Artificial Neural Networks and Deep Learning - Notes - v0.4.0

260236

December 2025

Preface

Every theory section in these notes has been taken from two sources:

- Ian Goodfellow and Yoshua Bengio and Aaron Courville, Deep Learning, MIT Press. [2]
- Course slides. [6]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on artificial neural networks and deep learning. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

Contents

1 Introduction to Deep Learning **5**
- 1.1 Machine Learning Foundations . . . 5
  - 1.1.1 Machine Learning Paradigms . . . 8
    - 1.1.1.1 Supervised Learning . . . 9
    - 1.1.1.2 Unsupervised Learning . . . 12
    - 1.1.1.3 Reinforcement Learning . . . 17
- 1.2 Towards Deep Learning . . . 20
- 1.3 Modern Pattern Recognition (Pre-DL) . . . 22
- 1.4 What is Deep Learning after all? . . . 24
- 1.5 What's Behind Deep Learning? . . . 29
- 1.6 Summary . . . 31

2 From Perceptrons to FNNs **32**
- 2.1 Historical Context . . . 32
- 2.2 The Perceptron . . . 38
  - 2.2.1 Who Invented It? . . . 38
  - 2.2.2 Mathematical Model & Logical Operations . . . 40
  - 2.2.3 Hebbian Learning Rule . . . 44
  - 2.2.4 Perceptron as Linear Classifier . . . 48
  - 2.2.5 Boolean Operators & Linear Separability . . . 51
- 2.3 Feed-Forward Neural Networks (FNNs) . . . 54
  - 2.3.1 Architecture . . . 54
  - 2.3.2 Activation Functions . . . 57
    - 2.3.2.1 Linear . . . 59
    - 2.3.2.2 Sigmoid . . . 61
    - 2.3.2.3 Hyperbolic Tangent (tanh) . . . 64
  - 2.3.3 Output Layer . . . 67
    - 2.3.3.1 Regression . . . 68
    - 2.3.3.2 Binary Classification . . . 72
    - 2.3.3.3 Multi-Class Classification . . . 76
  - 2.3.4 Neural Networks as Universal Approximators . . . 80
- 2.4 Learning and Optimization . . . 82
  - 2.4.1 Supervised Learning and Training Dataset . . . 83
  - 2.4.2 Error Minimization and Loss Function (SSE) . . . 86
  - 2.4.3 Gradient Descent Basics . . . 90
  - 2.4.4 Backpropagation (Conceptual Introduction) . . . 96
- 2.5 Maximum Likelihood Estimation (MLE) . . . 103
- 2.6 Perceptron Learning Algorithm . . . 110
- 2.7 Summary . . . 117

3 Neural Networks and Overfitting **118**
- 3.1 Universal Approximation Theorem . . . 118
- 3.2 Model Complexity . . . 123
- 3.3 Measuring Generalization . . . 126
- 3.4 Terminology Clarifications . . . 128
- 3.5 Cross-Validation Techniques . . . 130
  - 3.5.1 Hold-Out Validation . . . 131
  - 3.5.2 Leave-One-Out Cross-Validation (LOOCV) . . . 133

3.5.3 K-Fold Cross-Validation . . . 135
3.5.4 Nested Cross-Validation . . . 137
3.6 Preventing Overfitting . . . 140
3.6.1 Early Stopping . . . 142
3.6.2 Hyperparameter Tuning . . . 145
3.6.3 Weight Decay (L2 Regularization) . . . 151
3.6.4 Dropout (Stochastic Regularization) . . . 160
3.7 Tips & Tricks . . . 162
3.7.1 Activation Function Saturation . . . 163
3.7.2 ReLU and Variants . . . 166
3.7.3 Weight Initialization . . . 170
3.7.4 Batch Normalization . . . 174
3.7.5 Mini-Batch Training . . . 179
3.7.6 Learning Rate Scheduling . . . 182

4 Recurrent Neural Networks (RNNs) 185
4.1 Sequence Modeling . . . 186
4.2 Memoryless Models . . . 190
4.2.1 Autoregressive (AR) Models . . . 190
4.2.2 Feed-Forward Extensions: TDNNs . . . 192
4.3 Models with Memory . . . 194
4.3.1 Hidden State Dynamics and Outputs . . . 194
4.3.2 Linear Dynamical Systems and the Kalman Filter . . . 196
4.3.3 Hidden Markov Models (HMMs) . . . 202
4.3.4 Comparison to Deterministic Recurrent Systems . . . 211
4.4 Definition . . . 213
4.4.1 What is a RNN? . . . 213
4.4.2 Nonlinear Update Equations with Weights . . . 218
4.4.3 Universal Computation Capability (Hava Siegelmann) . . . 221
4.5 Backpropagation Through Time (BPTT) . . . 223
4.5.1 RNN unrolling over U time steps . . . 223
4.5.2 Shared weights across time . . . 227
4.5.3 Training Algorithm Steps . . . 231
4.5.4 Vanishing and Exploding Gradients Limitation . . . 233
4.5.5 Dealing with Gradient Problems . . . 239
4.6 Long Short-Term Memory Network (LSTM) . . . 241
4.6.1 Architecture . . . 241
4.6.2 Gates . . . 244
4.6.3 Lightweight Alternative: Gated Recurrent Unit (GRU) . . . 247
4.6.4 Networks . . . 250
4.6.5 Multi-layer LSTM . . . 254
4.6.6 Bidirectional LSTM (BiLSTM) Networks . . . 257
4.6.7 Practical Tips: Initialization & Conditioning . . . 259
4.7 Sequential Data Problems . . . 262

Index 268

1 Introduction to Deep Learning

1.1 Machine Learning Foundations

Humans and animals learn from experience. Computers, too, can improve performance when exposed to more data or feedback. But how do we formally define "learning" in a way that's precise enough for a engineering course? Tom Mitchell[1], in 1997, proposed a now-classic definition:

Definition 1: Task, Experience, Performance

A computer program is said to learn from experience **E** with respect to some class of tasks **T** and a performance measure **P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**.

- **Task (T)**: what the program is supposed to do. For example, classification (spam vs not spam), regression (predict house prices) or game playing (chess).
- **Experience (E)**: the data the algorithm is exposed to. For example, training set of labeled emails (spam vs ham), past games played by an agent, sensor data from a robot.
- **Performance measure (P)**: the metric used to evaluate progress. For example, classification accuracy (F1 score), mean square error for regression, total reward in reinforcement learning.

A system "learns" if, after seeing more data or interacting more with the environment, its **measured performance improves**.

Example 1: Definition in Action

Some scenarios:

1. **Email Spam Classifier**
   - **T (task)**: Classify emails as spam.
   - **E (experience)**: Training dataset of emails labeled as spam.
   - **P (performance measure)**: Accuracy on unseen emails.

   If accuracy improves as the classifier sees more labeled data, then computer program learning.

2. **Self-Driving Car**
   - **T**: Driving from A to B safely.
   - **E**: Millions of hours of driving footage + sensor readings.
   - **P**: Fewer accidents per mile, shorter trip times.

   If the car improves after more data, it has learned.

[1] Tom Mitchell is a *pioneer of machine learning*, both as a scientist and as an educator. His 1997 textbook, and especially that concise definition, shaped how an entire generation of students and researches understand Machine Learning (ML).

3. **Chess Playing Agent**
   - **T**: Win games.
   - **E**: Past games played against itself or others.
   - **P**: Win rate.

   More games, better play, computer program learning.

This definition matters because is **broad and general** (covers supervised, unsupervised, and reinforcement learning), it stresses **measurable improvement** (no improvement, no learning), and highlights the **central role of data** (E) and evaluation (P).

Why Mitchell's definition doesn't mentions "Machine Learning" explicitly

1. **It's meant to be general**. Mitchell wasn't defining *what ML is as a field*, but rather *what it means for a program to learn*. He avoided vague terms like "machine learning" or "artificial intelligence" and instead described the *process*:
   - A program improves at a **Task (T)**;
   - Thanks to **Experience (E)**;
   - As measured by **Performance (P)**.
2. **Machine Learning = building such programs**. So instead of saying "*Machine Learning is when...*", he framed it as: "*a computer program is said to learn if...*". That's why his definition became the **canonical operational definition of Machine Learning**.
3. **It links directly to practice**. The definition is testable: we can check if a system improves with experience. This is much stronger than a philosophical definition like "*machine learning is making computers intelligent*".

Example 2: Analogy

Think of physics. Newton didn't define "physics". He defined *laws of motion* and *gravity*. From those definitions, physics as a discipline could build itself consistently.
Similarly, Mitchell didn't define "Machine Learning" as a whole discipline. He defined **what it means for a program to learn**. The field then said: "Machine Learning is the study of programs that satisfy this definition".

Mitchell's definition tells us ML is **not about hardcoding solutions**, but about **improving performance with data-driven experience**, measurable by a task-specific metric.

Why we start with Tom Mitchell's definition

1. **Machine Learning is broad and fuzzy**. People use "learning", "AI", "intelligence" loosely. By giving a **formal, authoritative definition** at the beginning, the course sets a *clear baseline*: what do we mean by *learning*? How do we recognize it in a program?

2. **It frames the whole course**. Everything we explain later, supervised learning, neural networks, deep learning, must fit inside this triplet (Task, Experience, Performance). For example:

   - Neural Network training? It's about improving P on T given more E.
   - Reinforcement learning? Same template, different E and P.

3. **It's rigorous but simple**. Unlike philosophical definitions of intelligente, Mitchell's version is **operational**: it tells us *how to test if learning is happening*. It works as a **scientific foundation**, "*if we can't measure performance improvement, we can't claim the program learned*".

4. **It avoids confusion later**. If we started with supervised learning or deep learning right away, we'd lack the general umbrella. With this definition first, we can always check: "*what is our T? what is our E? what is our P?*".

