Day 3: Fundamentals of Deep Learning

Naeemullah Khan
naeemullah.khan@kaust.edu.sa

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

KAUST Academy
King Abdullah University of Science and Technology

December 28, 2025

Learning Outcomes

By the end of this session, you will be able to:

1. Define Deep Learning and explain why it scales better than traditional Machine Learning.

2. Dissect a Neural Network into its core components: Neurons, Layers, and Weights.

3. Trace the Learning Process, explaining how data flows forward and gradients flow backward (Backpropagation).

4. Compare Optimizers (SGD, Momentum, Adam) and understand how they work.

Table of Contents

What is Deep Learning?

Deep Learning is simply **Machine Learning**...

...but powered by **Artificial Neural Networks** with many layers.

- **"Neural":** Inspired by the human brain (neurons).
- **"Deep":** Refers to the number of layers (depth).

What is Deep Learning?

Deep Learning is simply **Machine Learning**...

...but powered by **Artificial Neural Networks** with many layers.

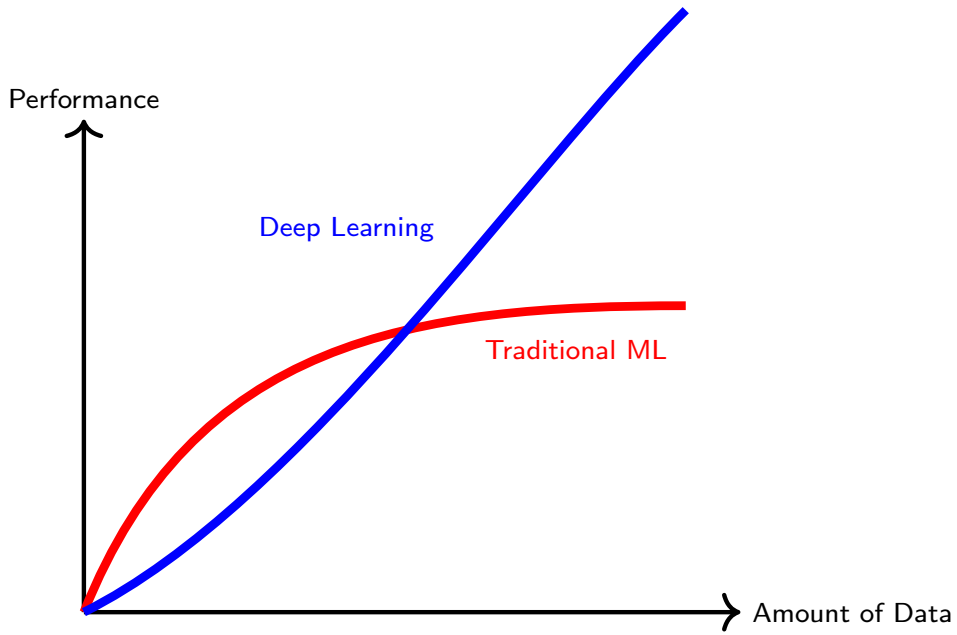
- **"Neural":** Inspired by the human brain (neurons).
- **"Deep":** Refers to the number of layers (depth).

But why did Deep Learning suddenly become so powerful?

Why Now?

Deep Learning isn't new. The ideas date back decades. But three things changed:

1. More **data**
   (internet, sensors, images)
2. Faster **compute**
   (GPUs, cloud computing)
3. Better **algorithms**
   (training techniques, architectures)

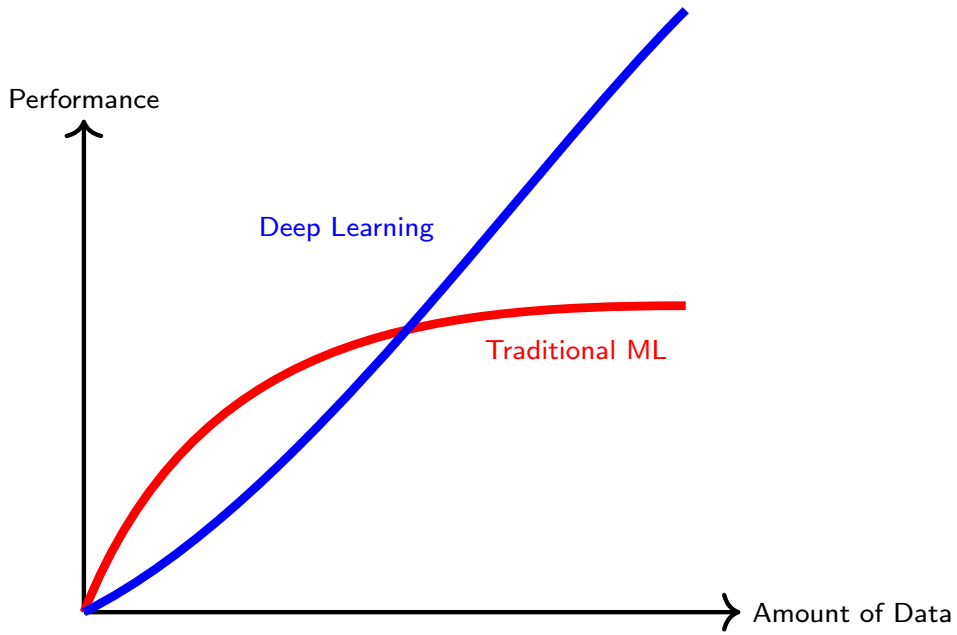


DL keeps improving as data grows.
ML hits a ceiling.

Why Now?

Deep Learning isn't new. The ideas date back decades. But three things changed:

1. More **data**
   (internet, sensors, images)
2. Faster **compute**
   (GPUs, cloud computing)
3. Better **algorithms**
   (training techniques, architectures)



DL keeps improving as data grows.
ML hits a ceiling.

But what makes Deep Learning so scalable?

The Secret: Simple Building Blocks

Traditional ML

Models are **fixed equations** designed by experts with specific assumptions about the data.

Example: Linear Regression $y = w_1 x + w_0$

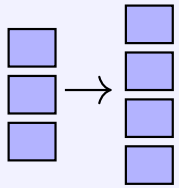
Benefit: Fast and interpretable (when assumptions hold).

Problem: Hard to adapt when data nature is different or unknown.

Deep Learning

Models are built from **simple, stackable blocks** (like LEGO).

Example: Stacking layers



Benefit: Very flexible. Can be adapted to any data type or task.

The Secret: Simple Building Blocks

أكاديمية كاوست
KAUST ACADEMY

Traditional ML

Models are **fixed equations** designed by experts with specific assumptions about the data.

Example: Linear Regression $y = w_1 x + w_0$

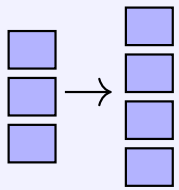
Benefit: Fast and interpretable (when assumptions hold).

Problem: Hard to adapt when data nature is different or unknown.

Deep Learning

Models are built from **simple, stackable blocks** (like LEGO).

Example: Stacking layers

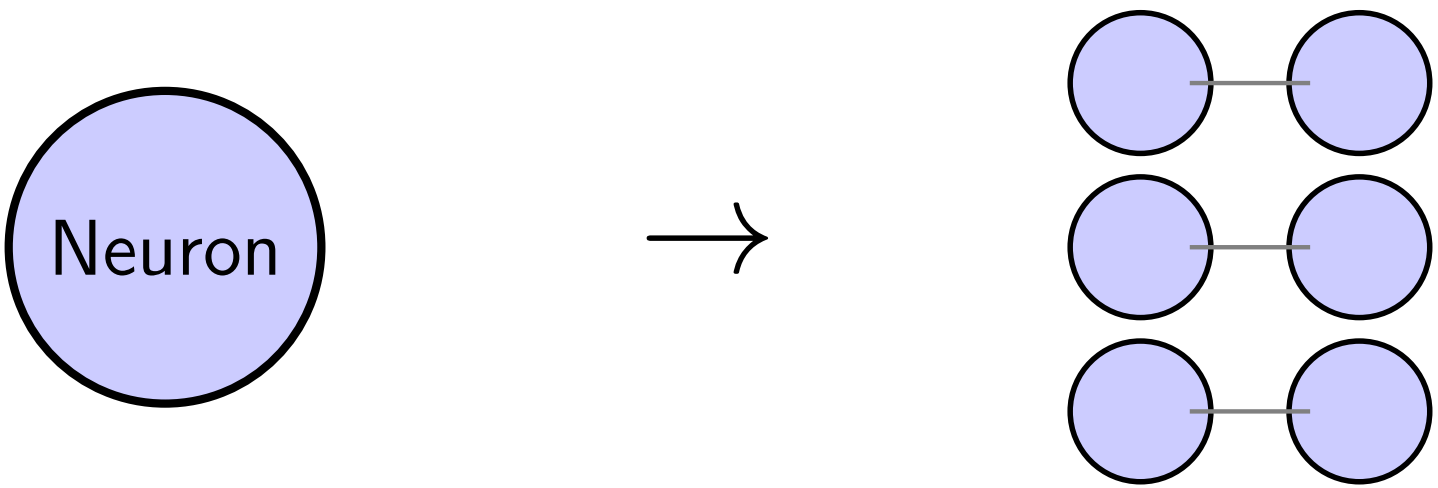


Benefit: Very flexible. Can be adapted to any data type or task.

So what is this magic building block?

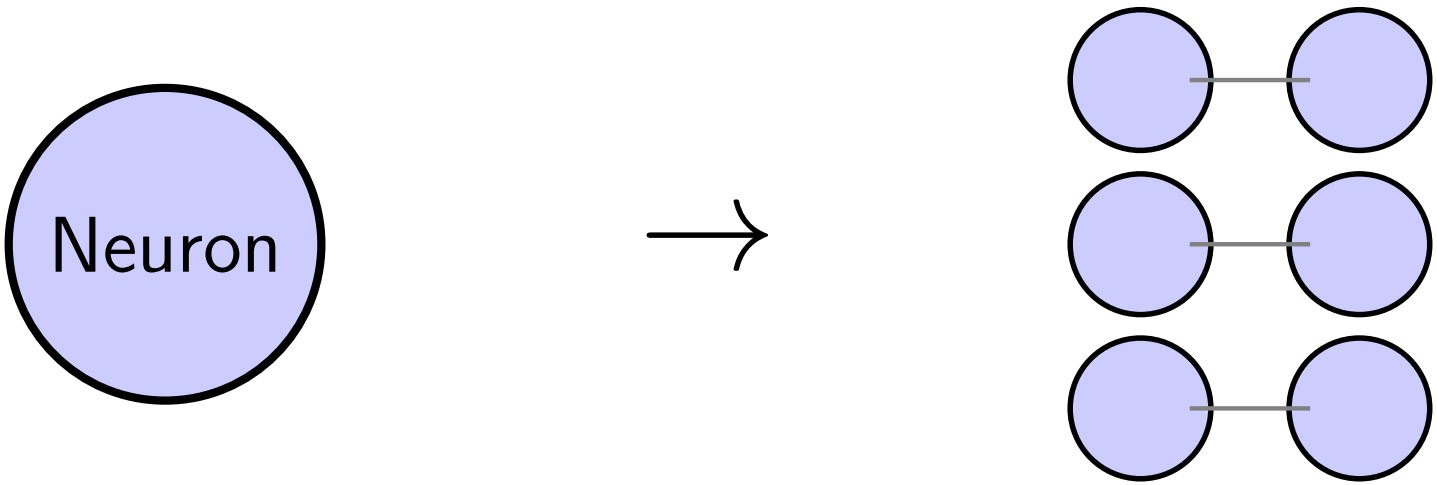
Meet the Neuron

The magic building block is called a Neuron.



