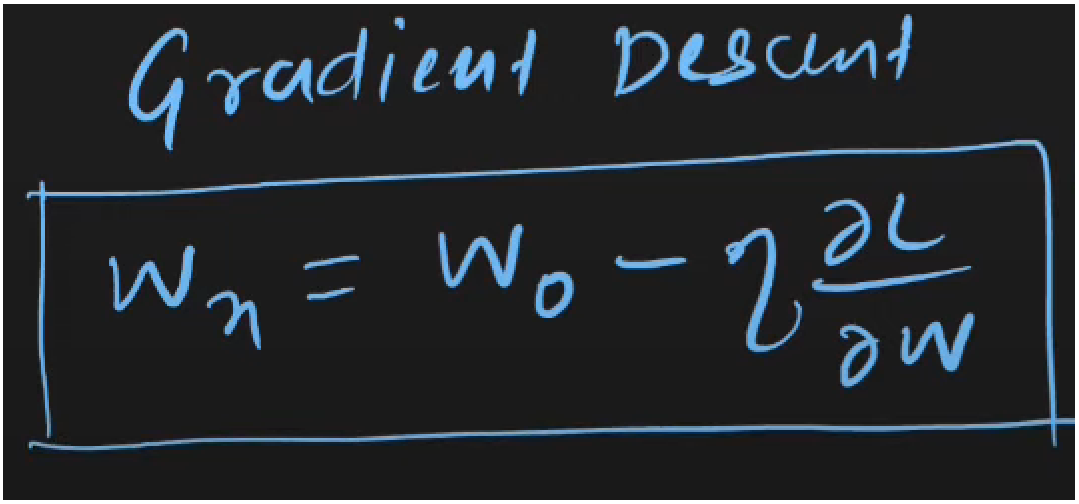
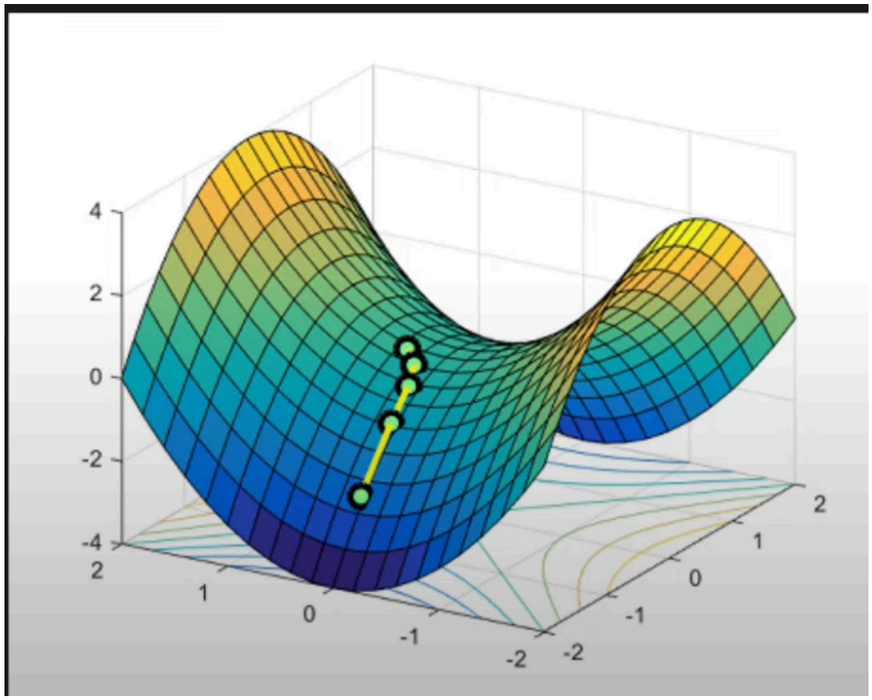
Optimizers in Deep Learning

The optimizer we use in DL is **Gradient Descent:**



Challenges with GD:

- Deciding the value of learning rate (η)
- Cannot set a custom learning rate for individual weights and biases
- Local minima problem
    - We can get stuck in Local Minima
- Saddle point
    - There's a point where slope does not change
    - Therefore, weights don't get updated.



The following optimizers solve the above issues:

1. Momentum
2. Adagrad (Adaptive Gradient Algorithm)
3. NAG
4. RMSprop (Root Mean Square Propagation)
5. Adam

Key concept used in above optimizers → Exponentially weighted moving average

Exponentially weighted moving average

- The **Exponentially Weighted Moving Average (EWMA)** is a smoothing technique that gives more weight to recent data while gradually "forgetting" older observations.

- It's a **weighted average**, but the weights **decay exponentially**
- EWMA gives more importance to recent weights
- As time passes, the weight of a point decreases exponentially

