β_1^{t-1} is already tiny
- The difference between the denominators is **second order small**

In short:

> After a few iterations,
> $1-\beta_1^t \approx 1-\beta_1^{t-1}$

The approximation error dies exponentially fast.

so, an honest answer would be
"The denominator mismatch is ignored because the difference between $1-\beta_1^t$ and $1-\beta_1^{t-1}$ vanishes exponentially, and replacing it allows the update to be interpreted cleanly as a Nesterov-style momentum correction."

Term 1:

$$\beta_1 \hat{m}_{t-1}$$

- This is the **expected momentum direction**
- It encodes “where momentum is already pushing us”
- It corresponds to the *lookahead shift*

Think:
“if nothing changes, this is where we’re heading”

Term 2:

$$\frac{(1-\beta_1)g_t}{1-\beta_1^t}$$

- This is the **current gradient contribution**
- Bias-corrected
- Represents *new information*

Think:
“but what does the loss say right now?”

Combined meaning

$$\beta_1 \hat{m}_{t-1} + \frac{(1-\beta_1)g_t}{1-\beta_1^t}$$

This is a **convex combination** of:

- momentum-predicted direction (past)
- current gradient (present)

That combination approximates:

$$\nabla J(\theta_t - \text{momentum lookahead})$$

Why $\hat{m}_{t-1}$ appears and not $\hat{m}_t$

Because:

- $\hat{m}_t$ already includes g_t
- Using it directly would double-count the current gradient
- Nesterov logic needs **previous momentum + current gradient**

Timeline view (this clears everything)

At iteration t:

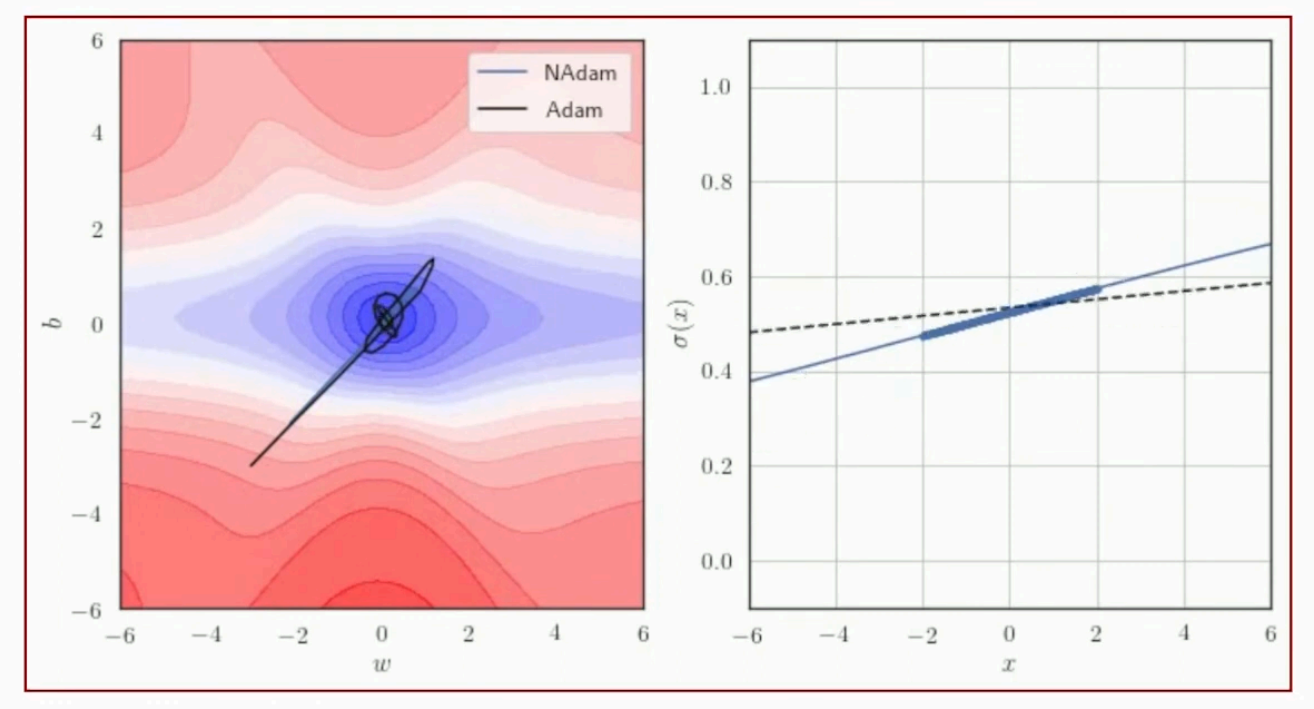
1. You **already know** m_{t-1}, $\hat{m}_{t-1}$, v_t
2. You compute $g_t = \nabla J(\theta_t)$
3. You form a **Nesterov-style corrected momentum**
4. You update θ_{t+1}

"NAdam does not look into the future; it approximates a Nesterov lookahead gradient using past momentum and the current gradient."

That it we get our update rule for nadam. ;)

Cool but boring.

To visualize the nadam vs adam



the animation can be seen in lecture: Source

Algorithm

INPUTS:

- Dataset: $X \in \mathbb{R}^{m \times n}$
- Targets: $y \in \mathbb{R}^m$
- Learning rate: η

- Exponential decay rate for first moment : $\beta_1 \in (0, 1)$
- Exponential decay rate for second moment : $\beta_2 \in (0, 1)$
- Small constant for numerical stability: ϵ
- Number of iterations: N

INITIALIZATION:

- Initial parameters: θ_0
- First moment (mean of gradients):

$$m_0 = 0$$

- Second moment (mean of squared gradients):

$$v_0 = 0$$

- Time step:

$$t = 0$$

COMPUTE:

For t=0 to N−1:

1. **Predict**

$$\hat{y}_b = X_b^\top \theta$$

2. **Compute loss**: $J(\theta)$
3. **Compute gradient**: $g_t = \nabla_\theta J(\theta)$
4. Update biased first moment estimate (momentum)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

5. Update biased second moment estimate (RMSProp)

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

6. Bias correction

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

   This step matters. Without it, early updates are wrong.
7. **Nesterov-corrected first moment**

$$\tilde{m}_t = \beta_1 \hat{m}_{t-1} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t}$$

   This term **approximates the lookahead gradient**.

8. Compute parameter update

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot \tilde{m}_t$$

9. Update parameters

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Code

```python
import numpy as np

def nadam_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float,
    beta1: float,
    beta2: float,
    n_iterations: int,
    batch_size: int,
    epsilon: float = 1e-8,
) -> np.ndarray:
    """
    NAdam optimizer for Mean Squared Error (MSE).

    Args:
        X (np.ndarray): Input data (m, n)
        y (np.ndarray): Targets (m,) or (m, 1)
        theta (np.ndarray): Initial parameters (n,)
        learning_rate (float): Learning rate (η)
        beta1 (float): First moment decay
        beta2 (float): Second moment decay
        n_iterations (int): Number of iterations
        batch_size (int): Mini-batch size
        epsilon (float): Numerical stability constant

    Returns:
        np.ndarray: Optimized parameters
    """

    m = X.shape[0]
```

```
    # Ensure column vectors
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    m_t = np.zeros_like(theta)
    v_t = np.zeros_like(theta)

    t = 0

    for _ in range(n_iterations):
        t += 1

        # ---- Mini-batch ----
        idx = np.random.choice(m, batch_size, replace=False)
        X_b = X[idx]
        y_b = y[idx]

        # ---- Gradient ----
        y_hat = X_b @ theta
        g_t = (2 / batch_size) * X_b.T @ (y_hat - y_b)

        # ---- Moment updates ----
        m_t = beta1 * m_t + (1 - beta1) * g_t
        v_t = beta2 * v_t + (1 - beta2) * (g_t ** 2)

        # ---- Bias correction ----
        m_hat = m_t / (1 - beta1 ** t)
        v_hat = v_t / (1 - beta2 ** t)

        # ---- Nesterov correction ----
        m_nesterov = beta1 * m_hat + (1 - beta1) * g_t / (1 - beta1 ** t)

        # ---- Parameter update ----
        theta = theta - learning_rate * m_nesterov / (np.sqrt(v_hat) +
epsilon)

    return theta
```

