ARTIFICIAL INTELLIGENCE 2

How to Train Your Neural Network

Francesco Spinnato

* francesco.spinnato@unipi.it
University of Pisa

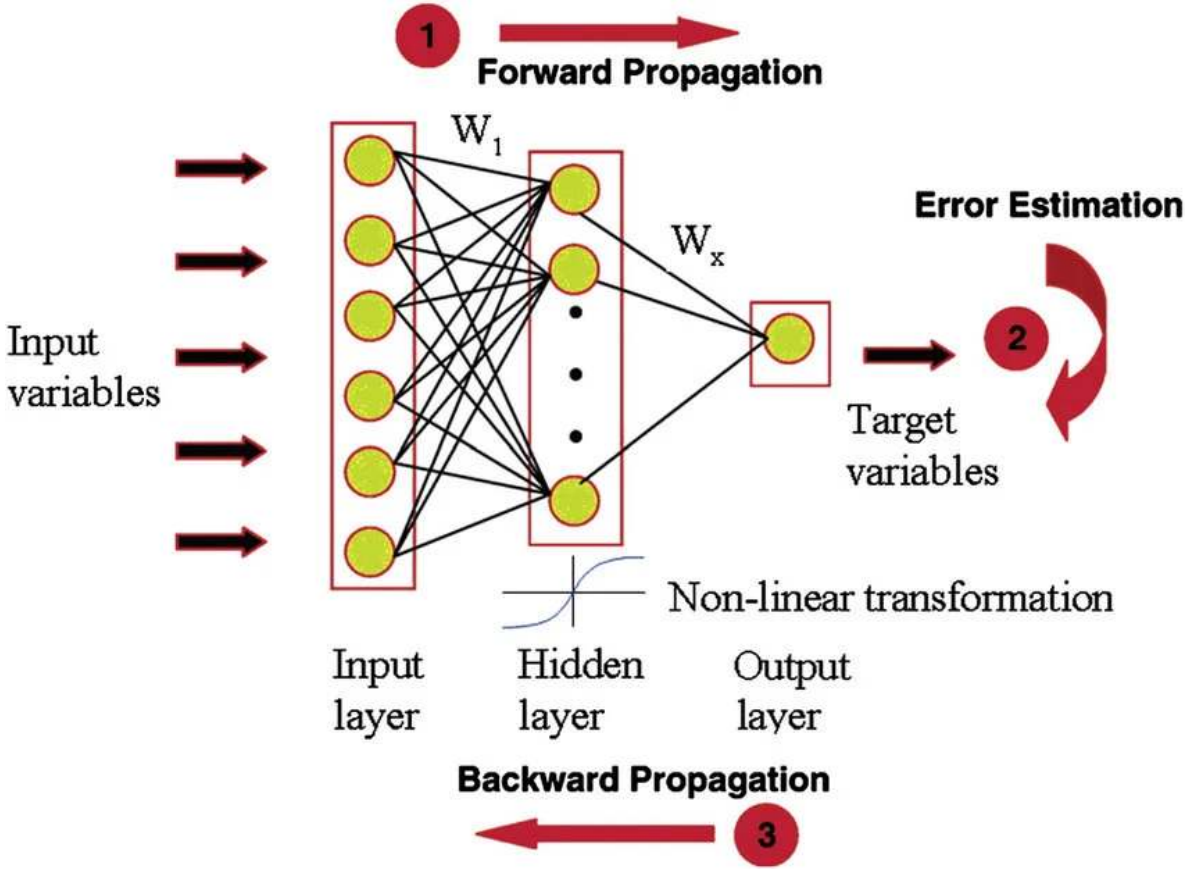
Why Training Neural Networks Is Non-Trivial

Training a neural network is not just a matter of writing down a model and pressing “run”.

Even for simple multilayer perceptrons, many choices strongly influence:

- Whether training converges at all
- How fast it converges
- Whether the final model generalizes or overfits

This lecture focuses on **practical rules of thumb**.



Parameters vs Hyperparameters

When training a neural network, we must distinguish two kinds of quantities.

Parameters are learned from data by optimization:

- Weights
- Biases

Hyperparameters are chosen by you:

- Learning rate
- Batch size
- Number of neurons and layers
- Regularization strength
- Optimizer
- Number of epochs

Performance depends more on hyperparameters than on the exact architecture.

The Standard Training Workflow

A clean experimental protocol is essential to avoid misleading results.

1. Split the dataset into training, validation, and test sets
2. Use the **training set to update the weights**
3. Use the validation set to **choose hyperparameters and training duration**
4. Once the choices are fixed:
    - retrain on training + validation (this is not mandatory in practice)
5. Evaluate **exactly once** on the test set

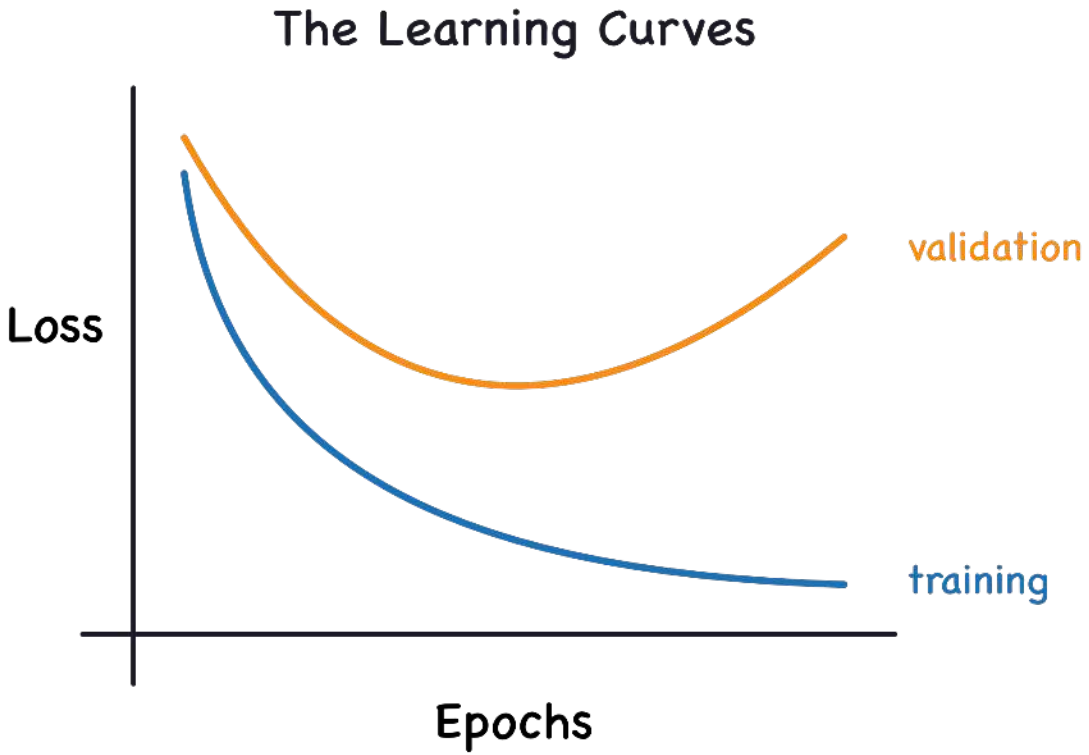
Violating this protocol almost always leads to overly optimistic performance estimates.

Train and Validation Curves

During training, one should always monitor at least two quantities:

- Training loss
- Validation loss

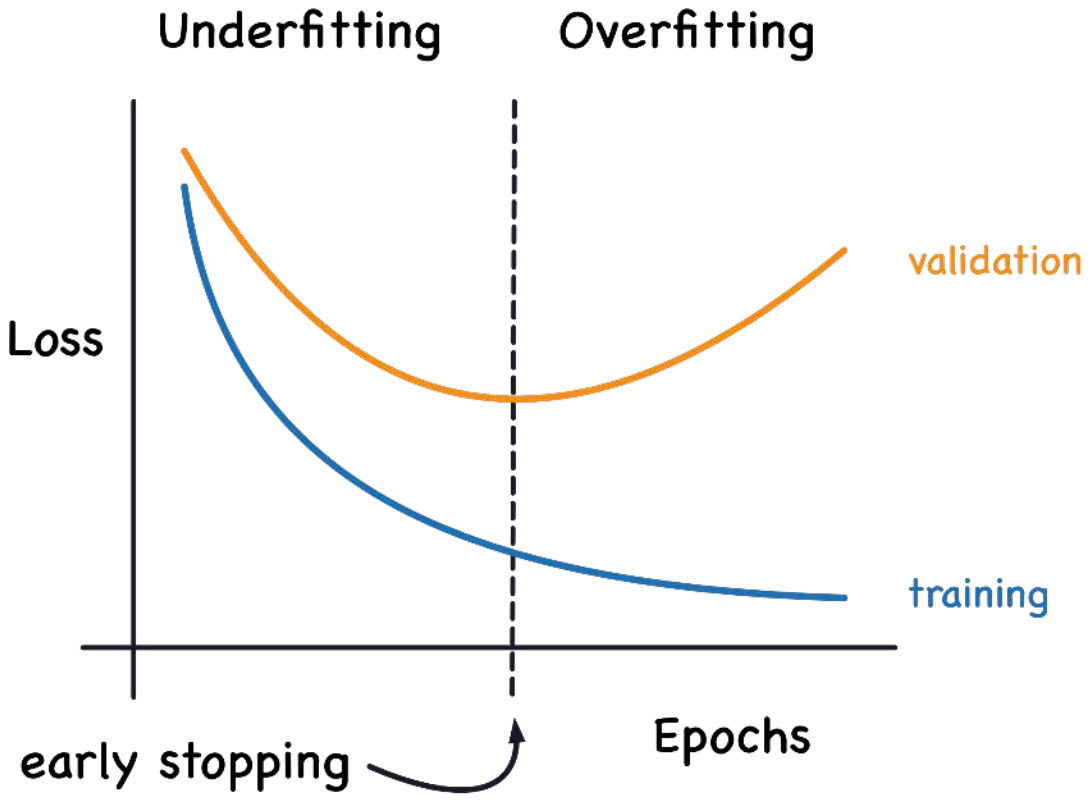
Their behavior tells us much more than a single final number.



