Important Concepts to Know

Definition of Sparsity in Neural Networks

sparsity refers to the concept of having a significant number of elements (e.g., weights, activations, or connections) in a neural network set to zero or being effectively unused.

General Idea: Sparsity occurs when a model contains many zero or near-zero values, reducing the number of active parameters. In the context of your deep learning setup, this could apply to:

- **Weight Sparsity**: A large fraction of the weight matrix in a layer (e.g., the kernel in your Dense layers) being zero.
- **Activation Sparsity**: Many neuron outputs being zero, often due to activation functions like ReLU (which outputs 0 for negative inputs).
- **Connection Sparsity**: Fewer active connections between neurons, as might occur with techniques like dropout or pruning.

Selecting Best Model (in classification tasks)

How to Detect Overfitting Using Loss

Overfitting can be identified by observing the behavior of **training loss** and **validation loss** over epochs. Here are the key indicators:

1. **Training Loss Decreases, but Validation Loss Increases**:

   As training progresses, the model optimizes to minimize training loss. If training loss continues to decrease while validation loss starts to **increase**, the model is overfitting. This suggests the model is memorizing the training data rather than learning general patterns that apply to the validation set.

2. **Large Gap Between Training and Validation Loss**:

If training loss becomes significantly lower than validation loss, it indicates the model performs much better on the training set than on the validation set. A large gap (e.g., training loss much lower than validation loss) is a sign of overfitting, as the model is overly tuned to the training data.

3. **Validation Loss Plateaus or Worsens**:

If validation loss stops decreasing (plateaus) or increases, even as training loss continues to drop, the model is likely overfitting. This shows that further training improves performance on the training set but not on unseen data.

4. **Context of Other Metrics**:

While loss is the primary indicator, you can cross-check with metrics like validation accuracy. If validation accuracy plateaus or decreases while validation loss increases, it reinforces the overfitting signal.

Calculation of Sparse Categorical Cross-Entropy Loss Using Sample MNIST Data

Example Setup

Batch Size (N): Let's use N=4 for simplicity (a small subset of your 128-sample batches) to demonstrate the process. In practice, N=128 would be used in your code. True Labels (y_i): A sample batch of 4 integer labels from MNIST (0-9):

$y_1=3$ $y_2=7$ $y_3=1$ $y_4=9$

Predicted Probabilities ($\hat{y}_{i,c}$): The softmax output for each sample is a 10-dimensional vector (one probability per class). Here are example predictions:

Sample 1 (i=1, true class 3): $\hat{y}_1$ =[0.05,0.05,0.05,0.70,0.05,0.05,0.0,0.0,0.0,0.05]

Sample 2 (i=2, true class 7): $\hat{y}_2$ =[0.10,0.10,0.10,0.10,0.10,0.10,0.10,0.15,0.05,0.10]
Sample 3 (i=3, true class 1): $\hat{y}_3$ =[0.05,0.60,0.10,0.05,0.05,0.05,0.0,0.05,0.0,0.05]

Sample 4 (i=4, true class 9): $\hat{y}_4$ =[0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.50]

Note: Each vector sums to 1 (e.g., $0.70+0.05\times8+0.0\times2=1$ for sample 1), reflecting softmax normalization.

Relevant Probabilities: We only need $\hat{y}_{i,y_i}$, the predicted probability for the true class:

$\hat{y}_{1,3}$ = 0.70 (class 3 for sample 1)

$\hat{y}_{2,7}$ = 0.15 (class 7 for sample 2)

$\hat{y}_{3,1}$ =0.60 (class 1 for sample 3)

$\hat{y}_{4,9}$ =0.50 (class 9 for sample 4)

Calculation

Using the formula $L = -\frac{1}{N}\sum_{i=1}^{N}\log(\hat{y}_{i,y_i})$:

Compute Logarithms:

$\log(\hat{y}_{1,3})=\log(0.70)\approx-0.357$ $\log(\hat{y}_{2,7})=\log(0.15)\approx-1.897$
$\log(\hat{y}_{3,1})=\log(0.60)\approx-0.511$ $\log(\hat{y}_{4,9})=\log(0.50)\approx-0.693$ (Using natural log, as is standard in TensorFlow's implementation.)