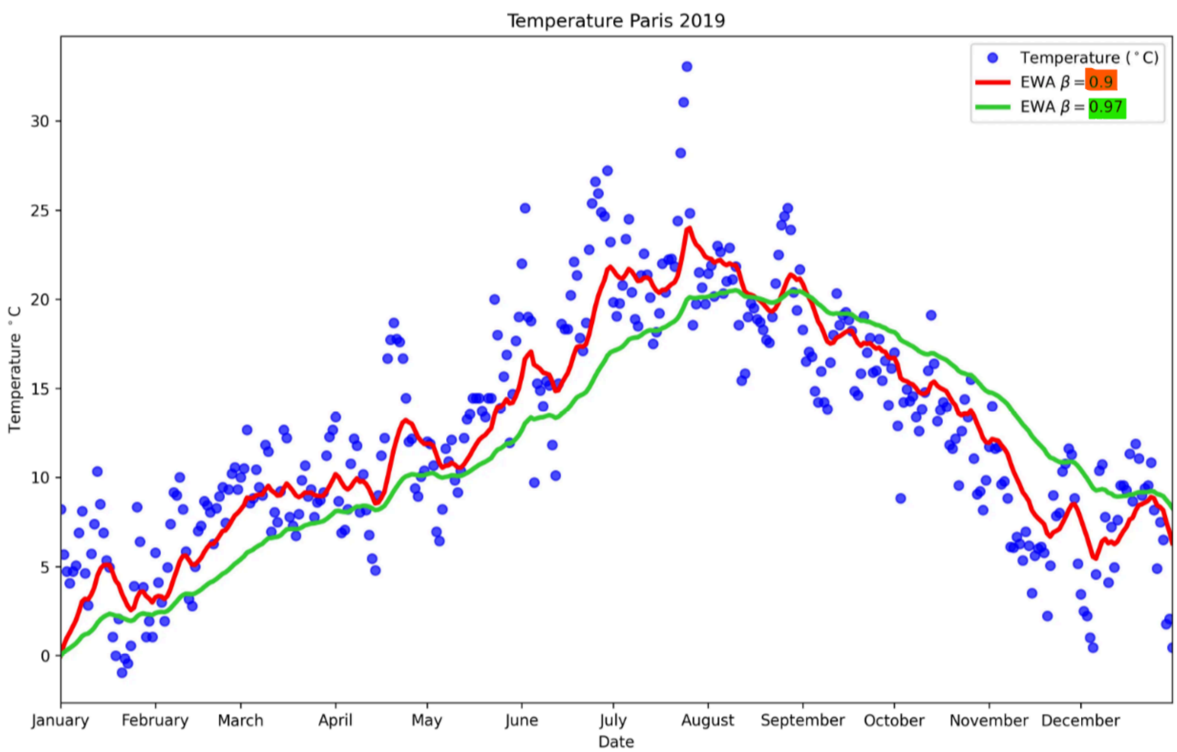
It's widely used in:

- **Time series forecasting** (e.g., stock prices, weather data).
- **Optimization algorithms** (e.g., Momentum in SGD, Adam, RMSprop).
- **Signal processing** (noise reduction).

🎯 Goal: Capture short-term trends while still considering past data.

🧠 Why Use EWMA?

Problem	EWMA Fix
Too much noise in time-series	✅ Smooths fluctuations
Need quick reaction to new data	✅ Gives recent data more weight
Need memory of older data	✅ Never fully discards old values



Formula:

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

- θ_t: Current observation (e.g., gradient, stock price).
- β: Decay rate (typically **0.9**, **0.99**, etc.).
    - Higher β = smoother but slower to adapt.
    - Lower β = noisier but more responsive.

Intuition

- Think of it as a **leaky average,** where older data decays exponentially.
- ***Example***: If β=0.9, the weight of past observations decays like:

Current=10%,

1 step back=9%,

2 steps back=8.1%,etc.

β Value:

Close to 1 (e.g., 0.9)	Fast response, recent data dominates (**Most common**)
Close to 0 (e.g., 0.1)	Slow response, more smoothing

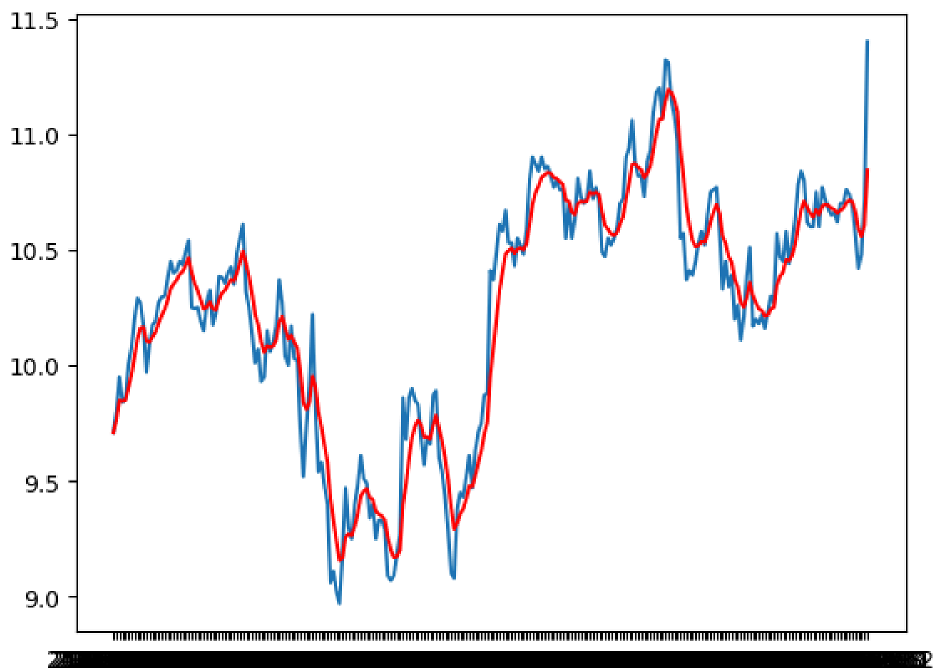
- **β closer to 1**: The weight on the most recent observation $x_i$is larger, meaning the EWMA is more sensitive to recent changes and reacts more quickly to new data.
    - This is typically used when you want to **prioritize recent values** more heavily.
- **β closer to 0**: The weight on the most recent observation is smaller, and the EWMA places more importance on older values, making it **less sensitive to recent changes**.
    - This is used when you want a **smoother** time series that reacts more slowly to new data.