Interpreting Learning Curves

Typical situations:

- **Underfitting**: both training and validation loss are high
  → the model is too simple or not trained long enough
- **Overfitting**: training loss decreases but validation loss increases
  → the model is memorizing the training data



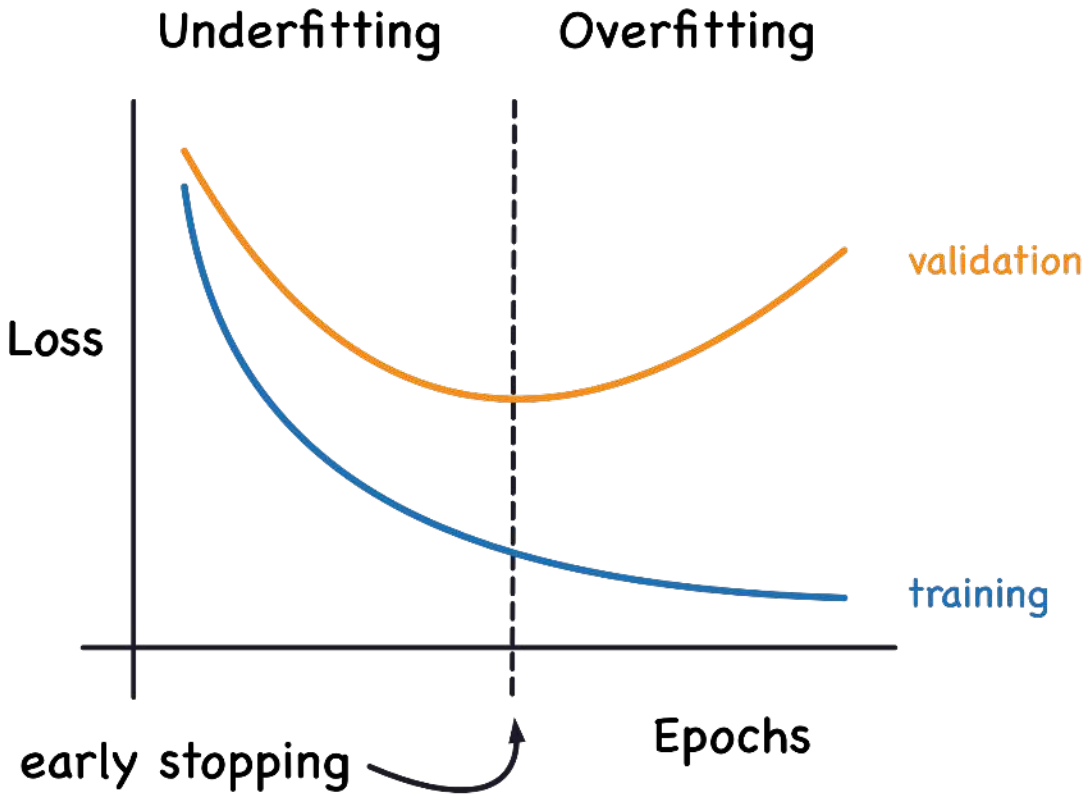
Early Stopping

The validation curve is used to:

- Choose the **number of epochs**
- Perform **early stopping** (halting trainign when the validation loss stops improving)

The **patience** parameter controls how tolerant the procedure is: it defines how many consecutive epochs without improvement to wait before stopping.

- **Low patience** → stops early, risks underfitting
- **High patience** → trains longer, more robust to noisy validation curves



Overfitting is a Central Problem

Most practical problems in neural network training are not about optimization, but about **generalization**.

A model can always be made large enough to:

- Fit the training set almost perfectly
- Perform poorly on new data

Controlling overfitting is one of the main goals of training strategy design.

Hyperparameters

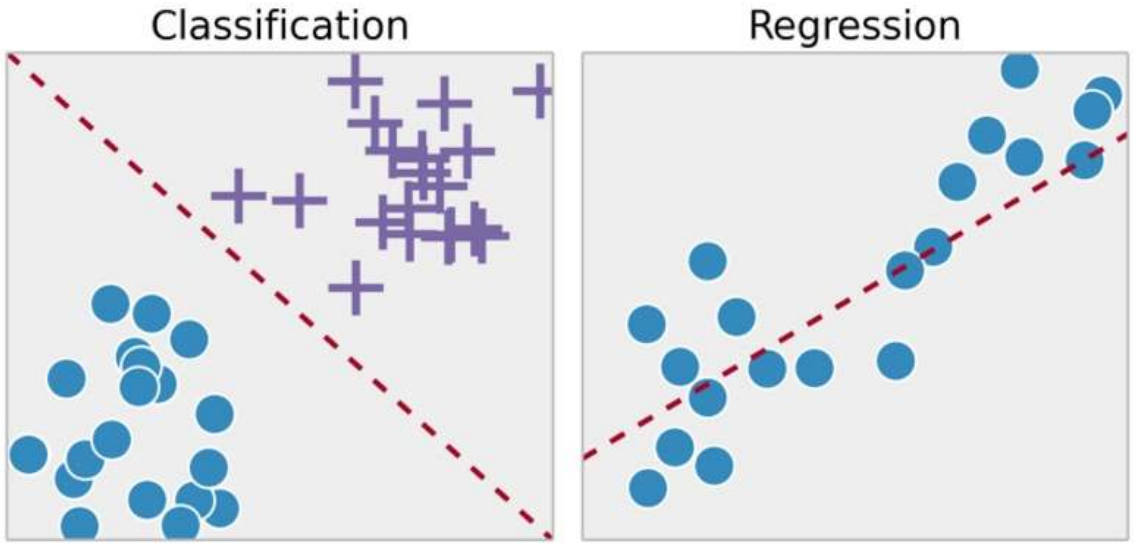
Loss

Loss is Task-dependent

Neural networks are used for many tasks, but the **training setup** depends strongly on whether we are solving:

- A **regression** problem: predict a real-valued quantity
- A **classification** problem: predict a discrete class

Choosing the wrong output activation or loss function can make training ineffective or meaningless.



Outputs, Activations, and Losses Must Match

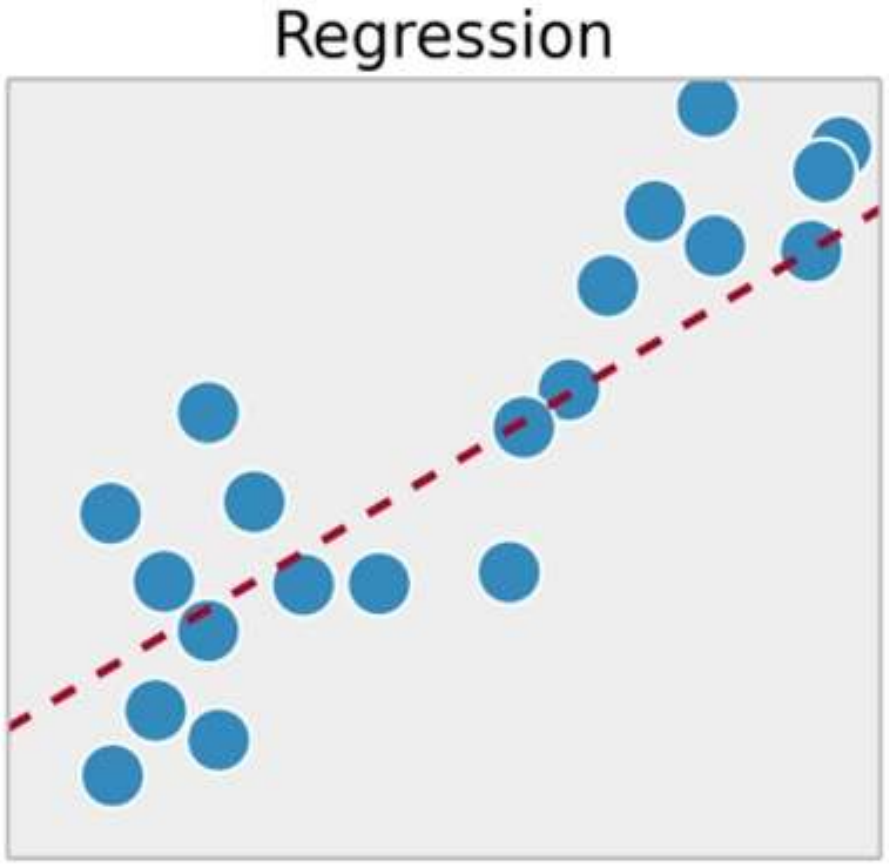
The **last layer** of the network and the **loss function** must be designed together.