Sum the Log Terms:

Sum = $-0.357+(-1.897)+(-0.511)+(-0.693) \approx -3.458$

Average Over Batch Size (N=4):

$L=-1/4 \times (-3.458) \approx 0.865$

Result

The sparse categorical cross-entropy loss for this sample batch of 4 is approximately 0.865. This value represents the average negative log-likelihood, where higher values indicate poorer predictions (e.g., y^ 2,7 =0.15 contributes more error due to its lower probability).

---

sep 28-2025
Relu

Generalized Rectifier (definition)

$$h_i = g(z_i, \alpha_i) = \max(0, z_i) + \alpha_i \cdot \min(0, z_i)$$

z_i : the **pre-activation input** to the unit (weighted sum of inputs before applying activation).

h_i : the **output (activation value)** of the hidden unit.

α_i \alpha_i : a parameter that controls how the function behaves for negative inputs.

If $\alpha_i = 0$:- we get standard **ReLU** max(0, z_i).

If $\alpha_i > 0$:- we get **Leaky ReLU** (small slope for negative values).

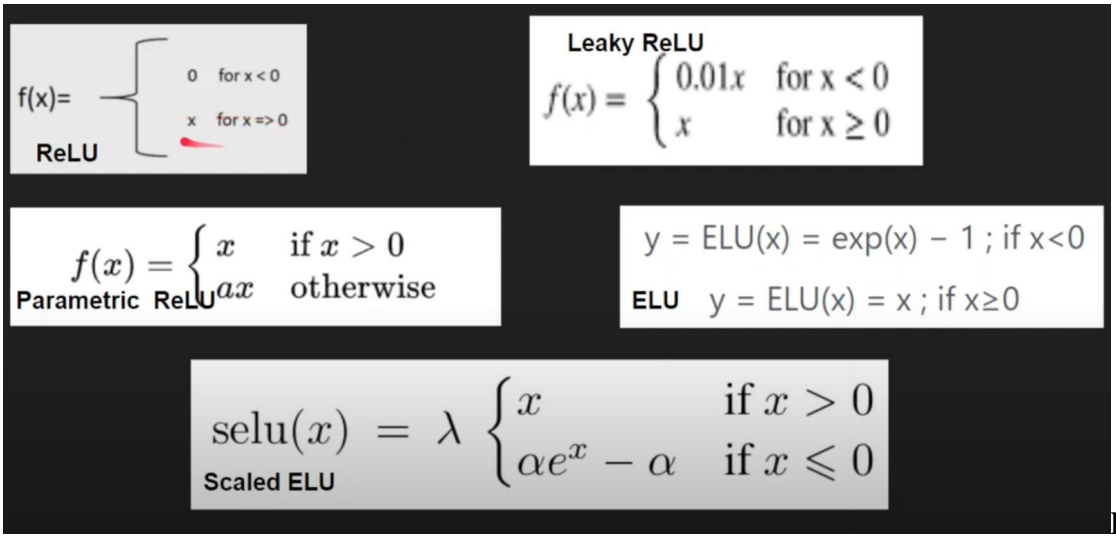
If α_i :- is learnable, we get **Parametric ReLU (PReLU)**.

Comparing with ReLU and Leaky ReLU

ReLU: f(x)=max(0,x) → gradient is 0 for negatives.

Leaky ReLU: f(x)=max(0,x)+ α min(0,x) → gradient is constant α \alpha α for negatives.

ELU: f(x)=xf(x)=x for positives, smooth exponential for negatives → gradient never exactly 0 on negatives.



Expected Findings:

ReLU:

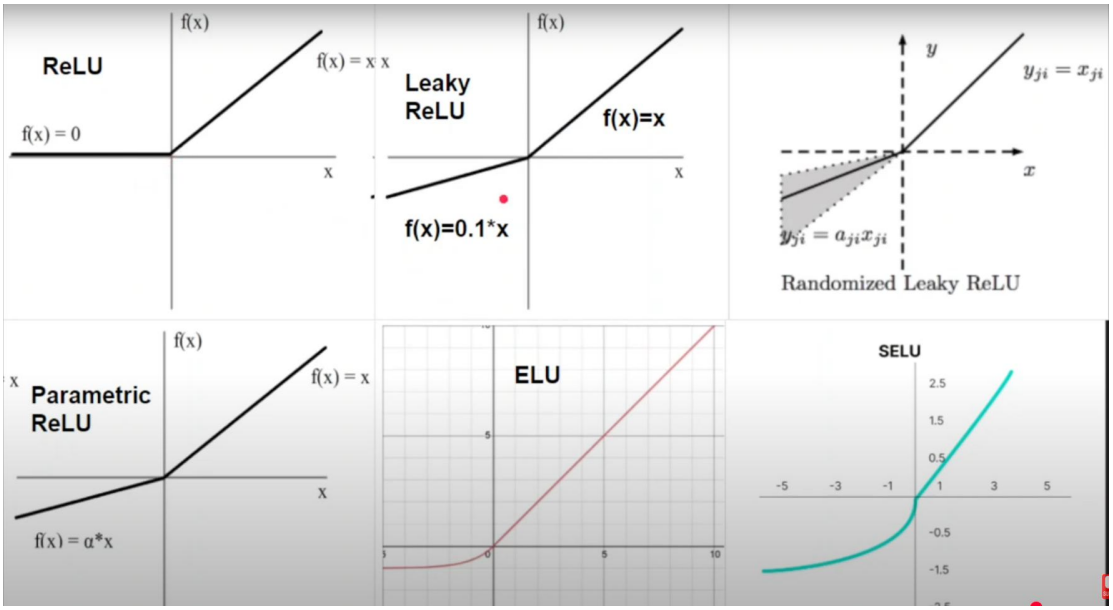
- **Performance**: As above, achieves ~97–98% validation accuracy and low loss.
- **Gradient Norms**: Stable and relatively large across layers, as ReLU passes positive gradients effectively.
- **Limitation**: Some neurons may "die" (always output zero) if many inputs are negative, leading to slightly reduced capacity in deeper layers.

Leaky ReLU:

- **Performance**: Similar to ReLU, likely ~97–98% validation accuracy, possibly slightly better or worse depending on the data.
- **Reason**: Leaky ReLU (f(x) = x if x > 0, else f(x) = 0.01x) allows a small gradient (alpha = 0.01 in the code) for negative inputs, preventing dead neurons.
- **Gradient Norms**: Slightly larger than ReLU in some layers, as negative inputs contribute small gradients, improving flow in early layers.
- **Loss/Accuracy Plots**: Very similar to ReLU, with minor improvements if dead neurons were a significant issue.

PReLU:

- **Performance**: Likely matches or slightly outperforms ReLU and Leaky ReLU (~97–98.5% validation accuracy).
- **Reason**: PReLU learns the slope for negative inputs during training (unlike Leaky ReLU's fixed alpha), adapting to the data. This can lead to better gradient flow and fewer dead neurons.
- **Gradient Norms**: Similar to or slightly larger than Leaky ReLU, as the learned slopes ensure more neurons contribute to gradients.
- **Loss/Accuracy Plots**: Comparable to ReLU, with potential for slightly faster convergence or higher accuracy due to adaptive slopes.



Feature	ReLU	Leaky ReLU	PReLU	ELU (For Context)
Formula (Negative x)	f(x)=0	f(x)= α x (α is small, fixed)	f(x)= α i x (α i is learned)	f(x)= α (ex−1)
Gradient for x<0	0	Constant α	Learned α i	Never exactly 0 (smooth exponential)
Dead Neuron Issue	High Risk	Mitigated (small gradient flow)	Best Mitigation (adaptive gradient flow)	Eliminated (non-zero gradient)

Imortant terms

1.Convex

Simple Definition:

A function is convex if it has one valley — one lowest point (global minimum).

You can imagine it like a **bowl** . No matter where you drop a ball, it will **roll down to the same lowest point**.

convex function is one where:

> The graph curves **upward** like a bowl.
>
> Any line drawn between two points on the graph lies **above the curve**.
>
> Most importantly, it has **only one minimum point** (called the **global minimum**), making optimization easier.

Why it matters:

> Convex loss functions are **easier to optimize**.
> You are guaranteed to reach the **best solution** (no confusing hills or multiple valleys).

2. Differentiable

Simple Definition:

> A function is **differentiable** if it has a **smooth slope** everywhere — no sharp corners or jumps.

Think of a **smooth road** — you can always say how steep it is at any point.