Weakness

1. **Not a guaranteed best choice:** There is no single best optimizer for all machine learning tasks. while Nadam can effective in many case. it might not always outperform

other optimizers like Adam or SGD. it still depends on factors like specific tasks, dataset properties, and model architecture
2. **Limited hyper parameter knowledge**
3. **Computational cost: adds extra calculation steps**
4. **Still potential for overshooting**: Nesterov momentum's look ahead strategy can sometimes lead the model to take larger steps than necessary during updates. this can cause the model to overshoot the minima point of the loss function. potentially leading to oscillation or even unstability in the training process. (we already seen this point in nesterov momentum)

10. AMSGrad

the machine learning practitioners noticed some cases where adaptive learning rate methods don't converge to an optimal solution and outperformed by SGD with momentum. (ahhh)

Reddi et al. (2018) figured out that "the exponential moving average of past squared gradients as a reason for the poor generalization behaviour of adaptive learning rate methods."

Recall that the introduction of the exponential average was well-motivated: It should prevent the learning rates to become infinitesimally small as training progresses, the key flaw of the Adagrad algorithm. However, this short-term memory of the gradients becomes an obstacle in other scenarios. (Wow our hero has some issues as well).

Adam also have convergence issues given author's example of simple convex optimization problem. #implementation

also the author gave a algorithm called "AMSGrad" that uses the maximum of past squared gradients v_t rather than the exponential average to update the parameters.

in adam:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

in AMSGrad:
instead of using v_t (or it's bias correction $\hat{v_t}$) directly, we employ previous v_{t-1} if is larger than the current one

$$\hat{v}_t = max(\hat{v}_{t-1}, v_t)$$

where ovio $\hat{v}_t$ not same as v_t.
"This way, AMSGrad results in a non-increasing step size, which avoids the problems suffered by Adam."
what they mean by "non-increasing step size " ?

For simplicity, the authors also remove the debiasing step that we have seen in Adam. #implementation

The full AMSGrad update without bias-corrected estimates are

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{v}_t &= \max(\hat{v}_{t-1}, v_t) \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t
\end{aligned}
$$

ha ha another variant of same step of update rule.

The authors observe improved performance compared to Adam on small datasets and on CIFAR-10. Link

More of optimizers Link

The more you see, the more you will be clear about what not to use or think of rather than what you should you use. so we are not too far to implement these in actual problems that already solved in past to understand the problem solving in deep learning more.

AMSGrad prevents Adam's adaptive learning rates from increasing by enforcing a non-decreasing second-moment estimate.

Intiution:
"Adam allows adaptive learning rates to increase during training due to decreasing second-moment estimates. AMSGrad prevents this by maintaining the maximum historical second moment, ensuring non-increasing effective learning rates and theoretical convergence guarantees."

Algorithm

INPUTS

- Dataset: $X \in \mathbb{R}^{m \times n}$
- Targets: $y \in \mathbb{R}^m$
- Learning rate: η
- Exponential decay rate for first moment : $\beta_1 \in (0, 1)$
- Exponential decay rate for second moment : $\beta_2 \in (0, 1)$
- Small constant for numerical stability: ϵ
- Number of iterations: N

INITIALIZATION:

- Initial parameters: θ_0
- First moment (mean of gradients):

$$m_0 = 0$$

- Second moment (mean of squared gradients):

$$v_0 = 0$$

- Maximum second moment:

$$\hat{v_0} = 0$$

- Time step:

$$t = 0$$

COMPUTE:

For t=0 to N−1:

1. **Predict**

$$\hat{y}_b = X_b^\top \theta$$

2. **Compute loss**: $J(\theta)$
3. **Compute gradient**: $g_t = \nabla_\theta J(\theta)$
4. Update biased first moment estimate (momentum)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

5. Update biased second moment estimate

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

6. Maintain maximum second moment

$$\hat{v_t} = max(\hat{v}_{t-1}, v_t)$$

   This is the **entire AMSGrad correction**.
7. Bias correction

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t}$$

   This step matters. Without it, early updates are wrong.
   (Second moment bias correction is unnecessary because of the max.)
8. **Nesterov-corrected first moment**

$$\tilde{m}_t = \beta_1 \hat{m}_{t-1} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t}$$

   This term **approximates the lookahead gradient**.
9. Compute parameter update

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot \hat{m}_t$$

10. Update parameters

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Code

```python
import numpy as np

def amsgrad_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float,
    beta1: float,
    beta2: float,
    n_iterations: int,
    batch_size: int,
    epsilon: float = 1e-8,
) -> np.ndarray:
    """
    AMSGrad optimizer for Mean Squared Error (MSE).

    Args:
        X (np.ndarray): Input data (m, n)
        y (np.ndarray): Targets (m,) or (m, 1)
        theta (np.ndarray): Initial parameters (n,)
        learning_rate (float): Learning rate (η)
        beta1 (float): First moment decay
        beta2 (float): Second moment decay
        n_iterations (int): Number of iterations
        batch_size (int): Mini-batch size
        epsilon (float): Numerical stability constant

    Returns:
        np.ndarray: Optimized parameters
    """

    m = X.shape[0]

    # Ensure column vectors
    y = y.reshape(-1, 1)
```

```
    theta = theta.reshape(-1, 1)

    m_t = np.zeros_like(theta)
    v_t = np.zeros_like(theta)
    v_hat = np.zeros_like(theta)  # max second moment

    t = 0

    for _ in range(n_iterations):
        t += 1

        # ---- Mini-batch ----
        idx = np.random.choice(m, batch_size, replace=False)
        X_b = X[idx]
        y_b = y[idx]

        # ---- Gradient ----
        y_hat = X_b @ theta
        g_t = (2 / batch_size) * X_b.T @ (y_hat - y_b)

        # ---- Moments ----
        m_t = beta1 * m_t + (1 - beta1) * g_t
        v_t = beta2 * v_t + (1 - beta2) * (g_t ** 2)

        # ---- AMSGrad correction ----
        v_hat = np.maximum(v_hat, v_t)

        # ---- Bias correction (first moment only) ----
        m_hat = m_t / (1 - beta1 ** t)

        # ---- Parameter update ----
        theta = theta - learning_rate * m_hat / (np.sqrt(v_hat) + epsilon)

    return theta
```

11. AdamW

Adam's L2 regularization is **mathematically incorrect**. So it solves this
What Adam was doing wrong ?

Explanations for Each Step

- **Step 1:** The gradient is calculated based on the current parameter value. For the function $f(\theta) = \theta^2$, the gradient $g_t = 2\theta_t$ represents the slope of the function at θ_t
- **Steps 2 and 3:** The first and second moment estimates (m_t and v_t) are updated using exponentially decaying averages of past gradients and squared gradients, respectively. These updates help the optimizer adjust the learning rate dynamically for each parameter, improving efficiency.
- **Steps 4 and 5:** Bias correction is applied to the moment estimates to address their initial bias toward zero. This correction is particularly important during the early stages of optimization, ensuring more accurate estimates.
- **Step 6:** The parameter is updated in two key parts:
  - Gradient Update: The parameter is adjusted in the opposite direction of the gradient. This adjustment is scaled by the learning rate and adapted using the corrected moment estimates.
  - Weight Decay: A regularization term is applied by reducing the parameter's value slightly. This encourages smaller parameter values, which helps to prevent overfitting.