EWMA in Python:

```
df['Close'].ewm(alpha=0.9).mean()
```

```
0       9.710000
1       9.791818
2       9.934324
3       9.849424
4       9.854442
          ...
246    10.573514
247    10.435351
248    10.475535
249    10.695554
250    11.329555
Name: Close, Length: 251, dtype: float64
```

```
plt.plot(df['Date'], df['Close'])
df['Close'].ewm(alpha=0.3).mean().plot(color='r')
```

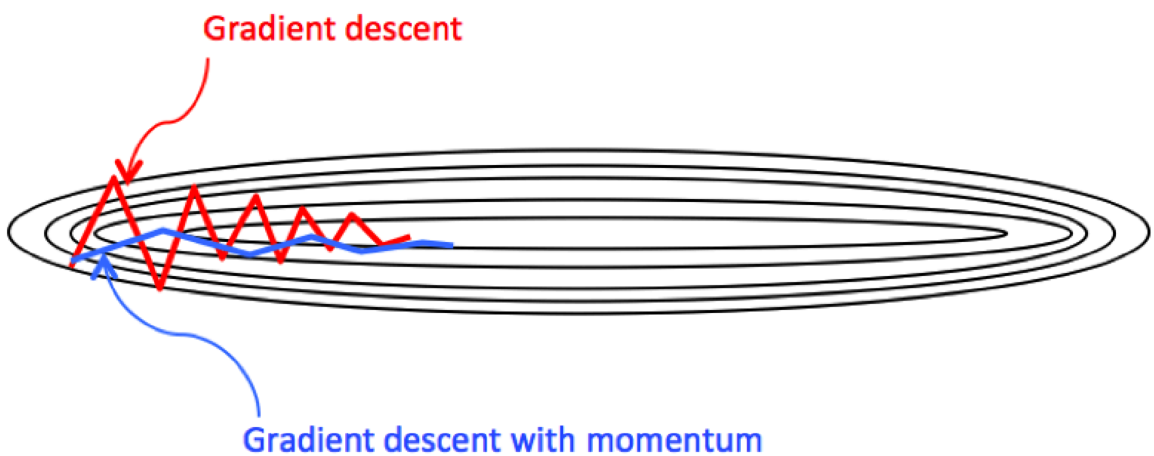


OPTIMIZERS 👇

1. SGD with *Momentum* (🚀*Speeed*)

💡 **SGD with Momentum** is an extension of the basic **Stochastic Gradient Descent (SGD)** optimization algorithm.

- **Stochastic Gradient Descent (SGD) with Momentum** is like rolling a ball downhill—it uses past gradients to **accelerate convergence** and **escape local minima**.



🚀 Key Benefits

✅ **Faster convergence**: Especially in flat or noisy loss landscapes.

✅ **Escapes local minima**: Momentum carries it through small bumps.

✅ **Less oscillation**: Smoother updates than vanilla SGD.

✅ **Solves High Curvature problem**

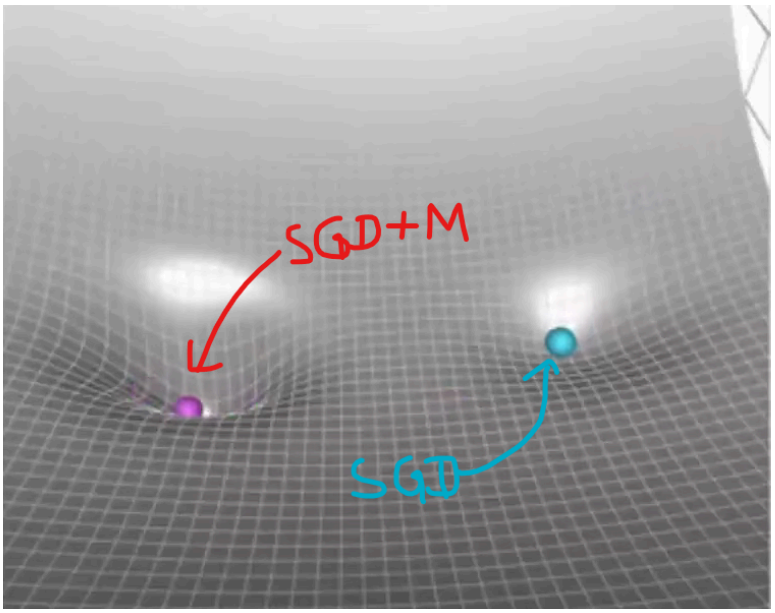
❌ Problem with vanilla SGD:

- Slow in narrow valleys

- Oscillates up and down in steep areas
- Gets stuck in local minima

✅ Momentum Fix:

- Adds **inertia** to the updates
- Helps SGD roll down like a **ball** with memory of past direction



Formula:

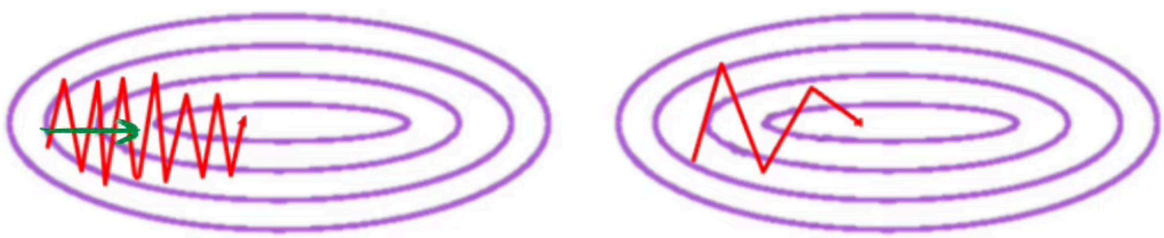
$$v_t = \beta v_{t-1} + (1-\beta)\nabla_\theta J(\theta_t)$$

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

- v_t: Velocity (exponentially weighted average of past gradients).
- β: Momentum hyperparameter (**0.9** is typical).
- α: Learning rate.

