

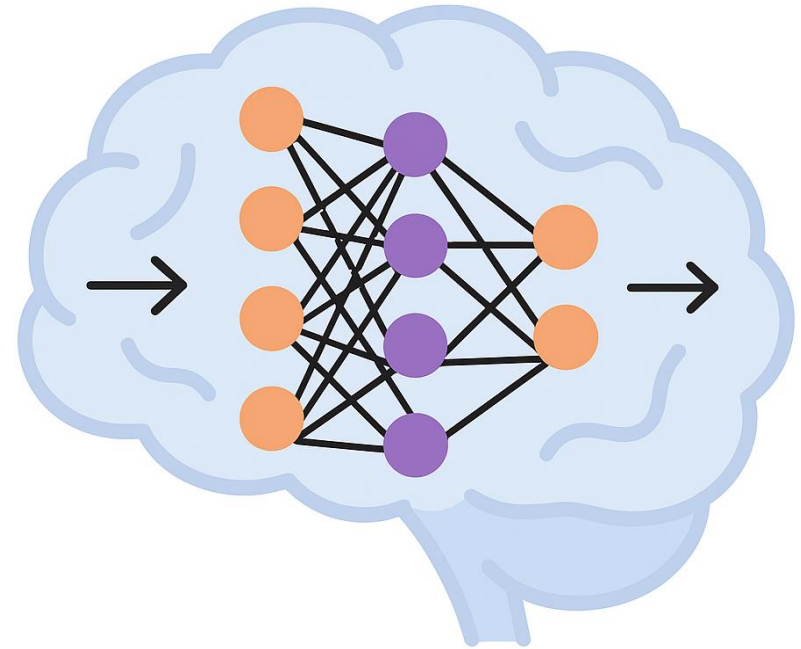
Training Neural Networks: Tips, Tricks, and Evaluation

Mohammadtaha Bagherifard

July 2025

Intro

- 🎯 What you'll learn today
- 🚫 What you don't need to know
- 🕒 Duration and format





What You'll Learn

- How to evaluate ML models beyond accuracy
- What makes training neural networks succeed or fail
- Practical tricks: learning rate, batch size, warm-up, etc.
- A peek into fine-tuning and efficient techniques.

What You Don't Need Today

- No deep math or backpropagation formulas
- No code implementation details
- No complex derivations



Duration and Format

- 2.5 hours of interactive lecture + 30 min Q&A
- Feel free to ask questions any time!

Agenda

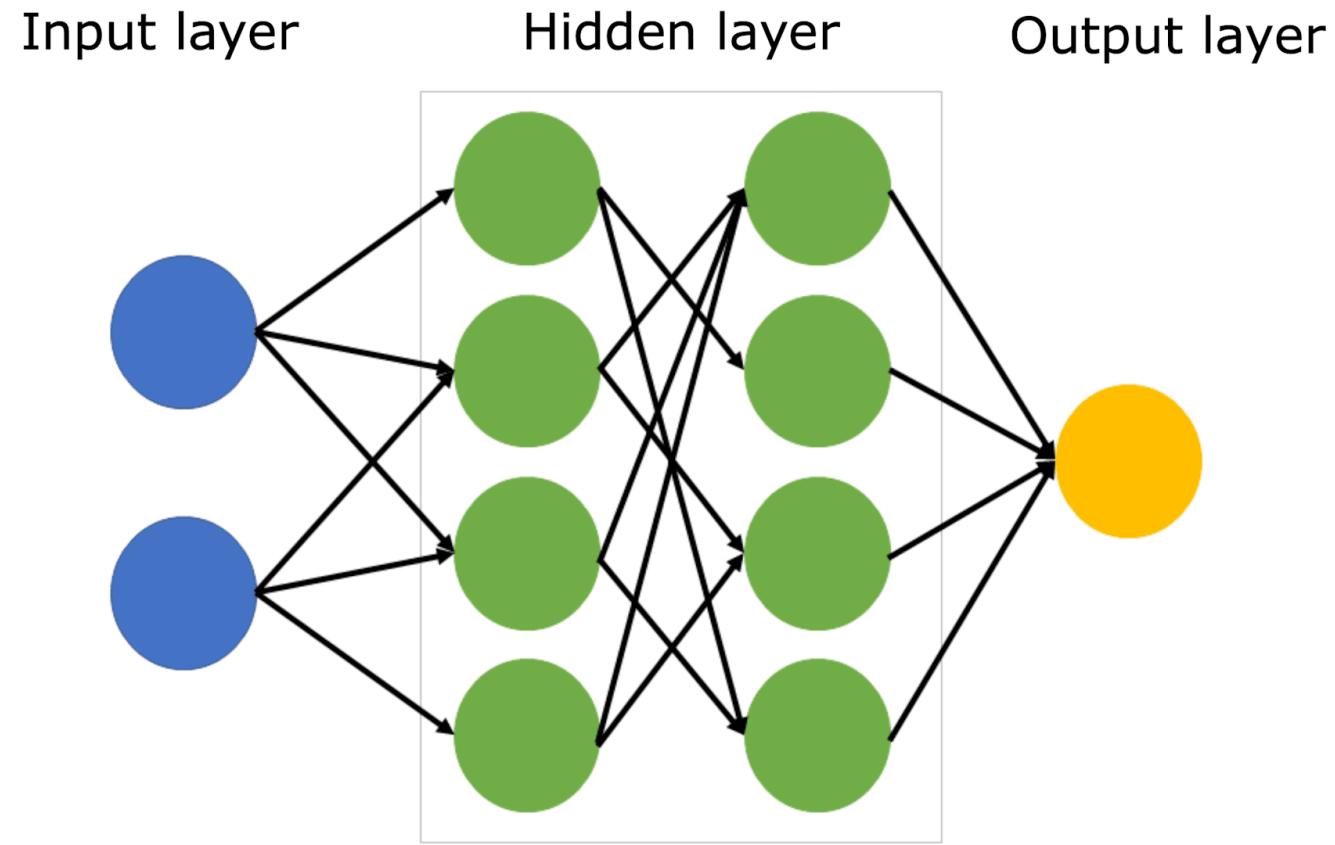
- 🧠 **Understanding Neural Networks**
What is an MLP? When & why we use it
- 📏 **Evaluating Model Performance**
Accuracy, Precision, Recall, F1, Confusion Matrix
- ⚙️ **Tuning the Model**
Layers, Neurons, Activations, Batch Norm, Residual Connection
- 🚀 **Training Tricks**
Batch size, Learning Rate, Initialization, Gradient Clipping, Weight Decay
- 🧪 **Advanced Notes**
Fine-tuning, Transfer learning, PEFT method (intro only)
- ❓ **Q&A + Open Discussion**
Your questions, curiosities, and deeper dives



What is an MLP?

- A **basic type of neural network** used for classification or regression
- Made up of **layers of “neurons”** connected to each other
- Has:
 - **Input layer** (what the model sees)
 - **Hidden layers** (where learning happens)
 - **Output layer** (final prediction)
- Learns by adjusting the connections between neurons during training
- Works best with **structured/tabular data** or **simple vision/NLP tasks**
- Learns features **automatically** by stacking layers and adding non-linearity

What is an MLP?



Artificial neural networks

MLP: Classification vs Regression

- MLPs can be used for **both classification and regression**
- Only the **output layer & loss function change**
- Differences:

Task	Output Layer	Output Example
Classification	One unit per class (with Softmax / Sigmoid)	“Dog” / “Not spam”
Regression	Single unit (no activation, or ReLU if needed)	3.27, 99.5, -12.4

- MLP learns by comparing predictions to the **true label or value**

Why Use MLPs? And When Not To?

Pros:

- Easy to build and understand
- Great for **structured/tabular data**
- Can solve basic **NLP / vision** tasks with enough data
- Could be used in more complex architectures like Transformers

Limitations:

- Doesn't scale well to high-dimensional input (e.g., images)
- Doesn't model **sequences or spatial structure**
- Doesn't assume **inductive bias**
- Has a fixed size input
- Sensitive to hyperparameters (LR, init, etc.)
- Fully connected → many parameters → **overfitting risk**
- **Not interpretable** out of the box

How Do We Evaluate ML Models?