Why it matters:

> If a loss function is differentiable, we can **compute gradients**.
> Gradients are the **directions we follow to minimize the loss**.

If it's **not differentiable**, the gradient is **undefined** at that point.

3. Smooth

Simple Definition:

A **smooth function** is differentiable **and** has no sudden changes in slope.

Think of it as a **very gentle and curved hill**, with **no sharp bends**.

What is a non-convex loss function?

A **loss function** measures how wrong a model's predictions are.

Convex loss → bowl-shaped, only one global minimum. Easy to optimize.

Non-convex loss → has **multiple local minima, saddle points, and plateaus**.

Where it occurs:

Most **deep neural networks** have non-convex loss functions because of:

Multiple layers with non-linear activations (ReLU, Sigmoid, etc.)

Complex architectures like CNNs or RNNs

Example: loss landscape of a 2-layer MLP on a 2D dataset is non-convex.

Does changing weights change the *shape* of the graph?

We need to be careful here:

Graph of the loss function $J(\theta)$ **vs. weights**

This graph is *fixed* for a given dataset and model architecture.

Changing the weights just means moving to a different point on this surface.

So the **shape doesn't change** — only your position on it changes.

Visualization analogy

Loss surface = fixed landscape.

Weights = your location on the landscape.

Decision boundary = the map you draw based on where you are. As you move to a better spot (lower loss), your map (predictions) improves.

Gradient Descent

1. Batch Gradient Descent

- Uses **all training data** to compute the gradient before updating weights.

- Gradient = average of all residuals.
- **Pros:** Smooth gradient, stable updates.
- **Cons:** Very slow for large datasets.

Batch size = all samples

Each update uses **all 6 points** to compute the gradient.

Compute gradient of the loss (say MSE) using all 6 points.

Update weights once.

Repeat this process for each epoch.

Effect: very stable updates, but **slow** on large datasets because you recalc on all points every step.

2. Mini-batch Gradient Descent

- Uses **a small batch of samples** (e.g., 32, 64) for each update.
- Gradient = average of residuals in that mini-batch.
- **Pros:** Balanced between speed and stability, widely used in practice.
- **Cons:** Needs batch size tuning.

Batch size = smaller group (say 2 samples at a time).

Dataset is split into **mini-batches**:

{(1,2),(2,4)},{(3,6),(4,8)},{(5,10),(6,12)}

Use the first 2 points, compute gradient, update weights.

Then use the next 2, update again.

Then the next 2, update again.

That's **one epoch** (all data has been seen once).

Effect: faster updates (since each step sees fewer samples), more noise, but often converges faster in practice.

3. Stochastic Gradient Descent (SGD)

· Uses **one sample at a time** to compute the gradient and update weights.

· Gradient = just from one sample.

· **Pros:** Very fast, can escape shallow local minima due to noisy updates.

· **Cons:** Updates are very noisy, loss curve is jumpy.

Batch size = 1 sample at a time.

Dataset is treated as individual samples:

{(1,2),(2,4),(3,6),(4,8),(5,10),(6,12)}

Example:

1. Take the first point (1,2), compute gradient, update weights.
2. Take the next point (2,4), compute gradient, update weights.
3. Take the next point (3,6), compute gradient, update weights.

Continue for all points.

That's **one epoch** (all data has been seen once).

Effect:

Very fast updates because each step uses only 1 sample.

Highly noisy gradients → loss curve jumps a lot.

Can escape shallow local minima because of randomness.

Often slower to converge smoothly compared to mini-batch, but still widely used especially for large datasets.

The **goal** of training is to find parameter values θ\thetaθ (weights & biases) that minimize this loss.

Method	# of samples per update	Noise	Speed	Stability
Batch GD	All	Low	Slow	High
Mini-batch GD	Small batch	Medium	Medium	Medium
SGD	1	High	Fast	Low

4. GD_momentum (Gradient Descent with Momentum)

Definition: Adds a "velocity" term to accumulate gradients, smoothing updates and speeding convergence.

5. GD_nesterov (Nesterov Accelerated Gradient, NAG)

Definition: Similar to momentum, but computes the gradient at the **lookahead position** (faster and more stable).