- The output activation determines:
  - The range and interpretation of the predictions
- The loss function determines:
  - What it means for a prediction to be “good” or “bad”

A mismatch between these two often leads to:

- Slow convergence
- Numerical instability
- Incorrect probabilistic interpretation

Regression: Typical Setup



In regression, the goal is to predict a real number.

Common choices:

- Final activation:
  - **None** (linear output)
- Typical losses:
  - Mean Squared Error (MSE)
  - Mean Absolute Error (MAE)
  - Huber loss

This setup allows the network to:

- Predict any real value
- Directly model continuous targets

When to Constrain the Output in Regression

Sometimes the target is known to live in a specific range:

- Positive only (e.g., variance, count, price)
- Between 0 and 1 (e.g., probability, ratio)

In these cases:

- Use an activation such as:
  - ReLU or softplus for positive outputs
  - Sigmoid for outputs in $[0, 1]$

The loss is still typically MSE or MAE, but applied to the transformed output.

Binary Classification: Typical Setup

In binary classification, the target is one of two classes.

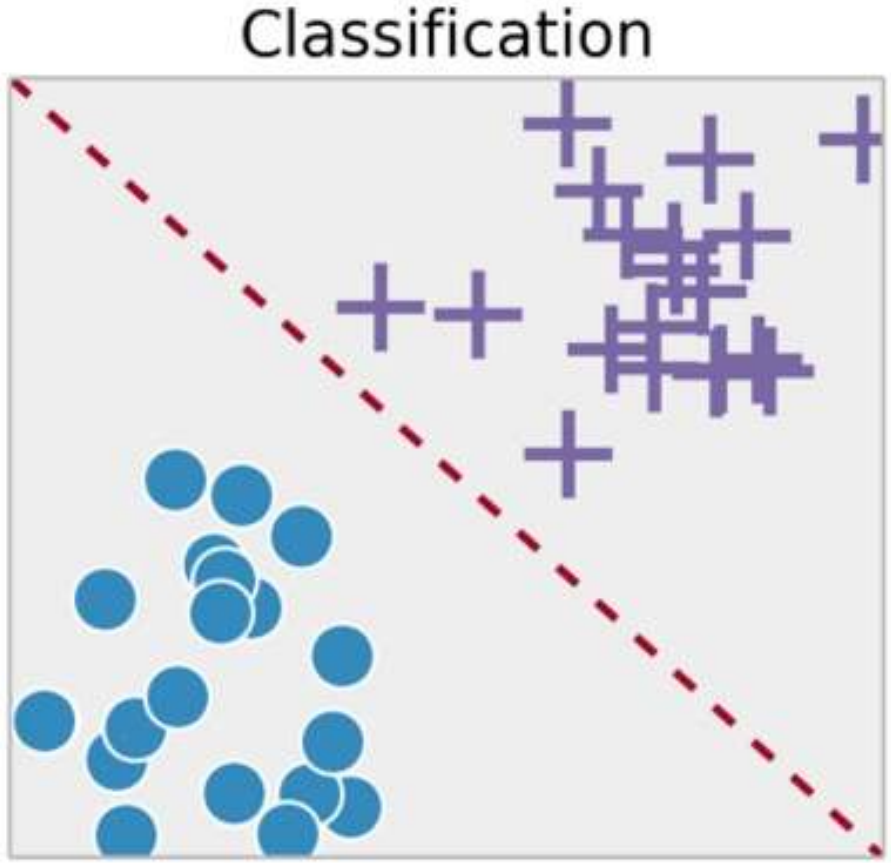
Standard design:

- Final activation:
  - **Sigmoid**
- Loss:
  - **Binary cross-entropy**
    $\ell(\hat{p}, y) = -\left[y \log \hat{p} + (1 - y) \log(1 - \hat{p})\right]$

The output is interpreted as:

- $p(y = 1 \mid x)$

A threshold (e.g., 0.5) is used to convert probabilities into class labels.



Why Not Use MSE for Classification?

Although it is possible to use MSE for classification, it is usually a bad idea:

- Gradients become very small when predictions saturate
- Training becomes slow and unstable
- The probabilistic interpretation is poor

Cross-entropy provides:

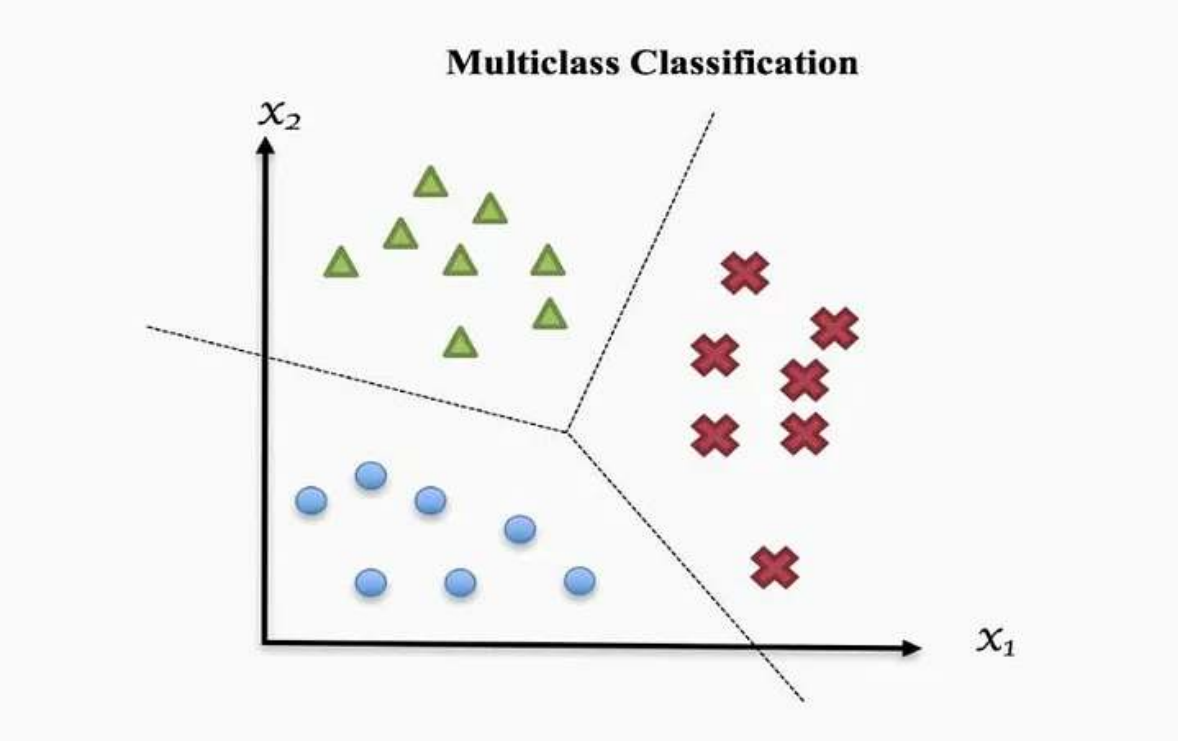
- Better gradients
- Faster convergence
- A principled likelihood-based objective

Multiclass Classification: The Usual Setup

In multiclass classification with K classes, the output is a probability distribution over classes: $\sum_{k=1}^{K} p_k = 1$

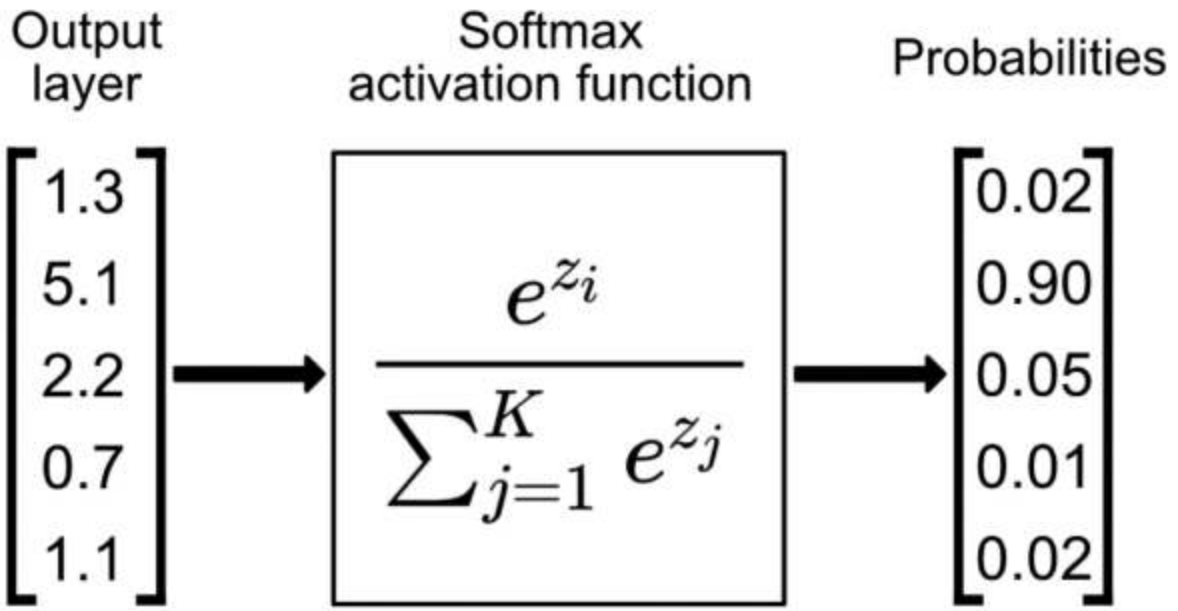
Standard design:

- Final activation:
  - **Softmax**
- Loss:
  - **Categorical cross-entropy**
    $\ell(\hat{\mathbf{p}}, y) = -\log \hat{p}_y$



Multiclass Classification: Softmax

Softmax is a mathematical activation function to convert a vector of raw scores (**logits**), into a probability distribution, where each value is between 0 and 1 and all values sum to 1



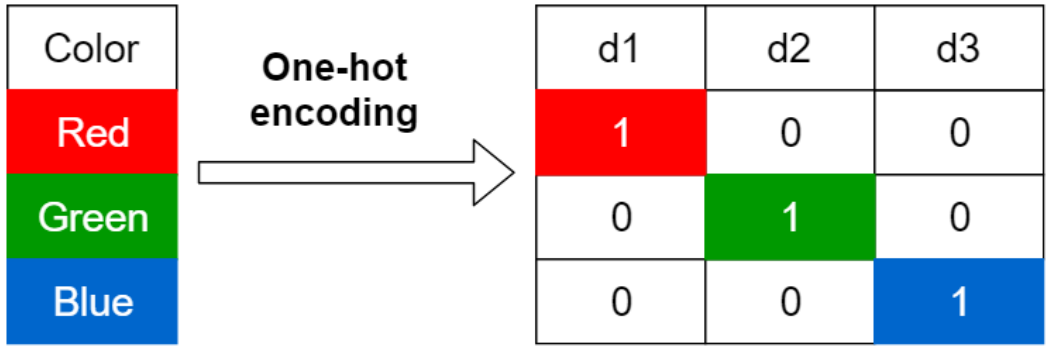
One-Hot vs Sparse Labels

There are two common ways to encode targets:

- **One-hot encoding**:
  - Target is a vector of length K
- **Sparse encoding**:
  - Target is an integer class index

The loss function must match the encoding:

- Categorical cross-entropy for one-hot labels
- Sparse categorical cross-entropy for integer labels



Summary

Besides the loss that you are optimizing, you can also track other metrics.

Task	Output Layer	Loss Function(s)	Common Metrics
Regression	Linear (or constrained)	MSE, MAE, Huber	MAE, RMSE, R^2
Binary classification	Sigmoid	Binary cross-entropy	Accuracy, F1, AUC-ROC
Multiclass classification	Softmax	Categorical cross-entropy	Accuracy, macro F1, weighted F1

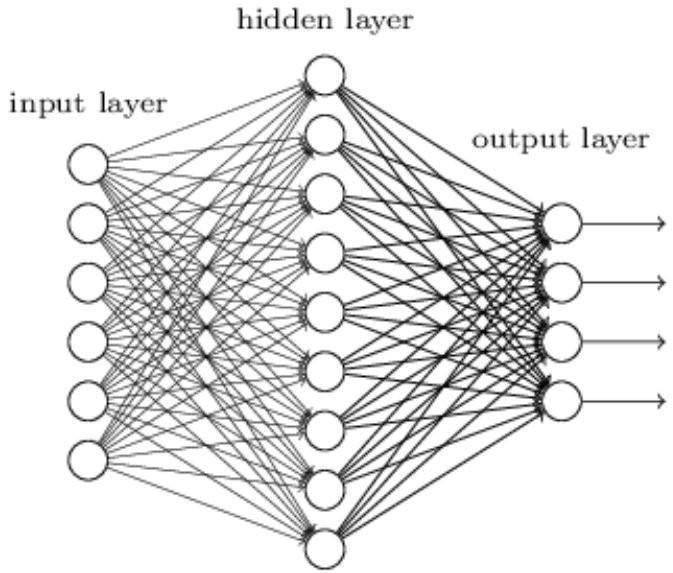
Hyperparameters

Width vs Depth

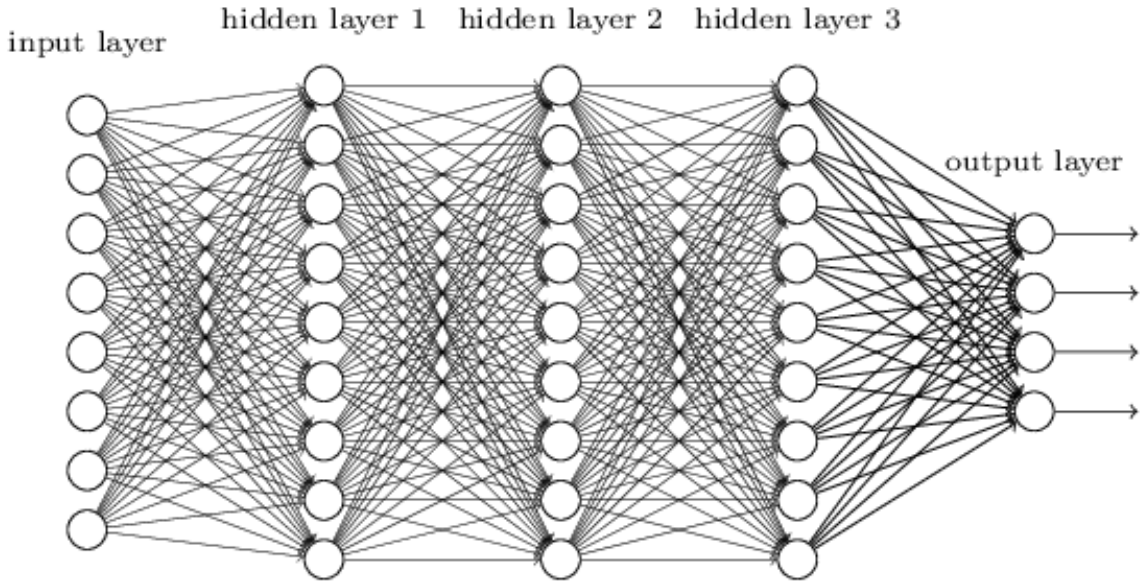
Choosing the Architecture (In Theory)

In principle, model capacity can be increased to fit more complex functions.

"Non-deep" feedforward neural network



Deep neural network



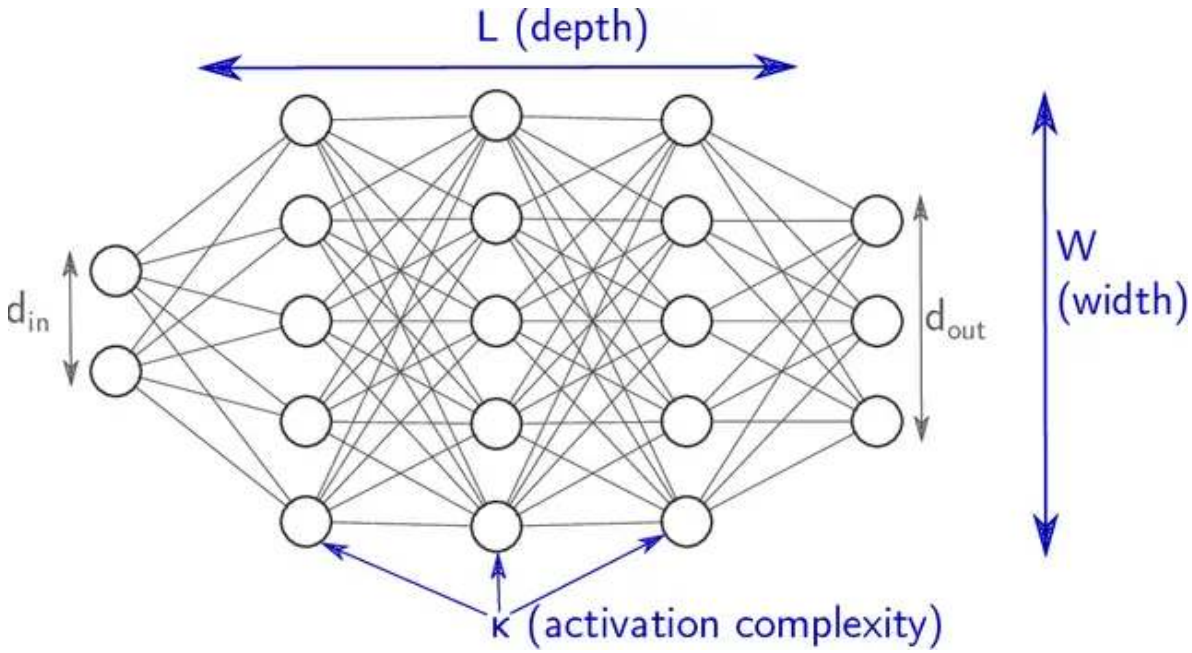
Choosing the Architecture (In Theory)

There are two main ways to do this:

- **Wider**: more neurons per layer
- **Deeper**: more layers

Both increase expressive power, but also:

- Increase computational cost
- Increase the risk of overfitting



Universal Approximation (But Not in Practice)

It is a classical result that:

- An MLP with a single sufficiently large hidden layer can approximate any continuous function.