Mathematical View

Formally, suppose we have:

- Dataset $D = \{(x_i, t_i)\}_{i=1}^{N}$ (inputs + targets).
- A model $f_\theta(x)$ with parameters θ.
- A loss function $L(f_\theta(x), t)$ that measures errors (P).

Learning means finding θ^* that minimizes the expected loss:

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{(x,t)\sim E} \left[L\left(f_\theta(x), t \right) \right]$$

This equation will be explained more thoroughly in the following sections.

1.1.1 Machine Learning Paradigms

When Tom Mitchell gave us the **triplet (T, E, P)**, he provided a general definition of learning. But in practice, machine learning problems usually fall into a few **big paradigms**; categories defined by *what kind of data (experience) we provide* and *what kind of task we want solved*. These paradigms are like **different ways of framing the learning problem**:

1. **Supervised Learning**: we give the algorithm examples of input and desired output. The goal is learn to map new inputs to outputs.
2. **Unsupervised Learning**: we only give input data, no labels. The goal is discover hidden structures or representations.
3. **Reinforcement Learning**: we don't provide explicit labels. The system interacts with an environment, receives **rewards or penalties**, and learns a strategy (policy) to maximize reward over time.

These paradigms are important because are the **building blocks of the field**. Almost any ML problem can be described belonging to (or combining) these three. They differ mainly in the **nature of the data (E)** and the **type of feedback (P)** available. Understanding them helps in choosing the right algorithms and models for a problem.

Example 3: Analogy

Imagine teaching three kinds of students:

- **Supervised Learning student**: we show them math problems *with answers*, and they learn how to solve similar ones.
- **Unsupervised Learning student**: we give them a pile of problems *without answers*, and they try to find patterns (like grouping similar problems together).
- **Reinforcement Learning student**: we give them a puzzle game. They don't know the rules, but they learn through *trial and error* because we give them rewards when they succeed.

1.1.1.1 Supervised Learning

Supervised Learning is like learning *with a teacher*:

- The algorithm is given **examples of inputs and their correct outputs (labels)**.
- The goal is to learn a **mapping function** that predicts the correct output for new, unseen inputs.

Formally:

- Training dataset:

$$D = \{(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)\}$$

  Where x_i are inputs and t_i are targets.
- Model: $f_\theta(x) \approx t$.
- Learning: choose parameters θ that minimize a loss function measuring error.

In other words, **Supervised Learning** is a type of machine learning where the algorithm is trained on a labeled dataset, meaning each training example includes both the input data and the correct output. And the goal is to learn a function that maps inputs to outputs, in order to make predictions on new, unseen data.

Types of Supervised Learning

In supervised learning we always have:

- **Inputs** x (features).
- **Outputs** t (labels/targets).
- A **model** $f_\theta(x)$ that learns a mapping from inputs to outputs.

The distinction between **classification** and **regression** depends on the **nature of the output**.

- **Classification**: Predict a **discrete class label**. The output space is a finite set of categories. For example:
  - Binary: $\{0, 1\}$, e.g. spam vs not spam.
  - Multi-class: $\{1, \ldots, K\}$, e.g. digits 0-9.

  From a mathematical point of view:

$$f_\theta(x) : \mathcal{X} \to \{1, 2, \ldots, K\}$$

Example 4: Cars vs Motorcycles

Use the classic triplet:

- **Task (T)**: distinguish between two categories (binary classification).
- **Experience (E)**: dataset of images labeled "car" or "motorcycle".
- **Performance (P)**: accuracy (percentage of correct predictions).

Pipeline (how supervised learning was traditionally done before deep learning):

- **Feature Extraction (Hand-Crafted Features)**. Raw data (like an image, sound, or text) is often too complex to give directly to a simple model. Traditionally, humans designed *rules* or *functions* to extract **features** from raw data.
  - Example (images): count edges, corners, textures, or wheel shapes.
  - Example (text): word frequencies, presence of certain keywords.
  - Example (audio): pitch, energy, Mel-frequency coefficients (MFCCs).

  These features are **manually engineered** to capture the most important aspects of the problem. The output is a vector of numbers (feature vector) that represents each example. This step is about "*what information to feed into the model*".

  In this example, hand-crafted features are:
  - Extract "number of circular shapes" (wheels);
  - Extract "dominant color";
  - Extract "edge orientation histograms".

  The photo is now a vector like $[2, 0.6, 0.8]$
- **Learning a Model (Classifier)**. Once we have feature vectors, we train a **machine learning model** that learns to map those features to outputs (labels or numbers). The model **learns decision boundaries** (for classification) or **functions** (for regression) that separate categories or fit numeric values. This is the **actual learning step**: the algorithm adjusts its parameters from the data.

  In this example, the classifier could be a Support Vector Machine (SVM) model, which learns as follows: if "number of wheels ≈ 2" then is a motorcycle; if "number of wheels ≈ 4" then is a car.

- **Regression**: Predict a **continuous value**. The output space is the set of real numbers ($\mathbb{R}$). From a mathematical point of view:

$$f_\theta(x) : \mathcal{X} \to \mathbb{R}$$

Example 5: Price Prediction

Use the classic triplet:

- **Task (T)**: predict a **continuous value** instead of a discrete label.
- **Experience (E)**: dataset of houses (features: size, location, rooms) with their selling prices.
- **Performance (P)**: Mean Squared Error (MSE), Mean Absolute Error (MAE), or R^2 score.

Pipeline:

- **Hand-crafted features**: e.g., number of rooms, square meters, distance to city center.
- **Learned regressor**: a model that predicts a continuous output.

In simple terms, if our labels are:

- Categories, it's **classification**.
- Numbers, it's **regression**.

Why Deep Learning Changed This

In **deep learning**, feature extraction and learning are **not separated anymore**. Neural networks **learn features automatically from raw data** (pixels, sound waves, text). So the pipeline becomes **one end-to-end step**: input raw data → neural network → prediction.

More resources about Supervised Learning can be found in the notes for the Applied Statistics course:

1.1.1.2 Unsupervised Learning

Unsupervised Learning is like learning *without a teacher*:

- We only provide the algorithm with **inputs** $x_1, x_2, \ldots, x_N$.
- There are **no labels/targets** telling the algorithm the "correct answer".
- The goal is to **discover hidden structures** or **representations** in the data.

Formally:

- Dataset:
$$D = \{x_1, x_2, \ldots, x_N\}, \quad x_i \in \mathbb{R}^d$$
- Task: find structure in D, e.g., groups, manifolds, lower-dimension embeddings.
- Performance measure: less obvious (since no labels). It can be internal measures (compact clusters, variance explained) or extrinsic measures (utility in downstream tasks).

The most intuitive unsupervised task: Clustering

In supervised learning, we had "car vs motorcycle", categories are known. In unsupervised, no labels are given. The simplest question becomes: "*can we group the data into natural categories, even if we don't know their names?*". That's exactly what clustering does. **Clustering** is the process of grouping data points into **clusters** such that:

- Points in the same cluster are **similar** to each other.
- Points in different clusters are **dissimilar**.

Clustering uses a **similarity measure**, such as Euclidean distance. The algorithm groups data into clusters that minimize within-cluster distance and maximize between-cluster distance. Some common algorithms include:

- **Hierarchical Clustering**. Build a tree of clusters by progressively merging or splitting. Exists two approach: Agglomerative Clustering (Bottom-Up) or Divisive Clustering (Top-Down).

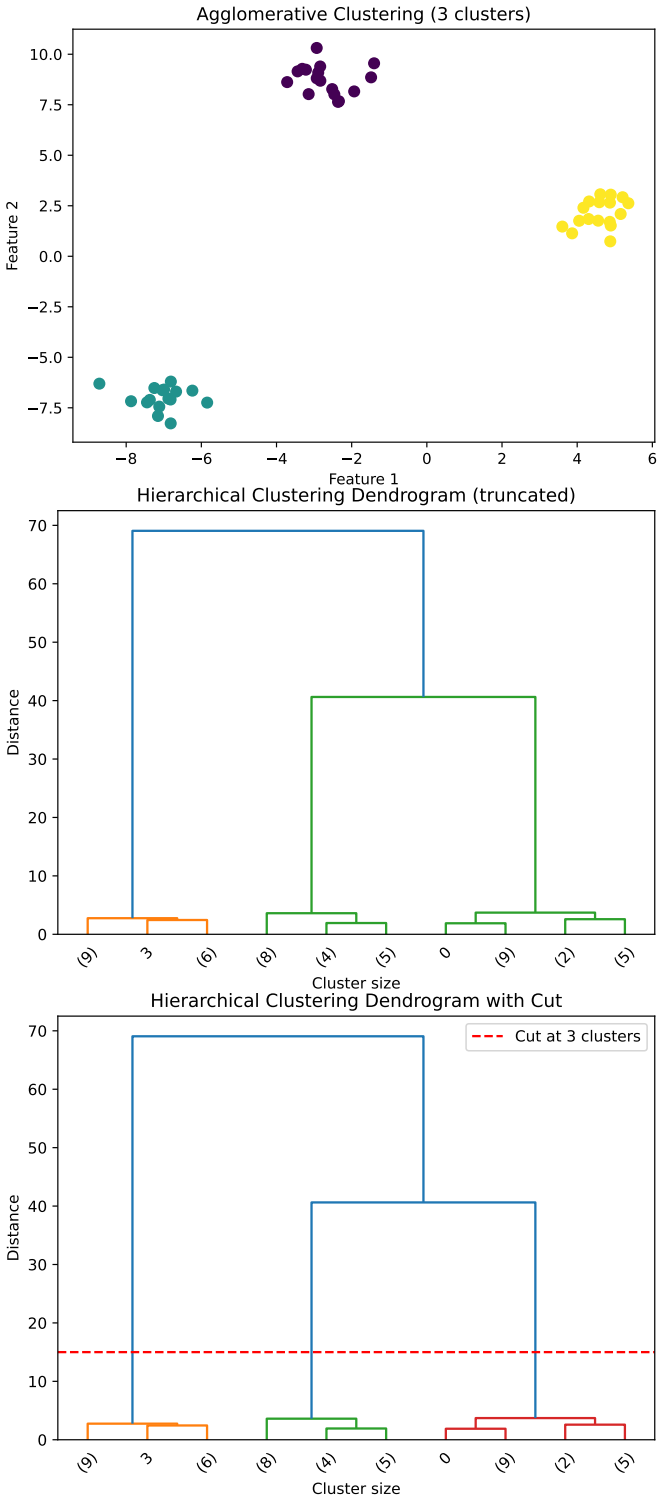


Figure 1: Agglomerative Clustering (top plot), Dendogram (mid plot) and Dendogram with cut (bottom plot).