6. Adagrad (Adaptive Gradient Algorithm)

Definition: Adapts learning rate per parameter — larger updates for infrequent features, smaller for frequent ones.

1. GD_batch (Batch Gradient Descent)

Definition:

Computes the gradient using the **entire dataset** at each step.

Very stable but **slow for large datasets** because it requires a full pass over the data each iteration.

Equation:

$\theta \leftarrow \theta - \eta \nabla\theta J(\theta)$

Terms Table:

Symbol	Meaning
θ	Model parameters (weights)
η	Learning rate (step size)
$J(\theta)$	Loss function evaluated on **all training data**
$\nabla\theta\ J(\theta)$	Gradient of the loss w.r.t θ using all data

Advantage: Stable and guarantees smooth convergence.
Limitation: Slow for large datasets, cannot escape shallow local minima easily.

---

2. SGD_minibatch (Mini-Batch Stochastic Gradient Descent)

Definition:

Uses a **small batch** (e.g., 32 samples) per update.

Compromise between stability (batch GD) and speed (stochastic GD).

Equation:

$$\theta \leftarrow \theta - \eta \nabla\theta J(\theta; B)$$

Terms Table:

Symbol	Meaning
θ	Model parameters
η	Learning rate
B	Mini-batch subset of training data
$J(\theta; B)$	Loss function evaluated on mini-batch B
$\nabla\theta J(\theta; B)$	Gradient using mini-batch B

Advantage: Faster than batch GD and less noisy than stochastic GD.
Extra: Enables GPU parallelization.

---

3. SGD_stochastic (Stochastic Gradient Descent)

Definition:

Updates parameters **using a single data point** at a time.

Fast and noisy; can help escape local minima.

Equation:

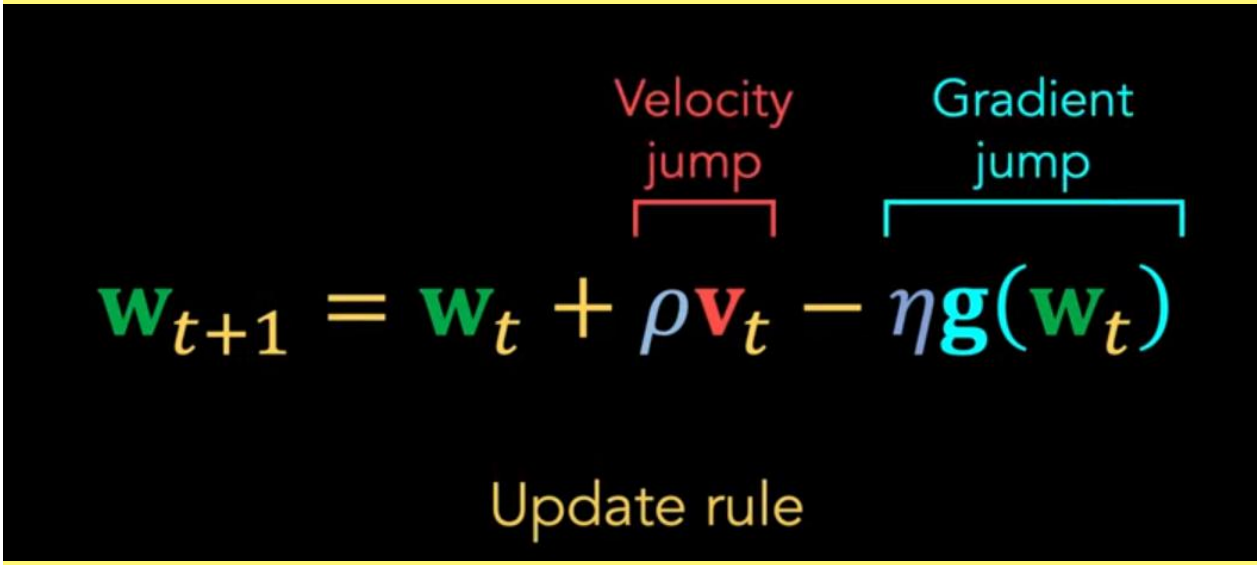
$$\theta \leftarrow \theta - \eta \nabla\theta J(\theta; xi, yi)$$