Meet the Neuron

The magic building block is called a Neuron.



But how powerful can these simple blocks really be?

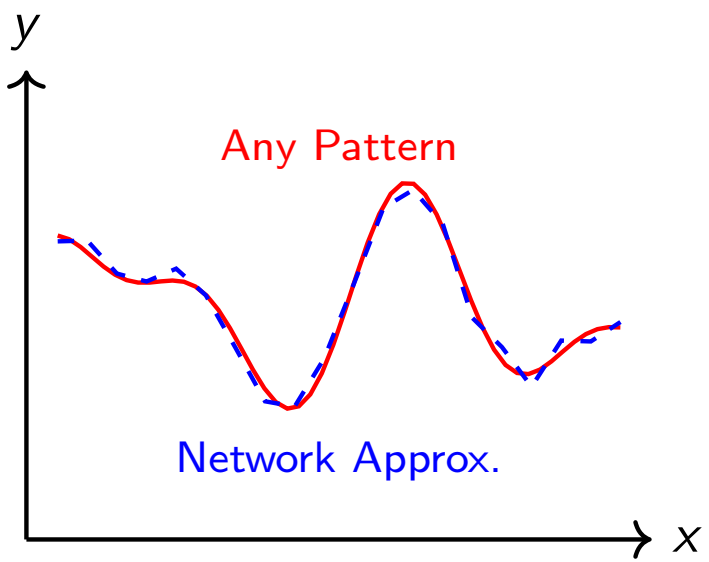
Surprisingly Powerful: Universal Approximation

Here's the remarkable part: if you stack enough neurons together, they can learn to fit **any pattern**, no matter how simple/complex it is.

Universal Approximation Theorem

A Neural Network with enough neurons can approximate **any continuous function** to arbitrary accuracy.

This is why neural networks are so flexible and powerful!



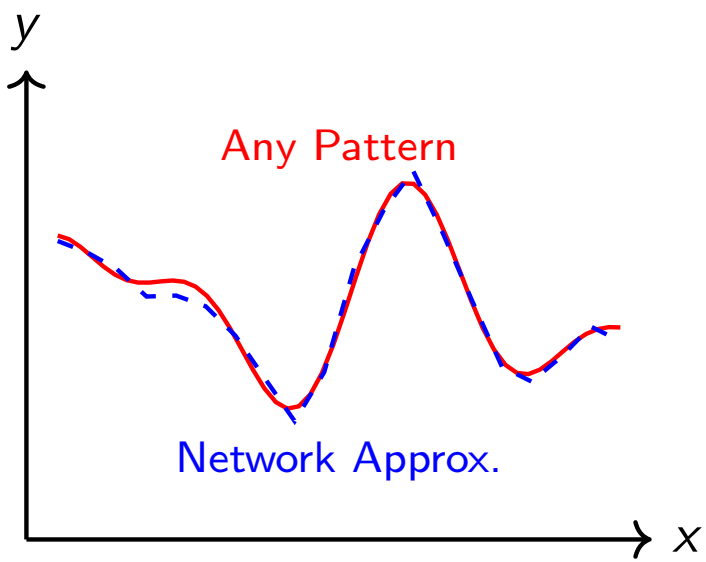
"Give me enough neurons, and I can fit any function."

Surprisingly Powerful: Universal Approximation

Here's the remarkable part: if you stack enough neurons together, they can learn to fit **any pattern**, no matter how simple/complex it is.

Universal Approximation Theorem

A Neural Network with enough neurons can approximate **any continuous function** to arbitrary accuracy.



"Give me enough neurons, and I can fit any function."

This is why neural networks are so flexible and powerful!

So let's look inside: what does a neural network actually look like?

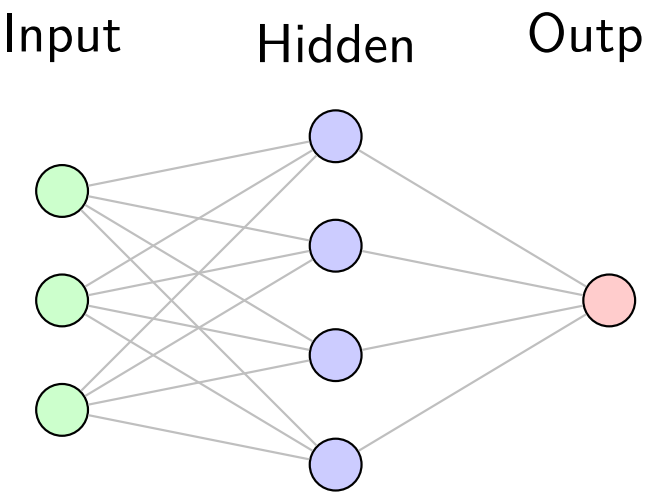
Inside a Neural Network

A Neural Network is organized into **layers** of neurons, where each layer transforms the data step by step.

The Three Parts:

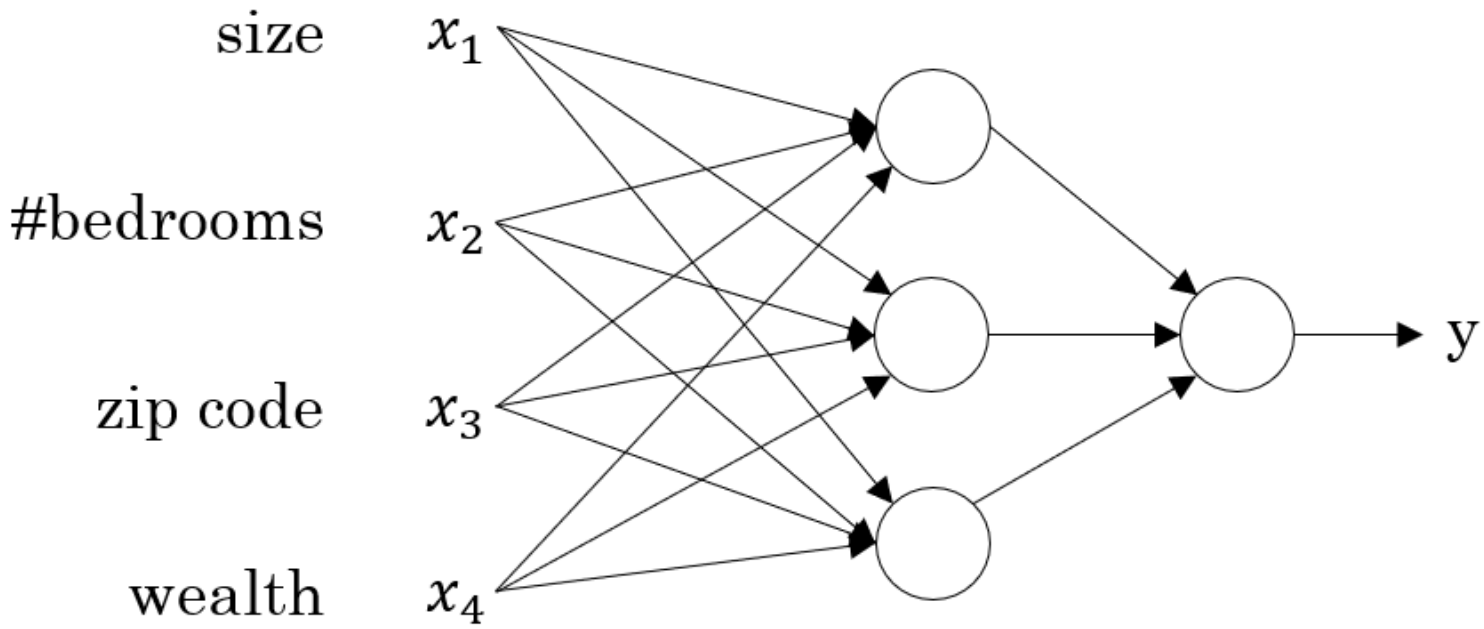
- **Input Layer:** Receives your raw data (x).
- **Hidden Layers:** Extract patterns and features.
- **Output Layer:** Produces the final prediction ($\hat{y}$).

The "deep" in Deep Learning comes from having many hidden layers.



Each circle is a neuron, each line is a connection.

Example: Predict House Price



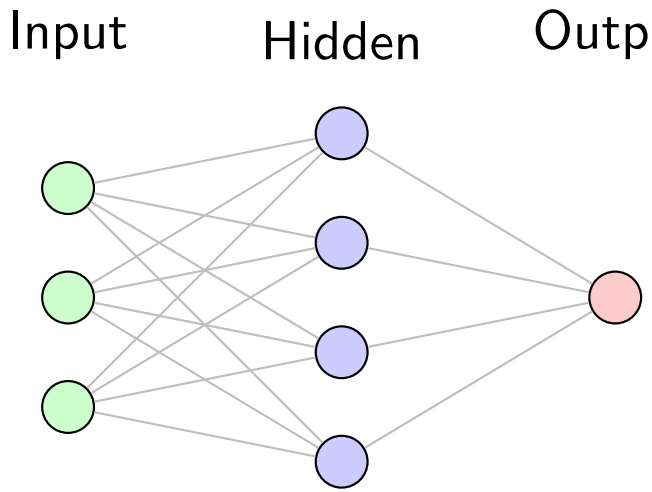
Inside a Neural Network

A Neural Network is organized into **layers** of neurons, where each layer transforms the data step by step.

The Three Parts:

- **Input Layer:** Receives your raw data (x).
- **Hidden Layers:** Extract patterns and features.
- **Output Layer:** Produces the final prediction ($\hat{y}$).

The "deep" in Deep Learning comes from having many hidden layers.