About Figure 1, page 13. In the Agglomerative Clustering result, each dot is a **data point** (here we generated 50 synthetic points). The algorithm grouped them into **3 clusters**. We can see points within each cluster are **close together** in space. Also, the clusters are **well separated**, this is why hierarchical clustering works well here. The Dendogram shows the **hierarchical merging process**:

- At the **bottom**, each point starts as its own cluster.
- Going **upwards**, clusters that are close together are merged.
- The **height of each merge** (y-axis = distance) indicates how far apart the clusters were when merged.
- At the **top**, all points are eventually merged into a single cluster.

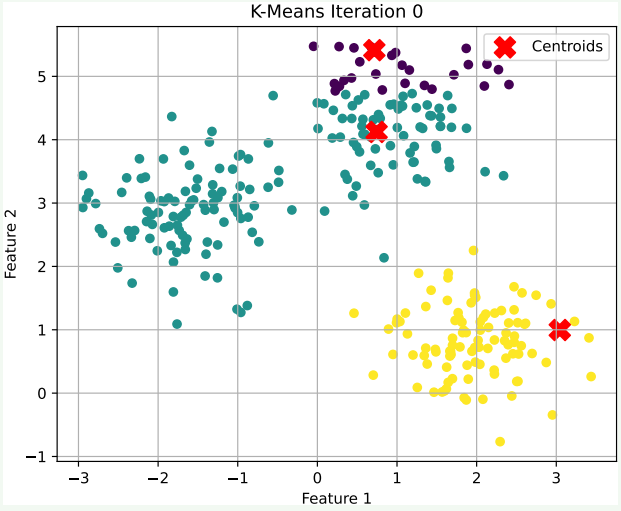
In the last figure, we "cut" the dendogram horizontally at a certain height (distance threshold), and we obtain a chosen number of clusters (here, 3). Everything **below the line** remains as separate clusters. Everything **above the line** (higher merges) is ignored. In the Dendogram, cutting at ≈ 15 gives **3 vertical "branches" crossing the red line**. Each branch corresponds to one cluster. These branches include **all 3 groups of points**.

- **K-Means**. Choose k clusters; assign points to the nearest cluster centroid; and update centroids until convergence.

Example 6: K-Means, taken from the Applied Statistics course

Below is a simple run of the K-means algorithm on a random dataset.

- Iteration 0 - **Initialization**

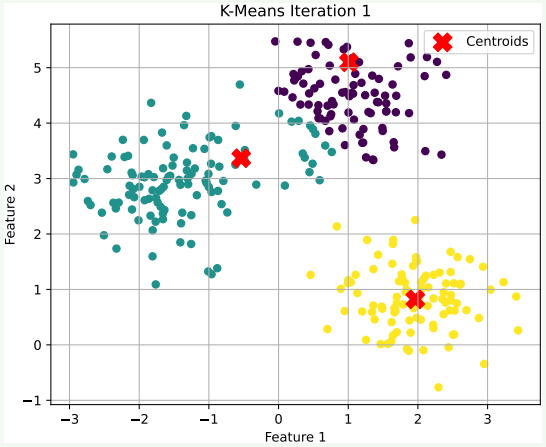


This is the starting point of the K-Means algorithm. **Three centroids are randomly placed in the feature space**.

At this point, no data points are assigned to clusters yet, or all are assumed to be uncolored/unclustered. The positions of the centroids will strongly influence how the algorithm proceeds.

The goal here is to start with some guesses. The next step will use these centroids to form the initial clusters.

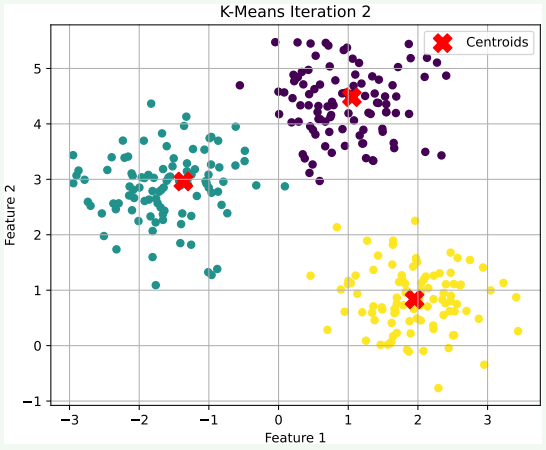
– Iteration 1 - **First Assignment and Update**



Each data point is assigned to the closest centroid, forming the first version of the clusters. New centroids are computed by taking the average of the points in each cluster. We can already see structure forming in the data, as points begin grouping around centroids.

This step is the first real clustering, and centroids begin to move toward dense regions of data.

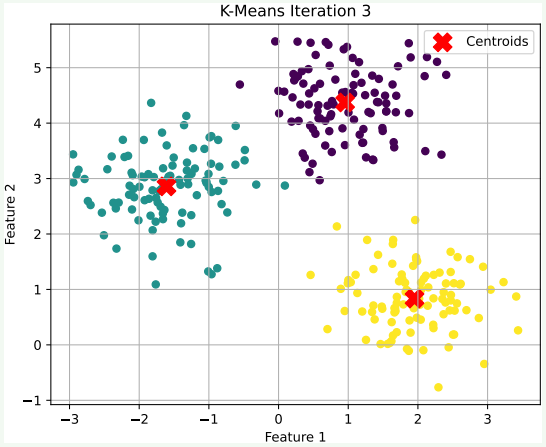
– Iteration 2 - **Re-Assignment and Refinement**



Clusters are recomputed based on updated centroids. Many points remain in the same clusters, but some may shift to a new cluster if a centroid has moved. Centroids continue moving closer to the center of their respective groups.

The algorithm is now refining the clusters and reducing the total distance from points to centroids.

– Iteration 3 - **Further Convergence**



At iteration 3, the K-Means algorithm reached convergence. The centroids no longer moved, and no points changed cluster.This means:

* The algorithm has found a locally optimal solution.
* Further iterations would not improve or change the clustering.
* The final configuration is considered the result of the algorithm.

In practice, this is how K-Means stops: it checks whether the centroids remain unchanged, and if so, it terminates automatically.

More resources about Unsupervised Learning and Clustering can be found in the notes for the Applied Statistics course:

1.1.1.3 Reinforcement Learning

Reinforcement Learning (RL) is like *learning by trial and error*. An **agent** interacts with an **environment** by taking **actions** and receiving **rewards** or **punishments**. The goal of the agent is to learn a policy that maximizes the cumulative reward over time.

At each step, the agent:

1. **Observes a state** s_t from the environment.
2. **Selects an action** a_t based on its current policy $\pi(a_t \mid s_t)$.
3. **Receives a reward** r_t and a **new state** s_{t+1}.

The agent's goal is to learn a **policy** $\pi(a \mid s)$ that maximizes the expected cumulative reward. Unlike supervised learning, no teacher gives the right answer; the agent learns from the **consequences** of its actions.

What is an Agent?

An **agent** is an *entity* that **makes decisions and takes actions in an environment to achieve a specific goal**. In reinforcement learning, the agent learns to optimize its behavior based on feedback from the environment.

With ***entity***, we mean anything that can perceive its environment through sensors and act upon that environment through actuators.

Example 7: Robot Navigation

For example, consider a robot navigating a maze. The robot (agent) perceives its surroundings (state), decides to move left or right (action), and receives feedback (reward) based on whether it gets closer to the exit or hits a wall. The robot's goal is to learn a strategy (policy) that maximizes its chances of reaching the exit while avoiding obstacles.

In simple terms, the robot through cameras and sensors perceives the maze (environment), decides its next move (action), and learns from the outcomes (rewards) to improve its navigation strategy (policy).

In summary:

- **Agent**: The robot.
- **Environment**: The maze.
- **State**: The robot's current position in the maze.
- **Action**: Moving left, right, forward, or backward.
- **Reward**: Positive reward for reaching the exit, negative reward for hitting a wall.
- **Policy**: The strategy the robot uses to decide its next move based on its current state.

The agent's **primary objective** is to **learn a policy that maximizes the cumulative reward** it receives over time by interacting with the environment.

Formalization of Reinforcement Learning

Reinforcement learning problems are often modeled using **Markov Decision Processes (MDPs)**. An MDP is defined by:

- **Task (T)**: learn a policy $\pi(a \mid s)$ mapping states to actions. In other words, the task is to find the best action to take in each state to maximize cumulative reward.
- **Experience (E)**: consists of sequences of states, actions, and rewards obtained by interacting with the environment.
- **Performance Measure (P)**: expected return (sum of discounted rewards):

$$P = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

  Where $\gamma \in [0, 1]$ is the discount factor that determines the importance of future rewards.

Key Concepts in Reinforcement Learning

The goal of this section is to introduce the Reinforcement Learning paradigm and its key concepts. These concepts will be covered in more detail in later sections. However, here are some of those concepts:

- **Exploration vs. Exploitation**: The dilemma of choosing between exploring new actions to discover their effects (***exploration***) and exploiting known actions that yield high rewards (***exploitation***).