- 👆 You use history of past velocities.

- It provides momentum



β Value:

- β = 0 → SGD
- β = 1 → Dynamic equilibrium. No decay
- β = 0.9 (Most common)

🧠 Intuition (Ball in a Valley Analogy)

- Plain SGD: Like dropping a ball in a bumpy valley; it jumps side to side.
- SGD with Momentum: Ball gains **velocity** and rolls faster, **smoother**, and more directly toward the bottom.

Python code:

```
from keras.optimizers import SGD

# SGD with momentum = 0.9
optimizer = SGD(learning_rate=0.01, momentum=0.9)

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

`learning_rate=0.01` **→ By default**

`momentum= 0.0` **(Default)**

🧾 SGD vs SGD + Momentum

Feature	SGD	SGD + Momentum
Uses gradient only	✅	✅
Uses past gradients	❌	✅
Smooth updates	❌	✅
Converges faster	❌	✅
Handles ravines	❌	✅

📊 Why Momentum Beats Vanilla SGD

Scenario	Vanilla SGD	SGD + Momentum
Flat regions	Crawls slowly	Accelerates through
Noisy gradients	Oscillates wildly	Smoothens updates
Local minima	Gets stuck	Rolls past them

Disadvantage of SGD + Momentum

- **Oscillations**: In regions of the loss surface where the gradient points in opposite directions (e.g., in valleys), SGD might oscillate back and forth.

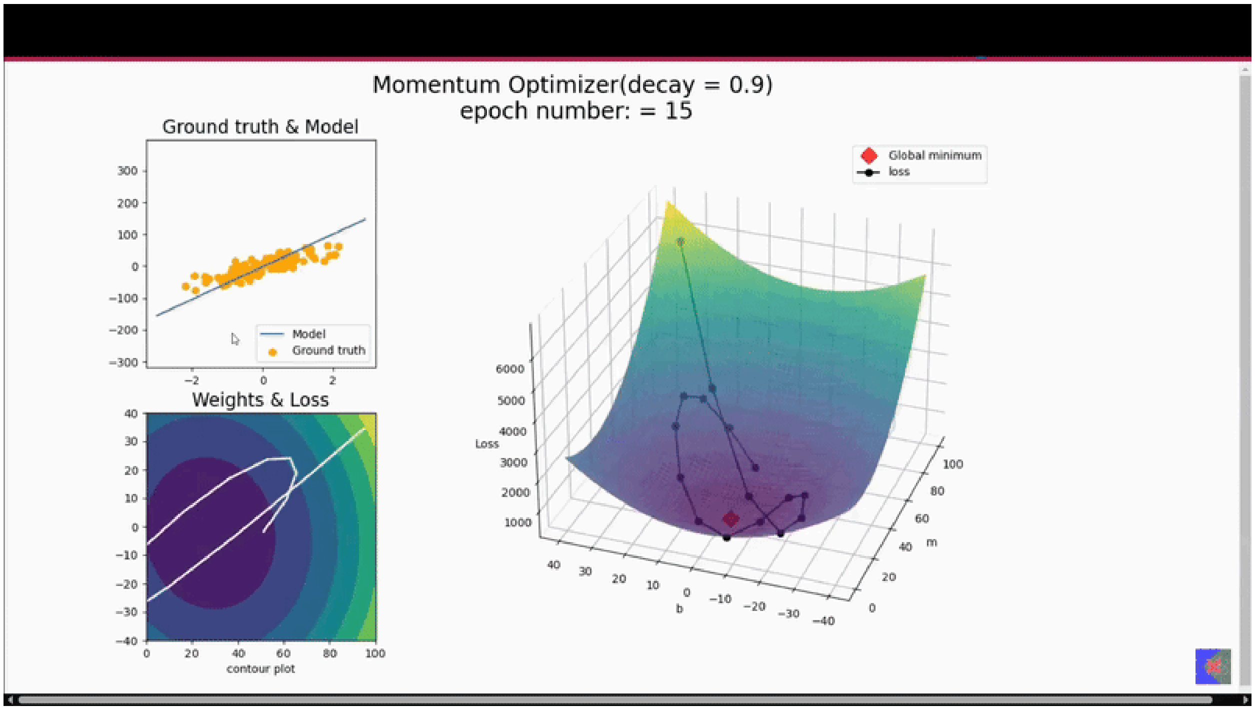
2. Nesterov Accelerated Gradient (NAG) 🐍