Each circle is a neuron, each line is a connection.

Now let's zoom in and see what a single neuron actually does...

What Does a Neuron Do?

A neuron performs two simple operations:

1. Linear Combination:

$$z = w^\top x + b$$

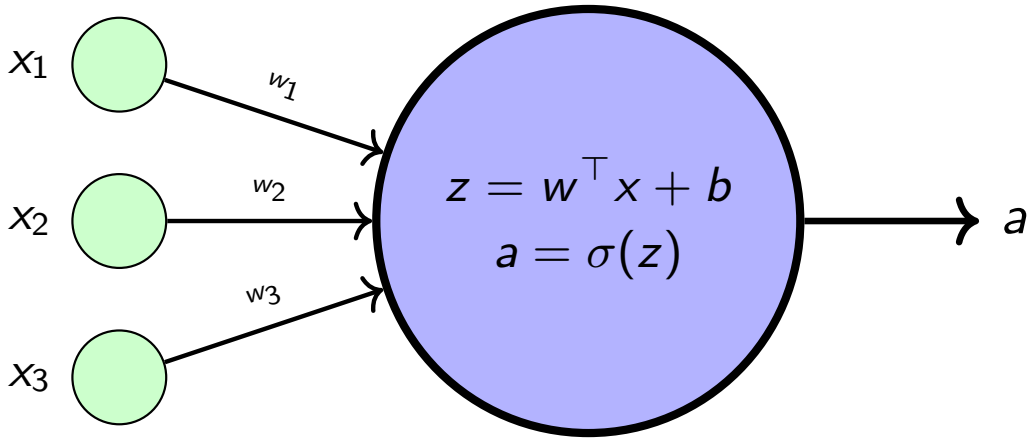
(Just like linear regression!)

2. Non-linear Activation:

$$a = \sigma(z)$$

(This is the new part)

The output a becomes the input for the next layer neurons!



What Does a Neuron Do?

A neuron performs two simple operations:

1. Linear Combination:

$$z = w^\top x + b$$

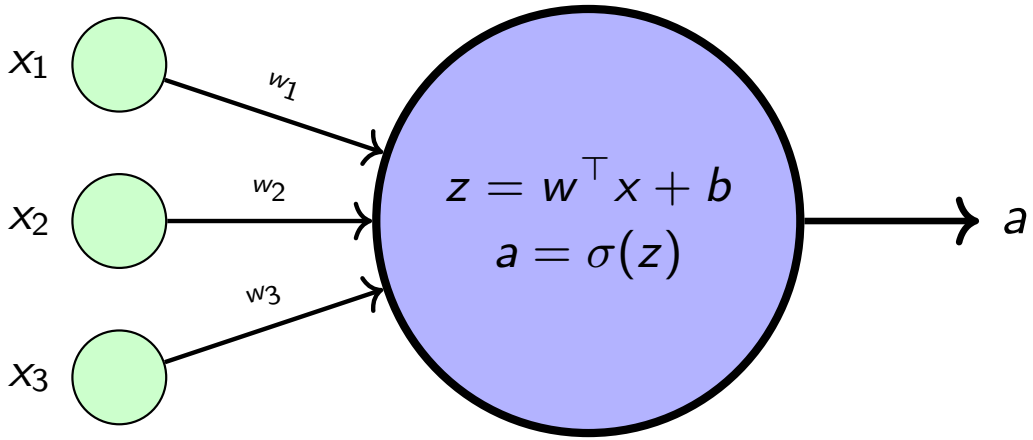
(Just like linear regression!)

2. Non-linear Activation:

$$a = \sigma(z)$$

(This is the new part)

The output a becomes the input for the next layer neurons!



But why do we need that activation function σ? Why don't we just use the linear part $(wx + b)$?

Why We Need Activation Functions

Question: Why don't we just use the linear part $(wx + b)$?

The Problem: Linearity

Without activations, stacking linear layers gives us... just another linear function!

$$\text{Linear} + \text{Linear} = \text{Linear}$$

No matter how many layers you stack, without activations, the whole network collapses into a single Linear Regression model!

Why We Need Activation Functions

Question: Why don't we just use the linear part $(wx + b)$?

The Problem: Linearity

Without activations, stacking linear layers gives us... just another linear function!

$$\text{Linear} + \text{Linear} = \text{Linear}$$

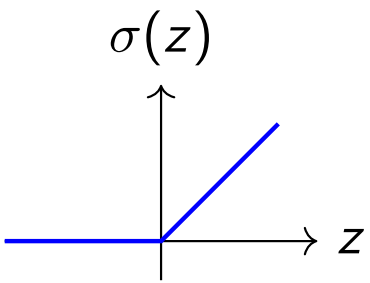
No matter how many layers you stack, without activations, the whole network collapses into a single Linear Regression model!

Solution: The "Bend": Activations (σ) introduce non-linearity. Think of it like bending a ruler. This allows us to fit curved, complex patterns.

Common Activation Functions

ReLU

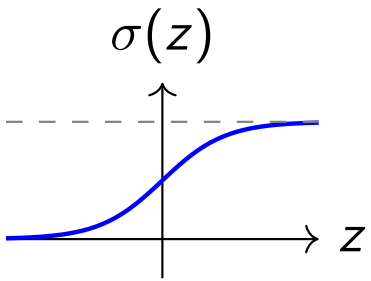
$$\sigma(z) = \max(0, z)$$



Most popular! Simple and works well in deep networks.

Sigmoid

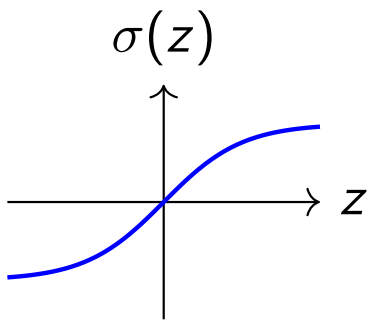
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



Output in $(0, 1)$. You know this from logistic regression!

Tanh

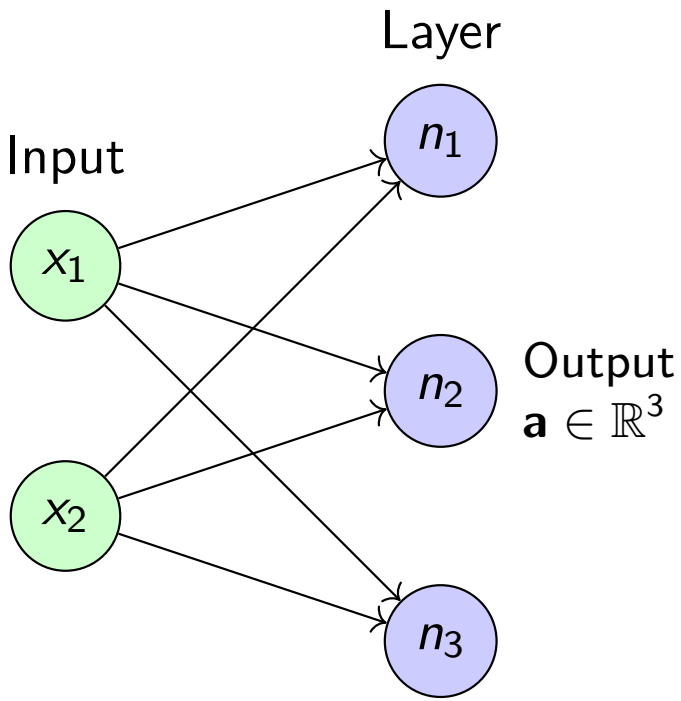
$$\sigma(z) = \tanh(z)$$



Output in $(-1, 1)$. Similar to sigmoid but centered at zero.

From Neurons to Layers

We don't just use one neuron. We stack them in parallel to form a **Layer**.



From Neurons to Layers

How do we decide the number of neurons?

- **The Input Layer:** If our data has k features (e.g., Age, Salary, Debt), the input layer must have k neurons.
- **The Output Layer:** If we are doing multiclass classification (e.g., Cat, Dog, Bird), the output layer must have k neurons (one score per class).
- **Hidden Layers:** This is a design choice (Hyperparameter). More neurons = captures more complex patterns.

Network Shapes for Different Tasks

Task	Input	Output of NN	Typical loss
Single output regression	$x \in \mathbb{R}^k$	$\hat{y} \in \mathbb{R}$	Mean squared error
Multi output regression	$x \in \mathbb{R}^k$	$\hat{y} \in \mathbb{R}^m$	MSE over all outputs
Binary classification	$x \in \mathbb{R}^k$	$\hat{p} \in (0, 1)$	Binary cross entropy
Multiclass classification	$x \in \mathbb{R}^k$	$\hat{p} \in \mathbb{R}^C$ softmax	Cross entropy

The Math: Vectorization

Since a layer is just a collection of neurons running in parallel, we can mathematically represent it using **Linear Algebra**.

- **Single Neuron:**

$$z = \mathbf{w}^\top \mathbf{x} + b$$

(Weights are a vector $\mathbf{w}$)

- **Full Layer:**

$$\mathbf{Z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

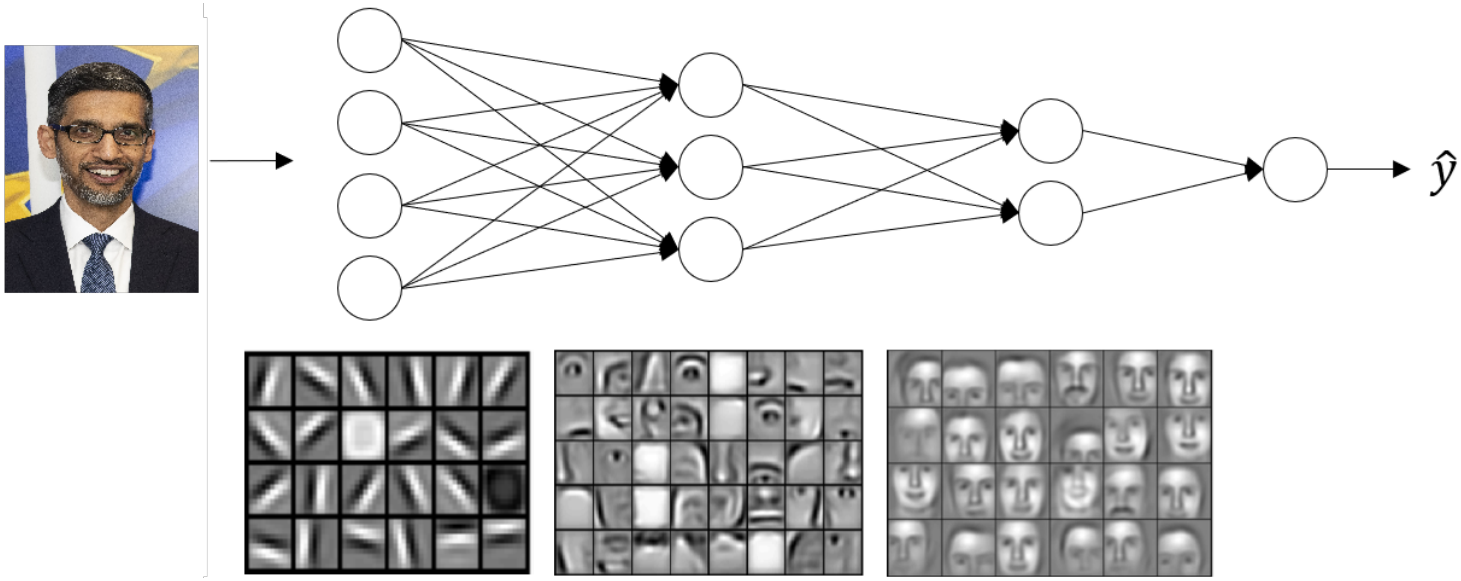
(Weights are a **Matrix W**)

