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)≈−3.458

Average Over Batch Size (N=4):

L=−1/4 × (−3.458)≈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).

99% accuracy

To achieve 99%+ validation accuracy and learn deep learning techniques:

BatchNorm: Stabilizes training by normalizing activations, reducing vanishing gradients in deep nets (Depth=10).
He Initialization: Prevents dead ReLUs by setting proper weight variance, crucial for deep ReLU networks.
Dropout (0.1): Prevents overfitting; low rate avoids over-regularization in deep nets.
Data Augmentation: Simulates more data (rotations, shifts, zoom) to improve generalization, critical for 99%+.
SGD with Momentum + LR Scheduling: Momentum accelerates convergence; annealing (step-down LR) refines optimization for high accuracy.
Residual Connections: Allow gradients to flow through deep layers, reducing vanishing gradient issues.
L2 Regularization: Small weight decay (1e-6) curbs overfitting without hurting learning.
Early Stopping: Saves training time and prevents overfitting by stopping when val_loss plateaus.
Experimentation Tips for 99%+:

- Increase units to 2048 or 4096 (wider layers boost capacity).
- Strengthen augmentation (add shear_range=0.1, or try elastic distortions via custom code).
- Try AdamW optimizer (weight decay built-in) or label smoothing in loss.
- Use ensemble: Train 3 models, average predictions (can push from 98.5% to 99.2%+).
- More epochs (100+) if not converging; monitor plots for plateaus.

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).

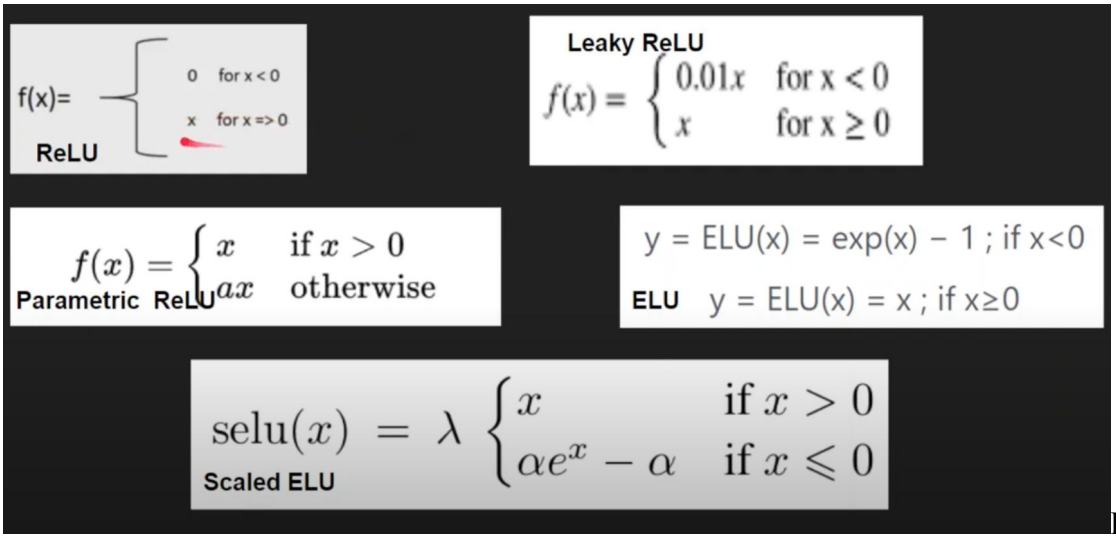
If α_i :- is learnable, we get **Parametric ReLU (PReLU)**.