  Why a dilemma? Because if the agent only exploits known actions, it may miss out on potentially better actions. Conversely, if it only explores, it may not accumulate enough reward.

- **Reward Signal**: The feedback received from the environment after taking an action, used to evaluate the action's effectiveness. It could be sparse or dense:
  - **Sparse Reward**: Rewards are infrequent, making it challenging for the agent to learn. For example, in a game, the agent might only receive a reward upon winning or losing.
  - **Dense Reward**: Rewards are given frequently, providing more immediate feedback. For example, in a driving simulation, the agent might receive small rewards for staying on the road and penalties for going off-road.

- **Delayed reward**: The reward for an action may not be immediate, making it challenging to associate actions with their long-term consequences. For example, in a chess game, a move may not yield an immediate reward but could lead to a win several moves later. The agent must learn to evaluate actions based on their long-term impact rather than immediate outcomes. This requires the agent to consider future rewards when making decisions.

⚖ RL vs. Supervised Learning

Reinforcement learning differs from supervised learning in several key ways:

Aspect	Supervised Learning	Reinforcement Learning
Data	Fixed labeled dataset (in-out pairs)	No labels; agent generates data by acting
Feedback	Correct answer for each example	Rewards (possibly delayed, sparse)
Goal	Minimize error (classification/regression)	Maximize cumulative reward
Typical methods	Regression, SVM, Neural Nets	Q-learning, Policy Gradients, Actor-Critic

⚠ Challenges of Reinforcement Learning

Reinforcement learning presents several challenges:

- **Exploration**: need to try enough actions to discover good strategies.
- **Delayed Feedback**: rewards may not be immediate, complicating reward assignment.
- **Sample inefficiency**: often requires millions of trials to learn effective policies.
- **Stability**: training can be unstable with neural nets.

Despite these challenges, RL has achieved remarkable success in various domains, including game playing, robotics, and autonomous systems.

In summary, reinforcement learning is a powerful paradigm for training agents to **make decisions in complex environments** by **learning from the consequences** of their actions. RL is distinct from supervised learning in its approach to data, feedback, and goals, making it suitable for a wide range of applications where direct supervision is not feasible.

1.2 Towards Deep Learning

This course, and this notes, focuses **mostly on Supervised Learning**, with some unsupervised learning concepts and techniques. ***Why?***

- Supervised Learning is the most widely used paradigm in practice (e.g., image classification, speech recognition, etc.);
- Many deep learning application (image recognition, NLP, etc.) are supervised tasks;
- Unsupervised learning will be touched when needed (e.g., representation learning, generative models, etc.);

Deep Learning is not a new paradigm, it's a **new approach** with supervised/unsupervised learning.

What about Deep Learning? Iris Flower Example

The Iris flower dataset is a classic dataset in machine learning, often used for classification tasks. It consists of 150 samples of iris flowers, each with four features: sepal length, sepal width, petal length, and petal width. The goal is to classify the flowers into three species: Iris setosa, Iris versicolor, and Iris virginica.

- **Traditional Machine Learning Approach:**
  - Extract "good features" from the raw data (e.g., petal length and width);
  - Train a classifier (e.g., decision tree) on these features;
- **Deep Learning Approach:**
  - Learn both **features** and **classifier** simultaneously from the raw data;

For example:

1. If **features are simple** (e.g., petal length and sepal width), then the classification task is **easy**, and a simple model (e.g., decision tree) can achieve high accuracy;
2. If **features are complex** (e.g., raw pixel values of flower images), then the classification task is **hard**, and the traditional approach **struggles to extract meaningful features**;
3. If **impossible to know** which features matter, then handcrafted features are **not enough**, and we need a model that can **learn features** from the data itself (e.g., a deep neural network).
4. Deep Learning learns features **directly from raw data**, making it suitable for complex tasks where feature engineering is challenging or infeasible (hierarchical representations).

⚖ Feature Engineering vs. Learned Features

- **Feature Engineering (Traditional ML)**:
  - Feature Engineering is the process of **using domain knowledge to extract features from raw data** that make machine learning algorithms work. It needs human experts to design and select features that are relevant to the task.
  - **Problem:** requires domain expertise, time-consuming, and may not capture all relevant information. It is often brittle and not transferable to new tasks or domains.
- **Learned Features (Deep Learning)**:
  - Learned Features are features that are **automatically learned by the model** from the raw data during training.
  - Layers learn progressively:
    - Lower layers learn simple patterns (e.g., edges, corners);
    - Middle layers learn more complex patterns (e.g., eyes, wheels);
    - Higher layers learn high-level concepts (e.g., faces, cars).
  - **Advantage:** optimized for the task at hand, can capture complex patterns, and are transferable to new tasks or domains. It requires less manual effort and often generalizes better to unseen data.

1.3 Modern Pattern Recognition (Pre-DL)

Before the rise of deep learning, modern pattern recognition techniques were primarily based on traditional machine learning algorithms and statistical methods. These techniques focused on feature extraction, dimensionality reduction, and classification using various algorithms.

Speech Recognition (early 1990s-2011)

Speech recognition systems used a **multi-stage pipeline** approach, which included:

- **Low-level features**: extracted from the raw audio waveform, such as MFCCs (Mel-Frequency Cepstral Coefficients), a compact representation of the spectral properties of the audio signal.
- **Mid-level features**: built by grouping/encoding low-level features over short time windows, capturing temporal dynamics. For example Mixture of Gaussians (MoG) used to model acoustic units (phonemes).
- **Classifier (high-level features)**: used to map mid-level features to words or phrases. Common classifiers included Hidden Markov Models (HMMs) combined with Gaussian Mixture Models (GMMs) to decode sequences of acoustic units into words.

This pipeline worked decently but was very **hand-crafted** and success depended heavily on the quality of feature engineering.

Object Recognition (2006-2012)

Computer vision systems followed a similar multi-stage pipeline approach:

- **Low-level features**: detect edges, corners, gradients using methods like SIFT (Scale-Invariant Feature Transform) or HOG (Histogram of Oriented Gradients).
- **Mid-level features**: combine low-level descriptors into higher-level "visual words". For example, clustering with k-means to create a codebook of visual words, and Sparse Coding to represent images as sparse combinations of these words.
- **Classifier (high-level features)**: train SVMs (Support Vector Machines) or Random Forests to classify images based on mid-level features.

Again, this approach was heavily reliant on hand-crafted features and required significant domain expertise to design effective features. However, before 2012, these methods were the state-of-the-art in many computer vision tasks.

General Pipeline (Pre-DL Pattern Recognition)

The general pattern recognition pipeline before deep learning can be summarized as follows:

1. **Low-level features**: raw signal transformation (e.g., edges, frequencies).

2. **Mid-level features**: encode or cluster low-level descriptors (e.g., visual words, acoustic units).
3. **Classifier (high-level features)**: learns categories from hand-designed representations.

⚠ Limitations

- **Domain expertise required**: Designing MFCCs, SIFT, HOG, etc. required significant knowledge of the specific domain (speech, vision).
- **task specific**: features built for one task often did not generalize well to others (e.g., MFCCs don't work well for images).
- **Brittleness**: sensitive to noise, illumination, scaling, speaker accents, etc.
- **Limited expressiveness**: as dataset grew, hand-crafted pipelines saturated in accuracy.

Before deep learning, pattern recognition was a **multi-stage pipeline** heavily **reliant on hand-crafted features and domain expertise**. While effective for its time, it had significant limitations in scalability, generalization, and robustness that deep learning would later address.

1.4 What is Deep Learning after all?

After showing the historical context, what Machine Learning is, the three paradigms and how pre-DL pattern recognition worked, we can finally answer the question:

Now that we know what ML does, what makes Deep Learning *different* from classic ML?

We will take our time answering this question. First, we need to understand the meanings of "features" and "classifiers".

What are "features"?

Features are **numerical representations of the raw data** that capture something meaningful for the task.

Type of Data	Raw data example	Example of features
Images	Pixels (RGB values)	Edges, corners, textures
Audio	Waveform (amplitude over time)	Pitch, frequency spectrum, MFCCs
Text	Words or sentences	Word counts, syntactic structure

In **classical ML**, these features were **manually designed** by humans; engineers decided *what* was important and *how to compute it*. For example:

Input image → extract edges manually → feed into SVM classifier

So we had:

Handcrafted Features → Learned Classifier

Where "handcrafted" means "coded by humans". So, before Deep Learning, the **feature extraction** and the **classifier** were two separate stages in the pipeline, and humans designed the first stage. This approach worked, but only if the human correctly guessed *what features matter* for the task.

What does "Learned Features" mean?

Deep Learning says: "*Stop handcrafting features; let the machine learn them automatically, layer by layer, together with the final classifier*".

In **Deep Learning**, the model itself learns how to transform raw data into useful internal representations. Each layer of a neural network acts as a **feature extractor** that learns automatically *what patterns matter*:

- First layers: detect edges, colors, or simple shapes.
- Intermediate layers: detect object parts (e.g., eyes, wheels, leaves).
- Deep layers: detect abstract categories (e.g., "face", "car", "flower").

So instead of telling the machine *what to look for*, we let it **discover patterns directly from data**. This is the "**learned features**" part.

What does "Learned Classifier" mean?

After features are extracted (automatically or manually), the model still needs to **make a decision**: classify, predict, or generate.

- In traditional ML, this is the final **classifier** stage (e.g., SVM, logistic regression, random forest).
- In Deep Learning, the **last few layers** of the network act as that classifier, they map high-level learned features to output labels.