Every row in matrix $\mathbf{W}$ *represents one specific neuron's weights.*

From Layer to Layers (Going Deep)

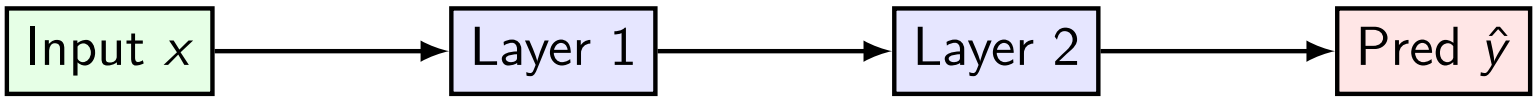
In theory: One layer is enough to fit anything.
In practice: Deeper networks are much more powerful and efficient.

- They learn **hierarchical features**:
  (Pixel $\rightarrow$ Edge $\rightarrow$ Shape $\rightarrow$ Face).



Putting it Together: The Neural Network

A Neural Network is just a **recursive chain** of these layers. The output of one layer becomes the input of the next.

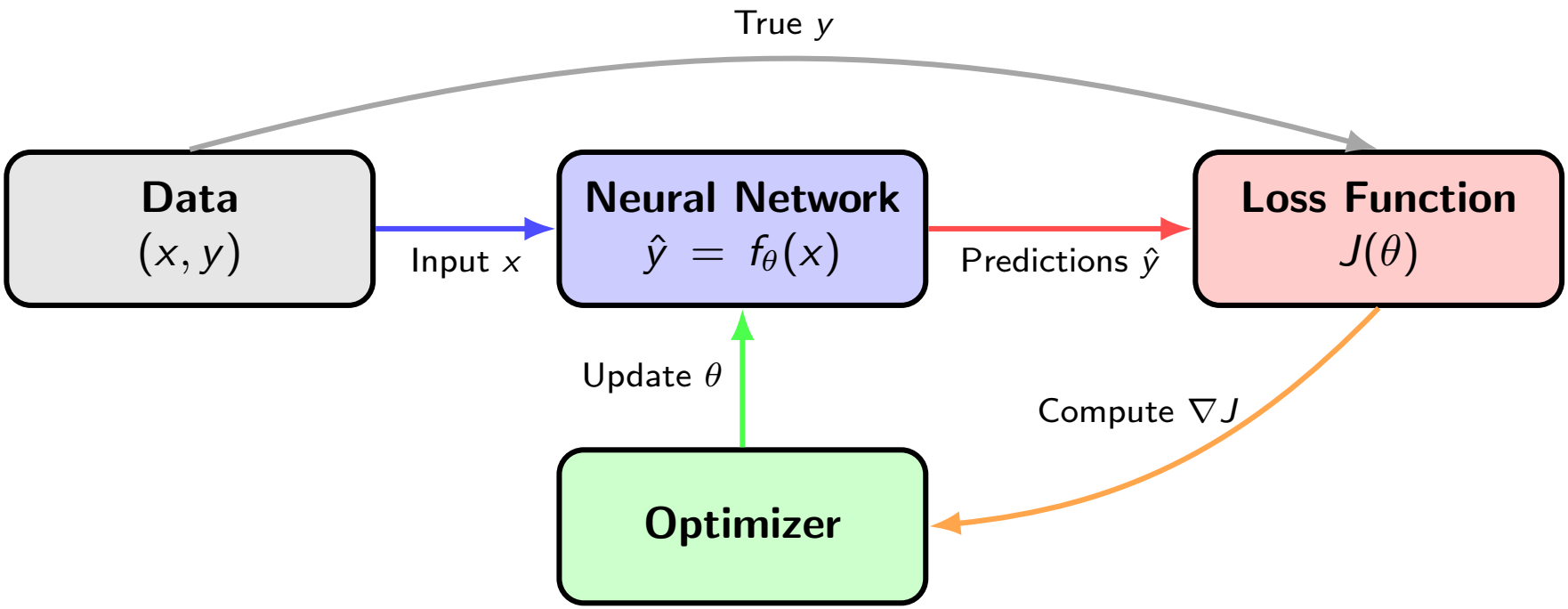


Mathematically (The Recursive Formula):

1. **Input:** $h^{(0)} = x$
2. **Hidden 1:** $h^{(1)} = \sigma(W_1 h^{(0)} + b_1)$
3. **Hidden 2:** $h^{(2)} = \sigma(W_2 h^{(1)} + b_2)$
4. **Output:** $\hat{y} = \text{Softmax}(W_3 h^{(2)} + b_3)$

Data flows from left to right, getting transformed at every step.

Neural Networks as Part of a Learning System



Training a Neural Network

To train this model we repeat two phases:

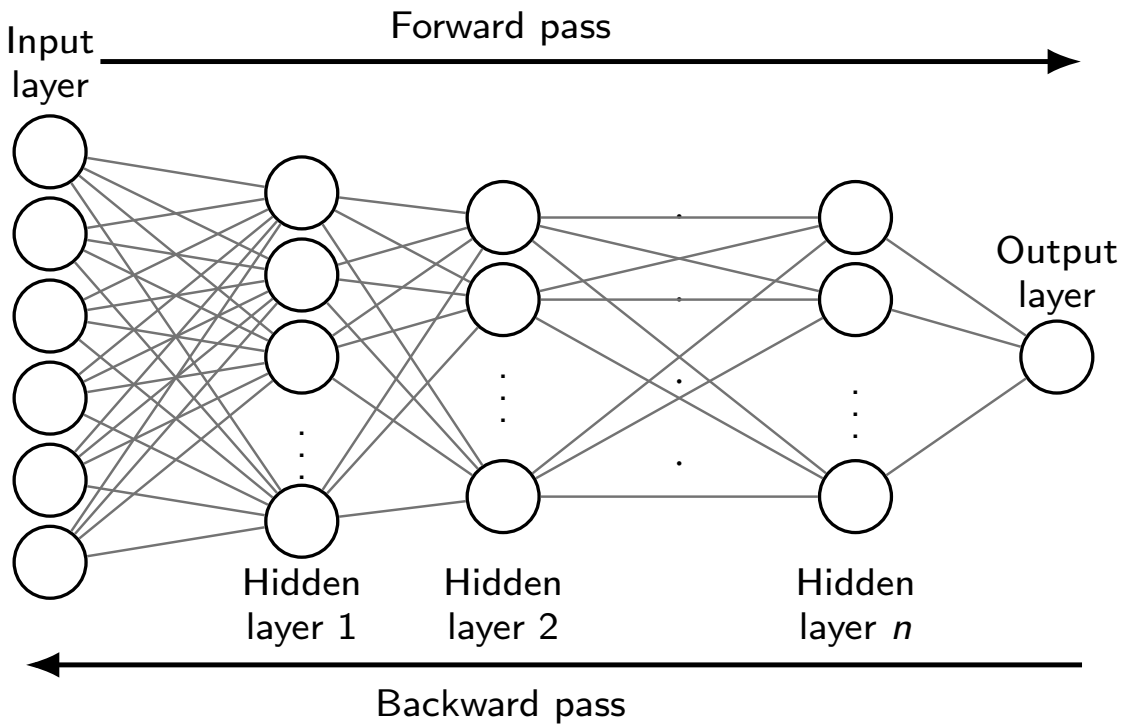
Forward Pass:

- Data $\rightarrow$ Model $\rightarrow$ Predictions
- Compute Loss

Backward Pass:

- Calculate gradients
- Update parameters

Let's discuss them in detail.



1. Forward Pass

Goal: Compute the network output $\hat{y}$ from input x, then measure the error.

What happens (layer by layer):

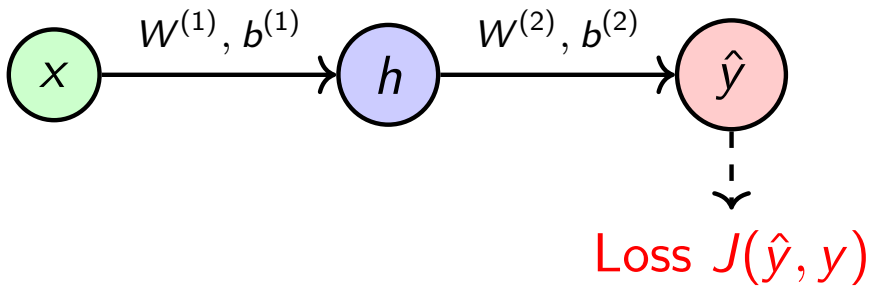
1. Start with input: $a^{(0)} = x$.
2. For each layer ℓ:

$$z^{(\ell)} = W^{(\ell)} a^{(\ell-1)} + b^{(\ell)}$$

$$a^{(\ell)} = \sigma\left(z^{(\ell)}\right)$$

3. Final layer gives the prediction: $\hat{y} = a^{(L)}$.
4. **Loss:** $J(\theta)$ compares $\hat{y}$ with the true target y.

Forward pass = just prediction (no learning yet).