Comparing with ReLU and Leaky ReLU

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

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

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



Expected Findings:

ReLU:

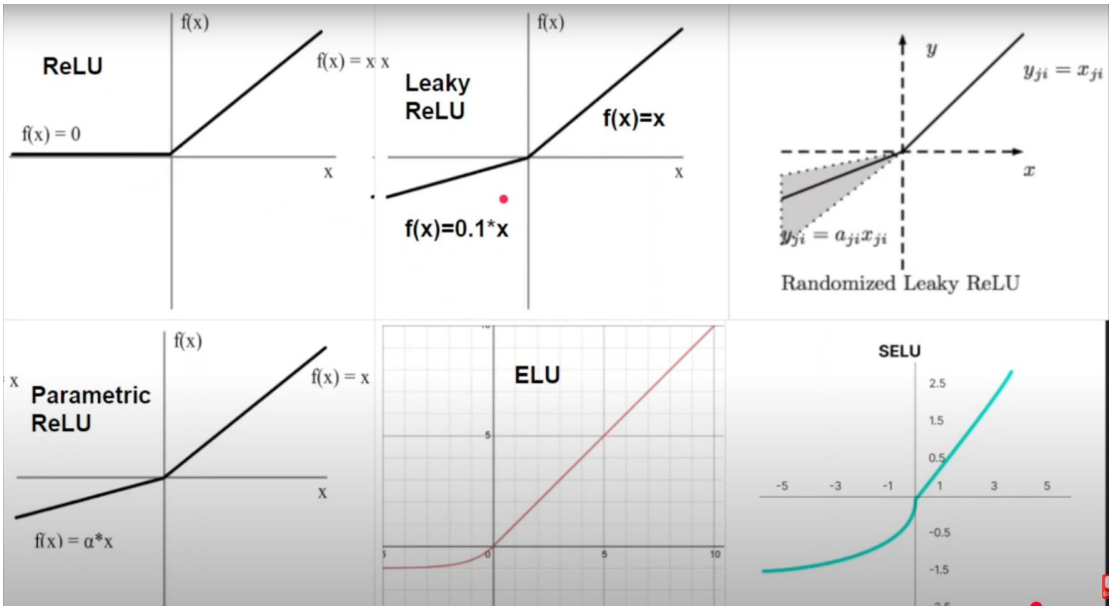
- **Performance**: As above, achieves ~97–98% validation accuracy and low loss.
- **Gradient Norms**: Stable and relatively large across layers, as ReLU passes positive gradients effectively.
- **Limitation**: Some neurons may "die" (always output zero) if many inputs are negative, leading to slightly 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)=αx (α is small, fixed)	f(x)=αi x (αi is learned)	f(x)=α(ex−1)
Gradient for x<0	0	Constant α	Learned αi	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

Symbol	Meaning
J(θ; B)	Loss function evaluated on mini-batch B
∇θ J(θ; 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:

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

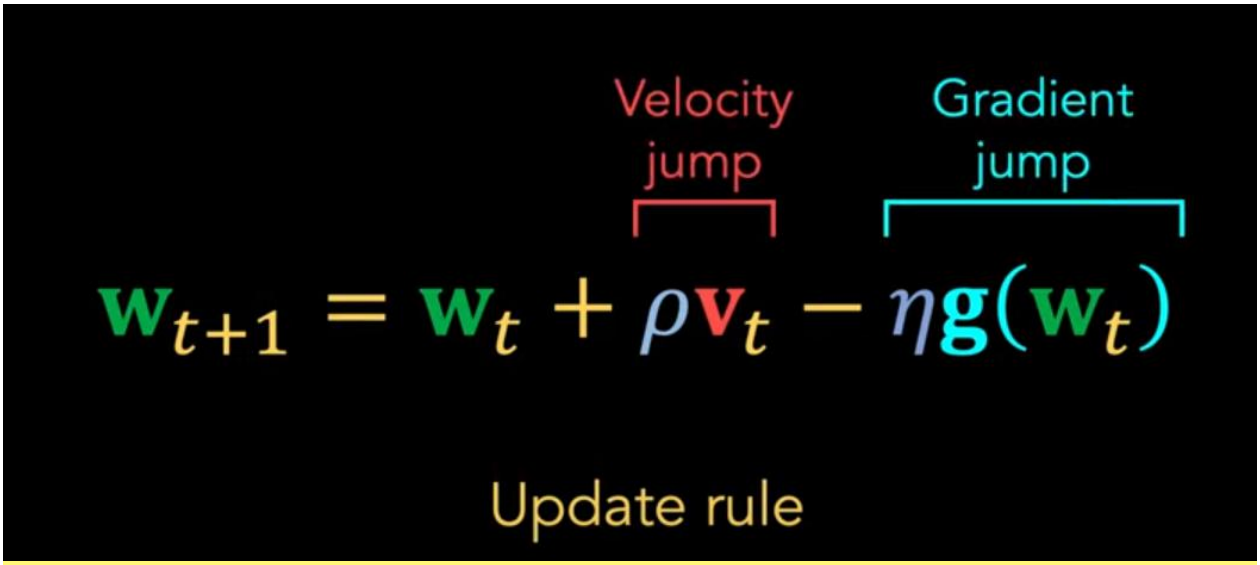
Terms Table:

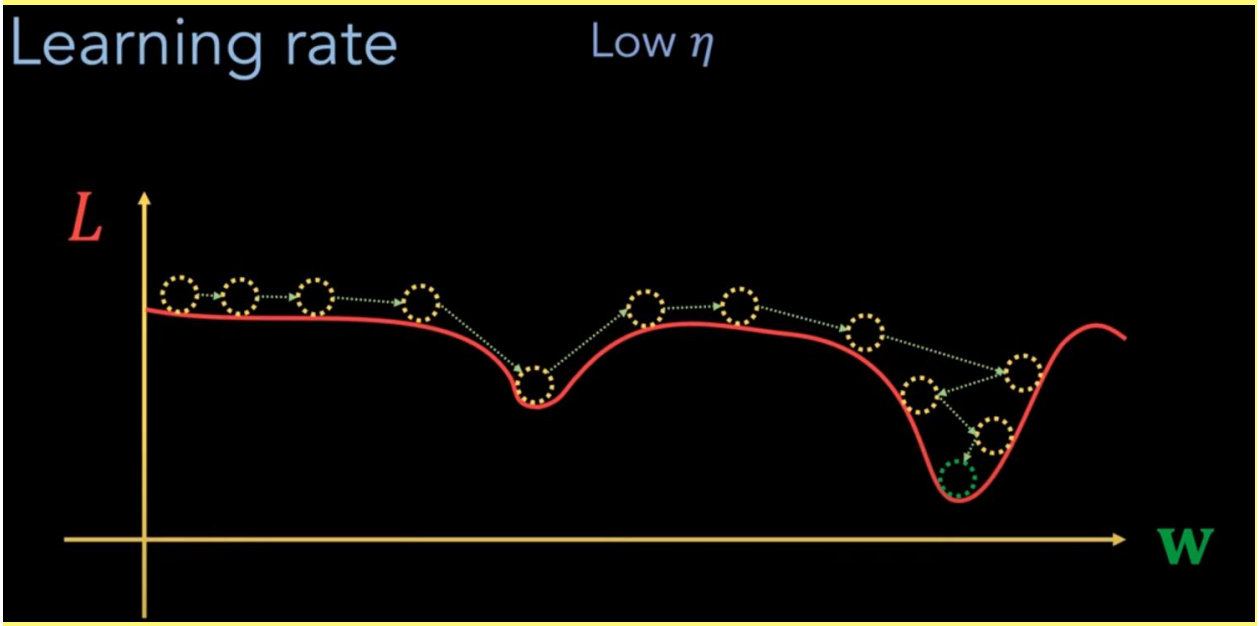
Symbol	Meaning
θ	Model parameters
η	Learning rate
x_i, y_i	Single sample (features, target)
$J(\theta; x_i, y_i)$	Loss function for this single sample
$\nabla\theta\, J(\theta; x_i, y_i)$	Gradient based on one sample

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
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 (change in θ J(θ))
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)
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.	$E[g^2]\_t = \varrho\ E[g^2](t-1) + (1-\rho)\ g\_t^2$; $\theta \leftarrow \theta - (\eta /$ $(sqrt(E[g^2]\_t) + \varepsilon)) * g\_t$	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: $m\_hat\_t = m\_t / (1 - \beta1^t)$ Bias correction for second moment: $v\_hat\_t = v\_t / (1 - \beta2^t)$ 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

