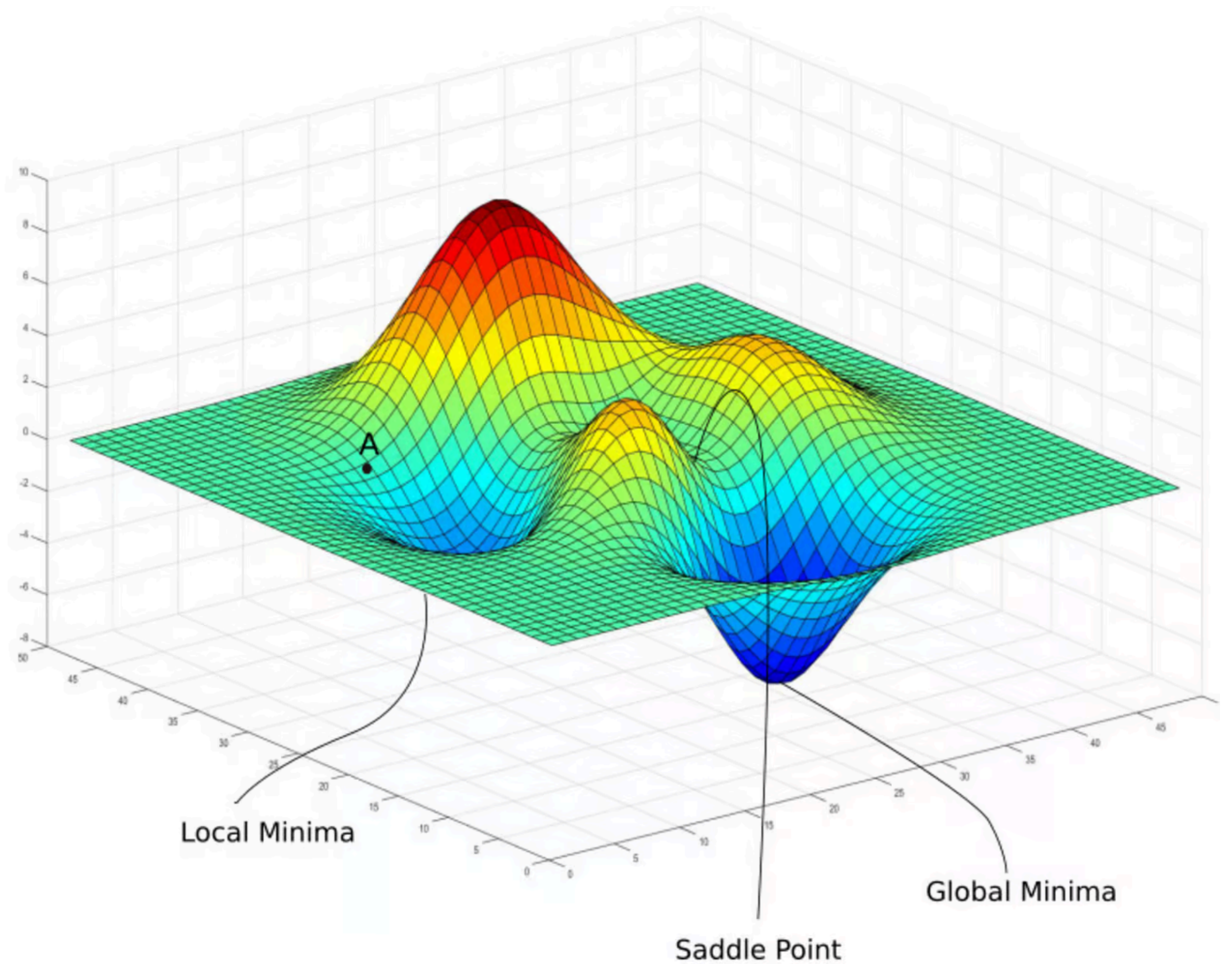
COURSE NAME

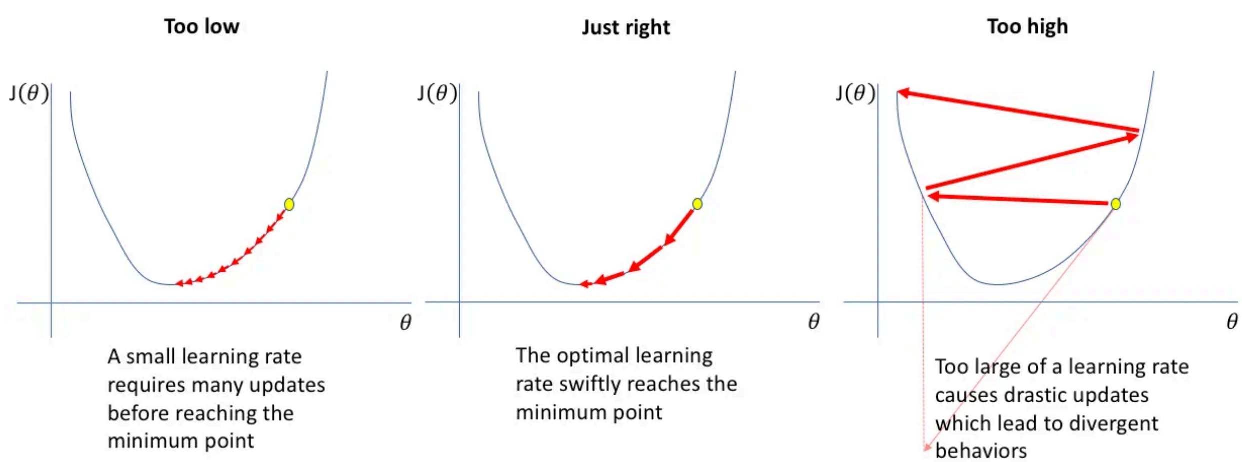
Optimizers



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

Learning Rate

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

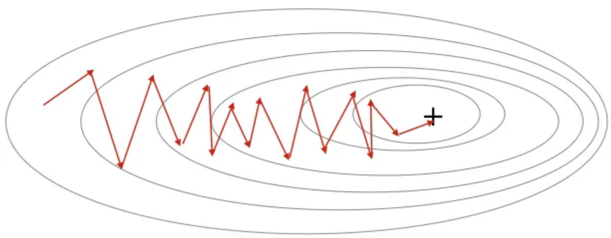


Introduction

1. Think of optimizers as an improvement chain:

Each optimizer builds upon the previous one to solve its shortcomings.

- **SGD** (Stochastic Gradient Descent) → Base method, but slow and struggles with complex landscapes.



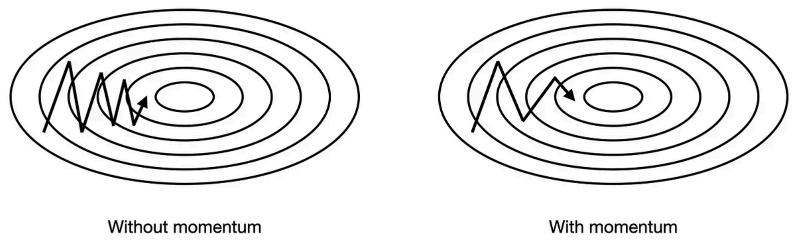
Copy

```
θ = θ - η * ∇J(θ)
```

Where:

- θ: parameters
- η: learning rate
- $\nabla J(\theta)$: gradient of the loss function

- **Momentum** → Momentum adds a **velocity** term to help **overcome local minima** and **speed up convergence**. Adds velocity to move faster. Adds "**physics**" - like a ball rolling down a hill. For faster convergence in deep networks.



Copy

```
v = γ * v + η * ∇J(θ)
θ = θ - v
```

Where:

- v: velocity (initialized as zero)
- γ: momentum coefficient (typically 0.9)

- **NAG (Nesterov Accelerated Gradient)** → NAG improves momentum by evaluating the gradient at the **"lookahead"** position. Smarter momentum to prevent overshooting. Adds "**foresight**" - looks ahead before committing. For faster convergence in deep networks.

Copy

```
v_prev = v
v = γ * v - η * ∇J(θ + γ * v_prev)
θ = θ + v
```

- **Adagrad** → AdaGrad **adapts learning rates for each parameter based on historical gradients**. Adaptive learning rate per parameter but can shrink too much. Adds "**personalization**" - different learning rates for parameters. Sparse data (e.g., NLP).

  Copy

  ```
  G = G + (∇J(θ))²
  θ = θ - η * ∇J(θ) / (√G + ε)
  ```

  Where:

  - G: sum of squared gradients (initialized as zeros)
  - ε: small constant for numerical stability

- **RMSprop** → **RMSProp modifies AdaGrad to prevent the learning rate from decreasing too rapidly**. Fixes Adagrad's shrinking issue using a moving average. Adds "**forgetting**" - focuses more on recent gradients. Sparse data (e.g., NLP).