By repeatedly performing these steps, the AdamW optimizer effectively moves the parameters closer to the function's minimum while controlling overfitting through the use of decoupled weight decay.

So What are the differences and problems it solved in Adam ?
the problem is with "L2 regularization with adam to have weight decay".

if you didn't completed the above required optimization please look into them before coming here to understand the terminology and formulas.

I will consider you have studied above optimization algorithms (specially the adam).

Let me elaborate
Loss

$$J(\theta) + \frac{\lambda}{2}\|\theta\|^2$$

Gradient

$$g_t = \nabla_\theta J(\theta_t) + \lambda\theta_t$$

Adam Update

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t$$

Where:
$\hat{m}_t$: contains the regularization term
$\hat{v}_t$: Scales it per parameter

This is where everything goes wrong.

Focus on the regularization part of the update:

$$-\frac{\eta}{\sqrt{\hat{v}_t}}\lambda\theta_t$$

This means:

- Each parameter is decayed by

$$\frac{\eta}{\sqrt{\hat{v}_{t,i}}}\lambda$$

So:

- Parameters with **large gradient variance** decay **less**
- Parameters with **small gradient variance** decay **more**
- Decay rate depends on optimizer history

That violates the **definition of weight decay**.
Weight decay means:

> "Shrink parameters uniformly, regardless of gradient behavior."

Adam does **not** do that.

Why SGD doesn't have this problem

Why SGD doesn't have this problem

$$\theta_{t+1} = \theta_t - \eta(\nabla J(\theta_t) + \lambda\theta_t)$$

Rearrange:

$$\theta_{t+1} = (1 - \eta\lambda)\theta_t - \eta\nabla J(\theta_t)$$

That is **true weight decay**.
SGD:

- does not rescale gradients
- does not distort regularization

Adam:

- rescales gradients
- accidentally rescales regularization

That's the bug.

How AdamW is solving that ?
Adamw separates the regularization term from optimization

it first
Step 1: optimization (pure Adam)

$$\theta \leftarrow \theta - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Step 2: regularization (pure weight decay)

$$\theta \leftarrow \theta - \eta\lambda\theta$$

That’s it.

Adam’s problem

- Regularization strength varies per parameter
- Depends on gradient history
- λ no longer means “regularization strength”
- Generalization suffers
- Training becomes optimizer-dependent in a bad way

AdamW’s fix

- Weight decay is **uniform**
- Independent of adaptive learning rates
- λ regains its meaning
- Better generalization
- Predictable behavior across architectures

AdamW doesn’t improve Adam’s *optimization*.
It fixes Adam’s **broken regularization semantics**.

One Liner:
"AdamW applies Adam’s adaptive update and weight decay as two separate steps, ensuring true L2 regularization."

"Adam mixes regularization into adaptive gradients, breaking weight decay; AdamW decouples them, restoring correct regularization."

Here is a image from the conference paper of adamw

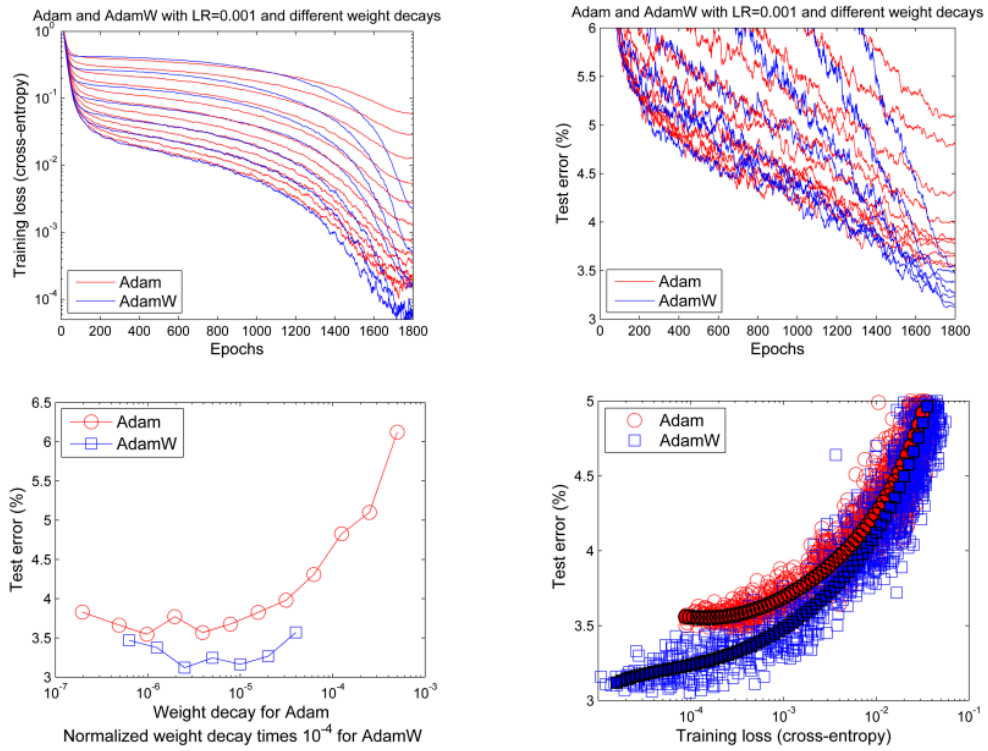
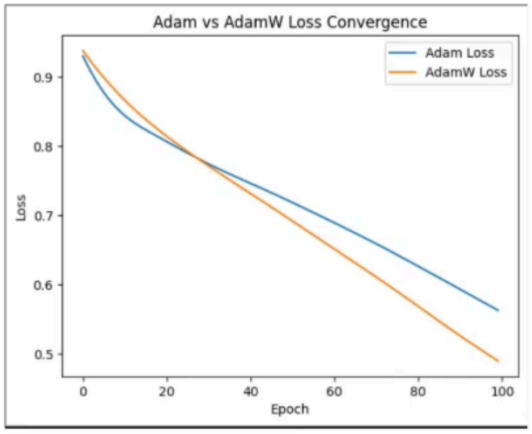
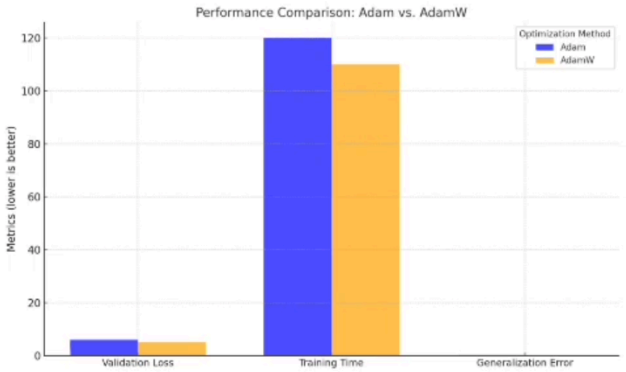


Figure 3: Learning curves (top row) and generalization results (bottom row) obtained by a 26 2x96d ResNet trained with Adam and AdamW on CIFAR-10. See text for details. SuppFigure 4 in the Appendix shows the same qualitative results for ImageNet32x32.

Source: Link

which proves Adamw Learn and generalize better as compare to Adam. as this is tested on ResNet.