- Evaluation depends on the **type of task**:
 - **Classification** → Accuracy, Precision, Recall, F1
 - **Regression** → MSE, MAE, R^2 (not covered here)
- Accuracy isn't always enough — we need to understand **how the model is wrong**
- Especially important in **imbalanced datasets** (e.g., 95% healthy, 5% sick)
- We'll focus on **classification metrics** — used in most ML competitions and practical systems



Understanding the Confusion Matrix

	Predicted: Positive	Predicted: Negative	
Actual: Positive	✓ TP (True Positive)	✗ FN (False Negative)	Recall
Actual: Negative	✗ FP (False Positive)	✓ TN (True Negative)	FPR
	Precision		

Understanding the Confusion Matrix

- **Accuracy** = $(TP + TN) / (TP + TN + FP + FN)$
- **Precision** = $TP / (TP + FP)$
 - "Of what we predicted positive, how many were actually correct?"
- **Recall** = $TP / (TP + FN)$
 - "Of all actual positives, how many did we catch?"
- **F1 Score** = $2 \times (Precision \times Recall) / (Precision + Recall)$
 - "Balance between Precision & Recall"

Understanding the Confusion Matrix

Examples

- Accuracy:



Balanced datasets, where both classes are equally important



Example: Image classification of animals (cat vs dog)

- Precision:



When false positives are costly



Example: Spam detection → we don't want to mark important emails as spam



Example: Cancer screening for expensive treatment → avoid false alarms

- Recall:



When false negatives are dangerous




Example: Medical diagnosis → missing a cancer case is worse than over-alerting

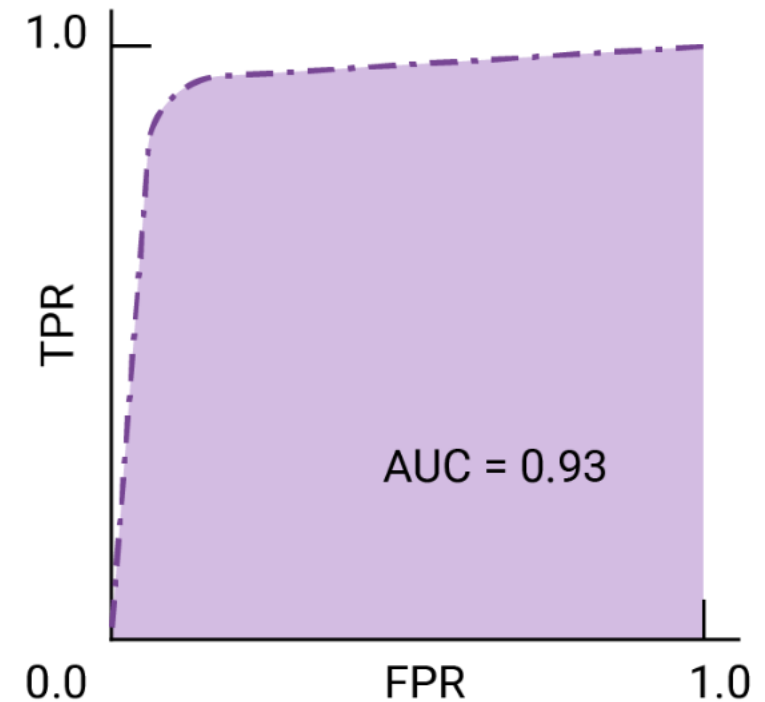
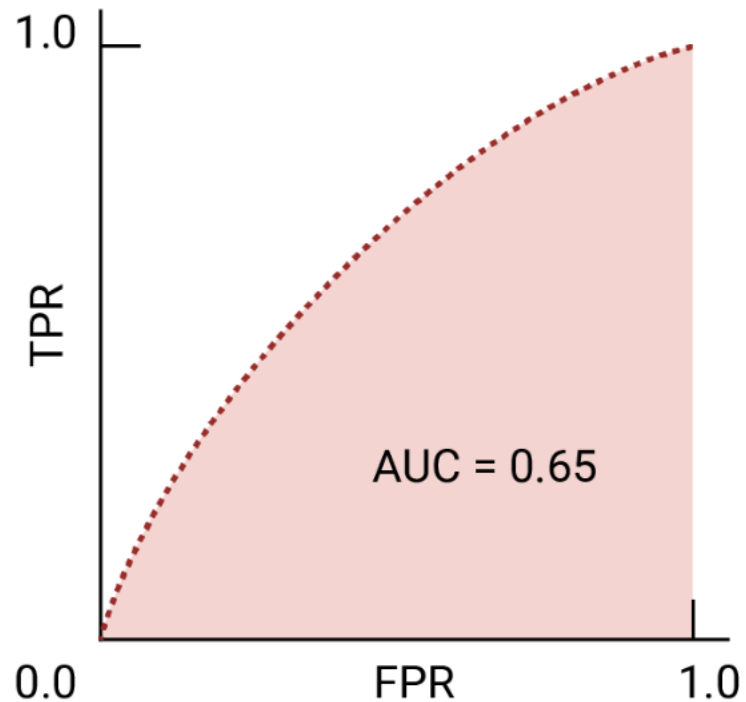


Example: Search engines → show as many relevant results as possible

ROC Metric


- **ROC** (Receiver-operating characteristic) Curve:
 - a visual representation of model performance across all thresholds
 - X-axis: False Positive Rate (FPR)
 - Y-axis: True Positive Rate (Recall)
- **AUC** = Area under ROC curve
 - Measures how well model ranks positives over negatives
- $AUC \approx 1 \rightarrow$ perfect classifier
- $AUC \approx 0.5 \rightarrow$ random guessing
-  Useful for comparing classifiers with **balanced datasets**.
 - FPR can look artificially low when negatives dominate

ROC Metric



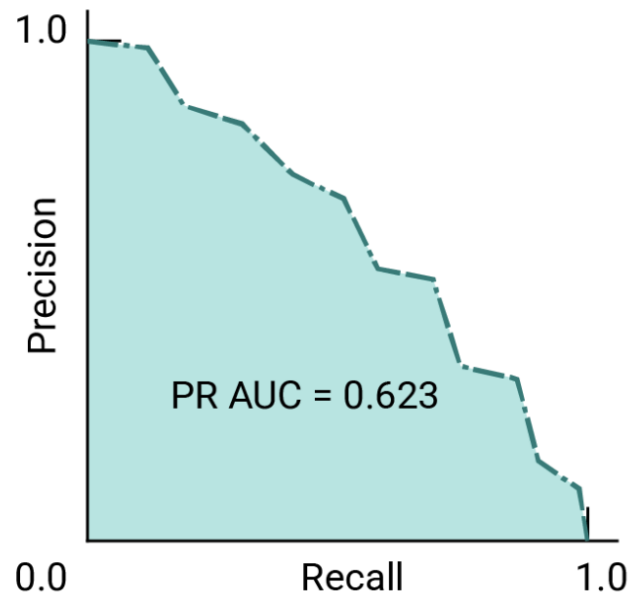
ROC and AUC of two hypothetical models. The curve on the right, with a greater AUC, represents the better of the two models.

Imbalanced Datasets





- In **imbalanced datasets**, one class dominates
 - (e.g., 99% healthy, 1% sick)
-  **Accuracy is misleading**
 - A model predicting “always healthy” gets 99% accuracy → but it's useless
- We care more about **how well the model separates the classes**
- **Precision-Recall curve** is the correct metric to compare the models.