```
SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
```

- Upgrade to SGD Momentum
- Mostly performs better than SGD Momentum

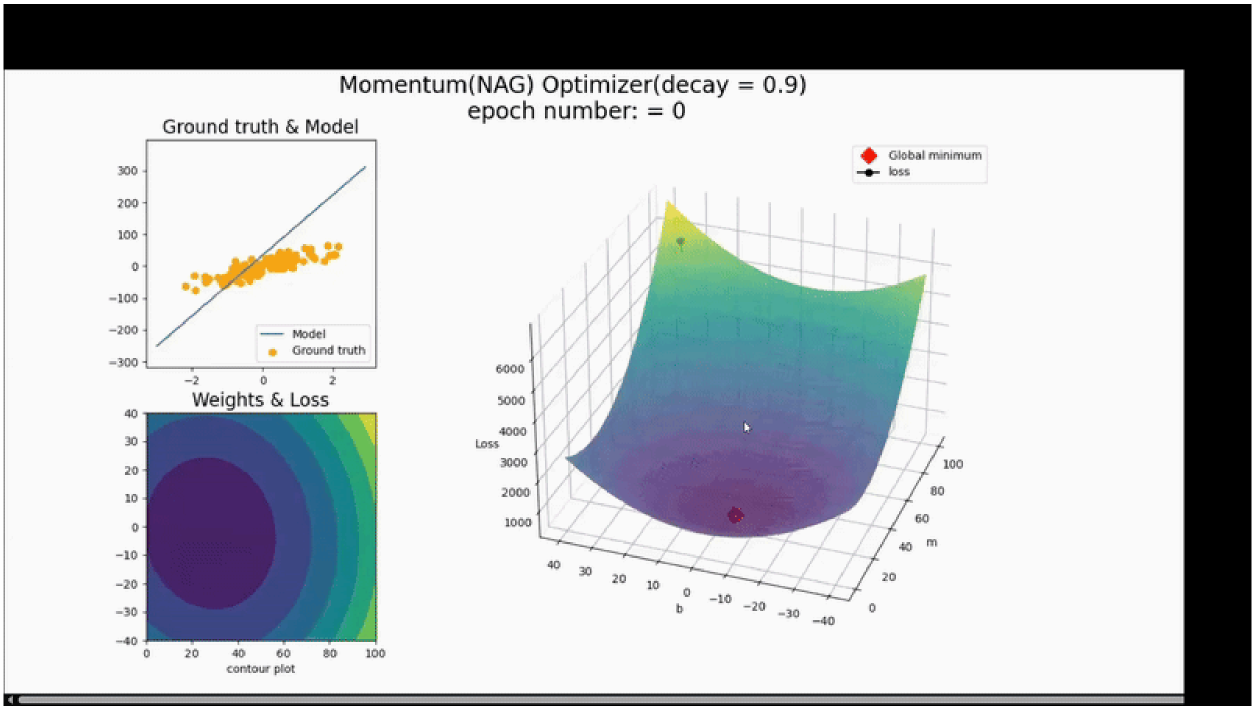
attachment:e9887393-6286-49e9-8a30-a05a3d4d7f45:2025-04-05_23-46-22.mp4

SGD Momentum 👆 👇



- The above👆 graph is taking more epochs to reach to minima cuz the decay factor is 0.9.
  - i.e. it's giving more importance to the recept weights
- If we decrease the decay factor, the oscillations will reduce
- **NAG helps to decrease the oscillations by keeping the decay constant.**

NAG with same decay i.e. 9 👇



🎯

Goal: Make more accurate and faster updates by "looking ahead" before calculating the gradient.

> **"Before I take the next step, let me check the slope a little ahead of where I'm going."**

Why Use NAG?

Issue in SGD + Momentum	NAG Solution
Momentum blindly trusts past direction	NAG looks ahead before moving
Overshoots minima	NAG applies **correction** before stepping
Slower convergence	NAG converges **faster and smoother**

📊 NAG vs. Classic Momentum

Aspect	**Classic Momentum**	**Nesterov Momentum**

Gradient Calculation	At current position	At "lookahead" position
Overshooting	More likely	Reduced
Convergence Speed	Fast	**Faster**
Stability	Good	**Better**

Update Rules – Step by Step

✅ Step 1:

- **Lookahead Step:** Instead of applying the momentum update directly on the current parameters

  θ_t, NAG first performs a lookahead step to predict where the parameters will be after the update:

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

✅ Step 2: Compute Gradient at Lookahead

- Gradient is taken at the **lookahead position**, not current position.

$$\nabla J(\tilde{\theta}_t)$$

✅ Step 3: Update Velocity

$$v_t = \gamma \cdot v_{t-1} + \eta \cdot \nabla J(\tilde{\theta}_t)$$

✅ Step 4: Update Weights

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

⚙️ Python (Keras) Implementation