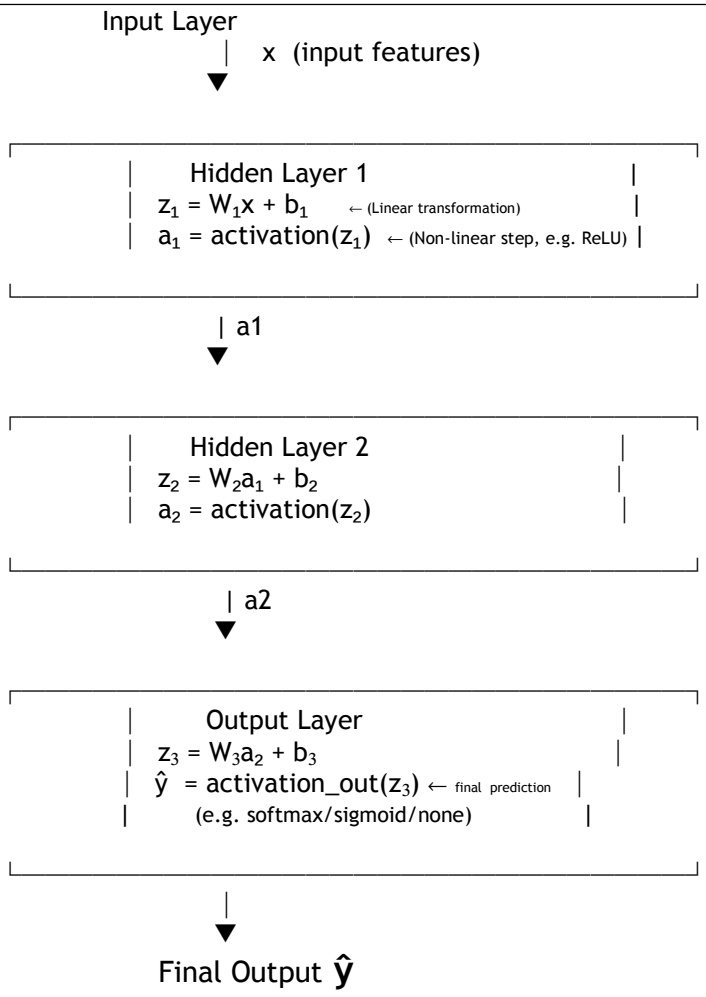
    Strong optimizers should generalize better with higher noise.

10/6/2025

Chapter 1A Deep Neural Network by Dr Fantahun

FEEDFORWARD NEURAL NETWORK

Input → Linear → Activation → Next Layer → Output.



NB: The output layer may use activation function depending of the type of task.
If
- Binary classification: use Sigmoid activation function
- Multiclass classification : use Softmax
- Regression : use none

Explanation of Flow

Input Layer:

Takes the data **x** (e.g., features).

Passes it to the first hidden layer.

Hidden Layer 1:

Computes $\mathbf{z_1 = W_1x + b_1}$ (linear step).

Applies **activation** (nonlinear step) → $\mathbf{a_1 = g(z_1)}$.

Sends $\mathbf{a_1}$ to the next layer.

Hidden Layer 2:

Takes $\mathbf{a_1}$ as input.

Does the same: $\mathbf{z_2 = W_2a_1 + b_2 \rightarrow a_2 = g(z_2)}$.

Sends $\mathbf{a_2}$ to the output layer.

Output Layer:

Computes $\mathbf{z_3 = W_3a_2 + b_3 \rightarrow \hat{y} = g\_out(z_3)}$.

$\hat{\mathbf{y}}$ is the final prediction (class label, probability, etc.).

Task type	Output neurons	Activation function
Binary classification	1	Sigmoid
Multi-class classification	n (number of classes)	Softmax

Task type	Output neurons	Activation function
Regression	1	None (linear)

Five quick takeaways

1. Feedforward networks approximate functions $f*$ by learning parameters.
2. Training = forward pass + loss + backprop + update (repeat).
3. Hidden layers are not shown the "right answer" — backprop teaches them indirectly.
4. Depth gives power but increases training difficulty and data need.
5. Watch learning rate, regularization, and initialization.

Is non-convexity good or bad?