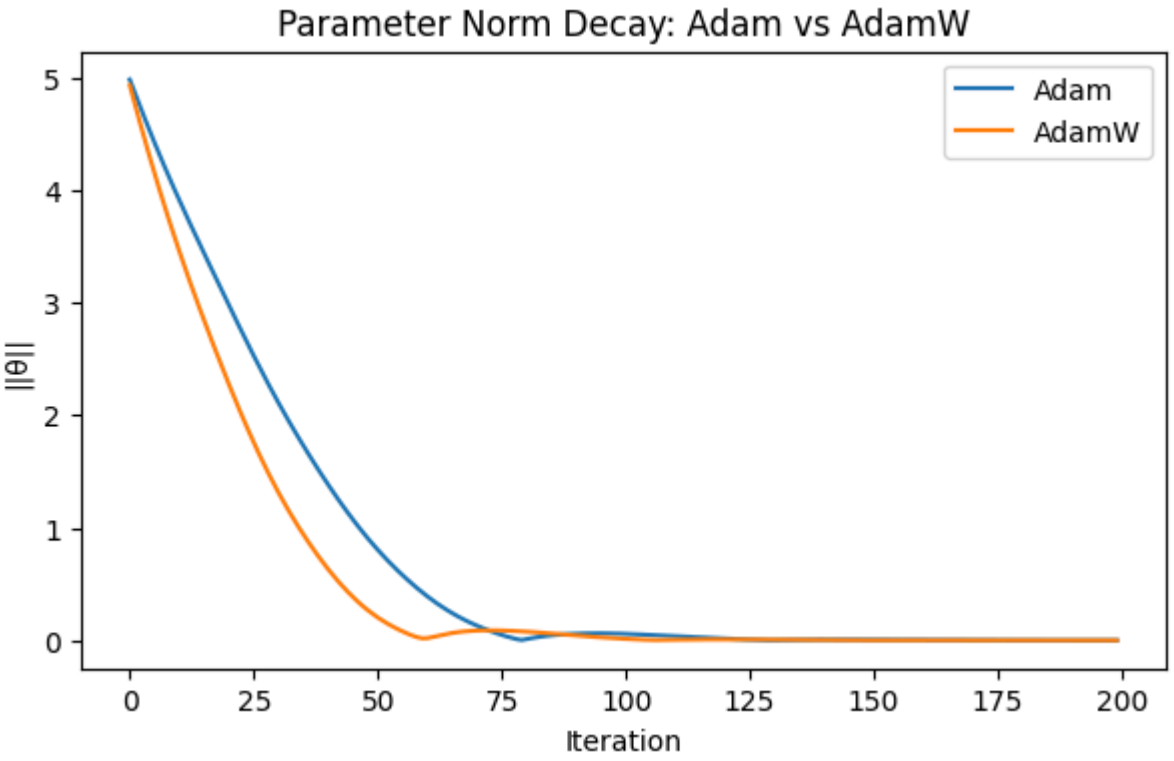




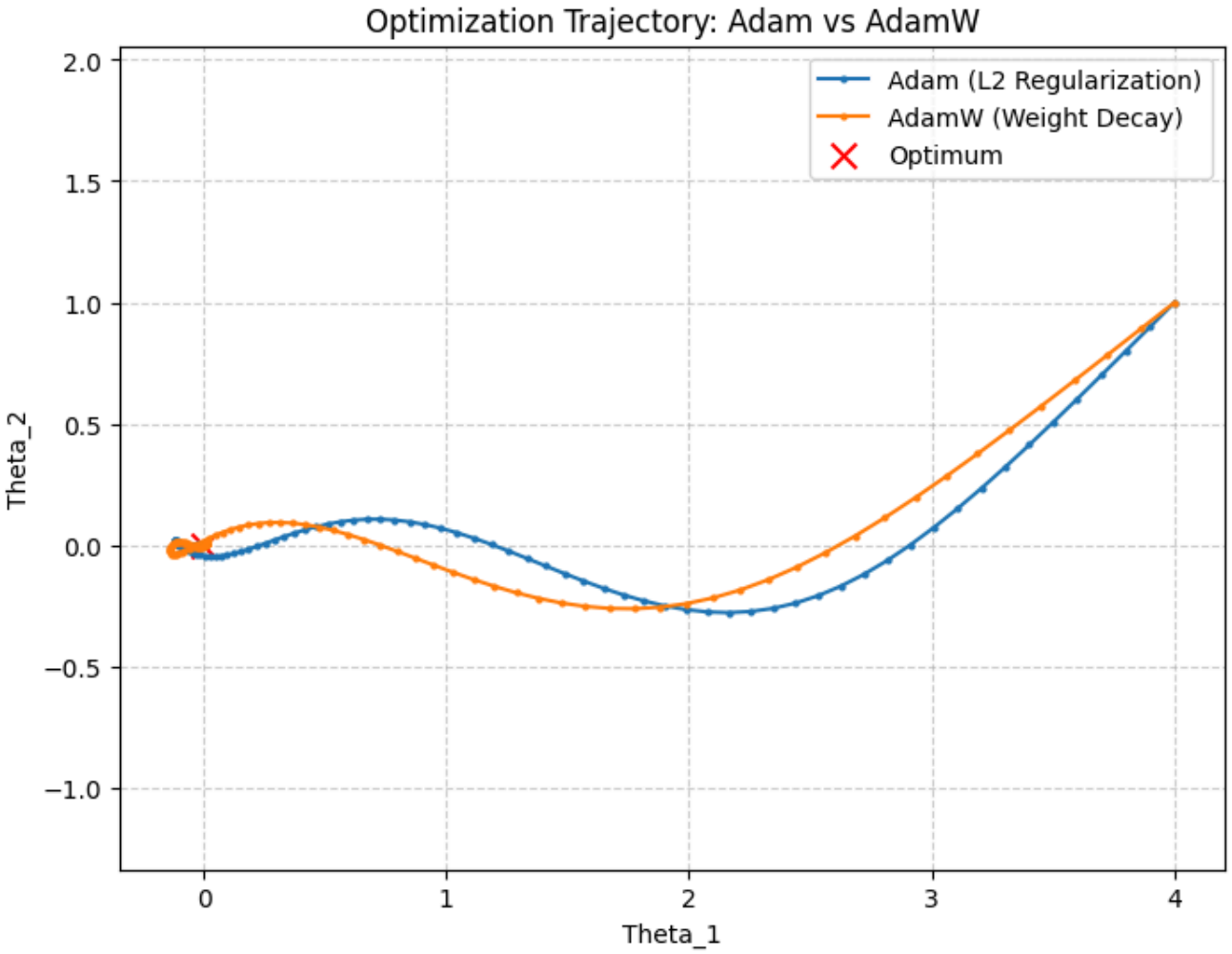
Source: Link

it converge better in less training time and there will be less validation loss as compare to adamW.

Where as we can see



- **Adam**: norm decays erratically, plateaus, sometimes barely shrinks
- **AdamW**: smooth, monotonic decay



Adam (L2 Regularization): The path is often more "curved" or aggressive in its early steps. Because the L_2 penalty is added directly to the gradient (g), it gets factored into the second moment (v_t). This means the optimizer "scales down" the regularization effect for dimensions with large gradients.

AdamW (Weight Decay): The path tends to be more direct. Since the weight decay is applied **after** the adaptive gradient update, it acts as a constant "pull" toward the origin $(0, 0)$ that isn't distorted by the historical gradient magnitudes.

Convergence behavior

- **Coupling vs. Decoupling:** In Adam, the regularization is "coupled" with the learning rate and the adaptive denominators. In AdamW, the decay is "decoupled."
- **Observation:** You will notice that AdamW often reaches a tighter region around the optimum. In practical deep learning, this decoupling is the primary reason AdamW usually yields better generalization than Adam with L_2 penalty.

Visualization

- **Starting Point:** Both start at $(4, 1)$.
- **The "Pull":** Both are being pulled toward $(0, 0)$ by two forces: the Loss Gradient (the "task") and the Regularization/Decay (the "constraint").