```
from keras.optimizers import SGD

# SGD with Nesterov Momentum
optimizer = SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
```

Explanation:

- `momentum=0.9` : Standard momentum value
- `nesterov=True` : Enables lookahead behavior

🧪 Simple Analogy

> Plain Momentum:
>
> **"I'm going downhill fast. Keep going!"**

> NAG:
>
> **"I'm going downhill fast. But let me look a little ahead first to see if I should slow down."**

🛠️ When to Use NAG?

✅ Ideal for:

- Deep neural networks
- Highly non-convex loss surfaces
- Image classification, NLP, time series forecasting

Disadvantage of NAG

- You can get stuck in local minima.
    - ***Momentum*** has enough power(momentum) to get out of the local minima.

🚫When to Avoid NAG?

❌ When you're using **Adam** or **RMSProp**, which already handle momentum smartly.

❌ Very noisy gradients (Adam/RMSprop may be better).
❌ Extremely large batches (momentum matters less).

3. AdaGrad Optimizer

AdaGrad (Adaptive Gradient Algorithm) **adjusts the learning rate individually for each parameter**, based on how frequently it's been updated.

> 💡 **Adagrad is not used in Neural Networks.**

🎯 Goal: Take larger steps for infrequent features, and smaller steps for frequent ones.

Why Use AdaGrad?

Problem with SGD:

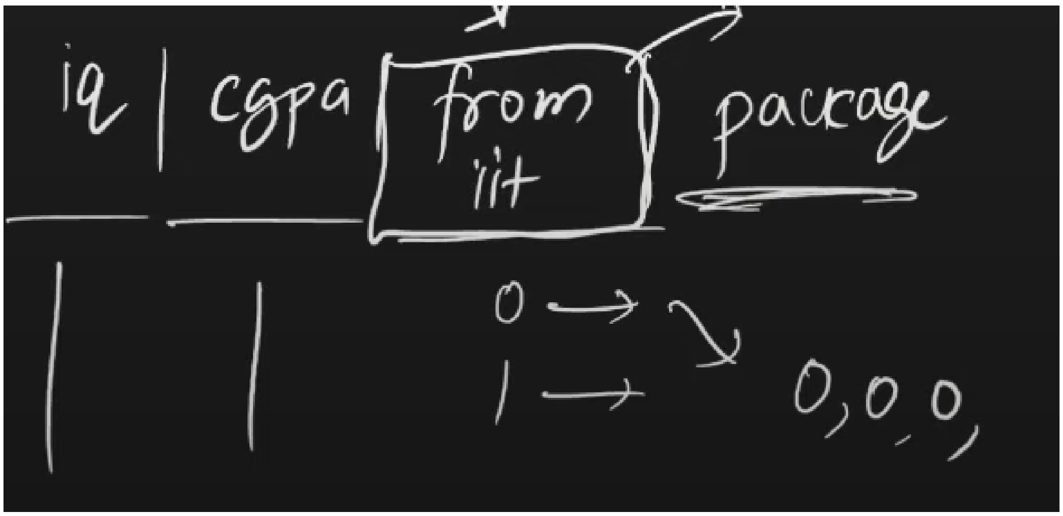
- Uses **same learning rate** for all weights → inefficient

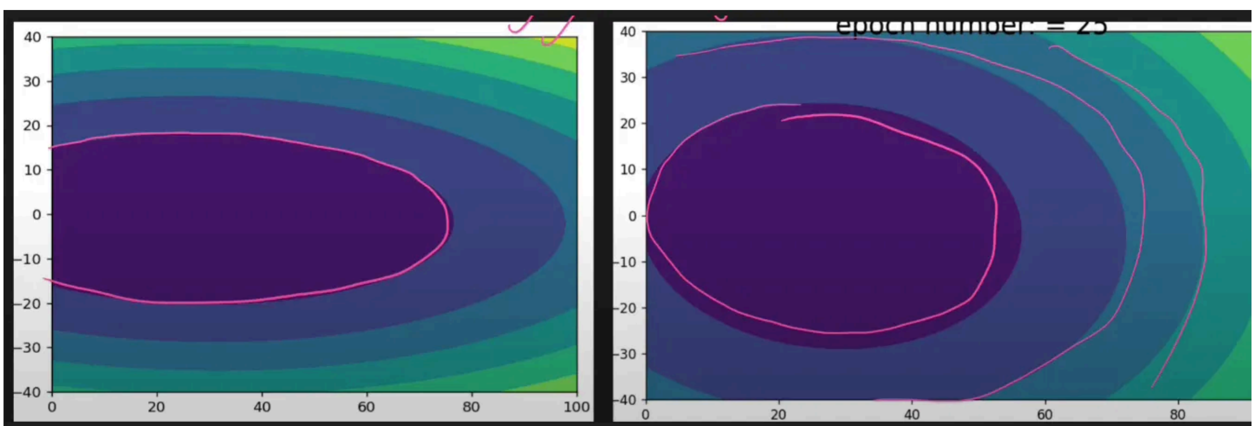
AdaGrad Fix:

- Tracks the **sum of squared gradients** for each weight
- Adjusts learning rate **based on parameter history**

⚡ *When to Use AdaGrad?*

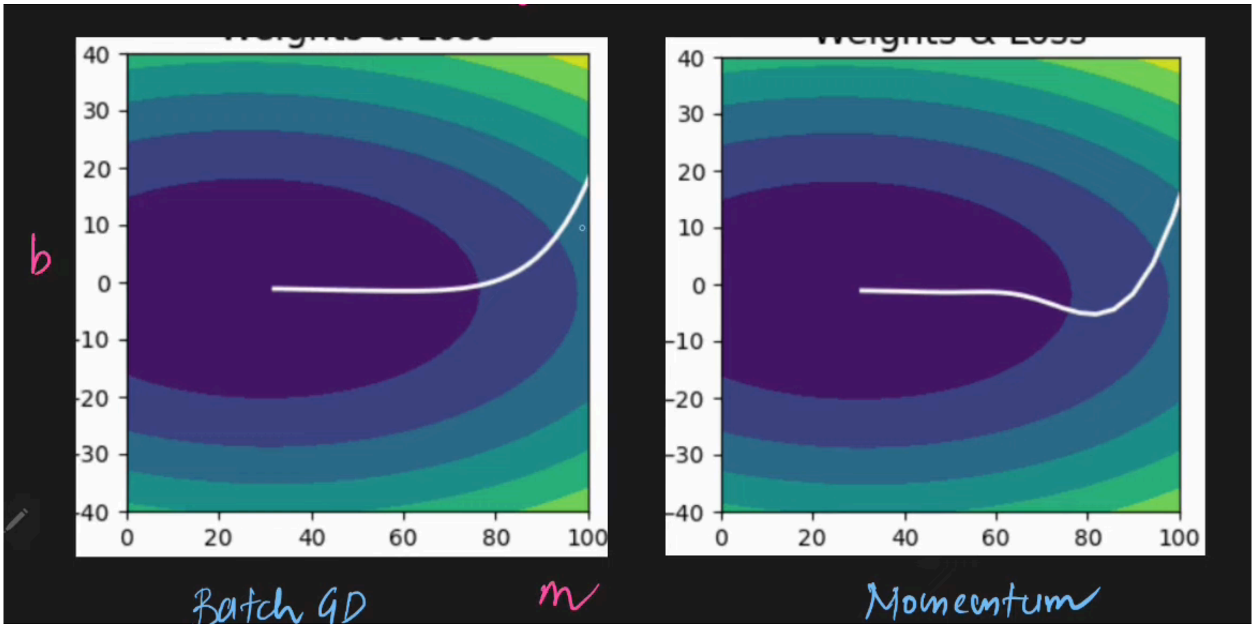
- When **scale of input features is different**
  - eg. CGPA (scale: 0 to 10) & salary (scale: in Lacs)
  - But we normalize the data in such case so Adagrad is not much useful.
- When **features** are **sparse** (most of the values are zero)





Sparse features

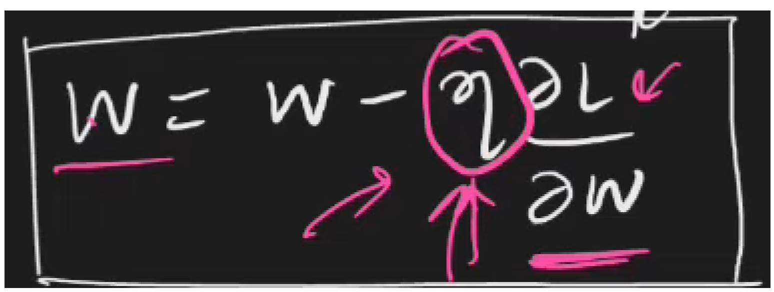
Non-sparse features



- The movement is slow for sparse columns cuz weight update is small

- To solve the above problem, adagrad uses adaptive LR📌👇

📌Big gradient → Small Learning Rate (& vice versa)



Formula:

Update Rule:

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

- G_t: Sum of squared past gradients (per parameter).
- ϵ: Small constant (~1e-8) to avoid division by zero.

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \nabla w_t \qquad \nabla w_t \rightarrow \boxed{\frac{\partial L}{\partial w}}$$

v_t: Sum of squared past gradients (per parameter)

ϵ: Small constant (~1e-8) to avoid division by zero.

$$v_t = v_{t-1} + (\nabla w_t)^2$$

📊 AdaGrad vs. SGD

Scenario	SGD	AdaGrad
Sparse features	Fails to train rare features	Excels
Learning rates	Fixed for all parameters	Adapts per parameter
Convergence	Slow for ill-conditioned data	Faster

Disadvantages of AdaGrad

- AdaGrad can get close to solution but can never converge
    - **Reason**: we divide LR by v_t. It eventually gets large and LR decreases → Small Updates

📈Behavior Summary:

Feature	Effect
Large gradient (frequent)	🚫 Step gets smaller
Small gradient (infrequent)	✅ Step stays large
Converges fast	✅ Initially
Long-term learning	❌ Learning rate may become too small to continue

Python/Keras Implementation