Pros:

Allows the network to **represent highly complex functions**.

Gives flexibility: the network can model intricate patterns that linear models **cannot**.

Cons:

Harder to train:

Gradient descent might get stuck in a **bad local minimum**.

Optimization can be slower due to plateaus or saddle points.

Surprisingly, **most local minima in large networks have almost the same loss as the global minimum**.

The high-dimensional nature of deep networks makes **bad local minima rare**, so gradient descent usually works well.

Saddle points (flat areas) are actually more of a problem than local minima.

MAXIMUM Likely Hood

In a binary classification task, the observed data are the inputs x and labels y. MLE finds network parameters θ so that the network's predicted probabilities make the observed labels as likely as possible.

Concept	Analogy
Observed data	The answer sheet (true labels)
Model prediction	Student's guess (network output)
MLE	Adjusting guesses so they match the answer sheet as closely as possible

· The linear equation y = Wx + b is your model prediction.
· MLE treats the model prediction as the mean or probability of the data.

Concept	Meaning
MLE	Find parameters θ that make observed data most probable
Likelihood (L)	Probability of data given θ
Log-Likelihood	Easier form (sum instead of product)
Negative Log-Likelihood (NLL)	What we minimize (used as cost)
Cross-Entropy	The same as NLL for classification tasks

Multinoulli and Bernoulli distributions
· Bernoulli: A Bernoulli distribution models a binary random variable — it has two possible outcomes: 0 or 1 (failure or success).Bernoulli → Binary → 2 outcomes → sigmoid.
· Multinoulli : A Multinoulli (sometimes called categorical) distribution models a discrete random variable with n possible outcomes.
Multinoulli → Multiple → >2 outcomes → softmax.

Distribution	Number of outcomes	NN output unit	Example
Bernoulli	2 (binary)	Sigmoid	Cat vs Dog
Multinoulli	n (>2)	Softmax	MNIST digits 0–9

The activation function of the output layer influences the choice of cost function because it determines the form of the network's predictions.
Output layer activation + label type → choose cost function.
Hidden layer activation → affects learning dynamics, not the cost function.

· Linear units → MSE: minimize difference between predicted and actual value.
· Sigmoid → binary cross-entropy: penalize wrong probabilities for a binary target.
· Softmax → multiclass cross-entropy: penalize probability assigned to wrong classes.

Step-by-step summary

1. Decide task type: regression, binary, multiclass.
2. Choose output unit: Linear, Sigmoid, Softmax.
3. Define cost function using maximum likelihood:
   Linear → MSE
   Sigmoid → binary cross-entropy
   Softmax → multiclass cross-entropy

4. Forward pass: compute predictions.
5. Compute cost/loss: compare predictions with true labels.
6. Backpropagate gradients to adjust weights.
   Repeat until network minimizes loss.

MAXout:

Oct - 7 - 2025

Maxout:

- Maxout is an advanced version (a generalization) of the ReLU activation function.
- Instead of applying a single activation function like ReLU, Maxout learns which activation shape works best.

Piecewise-linear function = many small straight lines connected to form a curve-like shape.

In Maxout, we don't apply a ReLU (or any fixed activation).
Instead, we replace ReLU with the max operation itself:

$$\text{Maxout}(x)=\max(z_1,z_2,\ldots,z_k)$$

That means:

You compute k different $z_j = w_j^T x+b_j$

then you just take the maximum of them.

· A **Maxout neuron** does not use a fixed formula such as ReLU(z) = max(0, z).

· Instead, it computes **several linear functions** and **selects the largest output** among them.

. Piecewise-linear function = many small straight lines connected to form a curve-like shape.

. Maxout can mimic ReLu and its variants if K = 2 ? (because ReLu selects max from 2 paramters so are the other variants)

. Why Maxout needs Regularization than ReLU
Think of it like this:

ReLU is like a student using one pen to take notes.
Maxout is like a student using 4 pens of different colors — they can make fancy notes, but might spend too much time decorating instead of learning
→ So, you (the teacher) tell them: "Use the pens, but don't overdo it!"
That "don't overdo it" = regularization.