Precision-Recall Curve

- **Precision-Recall Curve** focuses only on the **positive class**:
 - Precision (how many predicted positives were correct)
 - Recall (how many actual positives we caught)



How to Plot ROC and PR Curves


-  The model outputs **probabilities** (e.g., 0.91 cat, 0.23 cat, etc.)
-  Sweep the **classification threshold** from $0 \rightarrow 1$ (e.g., classify as positive if prob > 0.9 , then 0.8, 0.7, etc.)
-  For each threshold, calculate:
 - **Confusion matrix** \rightarrow then compute:
 - $\text{TPR (Recall)} = \text{TP} / (\text{TP} + \text{FN})$
 - $\text{FPR} = \text{FP} / (\text{FP} + \text{TN})$
 - $\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$
-  Plot the curves:
 - **ROC**: TPR vs FPR
 - **PR**: Precision vs Recall

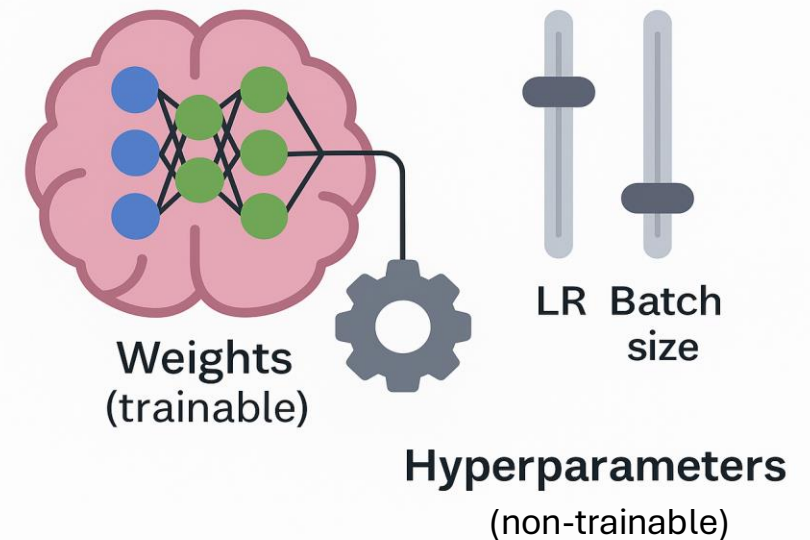
Training Neural Networks: Parameters vs. Hyperparameters

1. Trainable Parameters

- These are the **weights and biases** of the network
- They are **learned automatically** via:
 - Backpropagation
 - Gradient descent (or its variants)

2. Hyperparameters

- These are **manually set by the practitioner**
- They define **how the model is built** and **how it trains**
- They are **not learned** during training
-  **Examples:**
 - **Model architecture:** Number of layers, Neurons per layer, Activation functions (ReLU, sigmoid, etc.)
 - **Training hyperparameters:** Learning rate, Batch size, Number of epochs, Optimizer (SGD, Adam, etc.)
 - **Regularization:** Dropout rate, L2 penalty (weight decay), Early stopping



Model Architecture:

Layers and Neurons

- **Number of Layers (Depth)**

- More layers = can learn more complex patterns
- But also **harder to train**, more risk of overfitting



Shallow → faster, simpler



Deep → more expressive, but needs care

- **Neurons per Layer (Width)**

- More neurons = more capacity
- But also increases parameters → overfitting risk ⚠
- Should be **balanced** with dataset size

Model Architecture:

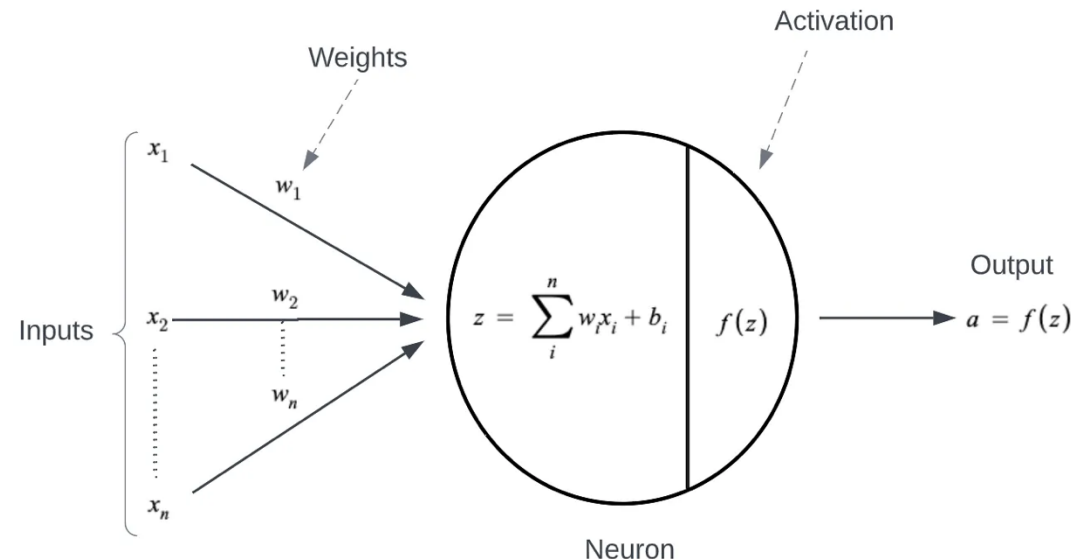
The Universal Approximation Theorem (UAT)

- A neural network with **just one hidden layer** (and enough neurons) can **approximate any continuous function**, given enough data.
- 🙄 Then why don't we use one huge hidden layer?
 - ❌ **Inefficient:** Shallow networks may need **millions of neurons** to model complex functions → slow, overfit easily.
 - ❌ **Poor Generalization:** Wide nets often **memorize**. Deep nets **learn patterns** (e.g., edge → shape → object).
 - ❌ **Harder to Optimize:** Flat loss landscapes make training unstable. Deep nets benefit from modern tricks (e.g., batch norm, skip connections).
 - ❌ **Depth = Expressiveness:** Some functions require **exponential width** in shallow nets but only **modest depth**.

Model Architecture:

Activation Function

- Without activations, neural nets are just **linear equations**
 - Stacked linear layers = still linear 🤪
 - Can't model complex relationships
- Activations introduce **non-linearity** so the network can learn curved decision boundaries



Model Architecture:

Activation Function

The **choice of activation** affects training behavior




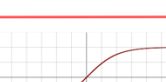






Training Effects:

- **Sigmoid / tanh** can cause **vanishing gradients**
 - Gradients get too small → slow or stuck training
Especially bad in deep networks
- **ReLU** avoids vanishing gradients
 - Doesn't squeeze the output
 - Gradient is constant (1) when active
Also sparse — many neurons output 0 → faster learning
- Some variants (e.g., Leaky ReLU, GELU) help avoid **dead neurons**

Model Architecture:

Activation Function

ACTIVATION FUNCTION	PLOT	EQUATION	DERIVATIVE	RANGE
Linear		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary Step		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$
Sigmoid		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
Hyperbolic Tangent(tanh)		$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
Rectified Linear Unit(ReLU)		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$
Softplus		$f(x) = \ln(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$	$(0, 1)$
Leaky ReLU		$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-1, 1)$
Exponential Linear Unit(ELU)		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	$[0, \infty)$

Model Architecture:

Residual Connection



- As networks get deeper, training becomes **harder**
- Vanishing gradients → layers stop learning
 - **Residual Connections** (from ResNet) solve this:
 - Instead of learning $F(x)$, the model learns a **residual**: $F(x) + x$
 - These "skip connections" let the gradient **flow directly** to earlier layers
 - Makes it possible to train networks with **dozens or hundreds of layers**

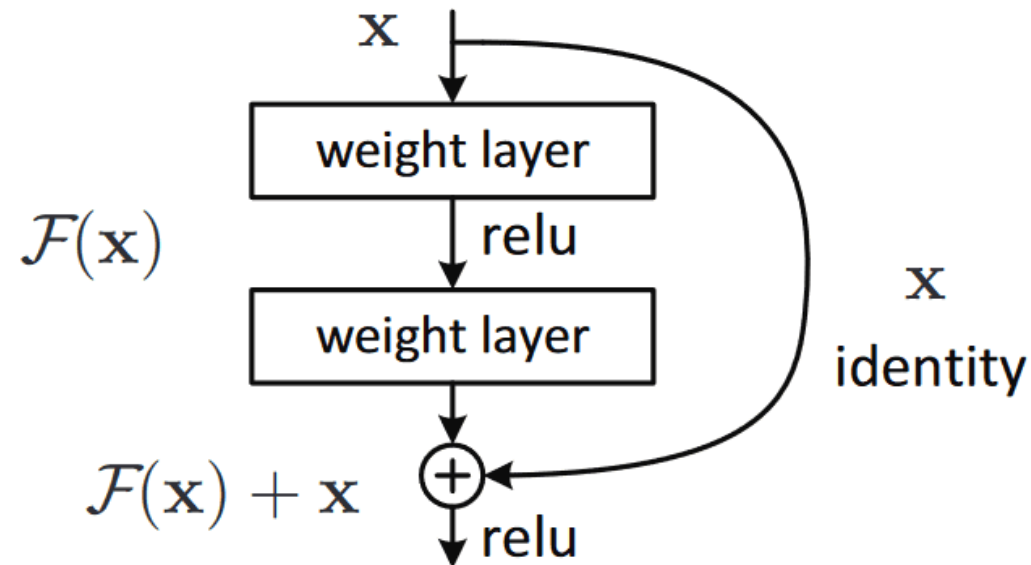
Model Architecture:

Residual Connection








Benefits

-  Solves **vanishing gradient** problem
-  Enables **very deep architectures**
-  Helps with **faster convergence** and **better generalization**

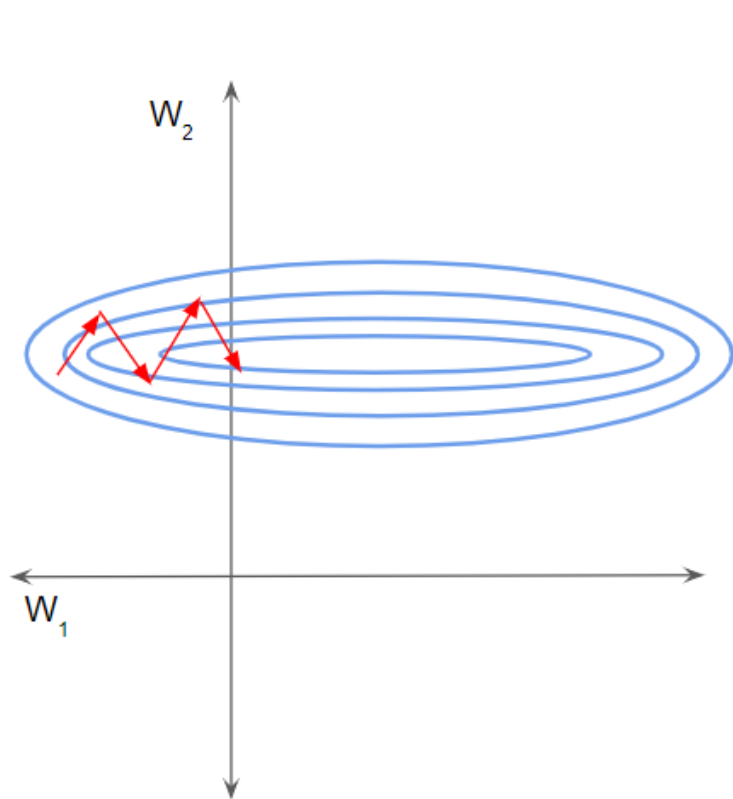


Model Architecture:

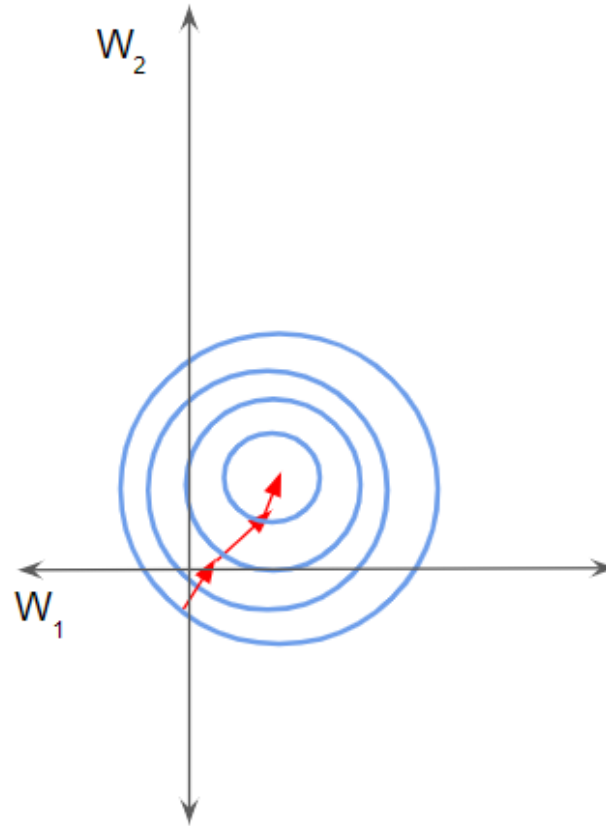
Batch Normalization

- During training, **distributions of layer inputs keep shifting ↗↘**
 - Makes training unstable and slow
 - Known as **internal covariate shift**
- **Batch Normalization (BN)** solves this by:
 - Normalizing activations in each layer
 - Rescaling & shifting them (learnable)
-  **Benefits of Batch Normalization:**
 -  **Faster convergence**
 -  **Smoother gradients**, more stable training
 -  **Less sensitive** to weight initialization & learning rate
 -  Acts like **regularization** → may reduce need for dropout

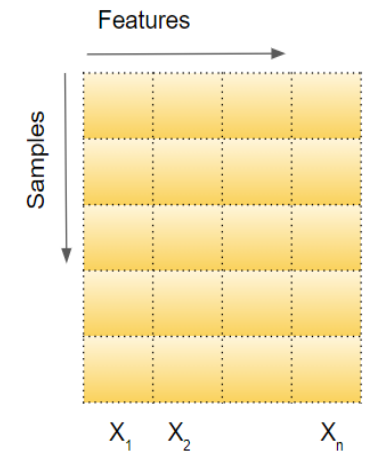
Model Architecture: Batch Normalization



Loss landscape before normalisation

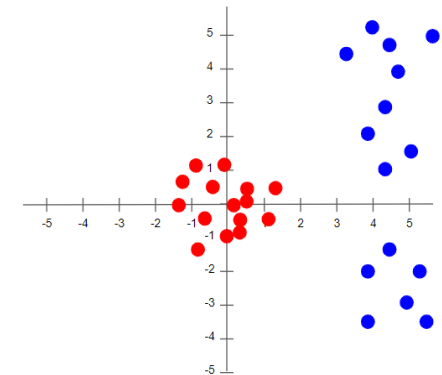


Loss landscape after normalisation



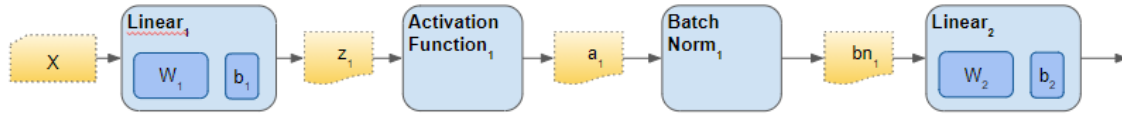
$$X_i = \frac{X_i - \text{Mean}_i}{\text{StdDev}_i}$$