So, both parts, the *feature extractor* and the *decision function*, are **learned jointly** through backpropagation.

$$\text{Raw Data} \xrightarrow[\text{learned weights}]{\text{Feature Extractor}} \text{Representations} \xrightarrow[\text{learned weights}]{\text{Classifier}} \text{Predictions}$$

So DL uses a single model to learn both **features** and **classifier** together: Learned Features + Learned Classifier. The model not only learns *how to decide* but also *how to see the world*, both are learned from data.

So, "What is Deep Learning after all?"

Deep Learning is **not just a new algorithm**, it's a new way of *approaching representation learning*. If we had to answer in one line: "Deep Learning is the automatic learning of hierarchical data representations and decision functions directly from raw data". That's why it's so powerful: it *adapts* to the data and the task, without relying on human intuition about features.

Deepening: Why Not Everything Is Deep Learning

Deep Learning is *powerful*, but it's not a silver bullet, it's not *free*. It's the best tool **when** we have: large amounts of diverse data, high compute, a task based on perception or pattern recognition. Otherwise, **traditional ML or statistical models** can be simpler, faster, and just as effective.

- **Deep Learning needs a *lot of data.*** Deep models have **millions (sometimes billions)** of parameters. They only generalize well when trained on **massive labeled datasets** (e.g., ImageNet: 14M images). If we have small data, like 300 samples from an industrial machine, a deep model will likely **overfit** and perform worse than simpler methods. In other words, Deep Learning shines when there is **data abundance**, but struggles in **data scarcity**.
- **Deep Learning needs a *lot of computation.*** Training is computationally heavy, requiring specialized hardware: GPUs, TPUs, clusters, or cloud computing. Classic ML (SVMs, Decision Trees, Random Forests) can run on a laptop. Deep nets require weeks of GPU training, hyperparameter tuning, and energy cost. So, if the

task doesn't justify the cost, simpler ML is more efficient.

- **Deep models are *black boxes.*** We can rarely explain *why* a deep network made a decision. For critical systems (healthcare, law, finance, safety) we need **interpretability** and **traceability**. Simpler models like linear regression or decision trees are **transparent**, easy to justify in front of regulators or domain experts. For example, a hospital won't risk a deep net saying "tumor" without being able to explain which features caused that prediction.
- **Deep models are *hard to train and tune.*** Choosing architecture (layers, neurons, learning rate, dropout, etc.) is an art. Training can **diverge** or **get stuck** (vanishing gradients, overfitting, exploding losses). We often need extensive experimentation and deep knowledge of optimization tricks. So, not every team or project can afford the expertise and trial cycles DL requires.
- **Deep Learning doesn't always fit the problem.** Some tasks simply:
  1. Have **structured or tabular data** (e.g., bank records, tabular logs). Here, traditional ML (XGBoost, Random Forests) often outperforms DL.
  2. Require **symbolic reasoning** or **logic**, not pattern recognition. Here, DL struggles to capture rules and relationships that classical AI or rule-based systems handle better.
  3. Need **causal inference**, not just correlations. DL finds patterns but doesn't understand cause-effect relationships, which are crucial in many scientific and policy domains. Let's think about a real-world example: predicting disease spread based on interventions (lockdowns, vaccinations) requires understanding causality, not just correlations in data (not just "if X happens, Y follows", but "if we do X, Y will change").
- **Deep Learning needs *good data.*** DL is extremely sensitive to: label noise (wrong annotations ruin learning); biases in the dataset (can reproduce or amplify them); distribution shifts (fails badly if test data differ from training). Traditional methods often handle noise and small variations more robustly. So, "Garbage in $\rightarrow$ garbage out" is even more true with DL.
- **Deep Learning doesn't mean *understanding.*** DL recognizes **patterns**, not meaning. It can detect a cat, but it doesn't *know* what a cat is. It can predict outcomes, but not always *why* they happen. That's why current research explores **hybrid systems** combining DL with: symbolic reasoning (neuro-symbolic AI), knowledge graphs, logic and interpretability layers.

Deepening: ChatGPT, LLaMA & Modern AI Models - What Are They?

ChatGPT, LLaMA, Gemini, Claude, etc. are all based on a specific kind of **Deep Neural Network** called a **Transformer**, introduced in 2017 by Vaswani et al. ("Attention is All You Need"). So, fundamentally:

$$\text{ChatGPT, LLaMA, Gemini, etc.} \in \text{Deep Learning}$$

They are not "beyond" DL, they are its **current frontier**.

What kind of Deep Learning model? They belong to the family of **Large Language Models (LLMs)**.

- **Architecture**: Transformer (a type of deep neural network specialized for sequences and attention).
- **Learning paradigm**: mainly *self-supervised learning*, a subform of unsupervised learning.
- **Objective**: predict the next word (token) given the previous ones.

Mathematically:

$$P\left(w_t \mid w_1, w_2, \ldots, w_{t-1}\right)$$

"Given this context, what's the next most probable word?". That's the only thing it learns. Everything else (reasoning, style, facts) *emerges* from learning this next-token distribution on vast text corpora.

Why are they still called "Deep Learning"? They perfectly fit the definition we discussed earlier: "Deep Learning is the learning data representation and decision functions directly from data".

- They learn **representations** of words, sentences, and even concepts automatically.
- They have **layers upon layers** (up to 100+ in GPT-4).
- They **don't rely on hand-crafted linguistic features** (no human tells them grammar rules).
- They learn everything **directly from raw text data** (syntax, semantics, even reasoning patterns).

So they exemplify:

$$\begin{gathered}\text{Learned Features (embeddings)} + \\ \text{Learned Classifier (next word predictor)}\end{gathered}$$

But at **massive scale**, with **billions of parameters** and trained on **terabytes of text**. This scale is what enables their surprising capabilities.

What makes them *different* from earlier Deep Learning. Traditional DL (e.g., CNNs, RNNs) had strong **task specialization**: CNNs for vision, RNNs for sequences, LSTMs for time series. Instead, Transformers with LLMs changed the game because they are **general-purpose learners**:

- They can handle language, code, images, audio, even multimodal data.
- Their **attention mechanism** learns relationships between all parts of the input simultaneously.

They are sometimes called: "Foundation Models", because they can be *fine-tuned* for many downstream tasks (translation, summarization, question answering, etc.).

Why do they feel intelligent? When we train on *massive data* (trillions of words) and *huge models* (hundreds of billions of parameters), the model starts showing **emergent behaviors**:

- Understanding context, humor, and nuance.
- Performing reasoning and arithmetic.
- Generating coherent, creative text.
- Translating languages fluently.
- Writing code snippets.

But still, it's pattern prediction. There is **no explicit symbolic reasoning** or understanding; it's just learned statistical structure at enormous scale. So we say: "They are **Deep Learning models**, trained on **massive dataset**, showing **emergent intelligence**".

1.5 What's Behind Deep Learning?

If the concept of neural networks exists since the 1950s, ***why did Deep Learning explode only after 2012?*** This is a natural question that comes *after* we've seen what Deep Learning is. To answer this question, we show two perspectives: the **MIT view** and **The Economist view**.

The MIT view: Computational Power

According to MIT and many early researchers, Deep Learning became possible only when **computational resources** caught up with the theory. It means that the mathematics and algorithms (backpropagation, perceptrons, convolutional nets) existed for decades, but **training deep networks** requires enormous computation:

- Millions of matrix multiplications.
- Thousands of gradient updates per sample.
- Gigantic datasets.

Before 2010, this was impractical. Around 2011-2012, **GPUs** (Graphics Processing Units) changed everything:

- They made large-scale matrix computations thousands of times faster.
- Deep learning frameworks (Theano, TensorFlow, PyTorch) made GPU computing accessible.
- Hardware parallelism allowed training networks with **hundreds of layers** instead of 3-4.

So from the MIT perspective: Deep Learning rose because **we finally had the computational power to train deep models**.

The Economist view: Big Data

In 2012, *The Economist* (yes, the famous magazine) proposed a different, and equally valid, explanation: "Deep Learning exploded because the world finally generated **enough data** to feed it". It means that the Internet, social media, smartphones, sensors, and cloud storage created **massive labeled datasets**:

- ImageNet (over 14 million labeled images).
- YouTube (millions of labeled videos).
- Text from web, Wikipedia, books, perfect for LLM pretraining.

Deep neural networks thrive on data volume: they don't generalize well with few examples. The more data, the better they learn **hierarchical representations**. So from the Economist perspective: "Deep Learning rose because **we finally had Big Data**, the fuel it needs to work".

The Real View: Both Matter

In reality, both perspectives are correct and complementary. Deep Learning's success is due to the **synergy of computational power and big data**:

- Before 2010, algorithms existed but computing was too slow and data too scarce. Then neural networks were limited to shallow architectures and small datasets.
- Around 2012, hardware (GPUs, TPUs, distributed training) made computation feasible. Simultaneously, the explosion of digital data provided the massive labeled datasets needed.

This combination triggered the **Deep Learning revolution**. The turning point was **ImageNet 2012**, where Krizhevsky, Sutskever, and Hinton demonstrated that a deep convolutional network (AlexNet) could drastically outperform traditional methods on image classification. This success was possible only because:

- They used two NVIDIA GPUs to train a deep network with millions of parameters.
- They trained on the large ImageNet dataset with 1.2M labeled images.

The result was an error rate of 15%, compared to 26% for the best traditional method. This landmark event showcased the **power of deep learning when both computational resources and big data are available**.

1.6 Summary

Everything we've seen, supervised, unsupervised, or reinforcement learning, ultimately depends on **how we represent data**. In traditional ML, features are *hand-crafted*. In Deep Learning, features are *learned automatically* through hierarchical representations. The revolution of Deep Learning wasn't new math, it was learning **what matters** in the data instead of coding it by hand.

$$\text{Success of ML} \Rightarrow \text{Success of its feature representation}$$

Deep networks just made the **representation learning** automatic and scalable.