Steps involved while using maxout:

Step 1:

Take an input vector x.

Step 2:

Compute k different linear responses:

$z_1 = w_1^T x + b_1$
$z_2 = w_2^T x + b_2$
...
$z_k = w_k^T x + b_k$

(each z has its own weight vector w_j and bias b_j)

Step 3:

Group these k values together → this group represents one neuron's candidates.

Step 4:

Take the maximum value from the group:
output = $\max(z_1, z_2, \ldots, z_k)$

Step 5:

Send this output to the next layer — this replaces the usual activation step like ReLU or tanh.

4. Properties

Each neuron is defined by k sets of weights, not just one.

If k = 2, Maxout can mimic ReLU, Leaky ReLU, or even other shapes.

With larger k, it can approximate any convex function.

Requires more regularization (e.g., dropout) or large training data to avoid overfitting.

Chapter 2B Optimization problem

Optimization for Deep Learning: Main Concepts (Flashcards)

Section 1: Optimization Overview and Parameter Initialization Strategies

Card Front (Concept/Question)	Card Back (Answer/Details)
Optimization for Deep Learning Agenda	The agenda includes: How Learning Differs from Pure Optimization, Challenges in Neural Network Optimization, Basic Algorithms, Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates, and Approximate Second-Order Methods.
Why is Parameter Initialization so critical in Deep Learning (DL)?	DL training algorithms are iterative and require specifying an initial point. The initial point strongly affects whether the algorithm converges at all, how quickly it converges, whether it converges to a high or low cost, and the final generalization error.
What is the only property known with certainty regarding initial parameters?	The initial parameters must "break symmetry" between different units. If two hidden units have the same activation function and are connected to the same inputs, they must have different initial parameters so that a deterministic learning algorithm updates them

	differently.
How are weights and biases typically initialized?	Weights are almost always drawn randomly from a Gaussian or Uniform distribution. Biases are typically set to heuristically chosen constants, often zero, although non-zero biases are used for output units (to obtain correct marginal statistics) or ReLU hidden units (to avoid immediate saturation).
What are the trade-offs regarding the scale of initial weights?	Larger initial weights yield stronger symmetry breaking and help avoid losing signal during forward or back-propagation. Too large initial weights may cause exploding values during forward/back-propagation, lead to chaos in recurrent networks, or cause activation functions to saturate, resulting in complete loss of gradient.
Describe the Sparse Initialization scheme.	Each unit is initialized to have exactly k non-zero weights. This method aims to keep the total amount of input to the unit independent from the number of inputs (m) without requiring the magnitude of individual weights to shrink with m.
What is the common heuristic for setting the initial scale of weights in a fully connected layer (m inputs, n outputs)?	Sample each weight from U(−1/m ,1/m). Other normalized initialization strategies exist, like Glorot and Bengio (2010), designed to compromise between preserving activation variance and gradient variance.

Section 2: Algorithms with Adaptive Learning Rates (AALR) (includes Adaptive Gradient, RMSprop, Adam)

Card Front (Concept/Question)	Card Back (Answer/Details)
Why are Adaptive Learning Rate Algorithms necessary?	The learning rate is often the most difficult hyperparameter to set, and the cost function can be highly sensitive to certain directions in parameter space and insensitive to others. AALR algorithms use a separate learning rate for each parameter, adapting them throughout training.
What is the core idea of the Delta-Bar-Delta algorithm?	If the partial derivative of the loss with respect to a parameter maintains the same sign, the learning rate should increase. If the partial derivative changes sign, the learning rate should decrease.
How does AdaGrad work, and what is its primary weakness in DL?	AdaGrad individually adapts learning rates by scaling them inversely proportional to the square root of the sum of all their historical squared gradient values (Duchi et al., 2011). The weakness is that the accumulation of squared gradients from the beginning of training can result in a premature and excessive decrease in the effective learning rate for deep neural networks.
How does RMSProp improve upon AdaGrad for non-convex optimization?	RMSProp (Hinton, 2012) modifies AdaGrad by changing the gradient accumulation into an exponentially weighted moving average. This allows it to discard history from the extreme past, enabling rapid convergence once the learning trajectory finds a locally convex bowl.
What is Adam, and what makes it distinct from RMSProp?	Adam (Kingma and Ba, 2014), derived from "adaptive moments," combines RMSProp and momentum. It incorporates momentum as an estimate of the first-order moment and includes bias corrections to the estimates of both the first- and second-order moments to account for their initialization at the origin. RMSProp lacks this correction factor.
Which optimization algorithms are currently the most popular?	SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta, and Adam.