Normalisation of features

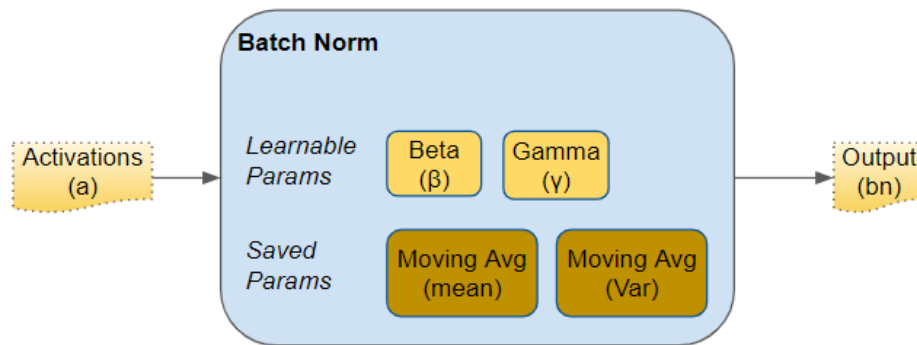


Effect of normalisation

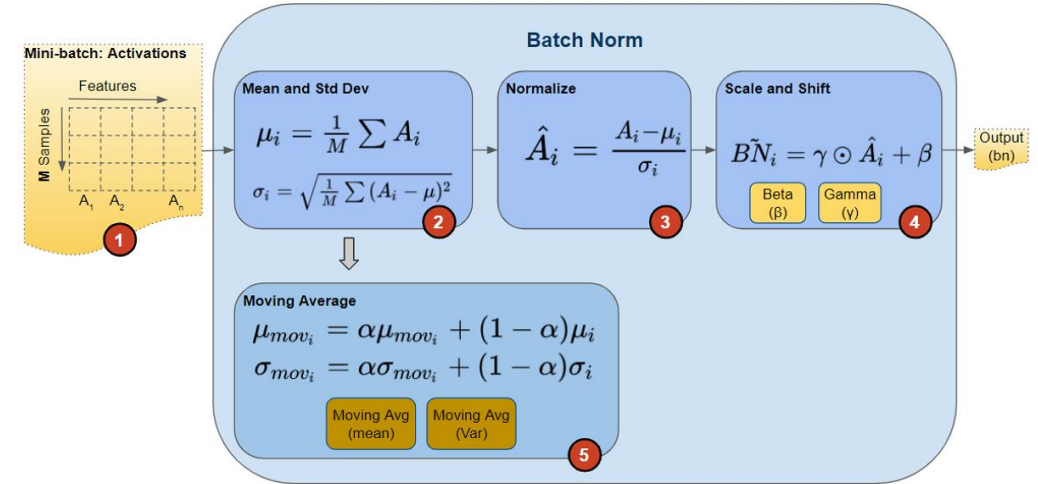
Model Architecture: Batch Normalization



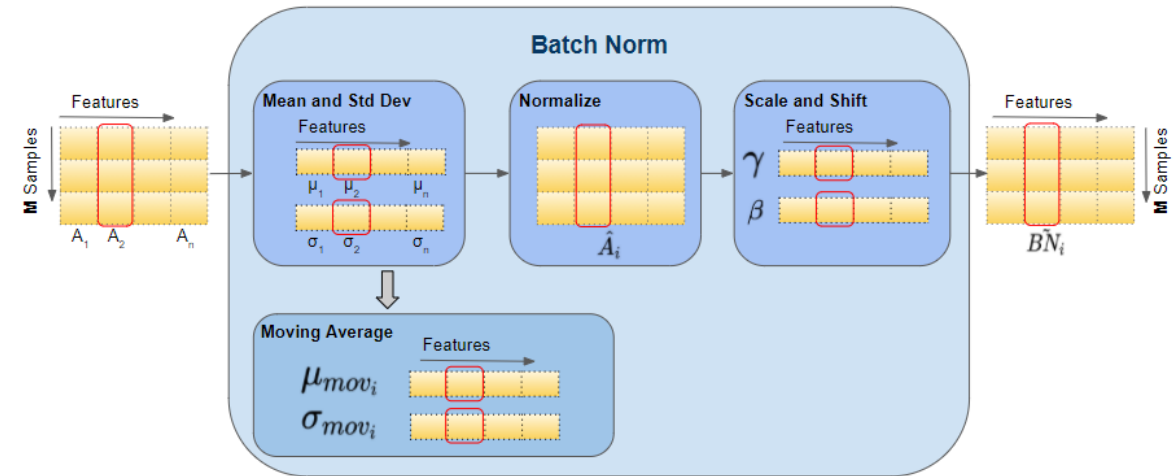
Batch norm layer's place in the network



Batch norm layer's parameters










Batch norm during training

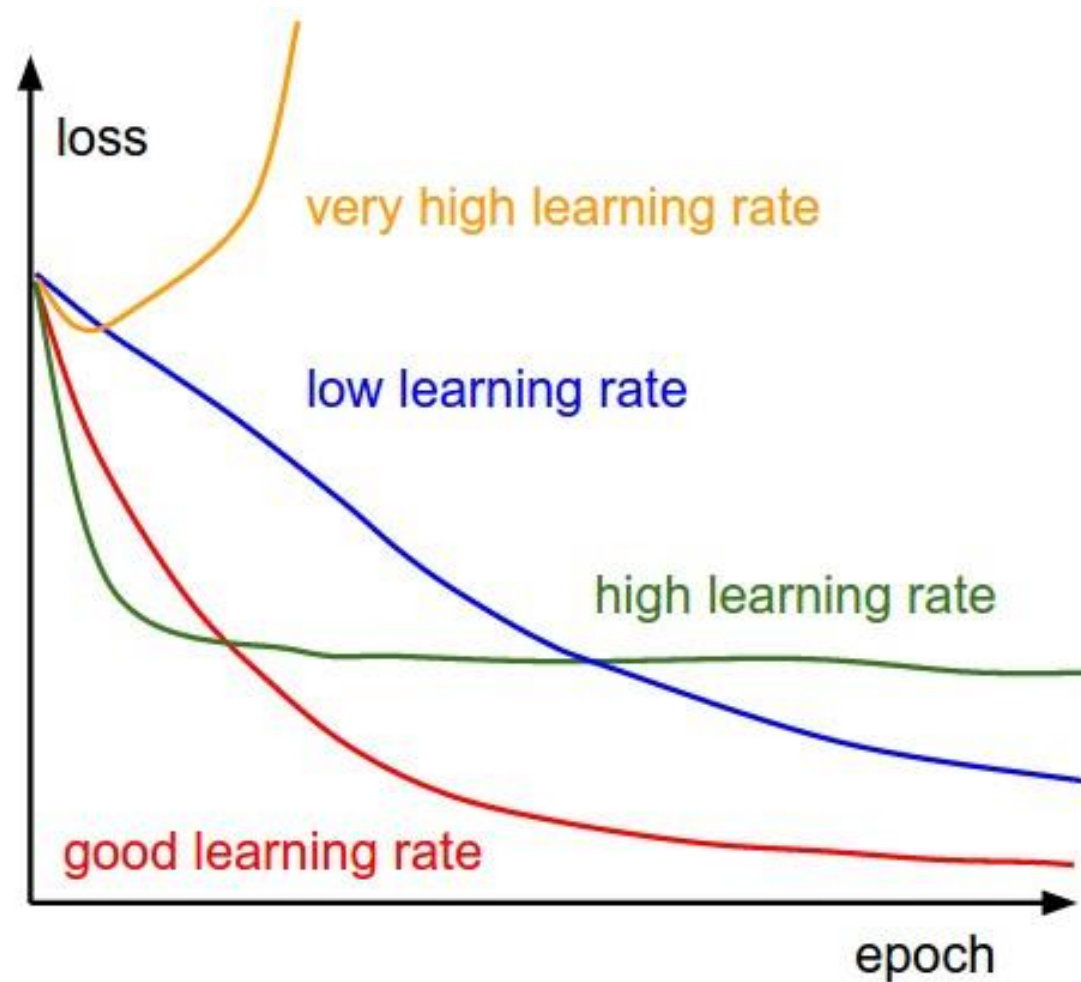


Batch norm during testing

Training Tricks: Learning Rate

- Learning rate (LR) controls the **size of each update** to the model's weights:
 - **Small LR:**
 -  Slow learning, Getting stuck on local minima
 -  Good for **fine-tuning / exploitation** (refining known good areas)
 - **Large LR:**
 -  May overshoot or diverge, Loss / gradient explosion,
 -  Good for **exploration** (searching broadly across the loss landscape)
-  **General Notes**
 -  You can **change LR during training** using **schedulers** (e.g., cosine decay, step decay, warm-up)
 -  Some optimizers (like **Adam**) adjust effective LR **per parameter**