- **The Divergence:** The gap between the two lines illustrates how much "signal" is lost when you mix the regularization into the adaptive moving averages.

Adam mixes L2 regularization into adaptive gradients, causing non-uniform and state-dependent parameter shrinkage. AdamW decouples weight decay from gradient adaptation, producing consistent norm decay.

1. Convergence Rate (The "Slope")

In the resulting graph, you will likely notice that **AdamW converges toward zero more efficiently** than Adam.

- **Adam (L2):** Because the penalty is inside the gradient g, it is divided by $\sqrt{v_t}$ (the adaptive learning rate). If the historical gradients are large, the regularization effect is inadvertently "squashed" or diminished.
- **AdamW:** The decay is constant and independent of the gradient's second moment. This ensures that the weights are penalized consistently regardless of how noisy or large the gradients are.

2. The Log Scale Observation

If you use a log scale for the Y-axis (`plt.yscale('log')`), the difference becomes even clearer. AdamW typically shows a more linear (faster) decay in log-space. This is why AdamW is the preferred optimizer for large models (like Transformers)—it prevents weights from growing too large more effectively than standard Adam.

Algorithm

INPUTS

- Dataset: $X \in \mathbb{R}^{m \times n}$
- Targets: $y \in \mathbb{R}^m$
- Learning rate: η
- Exponential decay rate for first moment : $\beta_1 \in (0, 1)$
- Exponential decay rate for second moment : $\beta_2 \in (0, 1)$
- Small constant for numerical stability: ϵ
- Number of iterations: N

INITIALIZATION

- Initial parameters: θ_0
- First moment (mean of gradients):

$$m_0$$

- Second moment (mean of squared gradients):

$$v_0$$

- Time step:

$$t = 0$$

COMPUTE

For t=0 to N−1:

1. **Predict**

$$\hat{y}_b = X_b^\top \theta$$

2. **Compute loss**: $J(\theta)$
3. **Compute gradient**: $g_t = \nabla_\theta J(\theta)$
4. Update biased first moment estimate (momentum)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

5. Update biased second moment estimate (RMSProp)

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

6. Bias correction

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

This step matters. Without it, early updates are wrong.

7. Adaptive update

$$\theta_{t+\frac{1}{2}} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t$$

8. Decoupled weight decay

$$\theta_{t+\frac{1}{2}} = \theta_t - \eta\lambda\theta_t$$

Code

```python
import numpy as np

def adamw_gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    learning_rate: float,
    weight_decay: float,
    beta1: float,
    beta2: float,
    n_iterations: int,
    batch_size: int,
    epsilon: float = 1e-8,
) -> np.ndarray:
    """
    AdamW optimizer for Mean Squared Error (MSE).

    Args:
        X (np.ndarray): Input data (m, n)
        y (np.ndarray): Targets (m,) or (m, 1)
        theta (np.ndarray): Initial parameters (n,)
        learning_rate (float): Learning rate (η)
        weight_decay (float): Weight decay coefficient (λ)
        beta1 (float): First moment decay
        beta2 (float): Second moment decay
        n_iterations (int): Number of iterations
        batch_size (int): Mini-batch size
        epsilon (float): Numerical stability constant
```

```
    Returns:
        np.ndarray: Optimized parameters
    """

    m = X.shape[0]

    # Ensure column vectors
    y = y.reshape(-1, 1)
    theta = theta.reshape(-1, 1)

    m_t = np.zeros_like(theta)
    v_t = np.zeros_like(theta)

    t = 0

    for _ in range(n_iterations):
        t += 1

        # ---- Mini-batch ----
        idx = np.random.choice(m, batch_size, replace=False)
        X_b = X[idx]
        y_b = y[idx]

        # ---- Gradient ----
        y_hat = X_b @ theta
        g_t = (2 / batch_size) * X_b.T @ (y_hat - y_b)

        # ---- Moments ----
        m_t = beta1 * m_t + (1 - beta1) * g_t
        v_t = beta2 * v_t + (1 - beta2) * (g_t ** 2)

        # ---- Bias correction ----
        m_hat = m_t / (1 - beta1 ** t)
        v_hat = v_t / (1 - beta2 ** t)

        # ---- Adam update ----
        theta = theta - learning_rate * m_hat / (np.sqrt(v_hat) + epsilon)

        # ---- Decoupled weight decay ----
        theta = theta - learning_rate * weight_decay * theta
```

```
    return theta
```

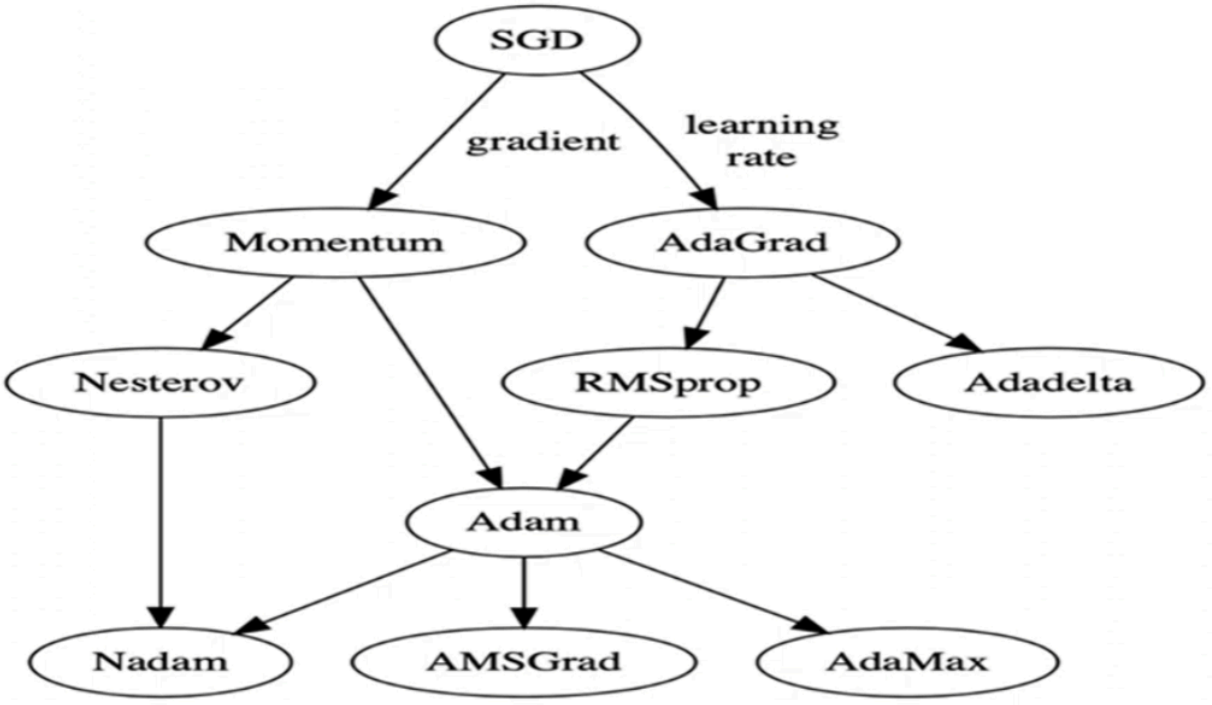
Weakness

- Heavy memory usage
- Complex state
- Overkills at scale

Solution

> It's like Optimization algorithms didn't get smarter.
> They got more **aware of geometry, history, and hardware**.