However, this is a **theoretical** result:

- The required layer may be astronomically large
- Training such a network can be extremely inefficient or unstable
- Generalization is not guaranteed

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural Networks, 2(5), 359–366.

Why Depth Helps in Practice

Although shallow networks are universal approximators, depth is often useful in practice:

- Many functions can be represented **more compactly** with deep networks
- Similar expressive power can often be achieved with:
  - Fewer neurons
  - Fewer parameters
  - More structured representations

This often improves parameter efficiency, but does **not** automatically make optimization or generalization easier.

Depth as Feature Hierarchy

Deeper networks naturally learn representations at multiple levels:

- **Early layers**: simple, local features
- **Middle layers**: combinations of features
- **Late layers**: task-specific, abstract features

This hierarchical structure is often crucial for good performance.

Choosing the Architecture (In Practice)

In real applications, architecture design is not guided by theory, but by robustness and empirical behavior.

There is no formula for the right size. The only reliable procedure is:

- Look at training and validation curves
- If both losses are high → model too small (underfitting)
- If training is low and validation is high → model too big (overfitting)

Architecture design is therefore an **experimental** process:

- you want a model with enough capacity to overfit
- but prevent it from actually overfitting

if you can't overfit a small batch, something is wrong!

A Reasonable Default Strategy

A practical and robust starting point:

- Use a **small, deep-ish** network:
  - Typically 2 to 5 hidden layers
  - Moderate width per layer
- Avoid:
  - Extremely wide single layers
  - Huge models unless you have huge datasets

Then:

- Increase capacity only if learning curves show clear underfitting.

How to Scale the Model

When you need more capacity:

1. First, increase **width** slightly
2. If that is not enough, add **one more layer**
3. Only then consider large jumps in model size

Every increase should be justified by:

- Clear underfitting in training and validation curves.

Width Is a Hyperparameter

There is no “correct” number of neurons.

Width controls:

- Model capacity
- Computational cost
- Overfitting risk

Like learning rate or regularization, it must be chosen empirically.

What Increasing Width Does

Increasing width:

- Increases the number of parameters
- Makes the function class richer
- Often makes optimization easier
- Increases memory and computation
- Increases overfitting risk

Sensible default ranges:

- Small tabular problems: 32–128 neurons per layer
- Medium-size problems: 128–512 neurons per layer
- Very large datasets: 256–1024 or more

These are starting points, not rules.

Sanity Checks on Width

A simple reality check:

- Hidden width is often:
  - Larger than input dimension
  - Larger than output dimension

Extreme mismatches are suspicious:

- 4 neurons for 300 inputs → probably too small
- 10,000 neurons for 10 inputs → probably too large

Layer Shapes

Two common patterns:

- **Uniform width**:
  - e.g., 256 → 256 → 256 → 256
  - Simple and robust
- **Tapering**:
  - e.g., 512 → 256 → 128 → 64
  - Gradual compression of representations

How to Tune Width in Practice

A simple procedure:

1. Start with a moderate width (e.g., 128 or 256)
2. Inspect learning curves:
   - Both losses high → increase width
   - Training much better than validation → reduce width or regularize
3. Change width by factors of 2, not small increments

Width vs Depth

Given a fixed parameter budget:

- Moderately deep + moderately wide usually works best
- Extremely wide + very shallow is rarely a good choice

Depth often gives:

- More structured representations
- Better parameter efficiency

When Very Wide Models Make Sense

Very wide layers can be useful when:

- The input is extremely high-dimensional
- The dataset is very large
- The model is clearly underfitting even with several layers

In these cases:

- Strong regularization and careful monitoring are required.

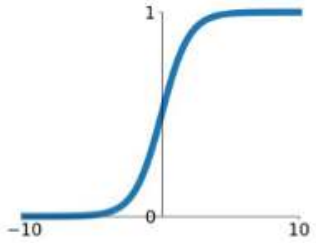
Hyperparameters

Activation Functions

Activation Functions

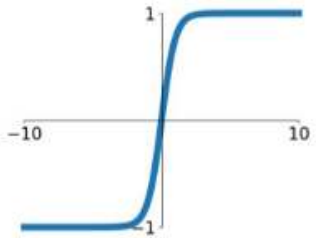
Sigmoid

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



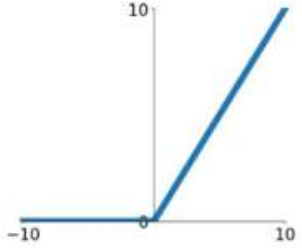
tanh

$$\tanh(x)$$



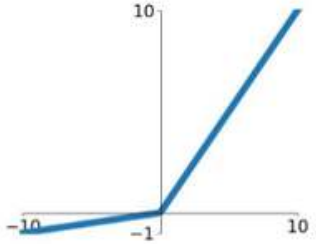
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

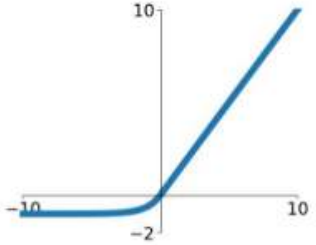


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation Function Choice

- **ReLU**
  - Default choice for hidden layers
  - Fast, simple, works well in most cases
- **Leaky ReLU / GELU / SiLU (Swish)**
  - Variants of ReLU
  - Better gradient behavior in some settings
- **Sigmoid**
  - Use for **binary outputs**
  - Not recommended in hidden layers (saturation, vanishing gradients)
- **Tanh**
  - Zero-centered alternative to sigmoid
  - Rare in modern deep nets, sometimes in RNNs

Hyperparameters

Initialization

Initialization in Practice

Neural networks are trained by gradient-based methods, so **the starting point does matter**.

With bad initialization:

- Gradients can vanish or explode
- Neurons can stay symmetric and learn the same thing
- Training can become very slow or fail entirely

Modern Defaults Are Already Good

In modern frameworks, the default initializations are usually:

- Xavier / Glorot (for tanh, sigmoid, linear)
- He initialization (for ReLU and variants)

These methods are designed to:

- Keep activations in a reasonable range
- Keep gradient magnitudes stable across layers

What This Means in Practice

For most applications:

- You should **not hand-design** initializations
- You should **trust the framework defaults**
- If training is unstable, the problem is almost always:
  - Learning rate
  - Architecture
  - Normalization
  - Not the initializer

When Initialization Actually Matters

You may need to think about initialization if:

- You use very unusual activation functions
- You build very deep networks without normalization
- You see immediate divergence or complete failure to learn

Otherwise, initialization is usually **not** a tuning knob in practice.

Hyperparameters

Optimizers

Optimizers

All optimizers minimize the loss by updating weights using gradients, but they differ in **how** they use gradient information.

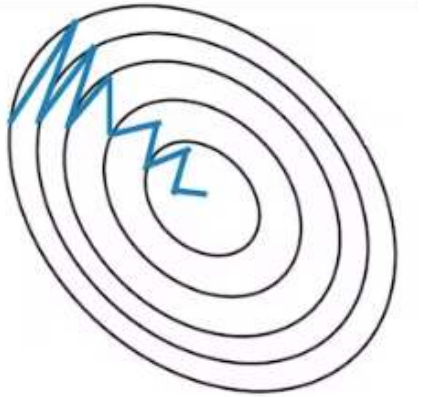
SGD Takes a fixed-size step in the gradient direction. Simple, but sensitive to the learning rate and slow in flat regions.

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}$$

SGD + Momentum Accumulates a "velocity", $\mathbf{v}$, i.e., a weighted average of past gradients. Like a ball rolling downhill, builds up speed in consistent directions and dampens oscillations.

$$\mathbf{v} \leftarrow \beta \mathbf{v} - \eta \nabla_{\mathbf{w}} \mathcal{L}, \quad \mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}$$

Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

Optimizers

RMSProp
Divides the gradient by a running estimate of its recent magnitude. Weights that receive large gradients get smaller updates; useful when gradients vary a lot across parameters.