Deep Learning = Learning Data Representation from Data

Deep Learning is not a specific architecture (like CNN, RNN, or Transformers) or algorithm. It's the **paradigm** where:

1. Input → raw data (e.g., pixels, text, audio)
2. Model → multiple non-linear layers learning internal representations.
3. Output → desired prediction/task.
4. Learning → end-to-end optimization of all layers together.

So instead of:

$$\text{Human designs features} \rightarrow \text{Model learns mapping}$$

We now have:

$$\text{Model learns both features and mapping} \rightarrow \text{directly from data}$$

This is the essence of Deep Learning: **learning hierarchical representations directly from raw data**.

"Which data?" - The key question of the course

This is the **transition line** to the rest of the course (notes). Now that we know *what* Deep Learning is, the next question is *what data we use and how*. Different data types define the upcoming sections:

Data Type	Upcoming Section
Tabular / numerical	Perceptrons & Feed-Forward NNs
Images	Convolutional Neural Networks (CNNs)
Sequential (text, time series)	Recurrent Neural Networks (RNNs) & Transformers
Unlabeled data	Autoencoders & Word Embeddings

So this question of "which data?" becomes the **roadmap** for the rest of the course (notes).

2 From Perceptrons to FNNs

2.1 Historical Context

When Artificial Intelligence first emerged as a field in the 1940s and 1950s, researchers were fascinated by the idea of creating machines that could *think*, *adapt*, and *learn* as the human brain does. At that time, traditional computers were already capable of executing precise, deterministic instructions with incredible speed. However, these **early machines lacked flexibility**: they could not interpret noisy or ambiguous input, nor could they modify their behavior from experience.

This limitation led scientists to look beyond the rigid Von Neumann architecture[2] and toward the **brain** as an alternative computational paradigm. The human brain, with its billions of interconnected neurons, represented a radically different kind of machine: **massively parallel**, **distributed**, **redundant**, and **fault-tolerant**. Each neuron is *simple*, yet together they form a system capable of extraordinary complexity and adaptability.

From this inspiration arose the idea of **neural networks**: mathematical models built from simple interconnected units that imitate, in a highly abstract way, the behavior of biological neurons. Interestingly, neural networks are not a recent invention of the deep learning era: they have existed since the birth of AI itself. In fact, the phrase "*Deep Learning is not AI, nor Machine Learning*" emphasizes that **deep learning is a later evolution within this larger historical continuum**. Neural networks have been a foundational approach to artificial intelligence from its inception, long before modern computational power and data made them successful.

In summary, the reason researchers in the 1940s and 1950s looked "beyond Von Neumann" was that they sought to create machines that could **learn from experience** and **adapt to new situations**, capabilities that traditional computers lacked:

- **1940s motivation**: classic computers excelled at precise, fast arithmetic but researches wanted systems that could **interact with noisy data**, be **parallel and fault-tolerant**, and **adapt**.
- **Brain as a computational model**: the brain offers a radically different architecture that is massively parallel, distributed, redundant system. These properties are an appealing template for computation, which inspired artificial neurons and later full neural networks.

The inception of AI

In the years immediately following the Second World War, a new scientific dream began to take shape: the idea that intelligence could be recreated in a machine. Early pioneers such as **Alan Turing**, **John von Neumann**, **Warren McCulloch**, and **Walter Pitts** laid the foundations of what would soon

[2] The sequential model where computation and memory are separated

be called *Artificial Intelligence*. Computers had just proven they could follow precise instructions and perform huge calculations at incredible speed, yet these machines were nothing more than rigid automata: they obeyed every command literally, unable to perceive, reason, or learn.

The emerging field of AI was born from the desire to bridge that gap, to make machines that could **adapt**, **generalize from experience**, and **interact intelligently** with the world. The 1940s and 1950s were therefore an era of conceptual excitement: *could the brain's mechanisms be modeled mathematically and implemented in hardware or software?* The earliest experiments sought to replicate the nervous system's structure, creating computational units that mimicked neurons and synapses. These units could, in principle, activate or remain silent depending on the inputs they received, a primitive form of reasoning.

At this stage, AI and neural networks were inseparable: **to build an intelligent machine meant to build an artificial brain**. Over the next decades, this vision would split into two main traditions. One emphasized *symbolic* reasoning (manipulating explicit rules and logic) while the other, the *connectionist* approach, pursued learning from examples through networks of simple computational nodes. The second line, though overshadowed for many years, would eventually resurface as what we now call **Deep Learning**.

From Von Neumann Machines to Brain-Inspired Models

In the 1940s, the **Von Neumann architecture** defined what we still call a *classical computer*: a machine with a central processor (CPU) that executes instructions stored in memory, step by step, following a deterministic sequence. This design is extremely powerful for arithmetic and logic, but it has key limitations when the goal is to emulate intelligence.

A Von Neumann computer is **serial**, **rigid**, and **exact**: it does exactly what it's told, line by line. Intelligence, however, requires something different, the ability to handle **noisy or incomplete data**, **recover from errors**, **adapt to change**, and **operate in parallel** on many signals at once. The human brain, in contrast, is a **massively parallel** and **distributed** system made of roughly 10^{11} neurons, each connected to thousands of others through 10^{14} to 10^{15} synapses.

This comparison motivated the idea of a **computational model inspired by the brain**. Instead of a single central processor, the brain uses huge numbers of simple processing units (neurons) working together. Each neuron performs a small, nonlinear operation, but their collective behavior gives rise to perception, reasoning, and learning.

Researchers realized that if intelligence in humans comes from these interactions, perhaps machines could become intelligent by simulating **networks of artificial neurons**, each following simple rules, but collectively capable of complex, adaptive computation.

In short:

- Von Neumann: deterministic, sequential, rigid.
- Brain-inspired: parallel, adaptive, fault-tolerant.

This shift marks the conceptual birth of **neural networks** as a new computational paradigm.

Neural Networks in the Early AI Era

The idea of taking the **human brain** as a model for computation stems from its extraordinary complexity and efficiency. A typical adult brain contains around **100 billion neurons** (10^{11}), and each neuron is connected to roughly **7'000** others, forming an estimated 10^{14} - 5×10^{14} **synapses**, even reaching 10^{15} in a three-year-old child.

Despite being slow compared to digital processors (neurons fire in milliseconds, not nanoseconds), the brain's power lies in its **massive parallelism** and **redundancy**. Each **neuron is a simple processing element**, but together they create a distributed, nonlinear, and fault-tolerant system capable of perception, reasoning, adaptation, and learning; functions that no single algorithmic machine of the 1940s could perform.

From a computational viewpoint, this means:

- **Processing is distributed**: no central control; intelligence arises from interactions.
- **Information is encoded collectively**: a concept survives even if some neurons fail.
- **Parallelism ensures speed and robustness**: thousands of operations occur simultaneously.
- **Adaptivity**: synaptic strengths (connections) change with experience, enabling learning.

These characteristics inspired the **first attempts to formalize "neurons" mathematically**, giving rise to the **perceptron** and to the field of *artificial neural networks*. The perceptron is, in essence, a simplified abstraction of how a biological neuron integrates inputs, applies a threshold, and produces an output. An idea that we'll explore in the following section.

⚖ What about the computation of biological versus artificial neurons?

🧠 In a **biological neuron**, information is transmitted through **electrochemical signals**:

- The **dendrites** receive inputs from other neurons through *synapses*.
- Each input can be **excitatory** (it increases activation) or **inhibitory** (it decreases activation).
- The neuron **integrates** all these signals in the **cell body (soma)**.
- When the total accumulated signal exceeds a **threshold**, the neuron **fires**, sending an output through its **axon** to other neurons.

Although this process is complex and involves various biochemical mechanisms, it can be summarized as:

$$\text{collect inputs} \rightarrow \text{integrate} \rightarrow \text{compare with threshold} \rightarrow \text{fire}$$

⚙ But how to model this computationally? In the **artificial version**, we simplify this biological process into a mathematical model:

$$h_j(x, w, b) = f\left(\sum_{i=1}^{I} w_i x_i - b\right) = f\left(w^T x\right)$$

Where:

- x_i are the input values (analogous to signals received by dendrites). They are like the neurotransmitter signals that a biological neuron receives from other neurons.
- w_i are the weights (analogous to synaptic strengths). They represent how strongly each input influences the neuron's activation.
- b is the bias (analogous to the threshold). It determines the level of input required for the neuron to activate.
- $f(\cdot)$ is the activation function (analogous to the firing mechanism). It decides whether the neuron fires based on the integrated input.

Each artificial neuron thus performs three main steps:

1. **Weighted sum** of its inputs (integration): $\sum_{i=1}^{I} w_i x_i$.
2. **Subtracts the bias** (thresholding): $\sum_{i=1}^{I} w_i x_i - b$.
3. **Applies the activation function** (firing decision): $f\left(\sum_{i=1}^{I} w_i x_i - b\right)$.

Definition 1: Artificial Neuron

An **Artificial Neuron** is a **mathematical model** inspired by the way a biological neuron works. It's the **basic computation unit** of a neural network.

While a real neuron collects electrical signals from thousands of connections (synapses) and "fires" if the total signal passes a threshold, an artificial neuron does the same thing, but with numbers.

Formally, it takes several inputs $(x_1, x_2, \ldots, x_I)$, multiplies each by a **weight** w_i, sums them, adds a **bias** b, and passes the result through an **activation function** $f(\cdot)$:

$$h_j(x, w, b) = f\left(\sum_{i=1}^{I} w_i x_i - b\right) = f\left(w^T x\right) \tag{1}$$

Where:

- **Inputs** (x_i): the signals coming from other neurons or from data (e.g., pixel values).
- **Weights** (w_i): how strong each input connection is (analogous to synaptic strength).
- **Bias** (b): shifts the activation threshold up or down.
- **Activation function** (f): decides whether the neuron "fires" (outputs a strong signal) or stays quiet.