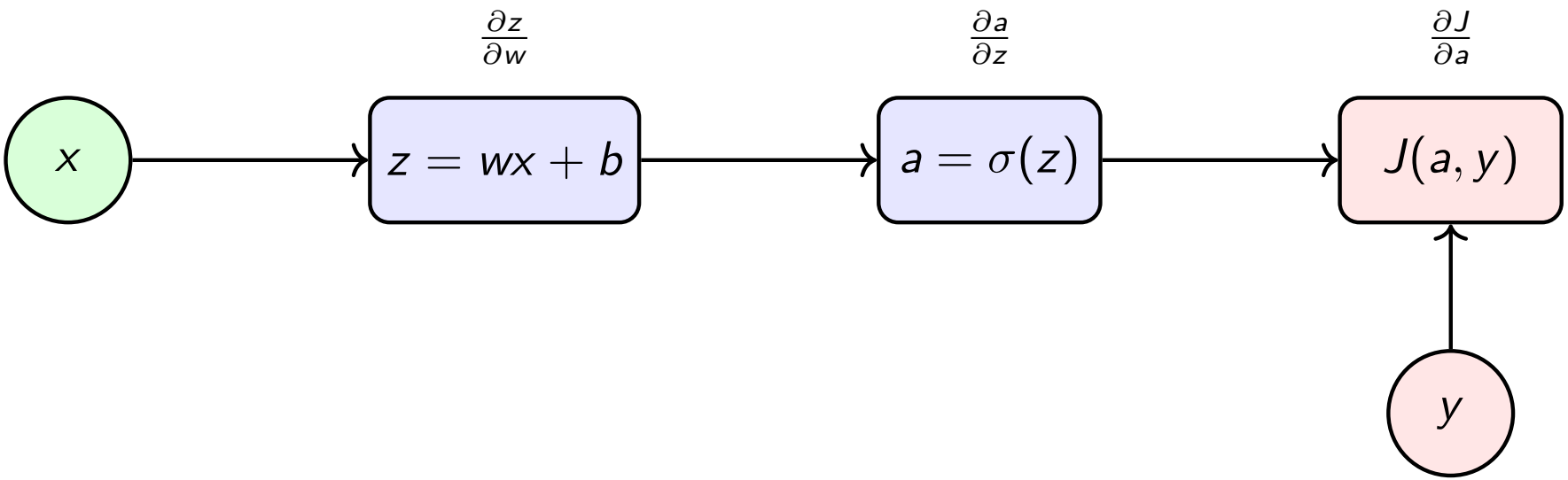


2. Backward Pass (Backpropagation)

Goal: Given feedback from the loss, calculate gradients for all weights, then update them via Gradient Descent.

By applying the **Chain Rule**, we multiply these partial derivatives to derive the gradient for every parameter.

We move from the end (the loss) **backwards** towards the beginning, passing through each weight to get its gradient. This is called **Backpropagation**.



Chain Rule Example: Compute $\frac{\partial J}{\partial w}$

Assume one neuron:

$$z = wx + b, \quad a = \sigma(z), \quad J = \frac{1}{2}(a - y)^2$$

Chain Rule (backprop)

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Now compute each piece:

$$\frac{\partial J}{\partial a} = (a - y) \qquad \frac{\partial a}{\partial z} = \sigma(z)(1 - \sigma(z)) = a(1 - a) \qquad \frac{\partial z}{\partial w} = x$$

Final result

$$\boxed{\frac{\partial J}{\partial w} = (a - y)\, a(1 - a)\, x}$$

Using the Gradient to Update Parameters

What do we do with $\frac{\partial J}{\partial w}$ **?** We use Gradient Descent to update the weights.

Parameter Update Rule (via Gradient Descent)

$$w \leftarrow w - \alpha \frac{\partial J}{\partial w} \qquad b \leftarrow b - \alpha \frac{\partial J}{\partial b}$$

- α : learning rate (step size).
- We compute $\frac{\partial J}{\partial (\cdot)}$ for **every** parameter using backprop.

Using the Gradient to Update Parameters

What do we do with $\frac{\partial J}{\partial w}$ **?** We use Gradient Descent to update the weights.

Parameter Update Rule (via Gradient Descent)

$$w \leftarrow w - \alpha \frac{\partial J}{\partial w} \qquad b \leftarrow b - \alpha \frac{\partial J}{\partial b}$$

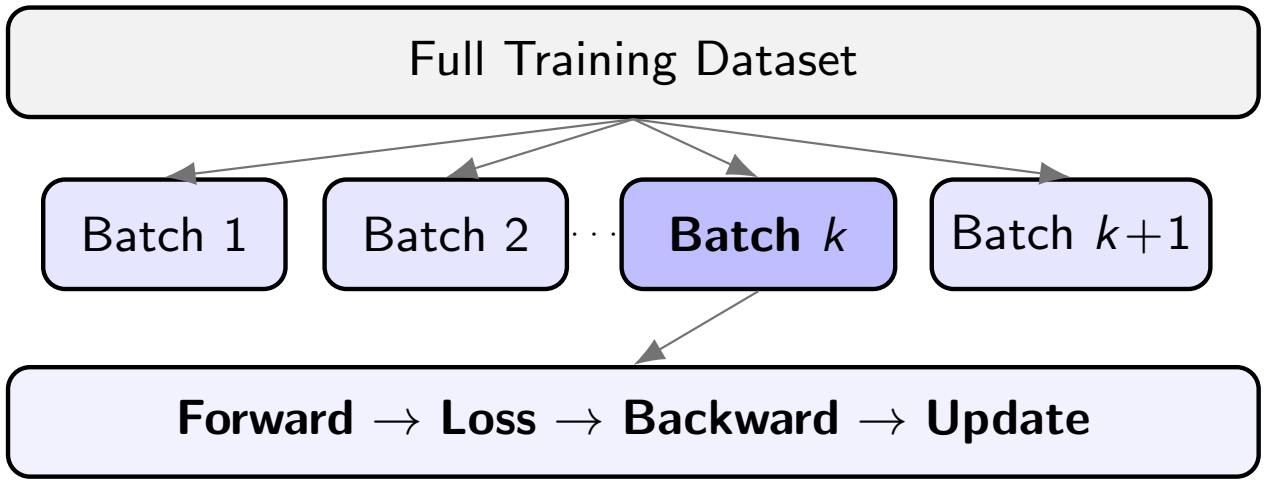
- α : learning rate (step size).
- We compute $\frac{\partial J}{\partial (\cdot)}$ for **every** parameter using backprop.

Q: Do we do this update using **all data at once**?

Training Step = One Update

We do not train using the full dataset at once.

Instead, we take a **small batch** (mini-dataset) and do one update:

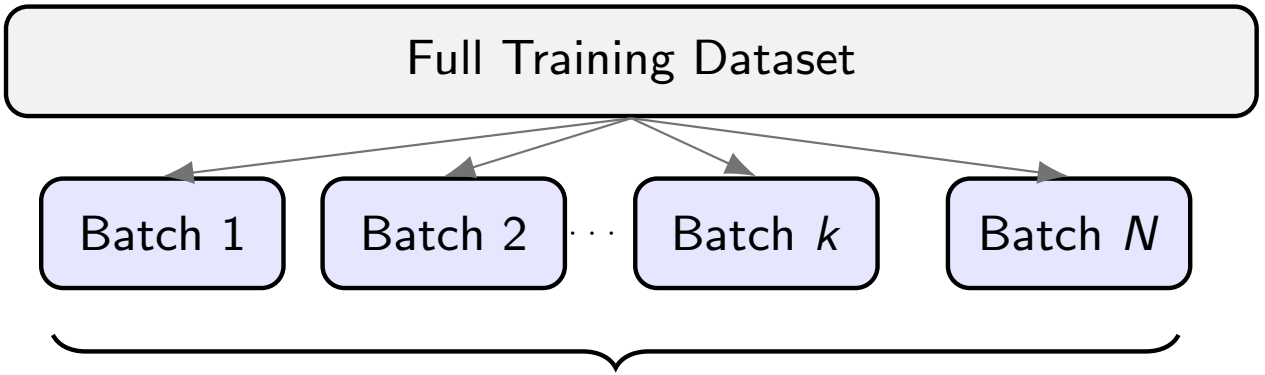


- This single update is called a **training step** (or **iteration**).
- **Batch size** = how many samples are used in one step.

Next: repeat this step for **all** *batches ⇒ one full* **epoch**.

Epoch = One Full Pass Over the Dataset

Dataset → split into **batches** → each batch makes **one step**.



- **Steps per epoch** $\approx \left\lceil \frac{\#\text{samples}}{\text{batch size}} \right\rceil$.
- Training usually runs for **many epochs**.

Next: before the next epoch, we usually **shuffle** *the dataset.*

Why Shuffle Between Epochs?

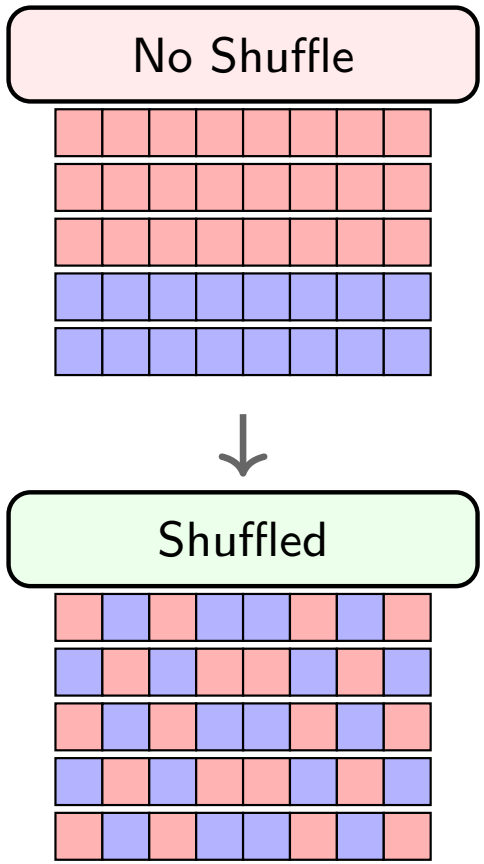
At the start of each epoch, we shuffle the dataset to randomize batch composition.