Training Tricks: Learning Rate



Effect of different learning rates during training

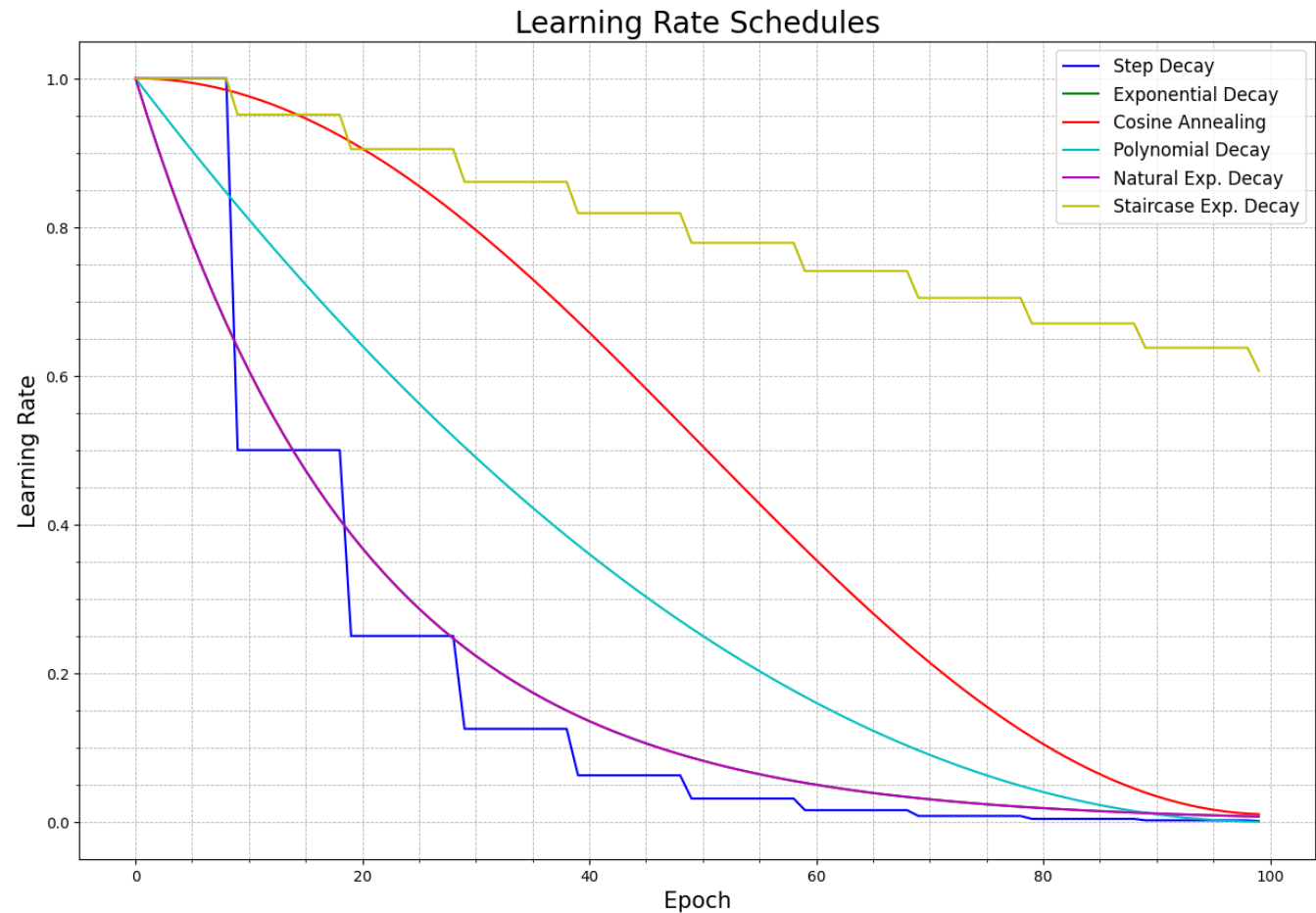
Training Tricks: Learning Rate Scheduling

Scheduling governs the pace as you train:


-  Using a **constant learning rate** may not be ideal for the full training process
-  **Schedulers** adjust the LR dynamically to improve convergence














Strategy	Idea 💡	When to Use
Step Decay	Reduce LR every N epochs	Classic; useful for staged training
Exponential / Linear Decay	LR shrinks gradually over time	Smooth convergence
Cosine Annealing	LR follows cosine curve to 0	Used in modern LLM training / fine-tuning
Warm-up	Start with small LR, then increase	Helps stabilize early training
Decay on Plateau	Reduce the LR when improvement stops	When validation loss flattens / model stalls

Training Tricks: Learning Rate Scheduling



Training Tricks: Batch Size

- **Batch size** = number of training examples used to compute one update step
- Affects: **speed, stability, generalization, and memory usage**
-  Trade-offs:

Small Batch	Large Batch
 Better generalization	 Prone to overfit
 Noisy gradients →  more exploration	 Smoother gradients →  more exploitation
 Slower per epoch	 Faster training (more parallelism)
 Fits on less memory	 Needs more memory
  Often results to flat minima (robust)	 Often results to sharp minima (sensitive)

Training Tricks: Gradient Accumulation

- Sometimes your GPU can't handle large batch sizes due to memory limits (e.g., on big models like Transformers)
- **Gradient Accumulation** solves this by:
 - Splitting a large batch into **smaller mini-batches**
 - Computing gradients for each mini-batch
 - **Accumulating** (summing) the gradients
 - Updating weights only after **N steps**






Example:

- You want a **batch size of 128**, but can only fit 32 in memory
 - Accumulate gradients for **4 forward-backward passes**
 - Then do **1 optimizer step**

Training Tricks: Gradient Accumulation





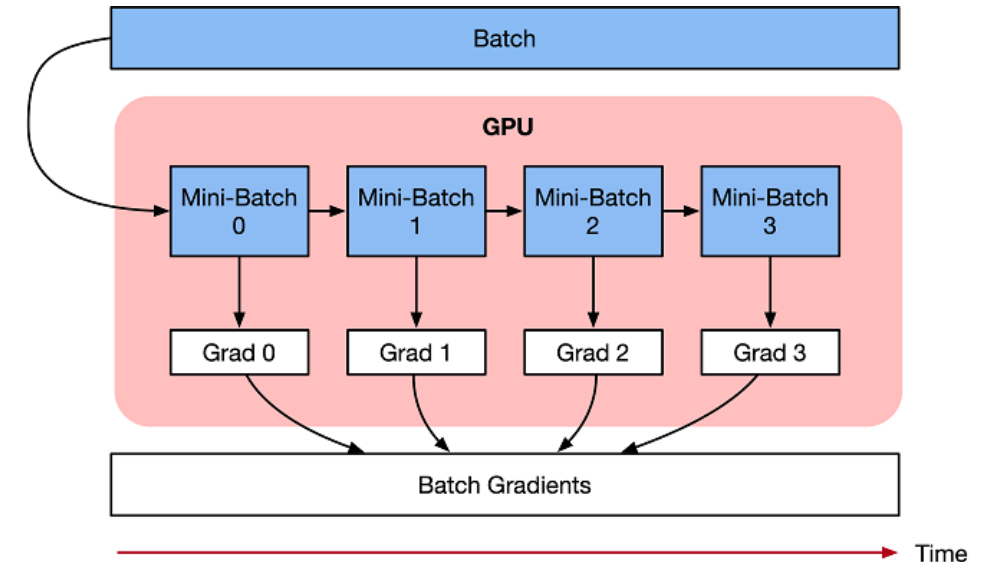
Benefits:

-  Simulates **large batch effects**
-  Works on **limited hardware**
-  Helps with **more stable training**



Trade-offs:

-  Slower per step (more forward/backward passes)
-  Must handle optimizer steps carefully



Training Tricks: Dropout

- A technique where we randomly **turn off** some neurons during training.