In essence, the pipeline of an artificial neuron is:

Weighted sum $\rightarrow$ Threshold/Bias $\rightarrow$ Nonlinear activation $\rightarrow$ Output

Definition 2: Bias

The **Bias** is an additional parameter in an artificial neuron that allows the activation function f to be shifted horizontally, providing the model with the ability to represent patterns that do not pass through the origin.

Mathematically, it appears as the constant term b in the neuron's activation equation (see page 36):

$$a = w^T x + b$$

The bias represents the **intrinsic tendency of a neuron to activate**, even in the absence of input. It acts like a tunable threshold that controls *when* the neuron fires.

Think of the bias as the neuron's **default tendency to fire**, it decides *how easy or hard* it is for the neuron to activate:

- A **large positive bias** → neuron tends to fire even with small input.
- A **large negative bias** → neuron needs strong evidence (large input sum) to fire.

In other words, the bias *shifts the activation threshold* left or right along the input axis, allowing the neuron to learn more complex decision boundaries.

Imagine a simple rule: "*if the weighted sum of our inputs is greater than 0, we output 1*". Now suppose all our inputs are zero ($x_1 = x_2 = 0$). If we want the neuron to still fire in that case, we need a **bias** to "push" it over the threshold. Bias gives the neuron a *baseline activity*, like saying: "*even if there's not input, we are slightly inclined to fire*".

So, an **artificial neuron** mimics the logical essence of a biological one: a small computing unit that combines multiple inputs into one output, depending on the learned connection strengths (weights) and a bias term. This is the foundation of the **perceptron**, the first neural network model, the topic of the next section.

2.2 The Perceptron

2.2.1 Who Invented It?

Once researchers realized that the brain could be viewed as a network of simple processing units, the next natural step was to formalize this idea into an actual **computational model**, what we now call a **neural network**.

Definition 3: Neural Network

A **Neural Network** is simply a **collection of artificial neurons** (page 36) connected by weighted links. Each neuron:

- Receives inputs,
- Computes a weighted sum,
- Applies an activation function,
- And produces an output that becomes the input for the next neuron.

Through these connections, the network forms a structure capable of **transforming input data into meaningful outputs**, a function approximator that *learns* by adjusting its weights.

The very first implementations appeared in the 1940s-1960s, with three major milestones:

McCulloch & Pitts (1943). They proposed the **Threshold Logic Unit (TLU)**, the first mathematical model of a neuron. Each unit:

- Received multiple binary inputs,
- Multiplied them by fixed weights,
- Summed them up,
- Compared the sum to a threshold,
- Output 1 if the threshold was exceeded, 0 otherwise.

They proved that a network of such units could represent **any logical function**, meaning it could, in theory, "compute" anything if properly wired.

Frank Rosenblatt (1957). He built the first **trainable model**, the **Perceptron**. Rosenblatt's perceptron could automatically **learn** the correct weights from examples using an update rule based on errors. His prototype was implemented in hardware:

- The weights were stored as adjustable electrical components (potentiometers),
- Electric motors updated them during learning. This was the first step from theoretical neuroscience to **machine learning**.

Bernard Widrow (1960). He developed the **ADALINE (Adaptive Linear Neuron)** and later the **MADALINE (Multiple ADALINE network)**. Widrow's key idea was to express the threshold as a **bias term**, simplifying the equations and making it easier to train models using gradient-based optimization, a cornerstone of modern networks.

Together, these models represent the **first generation of neural networks**: simple, linear systems inspired by the brain but operating with mathematics and electricity. They laid the groundwork for the more complex architectures that would follow, leading to the deep learning revolution we see today.

2.2.2 Mathematical Model & Logical Operations

The **Perceptron** is the **simplest neural network**, a single neuron that transforms multiple input signals into one output through a weighted sum and a thresholding function.

Formally, given inputs:

$$x = [x_1, x_2, \ldots, x_I]$$

And weights:

$$w = [w_1, w_2, \ldots, w_I]$$

The perceptron computes the quantity:

$$a = \sum_{i=1}^{I} w_i x_i + b = w^T x + b \tag{2}$$

Where:

- x_i are the input features,
- w_i are the learnable connection weights,
- b is the **bias** (representing the firing threshold).

Then, this **activation** a passes through a **step function** (also called **threshold** or **activation function**) to produce the final output y:

$$y = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

In some conventions, the output can also be -1 or $+1$ instead of 0 and 1, depending on how the data is encoded.

Sometimes, we include the bias directly as a weight w_0 associated with a fixed input $x_0 = 1$, rewriting the equations as:

$$y = f\left(w_0 x_0 + w_1 x_1 + \ldots + w_I x_I\right) = f\left(w^T x\right) \tag{4}$$

This makes formulas simpler and more uniform for training algorithms (compact vector notation).

Interpretation of the Perceptron math

The perceptron divides the input space into two regions separated by a **decision boundary** (a hyperplane[3]). If the weighted sum of inputs exceeds the threshold, the neuron "fires" (outputs 1); otherwise, it stays silent (outputs 0). Thus,

[3] A **hyperplane** is a **generalization of a line or a plane** to any number of dimensions. It's the mathematical way to describe a *flat surface* that separates space into two parts. In 1D, a hyperplane is just a point that splits the line into two halves; in 2D, it's a line that divides the plane into two regions, one where the perceptron outputs 1 and the other where it outputs 0; in 3D, it's a plane that separates space into two halves. In higher dimensions, it remains a flat subspace that partitions the input space.

the perceptron acts as a **linear classifier**: it determines which side of the hyperplane the input vector lies on.

We can express the **exact set of points where the neuron is undecided** (the **Decision Boundary Equation**) by setting the activation a to zero:

$$w^T x + b = 0 \quad \text{(decision boundary equation)} \tag{5}$$

What it can actually do: Logical Operations

Now, if the perceptron is a computational unit, *what kind of computations can it perform?* To answer that, we need simple, well-defined **functions** to test it on. The most basic functions are the **logical operations** used in Boolean algebra. Logical operations (like AND, OR, NOT) are perfect because:

- They have **binary inputs** (0 or 1), exactly like neuron activations.
- They produce **binary outputs** (true or false), like the perceptron's step function.
- They let us see immediately whether the neuron can separate input cases correctly.

Logical operations are the **first experiments** that show the perceptron's power as a *linear classifier*.

When the perceptron can reproduce logic operations like AND or OR, it proves that:

1. A single neuron can implement **decision-making**.
2. The model is capable of **classification** (separating inputs into categories).
3. We can assign **geometric meaning** (a hyperplane dividing true/false examples).

Example 1: Logical OR ($\vee$)

x_1	x_2	$y = x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

We want the perceptron to output **1** if *any* input is 1. A possible set of parameters is:

$$w_1 = 1, \quad w_2 = 1, \quad b = -0.5$$

This gives us the activation function:

$$a = w_1 x_1 + w_2 x_2 + b = x_1 + x_2 - 0.5$$

Or equivalently:

$$y = \begin{cases} 1 & \text{if } x_1 + x_2 - 0.5 > 0 \\ 0 & \text{otherwise} \end{cases}$$

So each neuron computes:

$$y = f\left(w_1 x_1 + w_2 x_2 + b\right) = f\left(1 \cdot x_1 + 1 \cdot x_2 - 0.5\right)$$

Checking all input combinations:

- For $(0,0)$: $a = 0 + 0 - 0.5 = -0.5 \Rightarrow y = 0$
- For $(0,1)$: $a = 0 + 1 - 0.5 = 0.5 \Rightarrow y = 1$
- For $(1,0)$: $a = 1 + 0 - 0.5 = 0.5 \Rightarrow y = 1$
- For $(1,1)$: $a = 1 + 1 - 0.5 = 1.5 \Rightarrow y = 1$

Thus, the perceptron correctly implements the OR function.

Example 2: Logical AND ($\wedge$)

x_1	x_2	$y = x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

We want the perceptron to output **1** only if *both* inputs are 1. A possible set of parameters is:

$$w_1 = 1, \quad w_2 = 1, \quad b = -1.5$$

This gives us the activation function:

$$a = w_1 x_1 + w_2 x_2 + b = x_1 + x_2 - 1.5$$

Or equivalently:

$$y = \begin{cases} 1 & \text{if } x_1 + x_2 - 1.5 > 0 \\ 0 & \text{otherwise} \end{cases}$$

So each neuron computes:

$$y = f\left(w_1 x_1 + w_2 x_2 + b\right) = f\left(1 \cdot x_1 + 1 \cdot x_2 - 1.5\right)$$

Checking all input combinations:

- For $(0,0)$: $a = 0 + 0 - 1.5 = -1.5 \Rightarrow y = 0$
- For $(0,1)$: $a = 0 + 1 - 1.5 = -0.5 \Rightarrow y = 0$
- For $(1,0)$: $a = 1 + 0 - 1.5 = -0.5 \Rightarrow y = 0$

- For $(1, 1)$: $a = 1 + 1 - 1.5 = 0.5 \Rightarrow y = 1$

Thus, the perceptron correctly implements the AND function. However, we can see that other weight/bias combinations could achieve the same result. For example:

$$w_1 = 1.5, \quad w_2 = 1.5, \quad b = -2.0$$

In both examples, the perceptron defines a **line (in 2D)** that separates input combinations giving output 1 from those giving output 0. For OR, the line lies closer to the origin, since only $(0, 0)$ should give 0; for AND, the line lies further away, since only $(1, 1)$ should give 1. So, by adjusting weights and bias, the perceptron can learn to classify inputs according to these logical rules. However, it's clear that **manually setting weights and biases for complex tasks is impractical**. This brings us to the next important topic: ***how can it learn those weights automatically instead of us setting them by hand?***

2.2.3 Hebbian Learning Rule

Now that we understand what the Perceptron does and who invented it, let's explore **how it learns** from data. When the first artificial neurons were proposed, researchers wanted them not just to compute, but to **learn from experience**, as biological neurons do. The earliest and most influential idea for this was the **Hebbian Learning Rule**, introduced by psychologist **Donald Hebb** in 1949.