Without shuffling:

- Batches see same patterns repeatedly
- Similar samples grouped together
- Biased gradient estimates

With shuffling:

- Each batch is a better mix
- More stable learning
- Better generalization



The Three Pillars of Learning

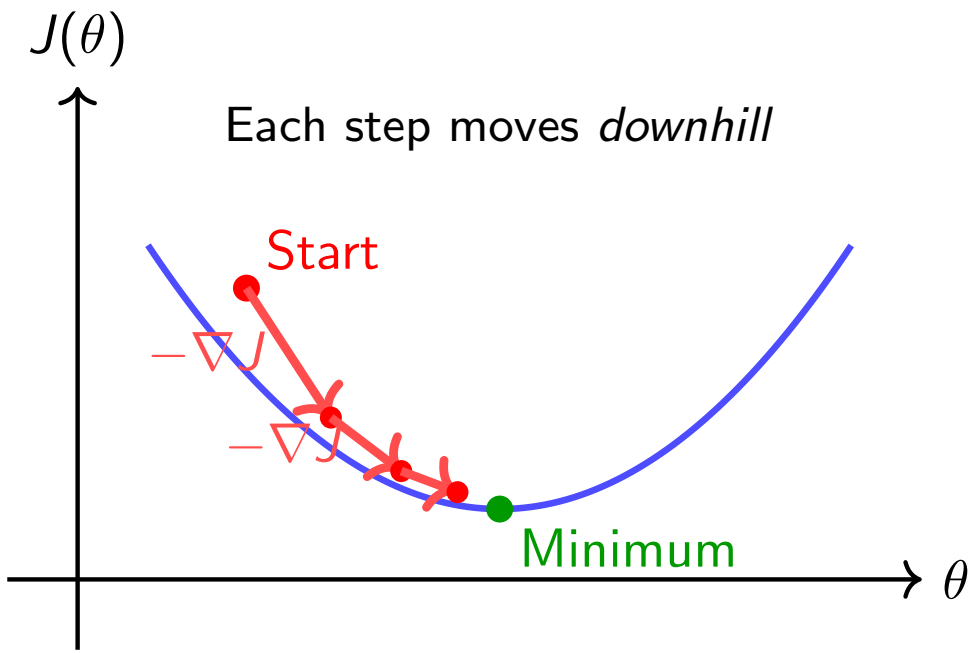
1. The Model (f_θ)	2. Loss Function (J)	3. Optimizer
Design Choice: Neural Networks (Architecture)	**Design Choice:** MSE / Cross-Entropy	**Design Choice:** SGD, Adam, RMSProp **Depends on:** Stability & Convergence Speed

We defined the Models (1) and the Losses (2).
Now, Let's discuss optimizers.

Gradient Descent: The Core Idea

Goal: Find parameters θ that minimize the loss $J(\theta)$.

Intuition: Follow the negative gradient (steepest descent direction).



The gradient $\nabla_\theta J(\theta)$ tells us which direction increases the loss most.
We go the **opposite direction** to decrease it.

Gradient Descent: Update Rule

Basic update equation:

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \nabla_\theta J(\theta)$$

- α is the **learning rate** (step size)
- $\nabla_\theta J(\theta)$ is the gradient of loss w.r.t. parameters
- Repeat until convergence (or for fixed number of steps)

Gradient Descent: Update Rule

Basic update equation:

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \nabla_\theta J(\theta)$$

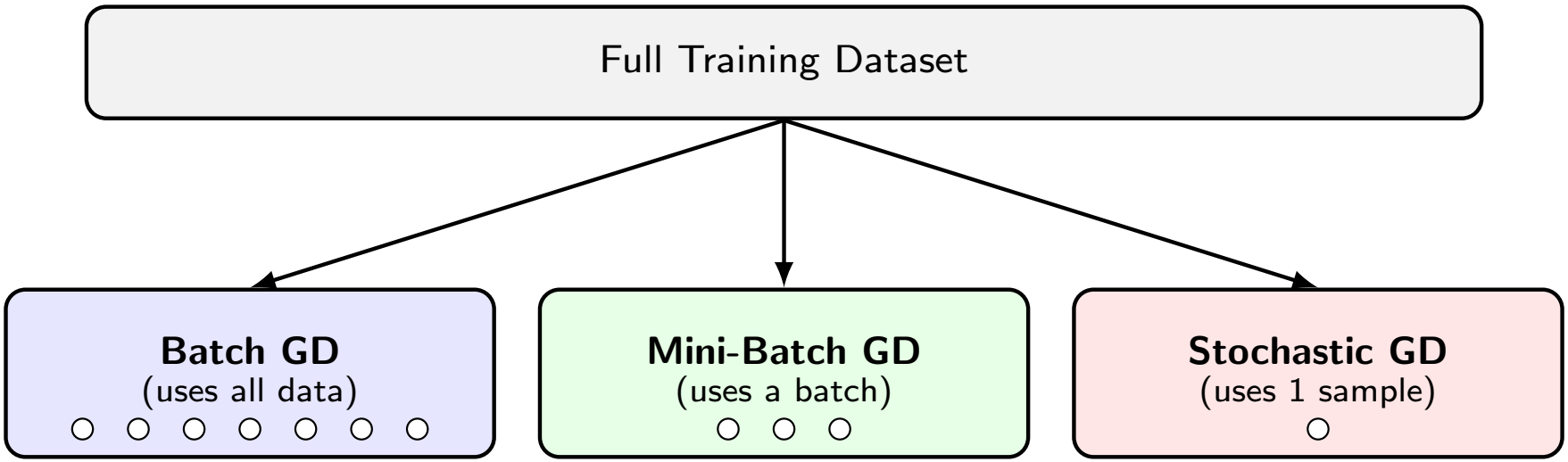
- α is the **learning rate** (step size)
- $\nabla_\theta J(\theta)$ is the gradient of loss w.r.t. parameters
- Repeat until convergence (or for fixed number of steps)

Three variants based on how much data we use:

1. **Batch Gradient Descent:** Use entire dataset
2. **Stochastic Gradient Descent (SGD):** Use one sample
3. **Mini-Batch Gradient Descent:** Use small batch of samples

Batch GD vs Mini-Batch GD vs Stochastic GD

How much data do we use per update (one step)?



Batch GD

- Stable updates
- Expensive per step
- Needs lots of RAM

Mini-Batch (Default)

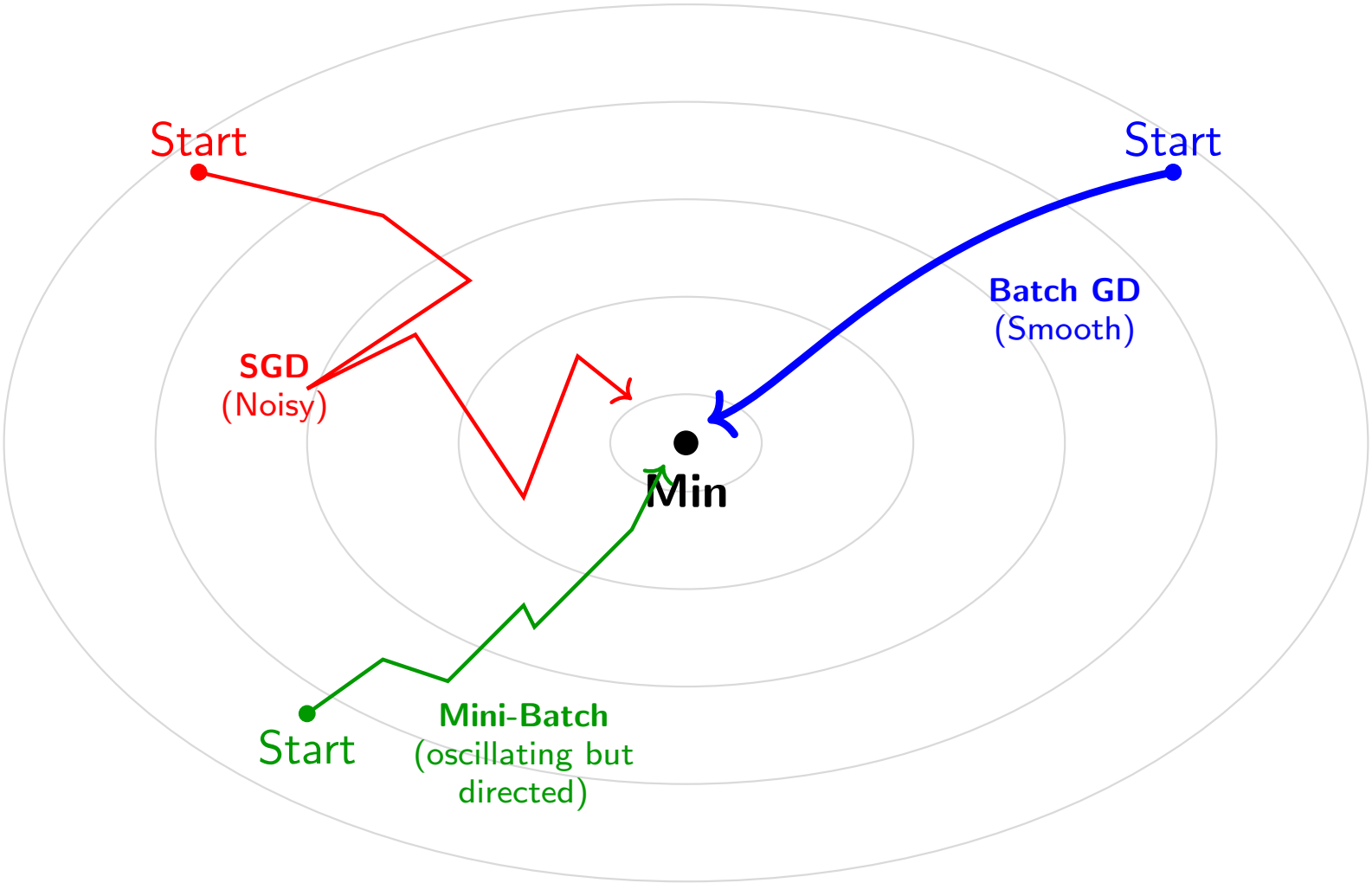
- Best of both worlds
- Stable & Fast
- GPU Efficient

Stochastic GD

- Very noisy updates
- Can escape local minima
- Slow (can't parallelize)

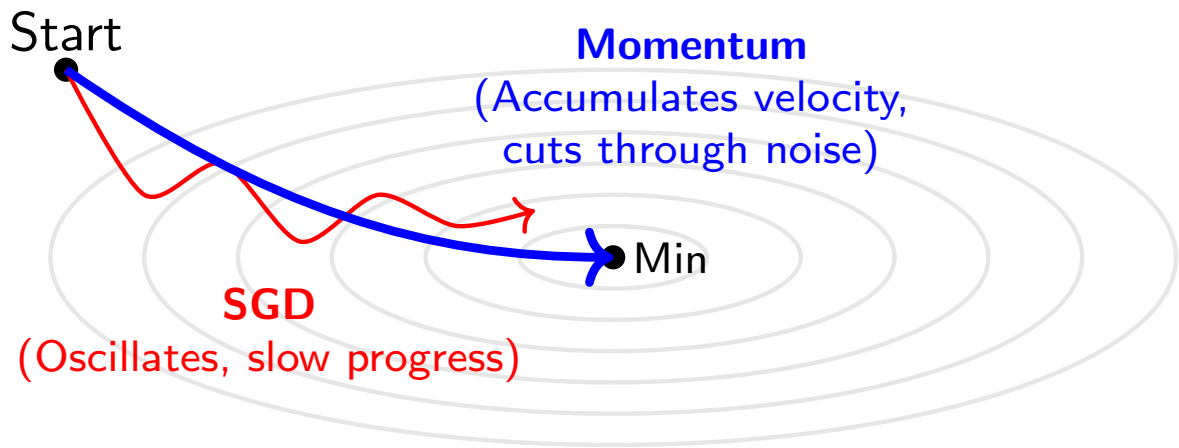
In practice: we almost always use Mini-Batch (whenever GD mentioned next, it refers to the mini-batch variant).