and we can vizualize their history with below diagram



Source: Link

In Next Note we will cover more Optimization algorithm

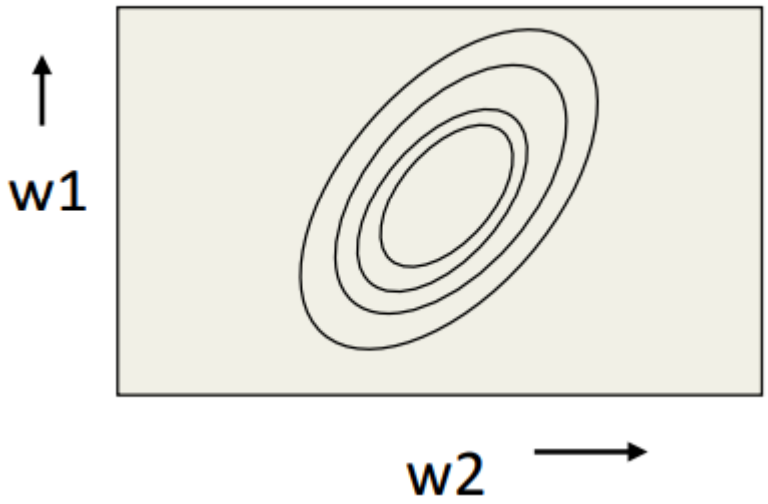
1. Lion
2. QHAdam
3. AggMo
4. YellowFin
5. QHM
6. Demon Adam
7. Shampoo / K-FAC
8. LAMB / LARC

9. Adafactor

Understand few more terminologies

Concept - 1

Below is a 2d image of a 3d terrain of a loss function. and
This is a **contour plot**, which is a common method for visualizing a three-dimensional surface in two dimensions.

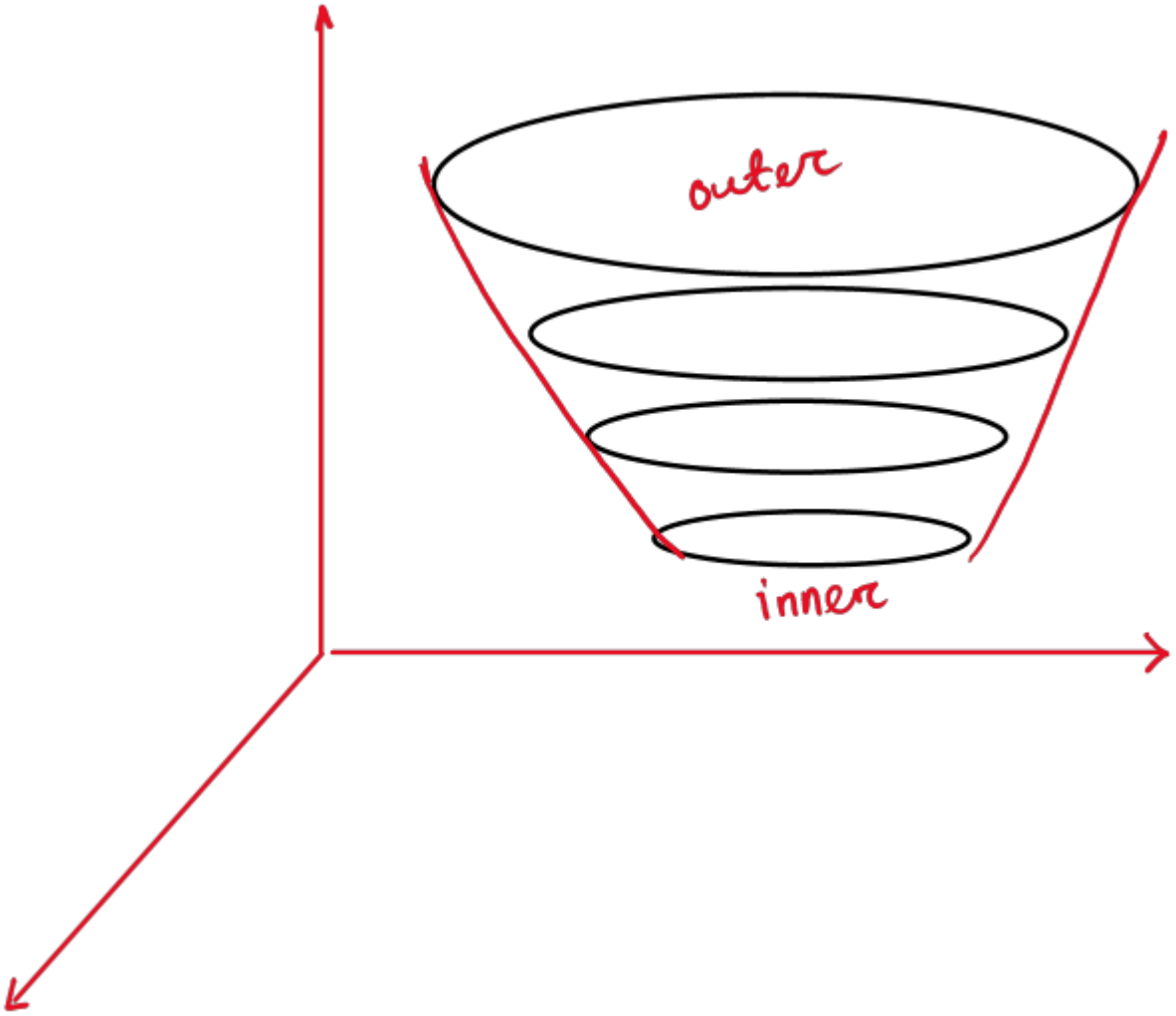


The w1 and w2 axes represent the two input dimensions (like coordinates on a map). and
The elliptical lines (contours) represent points that all have the same "height" or value in the third dimension.

it's like can be seen as 2 ways

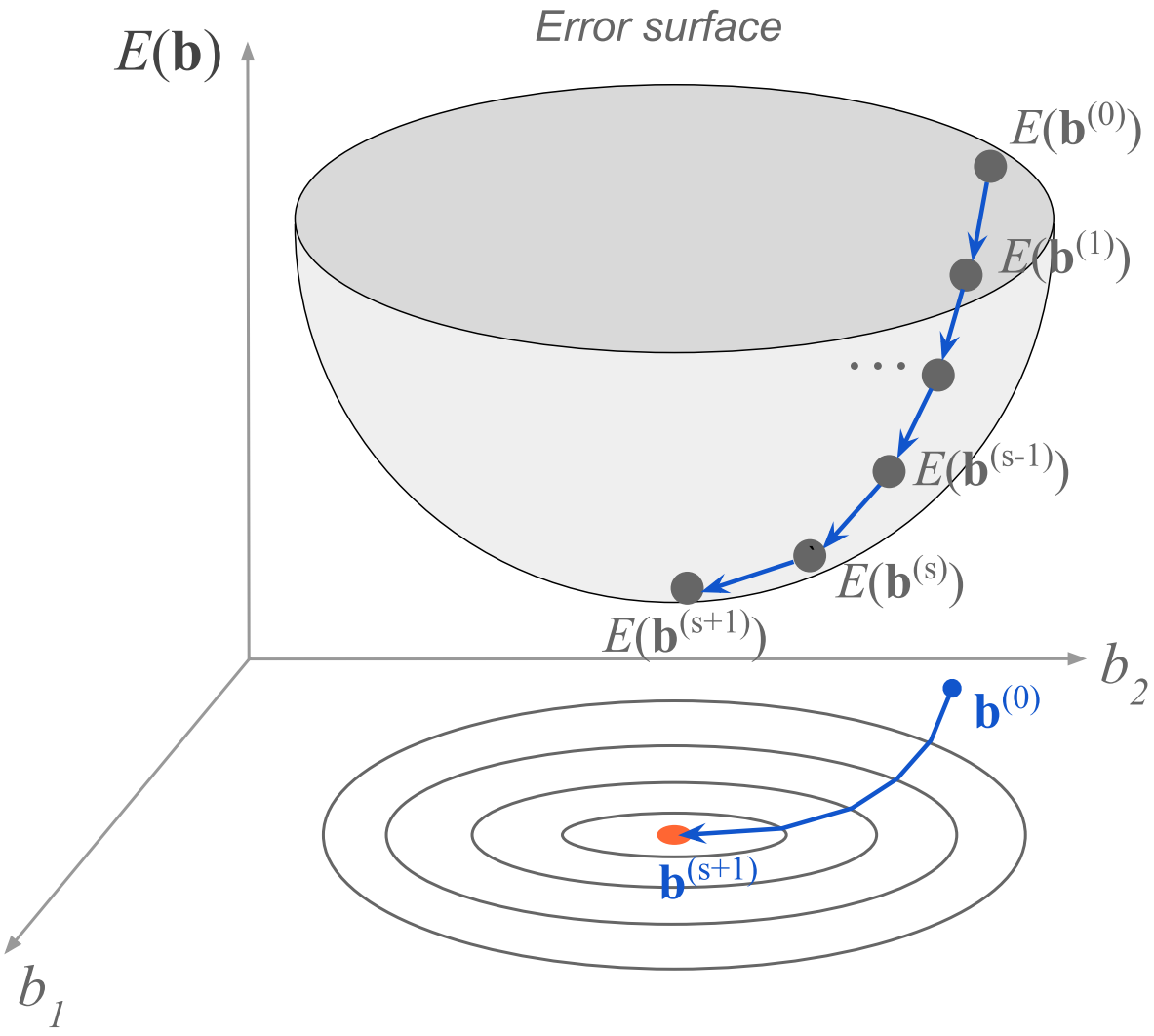
1. Hill (which is not likely in ML)