Section 3: Approximate Second-Order Methods (ASOM)

Card Front (Concept/Question)	Card Back (Answer/Details)
How do Second-Order Methods differ from First-Order Methods?	Second-order methods make use of second derivatives (the Hessian matrix, H) to improve optimization, unlike first-order methods which only use the gradient.
What is Newton's Method, and why is it impractical for large DL networks?	Newton's method uses a second-order Taylor series expansion to approximate the objective function. For a locally quadratic function, it can jump directly to the minimum by rescaling the gradient by the inverse Hessian (H−1). It is impractical for large DL networks due to the significant computational burden of computing and manipulating the Hessian matrix.
What problem does the method of Conjugate Gradients solve?	Conjugate gradients efficiently avoids the calculation of the inverse Hessian required by Newton's method. It addresses the weakness of steepest descent, where the iterative line searches follow orthogonal directions that often "undo" progress made in previous steps (the zig-zag pattern).
Define "conjugate directions" in the context of Conjugate Gradients.	At iteration t, the next search direction dt is chosen to be conjugate to the previous search direction dt−1, meaning dtT H(J)dt−1 =0. This prevents the new search step from undoing progress made in the previous direction.

RNN

RNNs share parameters in a different way.
Each member of the output is a function of the previous members of the output.

For the simplicity of exposition, we refer to RNNs as operating on a sequence that contains vectors x(t) with the time step index t ranging from 1 to τ .

is recurrent because the definition of s at time t refers back to the same definition at time t−1.

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}),$$

RNN Architectures:

Type 2 — Output-to-Hidden Recurrence (output feeds next hidden)

RNNs that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step

Less powerful than Type-1?

Type 3 — Sequence-to-Fixed-Size Output (summary / encoder pattern)

Architecture: run an RNN (usually hidden-to-hidden) across the whole variable-length input; after the final time step take the final hidden state h(τ)h^{(τ)}h(τ) (or apply pooling) to produce a single fixed-size output ooo.

Quick comparison (one-line each)

Type 1 (Hidden→Hidden): most flexible and powerful; sequential training (BPTT); good memory.

Type 2 (Output→Hidden): simpler flow, parallelizable training; less expressive because outputs must carry history.

Type 3 (Sequence→Fixed): condenses whole sequence into single vector; good for summarization/classification; might lose long-range detail.

Architecture: run an RNN (usually hidden-to-hidden) across the whole variable-length input; after the final time step take the final hidden state h(τ)h^{(τ)}h(τ) (or apply pooling) to produce a single fixed-size output ooo.

Bidirectional RNNs – Summary

Standard RNNs are **causal**: each state at time ttt only sees past inputs x(1)…x(t−1) and the current input x(t).

Some RNNs also use past outputs y if available.

Limitation: causal RNNs cannot use future inputs to improve predictions.

Bidirectional RNNs address this by processing the sequence **both forward and backward**, allowing each prediction y(t) to consider the **entire input sequence**.

Use case: speech recognition, where interpreting the current sound may depend on upcoming phonemes or words.

What actually happens in an RNN

The RNN has a **single set of weights** (same parameters at all time steps).

At each time step `t`, it takes the input `x(t)` and the **previous hidden state** `h(t-1)` and produces:

a new hidden state `h(t)`

and possibly an output `o(t)`

The process moves forward in time step by step.

When we **unroll** the RNN, we **draw or represent** the RNN **as if** it were a deep network with one layer per time step.

Each layer (time step) is a **copy of the same RNN cell** with **shared weights**.

This unrolled view helps us:

visualize how information flows over time, and

apply **Backpropagation Through Time (BPTT)** — the algorithm used for training RNNs.