How do they navigate the Loss Surface?



Momentum: Adding Inertia

The Ravine Problem: Gradient Descent makes slow progress because it "bounces" off the steep walls.



The Solution

We add **Momentum**, which acts like a heavy ball rolling downhill, building velocity and ignoring small bumps.
It maintains a **running average** of past gradients (v_t), allowing oscillations in opposing directions to cancel out while consistent directions accumulate and accelerate.

Momentum: Update Rule

Keep a moving average of past gradients:

$$v_t = \underbrace{\beta v_{t-1}}_{\text{past gradients}} + \underbrace{(1-\beta)\nabla_\theta J(\theta)}_{\text{new gradient}}$$

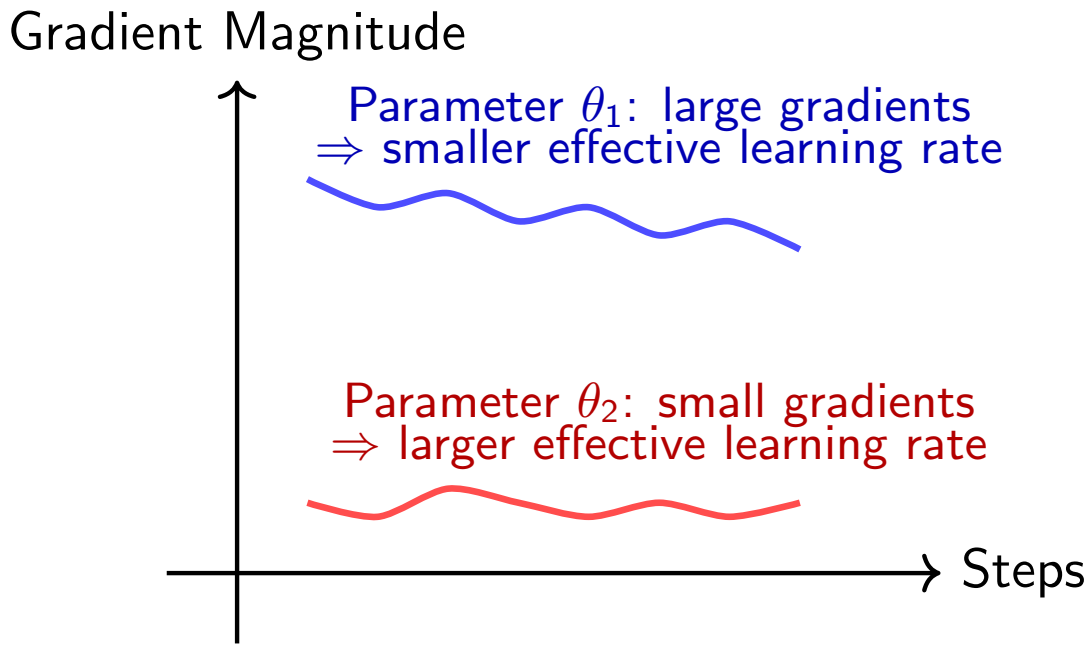
$$\theta_{t+1} = \theta_t - \alpha v_t$$

- v_t is the **velocity** (momentum term)
- β is the momentum coefficient (typically 0.9)
- α is the learning rate

AdaGrad: Adaptive Learning Rates

Problem: Same learning rate for all parameters may not be optimal.

Idea: Give each parameter its own adaptive learning rate based on its gradient history.



Parameters with large gradients get smaller steps (learning rates).
This prevents overshooting.

AdaGrad: Update Rule

Accumulate squared gradients for each parameter:

$$G_t = \underbrace{G_{t-1}}_{\text{past squared gradients}} + \underbrace{g_t^2}_{\text{new squared gradient}}$$

Update with parameter-wise learning rate:

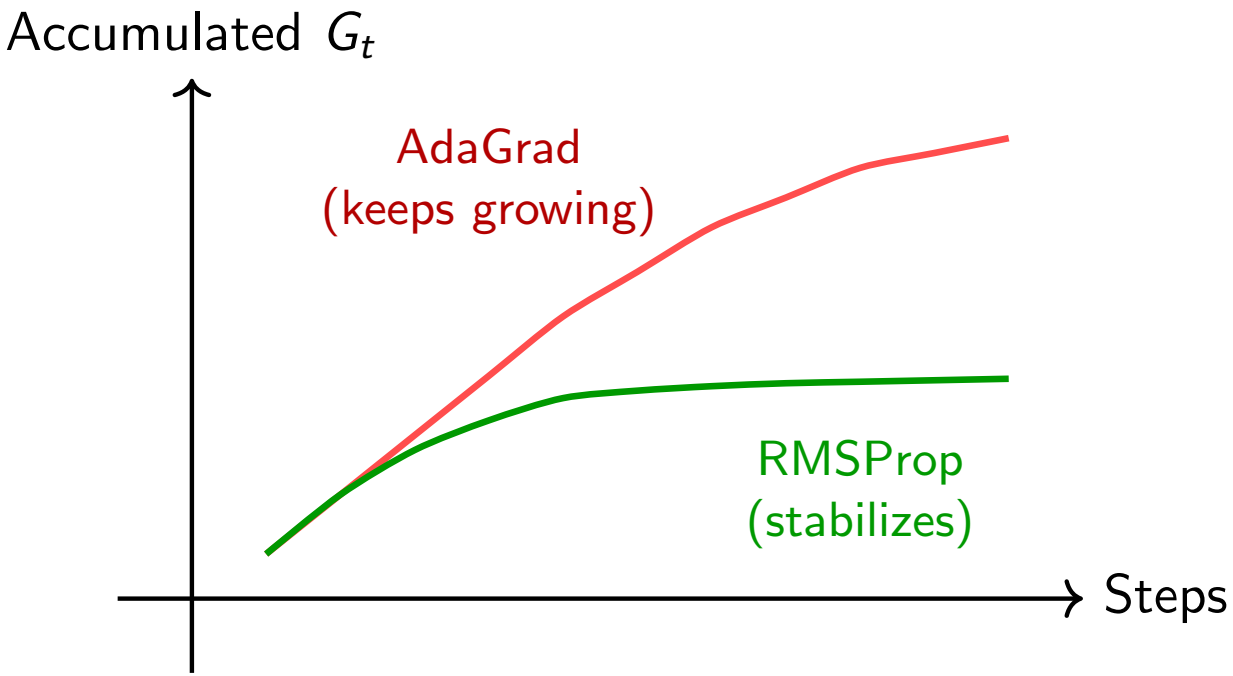
$$\theta_{t+1} = \theta_t - \underbrace{\frac{\eta}{\sqrt{G_t + \epsilon}}}_{\text{adaptive learning rate}} \underbrace{g_t}_{\text{gradient}}$$

- G_t accumulates all past squared gradients $\rightarrow$ larger G_t means smaller effective learning rate
- η is the base learning rate
- ϵ is a small constant for numerical stability ($\sim 10^{-8}$)
- Each parameter gets its own effective learning rate

RMSProp: Fixing AdaGrad

Problem with AdaGrad: Accumulates *all* past gradients → Learning rate can shrink too much over time (never forgets past gradients).

RMSProp solution: Use **exponential moving average** instead (forget old gradients!).



Effect: Learning rate adapts but doesn't vanish → training continues effectively!

Exponential moving average of squared gradients:

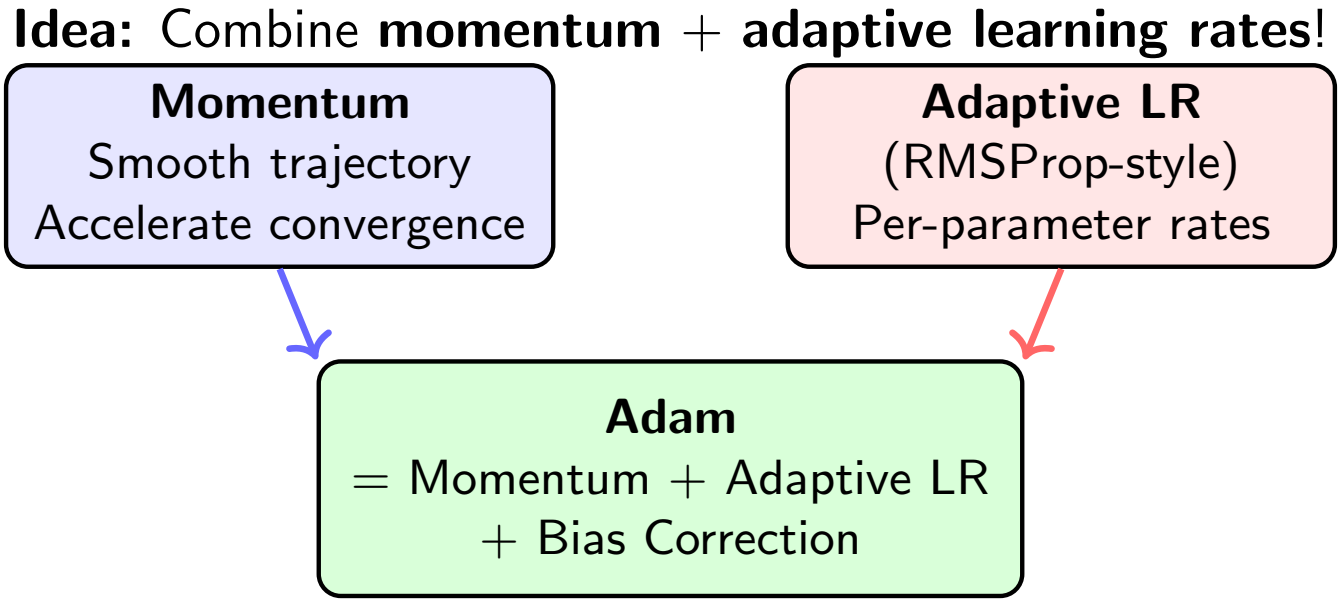
$$E[g^2]_t = \underbrace{\alpha E[g^2]_{t-1}}_{\text{decayed past gradients}} + \underbrace{(1-\alpha) g_t^2}_{\text{new squared gradient}}$$

Update rule:

$$\theta_{t+1} = \theta_t - \underbrace{\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}}_{\text{adaptive learning rate}} \underbrace{g_t}_{\text{gradient}}$$

- α is the decay rate (typically 0.9) (old gradients gradually fade away)
- Recent gradients have more influence than old ones

Adam: Best of Both Worlds



Bias Correction: At the start of training, moving averages are initialized to zero, causing them to be biased toward zero. Adam corrects this bias to get accurate estimates in early steps.