2. Valley (which is highly likely)



Valley representation

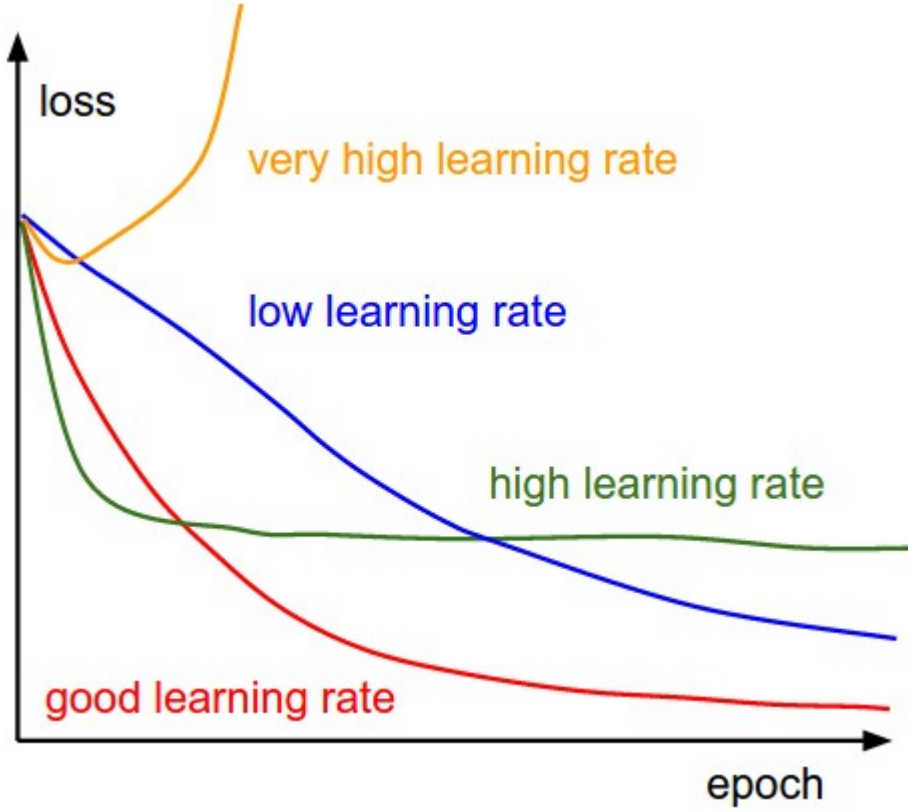
Here is a good one



Source: Link

Concept-2

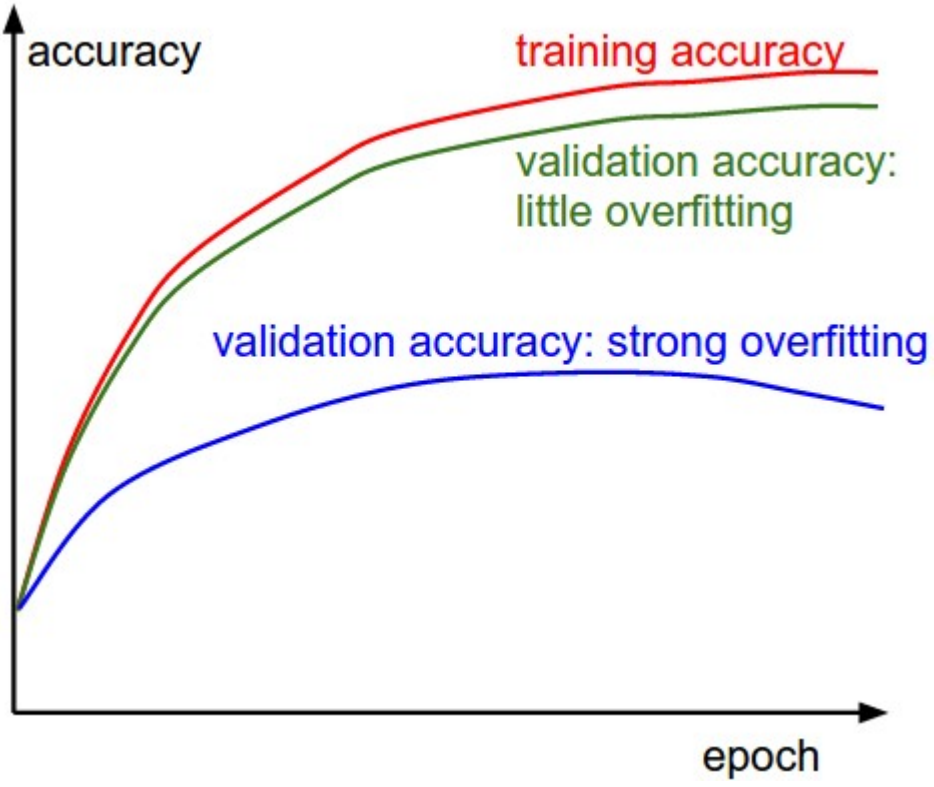
In machine learning, we always want the red line plot in below graph.