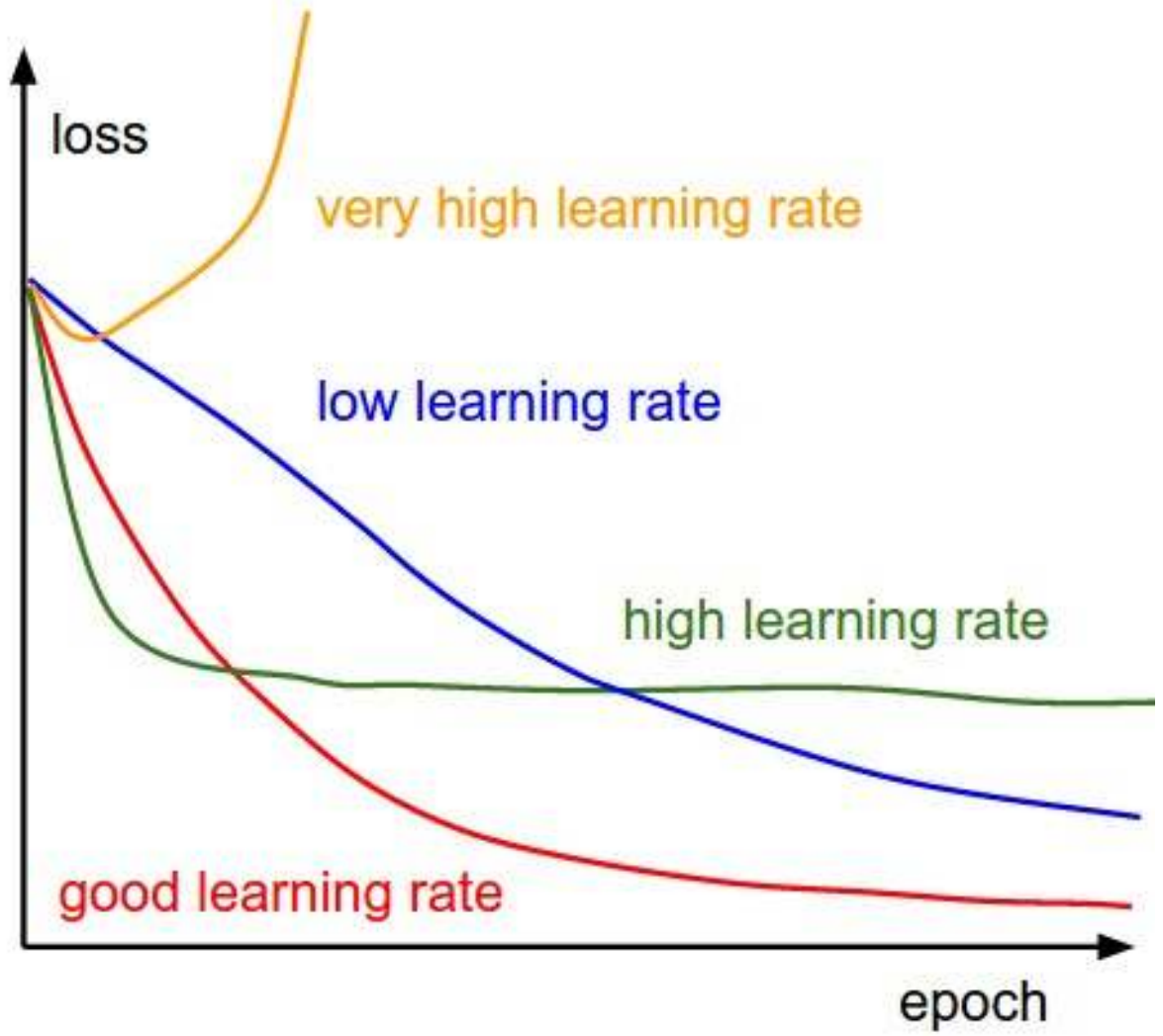
Adam *(most commonly used)*
Combines momentum + RMSProp: tracks both the direction and scale of past gradients, adapting the step size per parameter automatically. A robust default for most problems.

When in doubt, start with Adam. Switch to SGD with momentum if you need more control or reproducibility.

Learning Rate

The learning rate is a parameter in all optimizers:

- **Most important hyperparameter for training stability and speed**
- Controls the **step size** of each update:
  - Too large → divergence / unstable training
  - Too small → very slow convergence
- Common starting choices:
  - **Adam**: 10^{-3}
  - **RMSProp**: 10^{-3}
  - **SGD + Momentum**: 10^{-2} to 10^{-1}



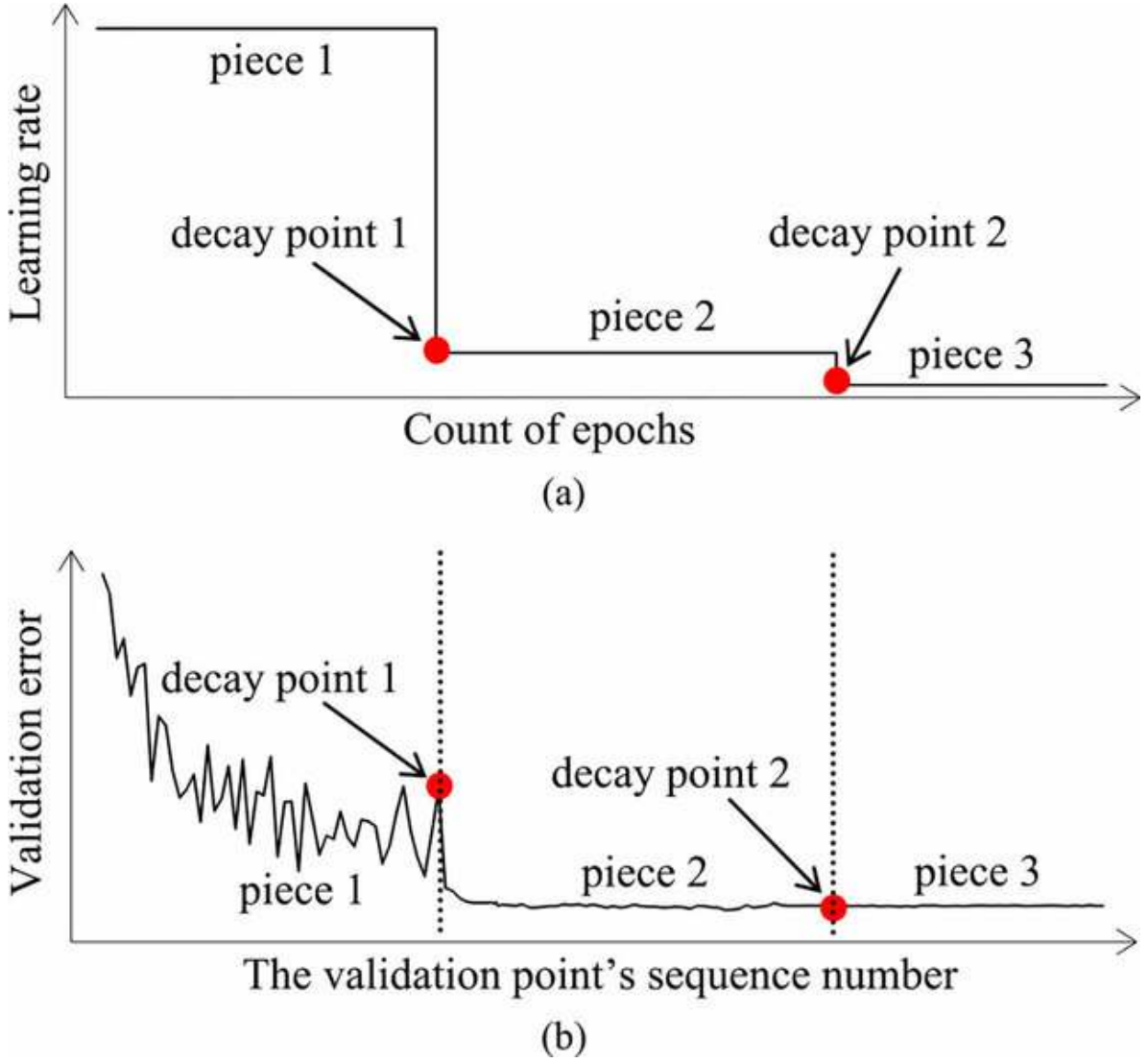
Learning Rate Scheduling

A fixed learning rate is rarely optimal throughout training.

Schedulers adapt the LR over time:

- **Step decay**: reduce by a factor every N epochs
- **Cosine annealing**: smoothly decay following a cosine curve
- **Warm-up**: start small, then increase before decaying
- **ReduceLROnPlateau**: decrease when validation loss stalls

A common strategy: start high to explore, then decay to fine-tune.