How it works:

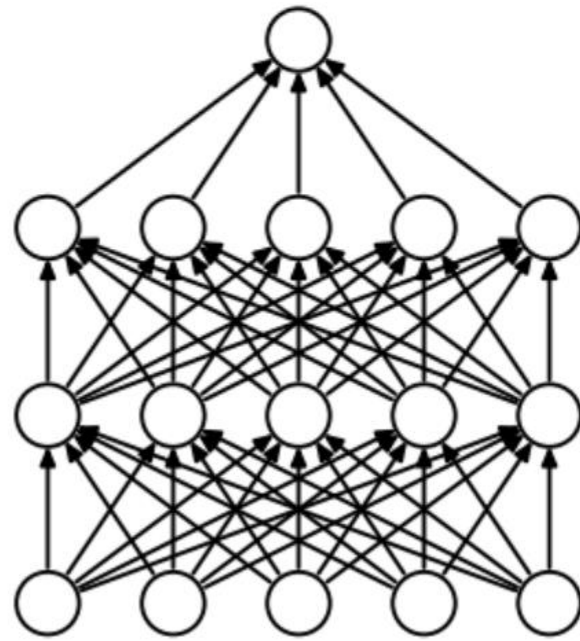
- For each training step, randomly "drop" (set to 0) a fraction of neurons
- Forces the network to **not rely too much** on any single path
- During inference, **all neurons are used**.



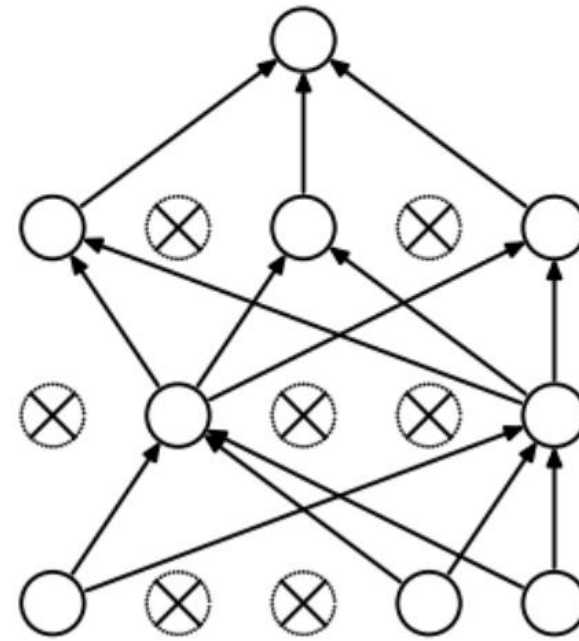
Why use it?

-  Prevents **co-adaptation** of neurons
-  Acts as **ensemble learning** (many thinned networks)
-  Reduces **overfitting**, especially on small datasets

Training Tricks: Dropout



(a) Standard Neural Net



(b) After applying dropout.

Training Tricks: Early Stopping

❏ A strategy to **stop training before overfitting starts** — based on how the model performs on a **validation set** (not the training set)



How it works:

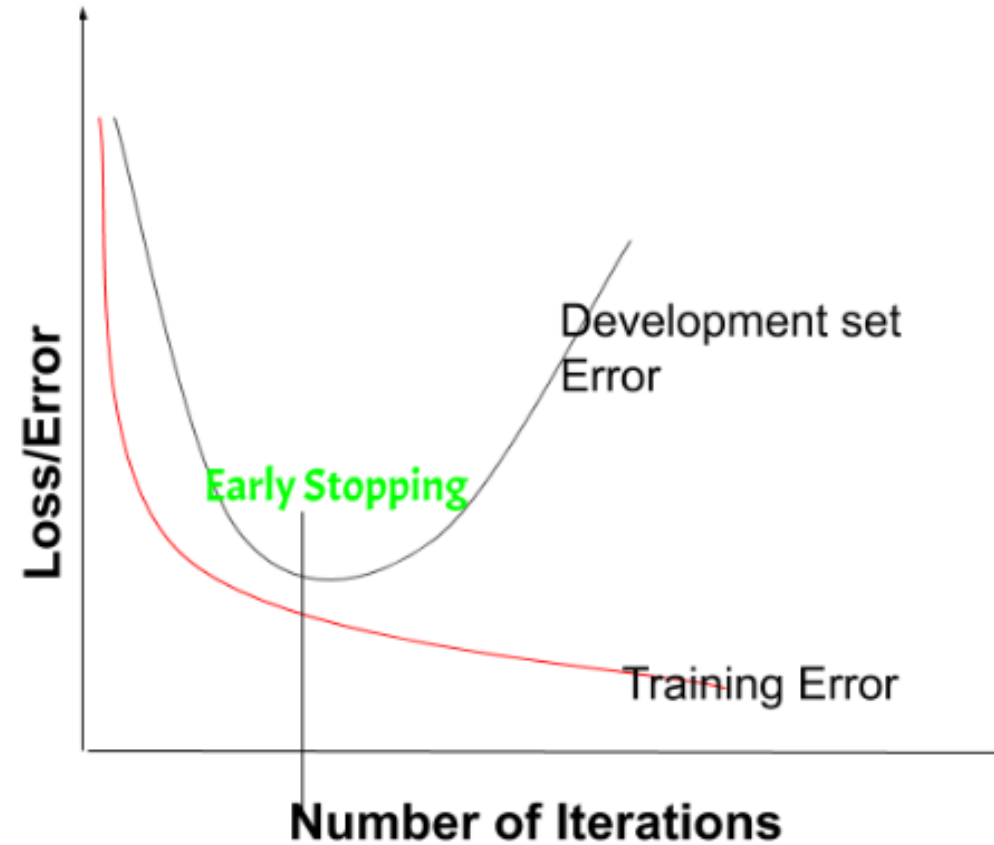
- Train the model as usual
- After each epoch (or few steps), evaluate on the **validation set**
- If validation performance **stops improving** for a number of steps (called *patience*) → **stop training**








Benefits:

- Prevents **overfitting**
- Saves **training time**
- Works well when you don't know the best number of epochs






Training Tricks: Early Stopping



Training Tricks: Gradient Explosion

- Sometimes gradients become **extremely large** during backpropagation
 - Especially in **very deep** or **recurrent** networks
 -  We call this phenomenon: “**Gradient Explosion**”
- This causes:
 -  Unstable updates
 -  Weights go to infinity / NaNs
 -   Loss diverges instantly

Training Tricks: Gradient Explosion

-  **Why does it happen?**
 - Gradients **compound multiplicatively** across layers
 - If each layer amplifies even a little, it **explodes exponentially**
 - Mainly happens due to bad weight initialization
-  **Symptoms:**
 - Training loss becomes **NaN** or **explodes**
 - Model **never converges**
- Two ways to address this:
 -  **Weight Initialization** → before training
 -  **Gradient Clipping** → during training
 -  **Weight Decay (L2-Regularization)** → during training

Training Tricks: Weight Initialization

- Neural networks start training with **random weights**
- But not all random is equal — poor initialization can cause:
 - 🔥 **Exploding gradients** (weights get huge)
 - ❄️ **Vanishing gradients** (weights go to zero)
 - 🐢 Slow or unstable learning
- 😬 **Why does it matter?**
 - Gradients flow **backward** through the network
 - If weights are too large/small:
 - Repeated multiplications cause **gradient instability**
 - Especially in deep networks




Training Tricks: Weight Initialization

- Popular Initialization Methods:

Method	Use Case
Xavier/Glorot Initialization	For Tanh or Sigmoid activations
He Initialization	For ReLU / LeakyReLU activations
Orthogonal Initialization	Sometimes used in RNNs and CNNs

- They all aim to keep the **variance of activations and gradients stable** across layers




Training Tricks: Xavier Initialization

-  **Designed for:** Sigmoid, Tanh
-  **Goals:**
 - Preserving activation variance during the forward pass
 - Preserving gradient variance during the backward pass
-  **Formula:**

$$W_{ij} \sim U \left[-\frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}} \right]$$

-  **Use when:** activations saturate easily (Tanh, Sigmoid)





Training Tricks: He Initialization

-  **Designed for:** ReLU, Leaky ReLU
-  **Goals:**
 - Maintain **activation variance** (ReLU deactivates roughly 50% of neurons)
 - Prevent **vanishing gradients** in deep ReLU nets
-  **Formula** (for normal distribution):

$$W \sim \mathcal{N}(0, \frac{2}{N})$$

-  **Use when:** you're using **ReLU-based activations**

Training Tricks: Orthogonal Initialization

-  **Designed for:** Stability in deep or recurrent networks
-  **Goals:**
 - Maintain **direction of gradient** (preserve norm)
 - Avoid **exploding/vanishing** gradients
-  **Key idea:**
 - Initialize weights as an **orthogonal matrix**
 - This means the derivative chain is better conditioned
 - Gradients neither explode nor vanish rapidly.
-  **Use when:** working with **RNNs, Transformers, or very deep CNNs**