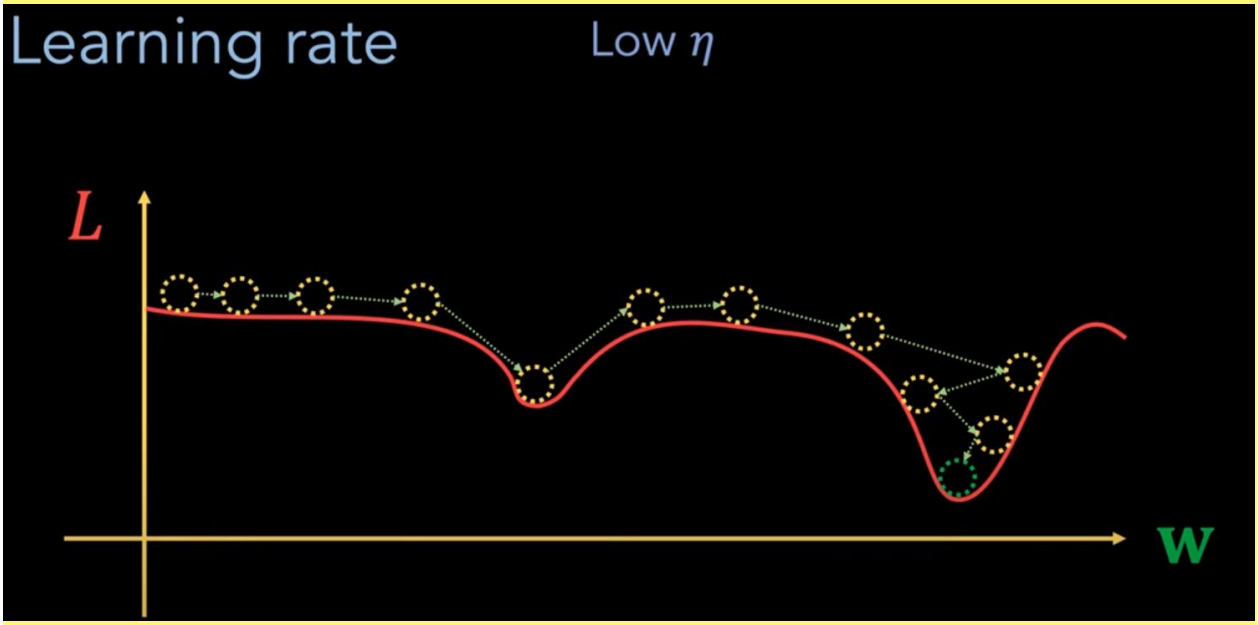
Terms Table:

Symbol	Meaning
θ	Model parameters
η	Learning rate
xi, yi	Single sample (features, target)
$J(\theta; xi, yi)$	Loss function for this single sample
$\nabla\theta J(\theta; xi, yi)$	Gradient based on one sample

Advantage: Fast updates, good for large datasets, can escape local minima.
Limitation: Noisy; convergence is less smooth.

4. GD_momentum (Gradient Descent with Momentum)





Definition:

Adds a **velocity term** that accumulates past gradients.

Reduces oscillations, accelerates in consistent gradient directions.

Equations:

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

$\theta \leftarrow \theta - v_t$

Terms Table:

Symbol	Meaning
v_t	Velocity (momentum term)
β	Momentum coefficient ($0 \leqslant \beta < 1$, e.g., 0.9)
η	Learning rate
$\nabla\theta J(\theta)$	Gradient of loss
θ	Model parameters

Advantage: Faster convergence, especially in **ravines** (steep sides and shallow bottom).

5. GD_nesterov (Nesterov Accelerated Gradient, NAG)

Definition:

Like momentum but **looks ahead** to compute gradient at predicted future position.

More accurate and slightly faster convergence.

Equations:

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

$$\theta \leftarrow \theta - v_t$$

Terms Table:

Symbol	Meaning
v_t	Velocity
β	Momentum coefficient
η	Learning rate
$J(\theta - \beta\, v_{t-1})$	Loss at "lookahead" position

Symbol	Meaning
θ	Parameters

Advantage: Less overshooting than regular momentum, faster in many cases.

---

6. Adagrad (Adaptive Gradient Algorithm)

Definition:

Adapts learning rate **for each parameter individually**.

Large updates for infrequent features, small for frequent ones.