  Copy

  ```
  G = β * G + (1 - β) * (∇J(θ))²
  θ = θ - η * ∇J(θ) / (√G + ε)
  ```

  Where:

  - β: decay rate (typically 0.9)

- **Adam** → Combines Momentum + RMSprop for the best of both worlds. The "**complete package**" - combines momentum and adaptive rates. Default for most deep learning tasks. Adam combines momentum and RMSProp, using first and second moments of gradients

Copy

```
m = β₁ * m + (1 - β₁) * ∇J(θ)      // First moment
v = β₂ * v + (1 - β₂) * (∇J(θ))²   // Second moment

// Bias correction
m̂ = m / (1 - β₁ᵗ)
v̂ = v / (1 - β₂ᵗ)

θ = θ - η * m̂ / (√v̂ + ε)
```

Where:

- β_1, β_2: decay rates (typically 0.9 and 0.999)
- t: iteration number

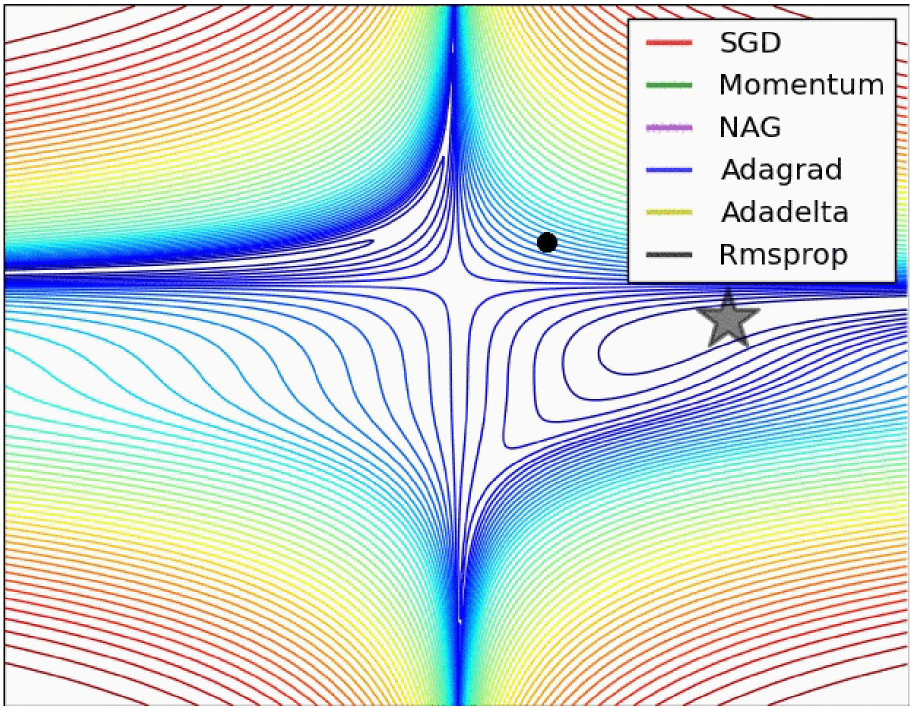
- **AdamW** → AdamW **decouples weight decay from gradient updates, properly implementing L2 regularization**. Fixes weight decay issues in Adam. The "**fix**" - properly implements regularization. Better for transformers, fine-tuning.

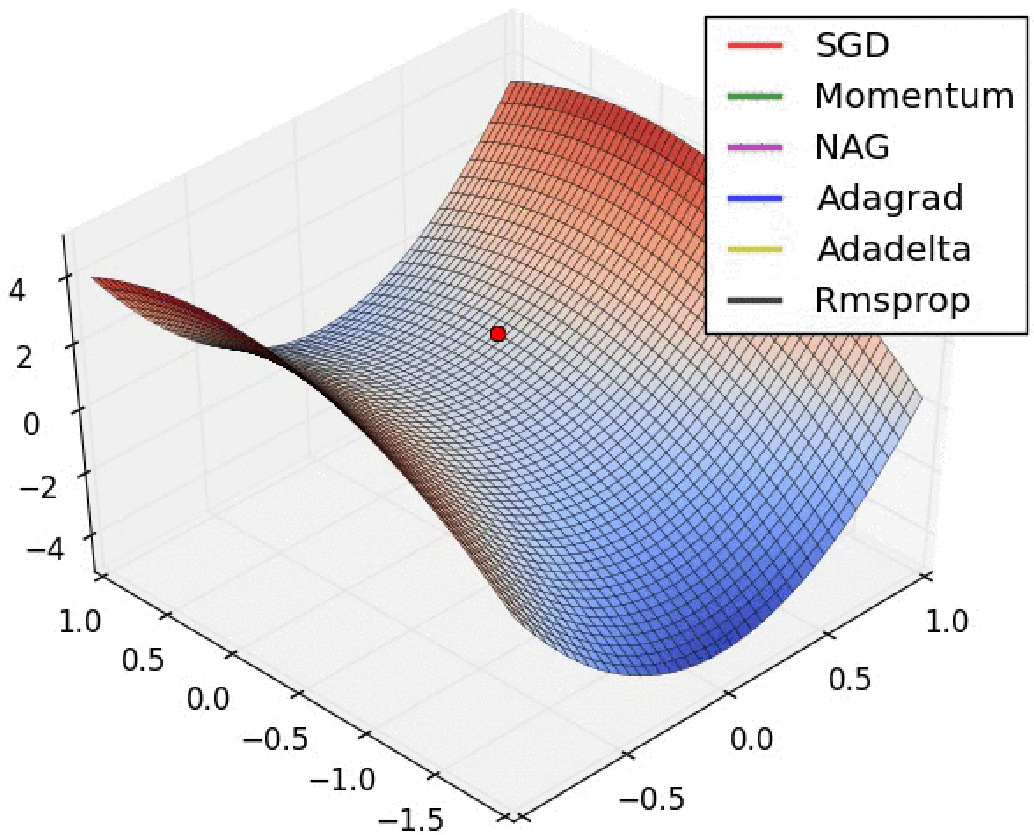
Copy

```
m = β₁ * m + (1 - β₁) * ∇J(θ)
v = β₂ * v + (1 - β₂) * (∇J(θ))²

m̂ = m / (1 - β₁ᵗ)
v̂ = v / (1 - β₂ᵗ)

θ = θ - η * (m̂ / (√v̂ + ε) + λ * θ)  // Weight decay term added
```