Regularization: The General Idea

Regularization means deliberately constraining the model or the training procedure in order to improve generalization:

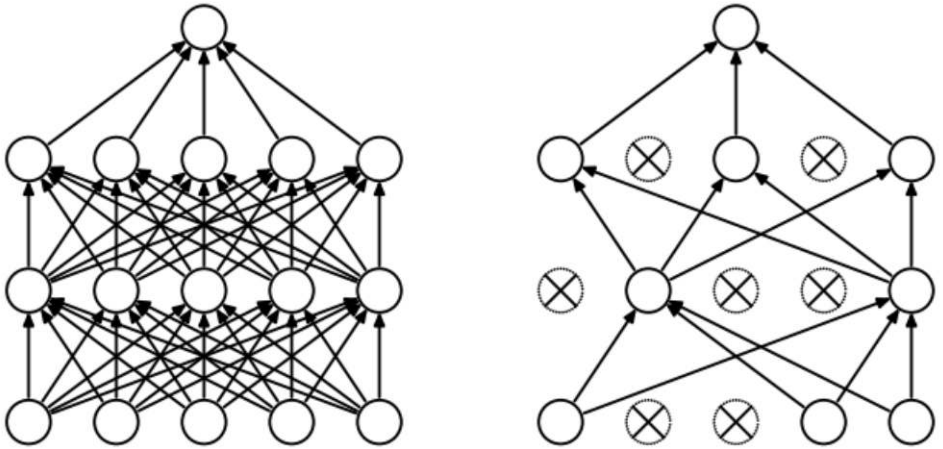
- **L2 regularization** or **weight decay**
  - In linear models, this corresponds to **Ridge regression**
  - In deep learning, it is implemented through **AdamW** or SGD with weight decay
- **L1 regularization**
  - In linear models, this corresponds to **Lasso regression**
  - Encourages sparse solutions
- **Early stopping**
- **Dropout**

Dropout

Dropout is a stochastic regularization technique:

- During training, some activations are randomly set to zero
- The network cannot rely on any single path through the model
- This forces the network to build redundant, robust representations

At test time, the full network is used, with appropriately scaled activations.



(a) Standard Neural Net (b) After applying dropout.

Batch Normalization

Batch Normalization normalizes the activations of each layer during training:

1. Compute per-column mini-batch mean and variance, $\boldsymbol{\mu}_B, \boldsymbol{\sigma}_B^2$:
2. Normalize (z-score):

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

3. Scale and shift with learnable parameters $\boldsymbol{\gamma}$, $\boldsymbol{\beta}$:
   $\mathbf{y}_i = \boldsymbol{\gamma} \odot \hat{\mathbf{x}}_i + \boldsymbol{\beta}$

At inference time, $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ are replaced by fixed running estimates of the mean and variance accumulated over the training set.
Where to place it: typically after the linear transformation and before the activation function.

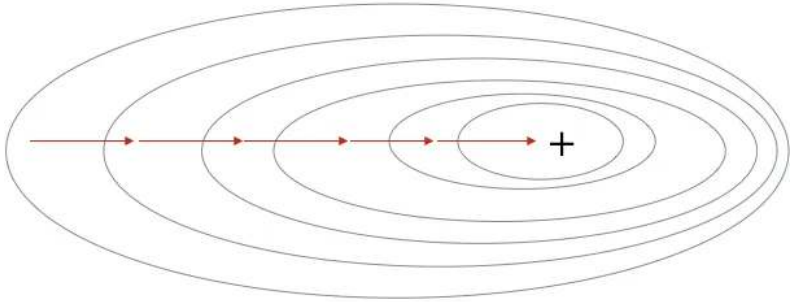
Hyperparameters

Batch Size

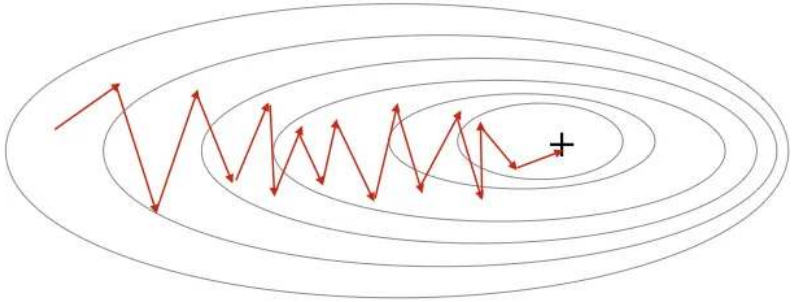
How to Choose the Batch Size?

The `batch_size` controls how many samples are used to estimate each gradient step.

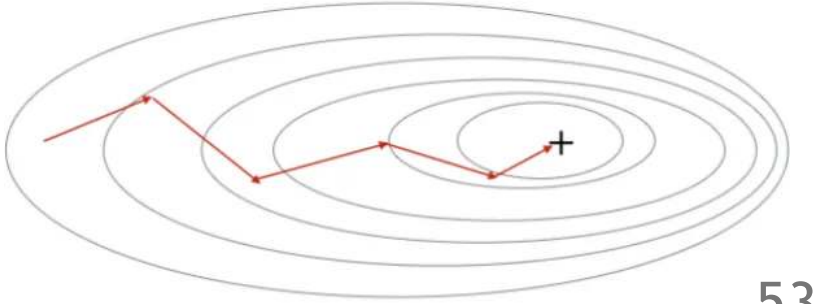
- `batch_size = 1`: online learning
- `batch_size = N`: full batch learning
- In practice: minibatches, often powers of 2, e.g., 8, 16, 32, 64, depending if it fits in memory



Stochastic Gradient Descent



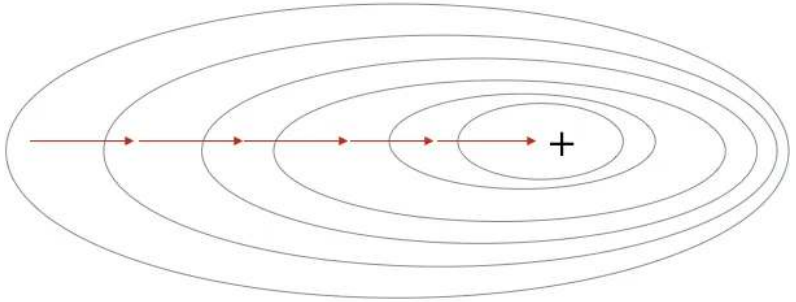
Mini-Batch Gradient Descent



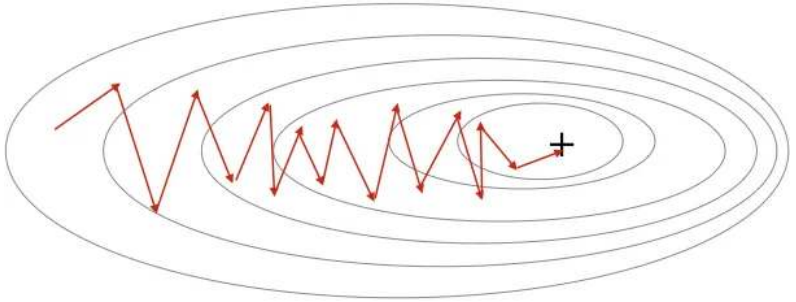
Effects of Batch Size

- Small batch sizes:
  - Noisy gradients
  - Sometimes better generalization
  - Slower per-epoch training
- Large batch sizes:
  - More stable gradients
  - Efficient on GPUs
  - Higher memory usage
  - Sometimes worse generalization

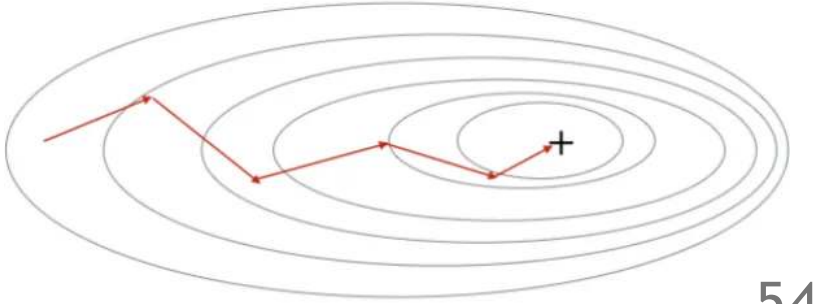
If the dataset does not fit in memory, reducing batch size is often the simplest solution.



Stochastic Gradient Descent



Mini-Batch Gradient Descent



Hyperparameters Search

Automating Hyperparameters Search

Finding the right model configuration (learning rate, depth, regularization, and more) is as important as the architecture itself.

The challenge: the search space is **enormous**. Even a modest network may have dozens of hyperparameters, each with a wide range of possible values, making exhaustive manual tuning **computationally infeasible**.