Equations:

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

$\theta \leftarrow \theta - (\eta / (\sqrt{G_t + \epsilon})) * g_t$

Terms Table:

Symbol	Meaning

Symbol	Meaning
G_t	Sum of squared gradients up to step t
g_t	Gradient at step t ($\nabla\theta J(\theta)$)
η	Global learning rate
ϵ	Small constant for stability
θ	Parameters

Advantage: Works well for **sparse data**.
Limitation: Learning rate can shrink too much over time.

---

7. RMSprop (Root Mean Square Propagation)

Definition:

Like Adagrad but uses **exponential moving average** of squared gradients.

Prevents learning rate from shrinking too fast.

Equations:

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

$\theta \leftarrow \theta - (\eta / (\sqrt{E[g^2]\_t + \epsilon})) * g\_t$

Terms Table:

Symbol	Meaning
$E[g^2]\_t$	Moving average of squared gradients
ρ	Decay rate (e.g., 0.9)
g_t	Gradient at step t
η	Learning rate
ϵ	Small constant
θ	Parameters

Advantage: Stable for non-stationary objectives, commonly used in deep learning.

---

8. Adam (Adaptive Moment Estimation)

Definition:

Combines **momentum (first moment)** and **RMSprop (second moment)**.

Widely used as **default optimizer**.

Equations:

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

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

Bias correction:

$\hat{m}_t = m_t / (1 - \beta_1^t), \hat{v}_t = v_t / (1 - \beta_2^t)$

Update:

$\theta \leftarrow \theta - (\eta / (\sqrt{\hat{v}_t} + \epsilon)) * \hat{m}_t$

Terms Table:

Symbol	Meaning
m_t	First moment (mean of gradients)

Symbol	Meaning
v_t	Second moment (uncentered variance)
β_1	Decay rate for m_t (0.9 typical)
β_2	Decay rate for v_t (0.999 typical)
$\hat{m}_t$	Bias-corrected first moment
$\hat{v}_t$	Bias-corrected second moment
g_t	Gradient at step t
η	Learning rate
ϵ	Small constant for numerical stability
θ	Parameters

Advantage: Fast, stable, works well in almost all cases, adapts learning rate automatically.

✅ Summary of What Each Adds Over the Others	
Algorithm	**Key Addition / Advantage**
Batch GD	Stability, smooth convergence
Mini-batch SGD	Balance speed & stability, GPU-friendly
Stochastic GD	Fast, can escape local minima
Momentum GD	Smooths updates, accelerates convergence
Nesterov	Lookahead gradient, less overshoot, faster convergence
Adagrad	Parameter-wise learning rate adaptation, good for sparse features
RMSprop	Fixes Adagrad's diminishing LR problem
Adam	Combines momentum + RMSprop, most robust, strong default

Optimization Algorithm Comparison table

Algorithm	Definition	Update Formula	Pros	Cons
Batch GD	Uses the entire dataset to compute the gradient. Very stable.	$\theta \leftarrow \theta - \eta * \nabla\theta J(\theta)$	Smooth convergence, guarantees descent	Very slow on large datasets
Mini-Batch SGD	Uses a small batch (e.g., 32 samples) for each update.	$\theta \leftarrow \theta - \eta * \nabla\theta J(\theta; B)$	Balance of speed & stability, GPU-friendly	Still somewhat noisy
Stochastic GD	Updates using one data point at a time.	$\theta \leftarrow \theta - \eta * \nabla\theta J(\theta; xi, yi)$	Very fast, escapes local minima	Highly noisy updates
Momentum GD	Adds a velocity term to accumulate gradients.	$v\_t = \beta\ v\_(t-1) + \eta\ \nabla\theta\ J(\theta)\ ;\ \theta \leftarrow \theta - v\_t$	Faster convergence, reduces oscillations	Can overshoot if β is large
Nesterov (NAG)	Computes gradient at the "lookahead" position.	$v\_t = \beta\ v\_(t-1) + \eta\ \nabla\theta\ J(\theta - \beta\ v\_(t-1))\ ;\ \theta \leftarrow \theta - v\_t$	Faster and more accurate than momentum	Slightly more computation per step
Adagrad	Adapts learning rate per parameter (smaller for frequent features).	$G\_t = G\_(t-1) + g\_t^2\ ;\ \theta \leftarrow \theta - (\eta\ /\ (sqrt(G\_t) + \varepsilon))\ *\ g\_t$	Great for sparse features	Learning rate shrinks too much over time
RMSprop	Uses exponential moving average of squared gradients.	$E[g^2]\_t = \rho\ E[g^2](t-1) + (1-\rho)\ g\_t^2\ ;\ \theta \leftarrow \theta - (\eta\ /\ (sqrt(E[g^2]\_t) + \varepsilon))\ *\ g\_t$	Works well for non-stationary problems	Extra hyperparameter (ρ) to tune
Adam	Combines momentum (m_t) + RMSprop (v_t).	**Adam Optimizer Update Rules (rewritten)** First moment (mean of gradients): $m\_t = \beta1 * m\_(t-1) + (1 - \beta1) * g\_t$ Second moment (uncentered variance): $v\_t = \beta2 * v\_(t-1) + (1 - \beta2) * (g\_t)^2$ Bias correction for first moment: $m\_hat\_t = m\_t / (1 - \beta1\hat{}t)$ Bias correction for second moment: $v\_hat\_t = v\_t / (1 - \beta2\hat{}t)$ Parameter update rule: $\theta = \theta - (\eta\ /\ (\ sqrt(v\_hat\_t) + \varepsilon\ ))\ *\ m\_hat\_t$	Very popular default, adaptive & robust	Slightly more memory use