Where:

- λ: weight decay coefficient


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

2. Key Concepts to Remember

Optimizer	Key Idea	Solves	Problem Solved from Previous
SGD	Basic gradient descent	-	High variance, slow convergence
Momentum	Adds velocity term ($\gamma \cdot v$)	Reduces oscillations	SGD's slow convergence in ravines
NAG (Nesterov)	Corrects momentum by "peeking ahead"	Better direction before update	Momentum overshooting
AdaGrad	Adapts LR per parameter ($\div\sqrt{sum\_g^2}$)	Sparse data optimization	Fixed LR for all parameters
RMSProp	Exponential moving avg ($\div\sqrt{E[g^2]}$)	Fixes AdaGrad's LR decay	Prevents vanishing LR
Adam	Momentum + RMSProp + bias correction	Combines best of both	Requires tuning β_1, β_2
AdamW	Decouples weight decay	Fixes Adam's weight decay	Adam's poor generalization

Optimizer	Update Rule (Formula)	Advantages	Disadvantages	Solves Problem of Previous Method
SGD (Stochastic Gradient Descent)	$\theta = \theta - \eta \cdot \nabla J(\theta)$	Simple, works well with large batches	Noisy updates, slow convergence	Baseline (no optimization tricks)
Momentum	$v = \gamma \cdot v + \eta \cdot \nabla J(\theta)$ $\theta = \theta - v$	Faster convergence, reduces oscillations	May overshoot minima	SGD's slow convergence in ravines
NAG (Nesterov Accelerated Gradient)	$v = \gamma \cdot v + \eta \cdot \nabla J(\theta - \gamma \cdot v)$ $\theta = \theta - v$	Better direction correction than Momentum	Slightly more computation	Momentum's overshooting issue
AdaGrad	$G \mathrel{+}= \nabla J(\theta)^2$ $\theta = \theta - (\eta/\sqrt{G}) \cdot \nabla J(\theta)$	Adapts learning rate per parameter, good for sparse data	Learning rate decays to zero too fast	Fixed LR for all parameters
RMSProp	$E[g^2] = \gamma \cdot E[g^2] + (1-\gamma) \cdot \nabla J(\theta)^2$ $\theta = \theta - (\eta/\sqrt{E[g^2]}) \cdot \nabla J(\theta)$	Fixes AdaGrad's aggressive LR decay	Still sensitive to γ hyperparameter	AdaGrad's vanishing LR problem

Adam (Adaptive Moment Estimation)	$m = \beta_1 \cdot m + (1-\beta_1) \cdot \nabla J(\theta)$ (1st moment) $v = \beta_2 \cdot v + (1-\beta_2) \cdot \nabla J(\theta)^2$ (2nd moment) $\hat{m} = m/(1-\beta_1{}^\wedge t)$, $\hat{v} = v/(1-\beta_2{}^\wedge t)$ $\theta = \theta - (\eta/\sqrt{\hat{v}}) \cdot \hat{m}$	Combines Momentum + RMSProp, fast convergence	Memory-intensive, sensitive to β_1, β_2	Requires tuning momentum & LR
AdamW (Adam with Weight Decay Fix)	Same as Adam, but weight decay is decoupled: $\theta = \theta - \eta \cdot (\hat{m}/\sqrt{\hat{v}} + \lambda \cdot \theta)$	Better generalization, fixes Adam's weight decay issue	Slightly more complex than Adam	Adam's poor weight decay handling