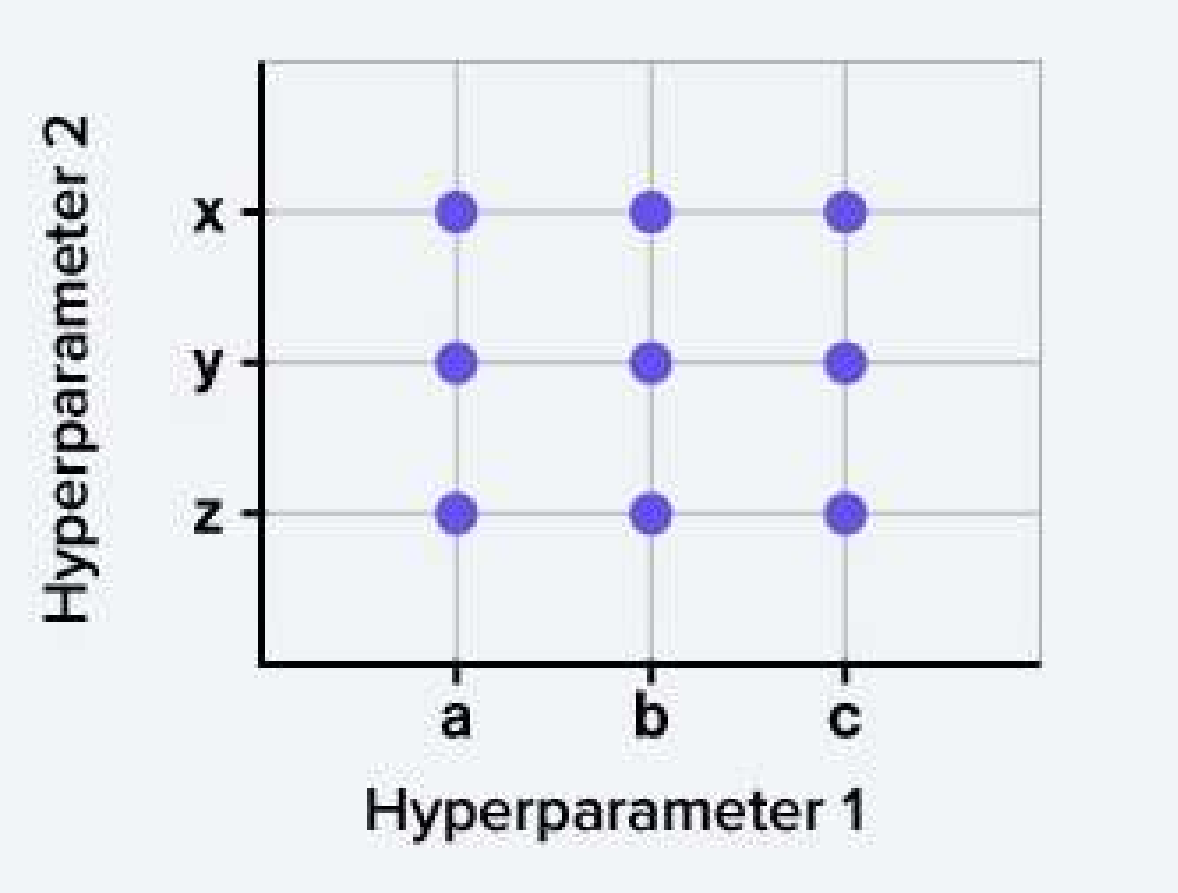
Systematic search strategies help navigate this space efficiently:

- **Grid Search** — exhaustively evaluate all combinations
- **Random Search** — sample combinations at random

Grid Search

Grid search is a **systematic hyperparameter optimization technique** used to identify the best combination of model parameters.

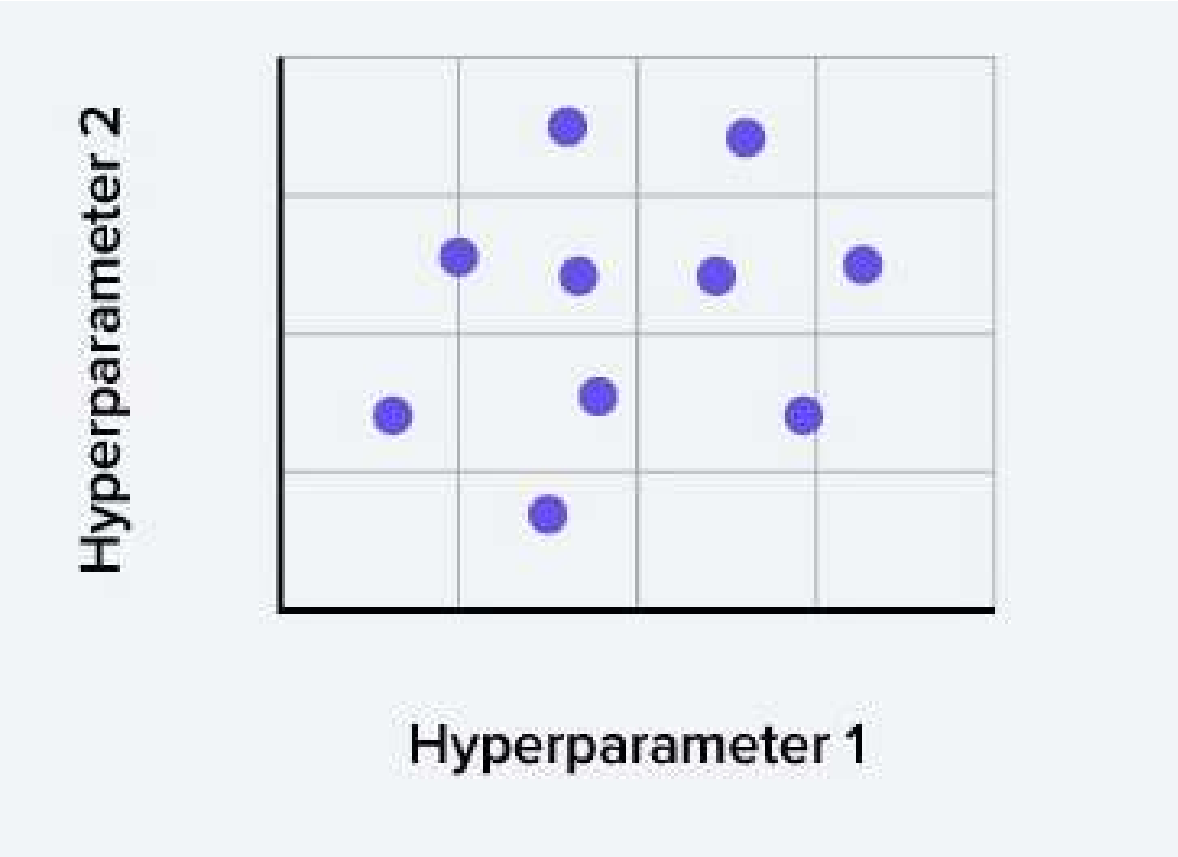
- Define a discrete set of candidate values for each hyperparameter
- Construct a grid of **all possible parameter combinations**
- Train and evaluate the model for each combination
- Select the combination that yields the best validation performance



Random Search

Random search is a **hyperparameter optimization technique** that samples parameter combinations at random rather than exhaustively.

- Define a **probability distribution** (or range) for each hyperparameter
- **Randomly sample a fixed number** of parameter combinations
- Train and evaluate the model for each sampled combination
- Select the combination that yields the best validation performance



Final Remarks

Each Run Can Give Different Results

Neural network training involves multiple sources of randomness:

- Weight initialization
- Data shuffling
- Dropout masks
- Non-deterministic GPU operations

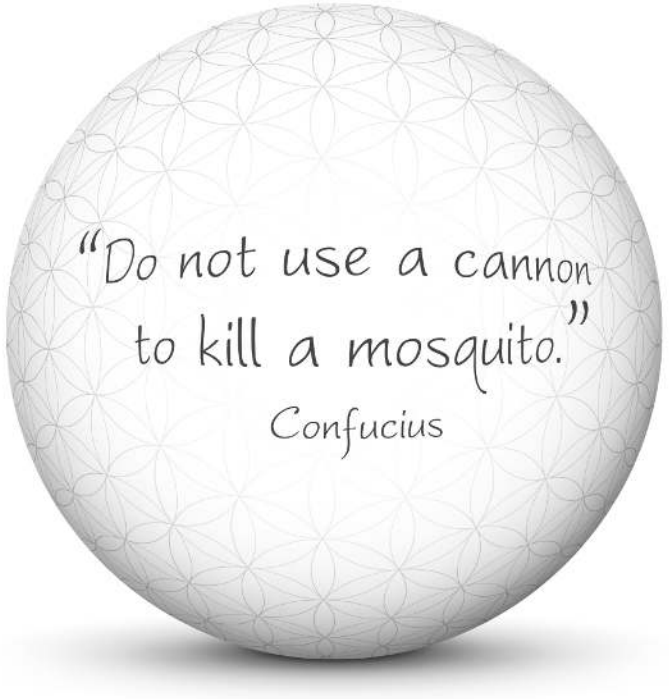
This means **two runs with identical code can produce different results.**

To improve reproducibility, set random seeds in all relevant libraries (Python, NumPy, PyTorch/TensorFlow). Even then, **full reproducibility on GPU is not always guaranteed.**

Practical advice: always report results as averages over multiple runs with different seeds, never cherry-pick a single lucky run.

Beware!!

- **Garbage in, garbage out**
  Bad data (labels, targets, scaling) cannot be fixed by any model.
- **Preprocessing still matters**
  Normalization and basic data cleaning are often as important as the model itself.
- **Rule of thumb**
  Start simple, fix the data and the pipeline first, increase complexity only if needed.



Neural networks are not the solution to everything.

If a simpler method works, use it.