```
from keras.optimizers import Adagrad

optimizer = Adagrad(learning_rate=0.01)

model.compile(optimizer=optimizer, loss='mse')
```

`learning_rate=0.001` **(Default)**

4. RMSProp (Root Mean Square Propagation)

- **Adaptive learning rate** optimization algorithm that **fixes AdaGrad's main weakness**: its learning rate decays too aggressively.
- **One of the best optimization techniques**

> 💡 **RMSProp was used before Adam.**

🎯
Goal: Maintain stable, adaptive learning rates for each weight without vanishing updates.

- **Problem with AdaGrad**: The sum of squared gradients (G_t) grows monotonically, causing learning rates to vanish over time.
- **RMSProp's Fix**: Replace Gt*Gt* with a **leaky average** (like momentum for gradients).

🧠 Real-World Analogy

> **AdaGrad**: "Every time I step, I get more cautious permanently."

> **RMSProp**: "I remember the recent terrain and adjust my step size based on it—not the whole history."

Update Rule:

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

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

- $E[g^2]_t$: Exponentially weighted average of squared gradients.
- β: Decay rate (typically **0.9**).
- ϵ: Small constant (~1e-8) for numerical stability.

🧮 3. RMSProp Update Rule – Step by Step

Let:

- θ: parameter (weight)
- g_t: gradient at time t
- $E[g^2]_t$: exponentially decaying average of squared gradients
- ρ: decay factor (default = 0.9)
- η: learning rate
- ϵ: small constant to avoid division by zero

✅ **Step 1: Update Moving Average of Squared Gradients**

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

➡️ Like an **Exponentially Weighted Moving Average (EWMA)**

✅ **Step 2: Parameter Update**

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

📊 RMSProp vs. AdaGrad

Scenario	AdaGrad	RMSProp
Learning Rates	Vanish over time	Stabilize
Convergence	Stalls in deep nets	Works well
Hyperparameter	None	Decay rate (β)
Deep networks	❌ Not good	✅ Reliable

Python/Keras Implementation