Training Tricks: Gradient Clipping

- What is Gradient Clipping?
 - ✂ It's a technique to **cap the magnitude** of gradients during training
 - ❓ Prevents **instability** and **divergent updates**

- ⚙ How it works:

Let: $g = \|\nabla \mathcal{L}(\theta)\|_2$

If $g > \tau$ (a threshold), scale the gradient

$$\nabla \mathcal{L} \leftarrow \nabla \mathcal{L} \cdot \frac{\tau}{g}$$

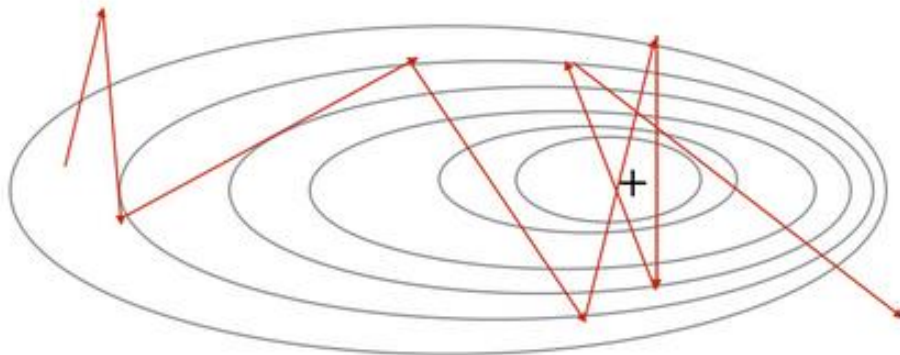
Training Tricks: Gradient Clipping



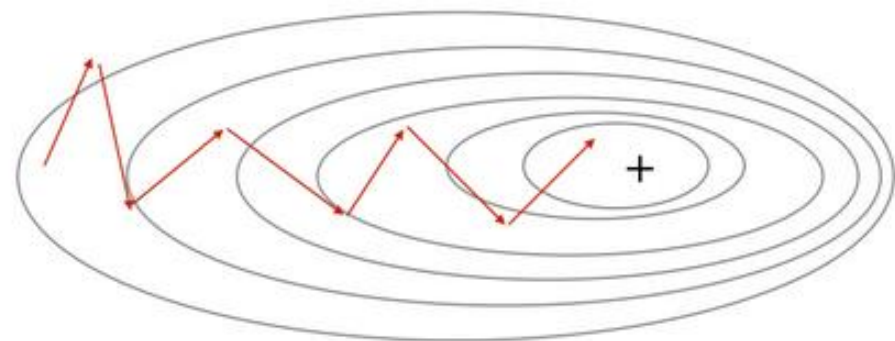
Benefits:

- Keeps updates within a **controlled range**
- Allows training to proceed even when gradients spike
- Prevents NaNs and exploding loss
- Helps stabilize **RNNs** and **transformer-style** models
- Makes training more **robust**, especially with large learning rates

Without gradient clipping



With gradient clipping



Training Tricks: Weight Decay

💣 During training, weights can grow **too large**, especially with:

- High learning rates
- Exploding gradients
- Overfitting on small datasets

❌ Large weights = unstable training & poor generalization

🛡️ **What is Weight Decay?**

- A regularization technique that **penalizes large weights**
- Encourages the model to **keep weights small**
- Adds an L2 penalty to the loss (SGD gradient):

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{task}} + \lambda \cdot \|W\|_2^2$$

Training Tricks: Weight Decay





Benefits:

-  Helps **prevent overfitting**
-  Encourages **simpler models**
-  Keeps gradients and weights more stable → fights gradient explosion



Adam vs. AdamW:

- **Adam**: Adds L2 penalty to the gradients ( not ideal)
- **AdamW**: Applies decay directly to weights ( better)

Advanced Note: Fine-tuning



We don't always train models from scratch.

- Most modern ML systems start with a **pretrained model**
- Then we **fine-tune** it for a **specific task, domain, or dataset**






What does fine-tuning mean?

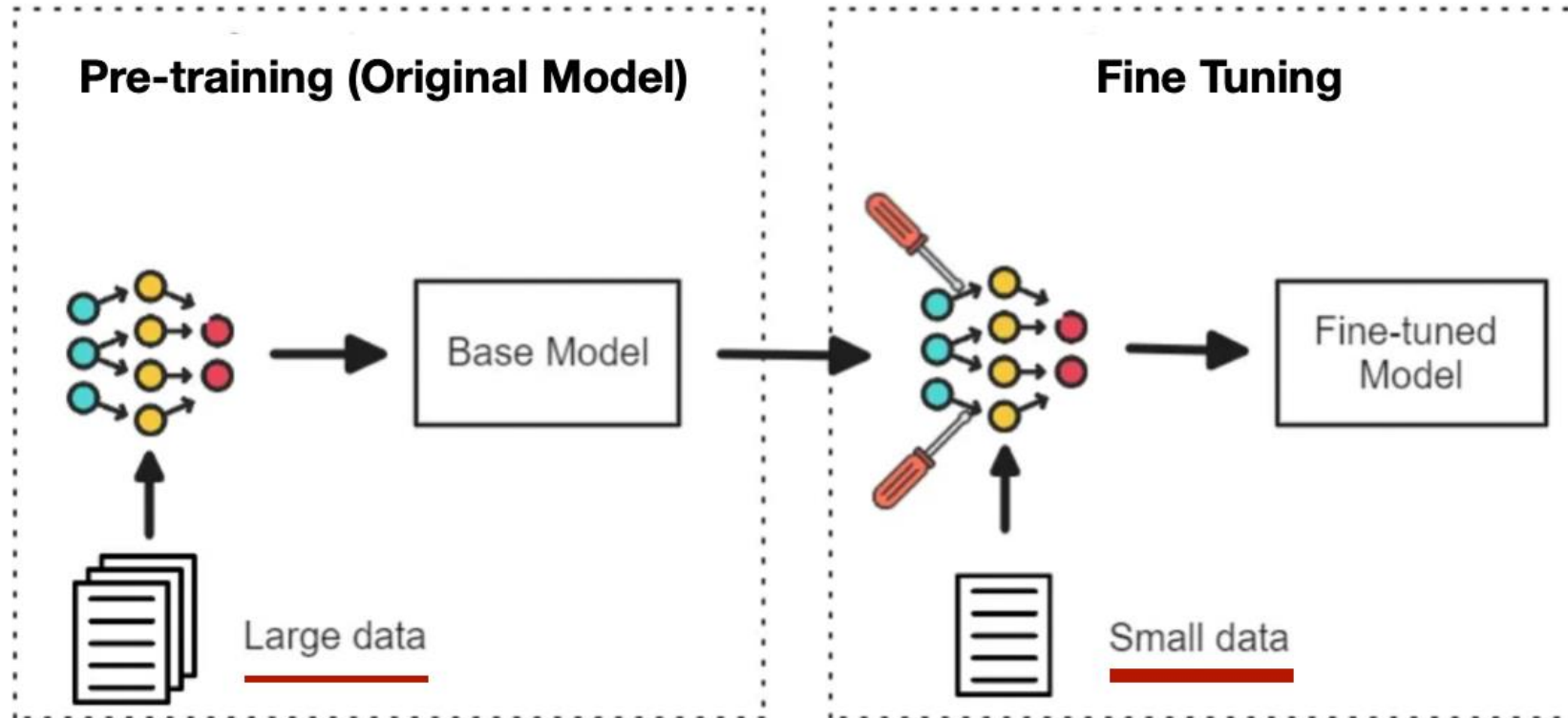
- Take a model that has already learned **general knowledge**, and continue training it to **specialize** in a new task.



Why?

-  Saves compute & time
-  Works well with small task-specific datasets
-  Leverages **transfer learning**: reuse what's already learned

Advanced Note: Fine-tuning



Thanks for your Attention