GD Momentum vs Nesterov Momentum

GD with Momentum:

Think of a ball rolling down a hill.

Momentum **adds "velocity"** to the updates, so the ball keeps moving in a consistent direction and can roll over small bumps (shallow local minima).

Updates are based on the **current gradient**.

GD with Nesterov Momentum:

Like **GD Momentum, but you look ahead**.

You compute the gradient **at the "future" position** (current position + momentum step), so it's like adding **acceleration** to the existing velocity.

This often allows more **precise and faster convergence**, especially in curved or steep regions of the loss surface.

Summary in your words:

GD_Nesterov ≈ GD_Momentum + acceleration (because it anticipates where the momentum will take you and corrects in advance).

Recomendations

For non-convex loss, use optimizers that handle noisy gradients and can escape

shallow minima:

Optimizer	Why it helps
SGD (mini-batch)	Noise in updates can help escape local minima and saddle points
Momentum / Nesterov	Adds velocity, helps overcome shallow local minima
Adam / RMSProp / Adagrad	Adaptive learning rates, faster convergence, more robust in complex landscapes

terms to know:

- accumulate gradients

Dataset:

$\{(1,2),(2,4),(3,6),(4,8),(5,10),(6,12)\} \rightarrow 6$

1. Batch Gradient Descent (Batch GD)

Uses **all 6 samples at once** to compute gradient.

Updates per epoch: 1

2. Mini-batch Gradient Descent

Batch size = 2 → dataset split into 3 mini-batches:

{(1,2),(2,4)},{(3,6),(4,8)},{(5,10),(6,12)}

Updates per epoch: 3 (one per mini-batch)

3. Stochastic Gradient Descent (SGD)

Batch size = 1 → each sample is a mini-batch of 1:

(1,2),(2,4),(3,6),(4,8),(5,10),(6,12)

Updates per epoch: 6 (one per sample)

Noisy updates but can converge faster in practice

Optimization comparison: vary these:

1. Learning rate (critical!)

    Try {0.001, 0.01, 0.05, 0.1}.

    Adam and RMSprop usually like lower rates. SGD needs tuning.

2. Batch size

    Compare {1, 16, 32, 128, full-batch}.

    You'll see tradeoffs in stability vs. generalization.

3. Momentum (for SGD, Nesterov)

    Values {0.5, 0.9, 0.99}.

    High momentum can overshoot, but speeds convergence.

4. Network architecture

    Currently (32, 16) hidden layers. Try deeper/wider: (64, 32), (128, 64, 32).

    This will show how optimizers scale with model complexity.

5. Regularization

    Add dropout or alpha (L2 penalty in MLPClassifier).

    See how optimizers behave under noisy conditions.

6. Dataset noise

    Increase noise in moons/circles.
    split it to 70/30

    Strong optimizers should generalize better with higher noise.