```
from keras.optimizers import RMSprop

optimizer = RMSprop(learning_rate=0.001, rho=0.9)

model.compile(optimizer=optimizer, loss='mse')
```

`learning_rate=0.001` **(Default)**

`rho=0.9` **(Default)**

⚡ Pro Tips

1. **Default Hyperparameters**:
    - Learning rate (α): **0.001** (start small).
    - Decay rate (β): **0.9** (standard).
2. **Combine with Momentum**:
    - RMSProp focuses on **gradient magnitude**; momentum handles **direction**.
    - Result: **Adam optimizer** (next-gen default).
3. **Use For**:
    - RNNs, non-convex problems.
    - Replaced by Adam in most cases, but still useful for some tasks.

5. Adam (Adaptive Moment Estimation)

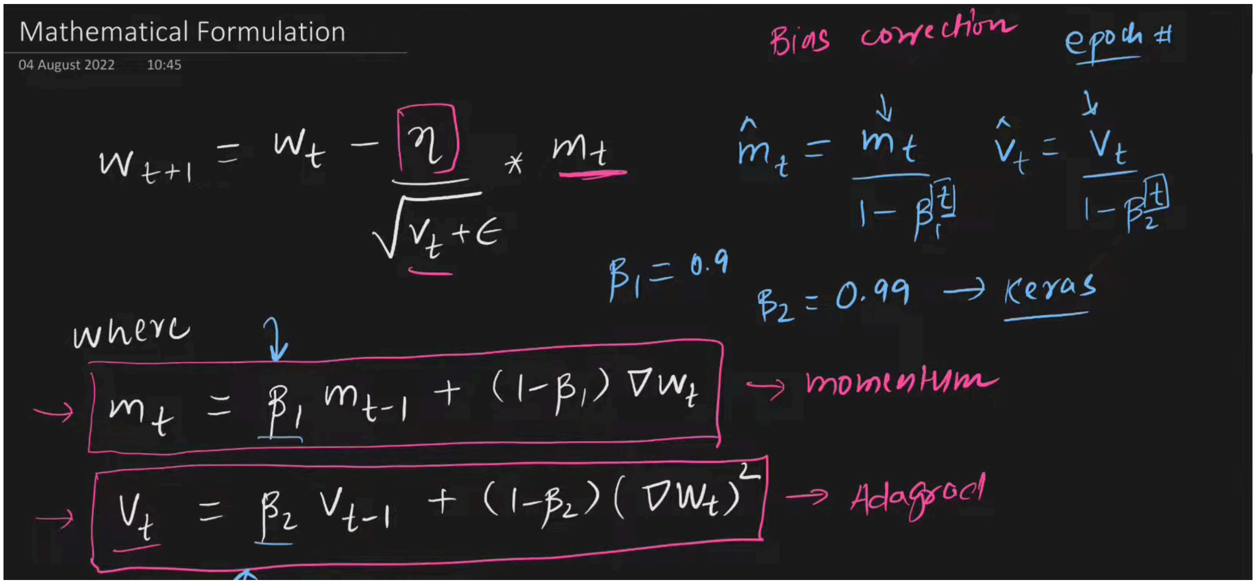
- Adam combines **RMSProp (adaptive learning rates) and Momentum (gradient history)** into one powerful optimizer.
- It's the **default choice** for most deep learning tasks due to its robustness and speed.

🎯 Goal: Achieve fast convergence with stable updates in deep learning models.

🧠 Why Use Adam?

Problem	**Fix Adam Provides**
Learning rate needs tuning	✅ Automatically adapts per weight
Momentum or RMSProp alone isn't enough	✅ Combines both
Noisy gradients in mini-batch training	✅ Smooth updates
Training unstable	✅ Stabilizes with bias correction

🔁 Update Rule – Step by Step



- θ_t: current weight
- g_t: current gradient
- m_t: moving average of gradient (1st moment)
- v_t: moving average of squared gradient (2nd moment)
- β1: decay for momentum (default = 0.9)
- β2 : decay for RMS (default = 0.999)
- ϵ: small constant (default = 1e-7)

✅ Step 1: Compute Gradients

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

✅ Step 2: Update First Moment (Momentum)

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

✅ Step 3: Update Second Moment (RMS)

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

✅ Step 4: Bias Correction

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

> Corrects for initialization bias early in training.

✅ **Step 5: Update Parameters**

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

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad \text{(1st moment)}$$

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

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \text{(bias correction)}$$

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

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

📊 Adam vs. Other Optimizers

Optimizer	Adaptive LR?	Momentum?	Best For
SGD	❌	❌	Simple convex problems
SGD + Momentum	❌	✅	Deeper networks
AdaGrad	✅	❌	Sparse data
RMSProp	✅	❌	RNNs, non-convex landscapes
Adam	✅	✅	**Most deep learning tasks**

Keras Code

```
from keras.optimizers import Adam

optimizer = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)

model.compile(optimizer=optimizer, loss='mse')
```

`learning_rate=0.001` (**Default**)

`beta_1=0.9` (**Default**)

`beta_2=0.999` (**Default**)

🚀 Why Adam Dominates?

✅ **Adaptive Learning Rates**: Each parameter gets a custom step size.

✅ **Momentum Acceleration**: Smoothens noisy gradients.

✅ **Bias Correction**: Fixes early training instability.

✅ **Works Out-of-the-Box**: **Default hyperparameters** suit most problems.

🎯 Use Cases

Task	Adam Use?
Deep neural networks	✅ Excellent
NLP & Transformers	✅ Standard choice
Image classification	✅ Common
RNNs & LSTMs	✅ Very effective
Tabular data	✅ Works well

🧾 Real-World Analogy

> Adam is like a self-driving car:

> It adjusts its speed based on road slope (gradient), remembers recent direction (momentum), and avoids overreacting (bias correction).

📌 When to Avoid Adam?

❌ **Theoretical convex problems** (SGD may generalize better).

❌ **Extremely large batches** (momentum becomes less useful).

🎯 Summary

- **Adam** = **RMSProp + Momentum + Bias Correction**.
- **Use For**: Almost all deep learning (CNNs, RNNs, Transformers).
- **Implementation**: 1 line in PyTorch/Keras.