Result: Fast, stable, and works well across many problems (the most popular optimizer in deep learning!)

Adam: Update Rule

Maintain two moving averages:

$$m_t = \underbrace{\beta_1 m_{t-1}}_{\text{past momentum}} + \underbrace{(1-\beta_1) g_t}_{\text{new gradient}} \quad \text{(momentum)}$$

$$v_t = \underbrace{\beta_2 v_{t-1}}_{\text{past variance}} + \underbrace{(1-\beta_2) g_t^2}_{\text{new squared gradient}} \quad \text{(adaptive LR)}$$

Bias correction (fixes initial zero bias):

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

As $t \to \infty$, $\beta^t \to 0$, so correction factor $\to 1$ (no correction needed later)

Update:

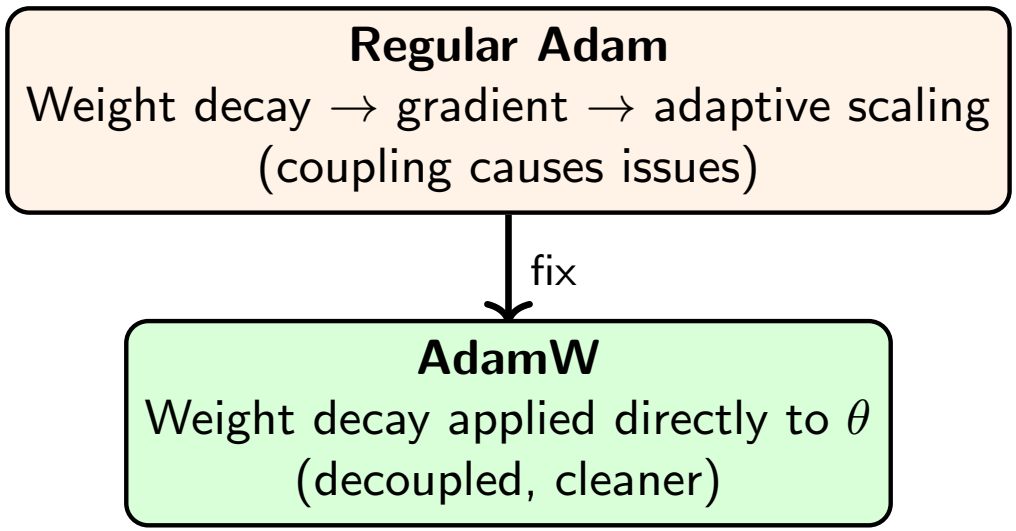
$$\theta_{t+1} = \theta_t - \underbrace{\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}}_{\text{adaptive step size}} \underbrace{\hat{m}_t}_{\text{momentum direction}}$$

- Typical: $\beta_1 = 0.9$, $\beta_2 = 0.999$

AdamW: Improved Regularization

Problem with Adam: Weight decay (L2 regularization) interacts poorly with adaptive learning rates.

Why? In Adam, weight decay is added to the gradient before the adaptive scaling. This means parameters with small gradients get disproportionately large weight decay, breaking the intended regularization.



Result: Better generalization, especially for large models (current best practice for training modern neural networks!)

AdamW: Update Rule

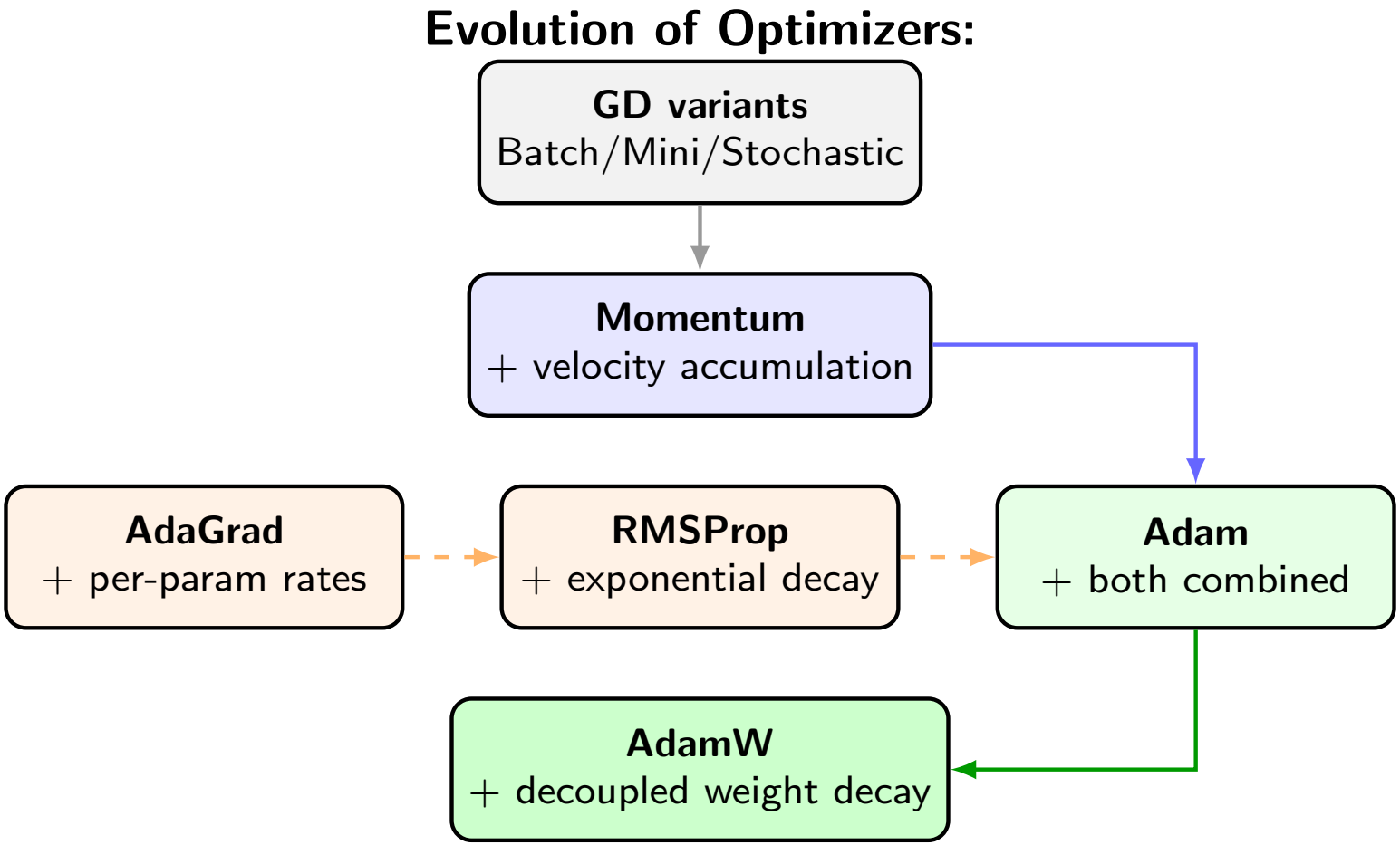
Same as Adam, plus decoupled weight decay:

$$\theta_{t+1} = \theta_t - \underbrace{\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t}_{\text{Adam update}} - \underbrace{\eta \lambda \theta_t}_{\text{weight decay (decoupled)}}$$

- First term: standard Adam update (gradient-based optimization)
- Second term: direct weight decay applied to parameters, independent of gradients
- λ is the weight decay coefficient (typically 0.01)

Why better? Cleaner separation of optimization and regularization.

Optimizer Summary



Rule of thumb: Start with **AdamW** (it's the most robust default choice!)

What We Choose When Building a Neural Network

Key Design Decisions:

1. Model Architecture

Structure:

- Number of layers
- Neurons per layer
- Network type (MLP/CNN/RNN)

Components:

- Activation functions (ReLU, sigmoid, tanh)
- Output layer shape & activation

2. Training Strategy

Learning:

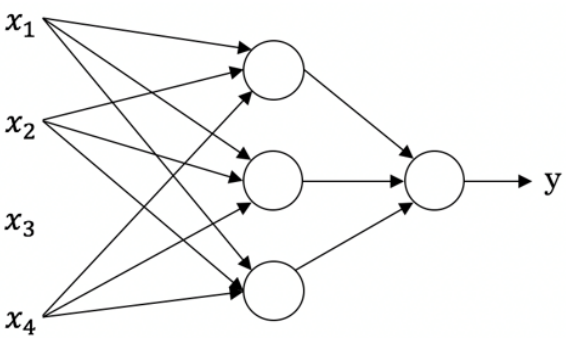
- Loss function (MSE, cross-entropy)
- Optimizer (SGD, Adam, AdamW)

Hyperparameters:

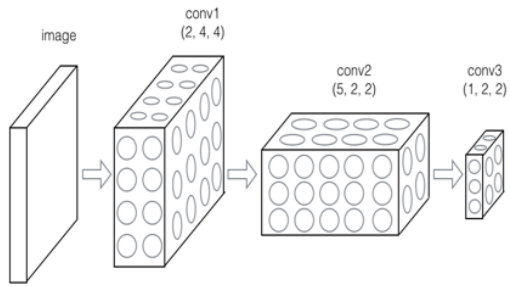
- Learning rate
- Batch size
- Number of epochs

Types of Neural Network Architectures

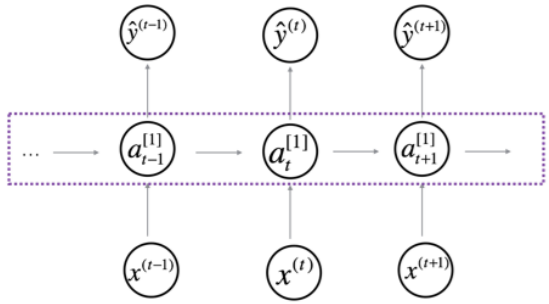
- **MLP** fully connected network for tabular data
- **CNN** convolutional neural network for images
- **RNN and variants** for sequences and time series



Standard NN



Convolutional NN



Recurrent NN

References

- Aurélien Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*
- Andrew Ng, *Deep Learning Specialization* (Coursera/DeepLearning.AI)

Slides contributed by Mohamed Eltayeb