Source: Link

Concept-3

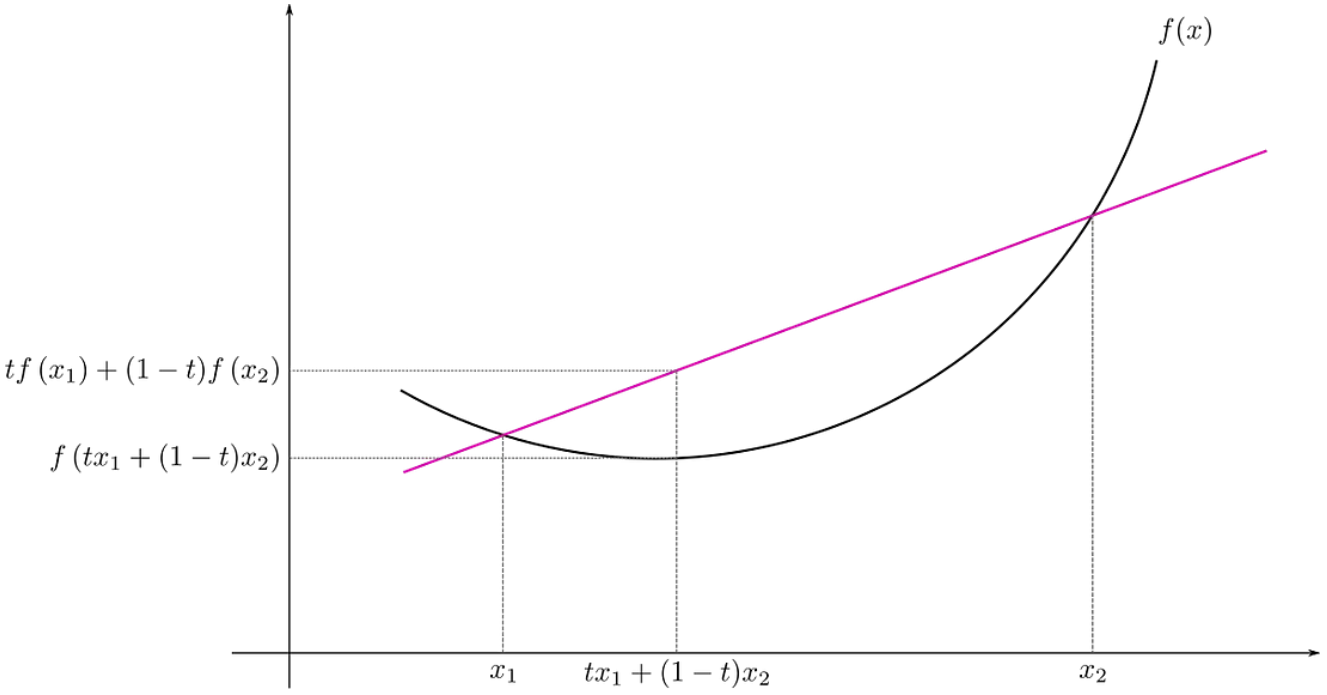
we always want the accuracy gap between training and validation should be more close (as close as possible) to avoid the case of overfitting.



Concept - 4

Convex function

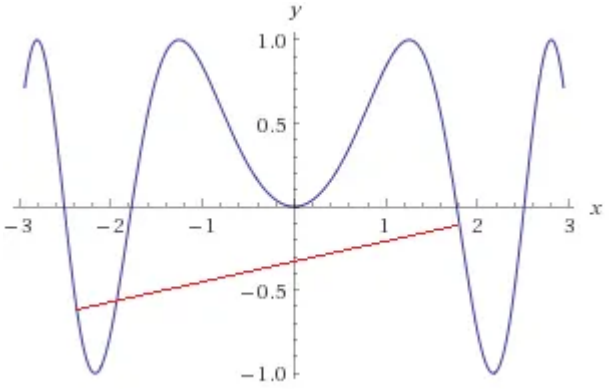
Convex functions are particularly important because they have a unique global minimum. (a single minima)
This means that if we want to optimize a convex function, we can be sure that we will always find the best solution by searching for the minimum value of the function.



Source: Link

Non Convex function

A function is said to be non-convex if it is not convex. Geometrically, a non-convex function is one that curves downwards or has multiple peaks and valleys.



Source: Link

Concept - 5

Problem with single global learning rate

1. Ill-conditioned curvature

The most common issue is that the error surface often looks like a narrow ravine or a taco shell—it is very steep in one direction and very flat in another.

- **The Dilemma:**
  - **If the rate is high:** You make good progress along the flat bottom of the ravine, but you **overshoot and oscillate** wildly up and down the steep walls.
  - **If the rate is low:** You stop oscillating on the steep walls, but you crawl **painfully slowly** along the flat bottom.
    Imagine this loss:

$$L(x, y) = x^2 + 15y^2$$

- x direction: shallow
- y direction: steep

Gradient:

$$\nabla L = (2x, 30y)$$

Now use one η.

If η is small

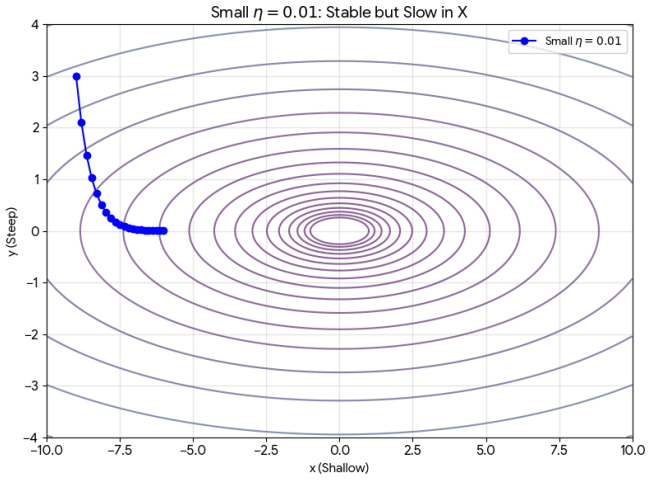
- Stable in y
- Painfully slow in x

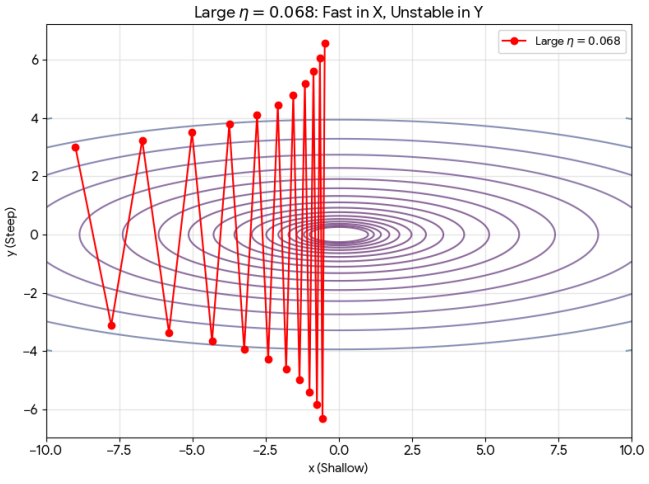
If η is large

- Makes progress in x
- Explodes or oscillates in y

There is **no η** that is optimal for both directions.

This isn't bad tuning. It's mathematically impossible.





2: Zig-zag inefficiency

In narrow valleys:

- Gradient always points toward steep walls
- Not toward the minimum along the valley

Single learning rate + gradient direction ⇒

- bounce left
- bounce right
- inch forward

You waste steps correcting errors caused by your own step size.

3: Feature scale sensitivity

If features aren’t normalized:

- One feature has values ~1000
- Another has values ~0.01

Same η means:

- one parameter explodes
- another barely moves

Now optimization depends on **units**, not learning.

4. Sparse vs dense features

- **Dense features:** Accumulate gradient signals constantly. A high global rate makes them explode.
- **Sparse features:** Receive signals rarely. A low global rate means they practically never update.
  In NLP, recommender systems, embeddings:
- Some parameters get gradients every step
- Some get gradients once in a while

Single learning rate:

- Frequent features dominate
- Rare features learn too slowly
  That's why we need different learning rate for different parameters.

5. Different training phases need different η

Early training:

- gradients large

- exploration needed

Late training:

- gradients small
- precision needed

One learning rate:

- either slow start
- or overshoot at the end

Schedulers try to patch this, but it’s still global and crude.

> Don't skip this

1. Start and complete each notes in sequence.
2. Refer to the source for more detail
3. if you get any doubt i am a message away.
4. Don't panic with vast syllabus. it's cool to complete these concepts in time.
5. Believe in yourself.

Have Fun ;)