The biological intuition

Donald Hebb was a psychologist, not a mathematician. In 1949, he was trying to explain **how the brain learns from experience**, without having explicit "teachers" or formulas. He observed that, in biological brains, learning seems to happen **through association**. That's the origin of his famous sentence:

"*Cells that fire together, wire together.*"

This means that if **two neurons** are **active at the same time** (one sending a signal and the other firing) then the **connection** (synapse) between them should **become stronger**. Over time, the brain reinforces useful associations automatically.

In other words, if neuron A consistently helps activate neuron B, the connection from A to B should be strengthened. This principle is thought to underlie learning and memory formation in the brain.

The Artificial Version: Mathematical Formulation

Now, we translate this biological intuition into a mathematical rule that can be applied to the Perceptron. In artificial neurons, "firing" means *output* is active (e.g., output is 1). So if both input and output are active at the same time, that's equivalent to "they fired together". The Hebbian learning rule says:

- **Increase** the weight of connections that are active when the neuron fires.
- **Decrease** or leave unchanged the connections that are inactive or misaligned.

To translate this into a mathematical rule for a Perceptron, we **express the weight update** as follows:

$$\Delta w_i = \eta \cdot x_i \cdot t \tag{6}$$

- If $x_i > 0$ (input is active) and $t > 0$ (target output is active), both are active, then Δw_i is positive, so the weight w_i **increases**, the connection strengthens.
- If $x_i > 0$ (input is active) but $t \leq 0$ (target output is inactive), mismatch, then Δw_i is zero or negative, so the weight w_i **decreases** or remains the same, the connection weakens.
- If $x_i \leq 0$ (input is inactive), regardless of t, then no update occurs since Δw_i is zero, the connection remains unchanged.

Where:

- η is the **learning rate**, a small positive constant that controls how much the weights are adjusted during each update. It ensures that learning is gradual and stable. To make an analogy, think of η as the **speed limit** on a road: it prevents the learning process from speeding ahead too quickly and potentially crashing (i.e., diverging).
- x_i is the i^{th} **input value** to the Perceptron.
- t is the **target output** (desired response) for the given input.
- Δw_i is the **change in weight** for the i^{th} input. This change is added to the current weight w_i to get the new weight.

The full update rule becomes:

$$w_i^{(k+1)} = w_i^{(k)} + \Delta w_i = w_i^{(k)} + \eta \cdot x_i \cdot t \tag{7}$$

This is the **Weight Update Rule**. It tells us *how to modify* each connection w_i after seeing one training example. Conceptually, at each learning step (each training example):

1. **Take the current weights** $w_i^{(k)}$.
2. **Compute** how much they should change $\Delta w_i = \eta \cdot x_i \cdot t$.
3. **Add that change** to get the new weights $w_i^{(k+1)}$.

How it works

1. **Initialize** all weights w_i to small random values (or zeros).
2. **Set** the learning rate η to a small positive value (e.g., 0.01).
3. For each **training example** (x, t):
   - **Compute the Perceptron's output** y using the current weights:
   $$y = f\left(w^T x\right)$$
   - **Compare with the target** t.
     - If $y = t$, the output y matches the target t, the neuron is already correct, so **no weight update is needed** since the association is already learned.
     - If $y \neq t$, the output y does not match the target t, the neuron is incorrect, and we need to **update the weights** to strengthen the association. This is done using the Hebbian learning rule:
     $$w_i^{(k+1)} = w_i^{(k)} + \eta \cdot x_i \cdot t$$
     This can be explained in informal steps:
       * For each weight w_i, compute the change $\Delta w_i = \eta \cdot x_i \cdot t$
       * Update the weight: $w_i \leftarrow w_i + \Delta w_i$
4. Repeat until all examples are correctly classified or a stopping criterion is met (e.g., a maximum number of iterations).

Example 3: Hebbian Learning Rule

Let's say we're learning a simple OR function with two inputs x_1 and x_2. The target outputs t for the four possible input combinations are:

- $x = [0, 0] \rightarrow t = 0$
- $x = [1, 0] \rightarrow t = 1$
- $x = [0, 1] \rightarrow t = 1$
- $x = [1, 1] \rightarrow t = 1$

We do not include a bias term in this example for simplicity. We'll use a step activation function:

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

The algorithm proceeds as follows:

1. **Initialize weights** $w_1 = 0.0$, $w_2 = 0.0$ and learning rate $\eta = 0.1$.
2. **First training example** $x = [0, 0]$, $t = 0$:
   - **Compute output**: $y = f(0.0 \cdot 0 + 0.0 \cdot 0) = f(0) = 0$
   - Output matches target, so **no weight update needed**.
3. **Second training example** $x = [0, 1]$, $t = 1$:
   - **Compute output**: $y = f(0.0 \cdot 0 + 0.0 \cdot 1) = f(0) = 0$
   - Output does not match target, so we **update weights**:

$$\begin{aligned} \Delta w_1 &= 0.1 \cdot 0 \cdot 1 = 0.0 \\ \Delta w_2 &= 0.1 \cdot 1 \cdot 1 = 0.1 \\ w_1 &\leftarrow 0.0 + 0.0 = 0.0 \\ w_2 &\leftarrow 0.0 + 0.1 = 0.1 \end{aligned}$$

4. **Third training example** $x = [1, 0]$, $t = 1$:
   - **Compute output**: $y = f(0.0 \cdot 1 + 0.1 \cdot 0) = f(0) = 0$
   - Output does not match target, so we **update weights**:

$$\begin{aligned} \Delta w_1 &= 0.1 \cdot 1 \cdot 1 = 0.1 \\ \Delta w_2 &= 0.1 \cdot 0 \cdot 1 = 0.0 \\ w_1 &\leftarrow 0.0 + 0.1 = 0.1 \\ w_2 &\leftarrow 0.1 + 0.0 = 0.1 \end{aligned}$$

5. **Fourth training example** $x = [1, 1]$, $t = 1$:
   - **Compute output**: $y = f(0.1 \cdot 1 + 0.1 \cdot 1) = f(0.2) = 1$
   - Output matches target, so **no weight update needed**.

After one pass through the training data, the weights are $w_1 = 0.1$ and $w_2 = 0.1$. Repeating this process over multiple epochs will further refine the weights until the Perceptron correctly models the OR function.

Should the bias be updated if the output doesn't match the target?

In the Hebbian learning rule, the **bias term can also be updated similarly to the weights**. The bias can be treated as a weight connected to an input that is always 1. Therefore, if the output does not match the target, the bias should also be updated to help correct the output. The update rule for the bias b would be:

$$\Delta b = \eta \cdot x_0 \cdot t = \eta \cdot 1 \cdot t = \eta \cdot t \qquad x_0 = 1 \tag{8}$$

So, if the **output** is **incorrect**, the bias would be adjusted by adding Δb to the current bias value:

$$b^{(k+1)} = b^{(k)} + \Delta b = b^{(k)} + \eta \cdot t \tag{9}$$

This adjustment helps shift the activation threshold of the Perceptron, making it more likely to produce the correct output in future iterations.

In other words, we can think of the **bias** as a *special weight* w_0 that connects to a *constant input* $x_0 = 1$. This trick lets us treat the bias **exactly the same** as all the other weights in the update rule. Therefore, the neuron computes:

$$y = f\left(w_0 \cdot 1 + w_1 x_1 + w_2 x_2 + \dots\right)$$

And the update rule applies uniformly to **every** w_i, including w_0 (the bias):

$$w_i^{(k+1)} = w_i^{(k)} + \eta \cdot x_i \cdot t \quad \text{for } i = 0, 1, 2, \dots$$

Thus, at every iteration:

- The **normal weights** $(w_1, w_2, \dots)$ adapt based on the input features and target output.
- The **bias** b (also considered a weight, like w_0) is updated to help the Perceptron better fit the data. It adapts based on the target output t alone, since its associated input is always 1 (i.e., $x_0 = 1$).

The bias learns to **adjust the overall tendency** of the neuron to fire. If the network often needs to output 1 (positive target), the bias weight increases, making it easier for the neuron to activate. Conversely, if the network often needs to output 0 (negative target), the bias weight decreases, making it harder for the neuron to activate. This dynamic adjustment of the bias is crucial for the Perceptron to learn effectively from data.

2.2.4 Perceptron as Linear Classifier

A **classifier** is a model that assigns input data points to one of several classes. In the case of the perceptron, it classifies input vectors into two classes based on a linear decision boundary.

A **linear classifier** is a type of classifier that makes its decisions based on a linear combination of the input features. In poor words, it makes a decision by checking on which side of a *line* (in 2D), *plane* (in 3D), or *hyperplane* (in higher dimensions) the input data point lies.

The perceptron computes:

$$a = w^T x + b$$

where w is the weight vector, x is the input vector, and b is the bias term, and decides:

$$y = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a \leq 0 \end{cases} \tag{10}$$

So the **decision** happens depending on the *sign* of a: positive values lead to class 1, while zero or negative values lead to class 0.

The **Decision Boundary** is the exact set of points where the model is **undecided**, where it switches from one class to the other. That happens precisely when the condition changes sign from negative to positive. The "border" between those two cases is when the activation a equals **zero**. Formally, this occurs when:

$$w^T x + b = 0$$

That's where the perceptron's decision flips, and therefore it's the **boundary line (or hyperplane)**. This boundary divides the input space into two halves:

- Points where $w^T x + b > 0$ are classified as class 1.
- Points where $w^T x + b < 0$ are classified as class 0.
- Points where $w^T x + b = 0$ lie exactly on the decision boundary.

Wait, why is zero special? In the above equation (10), the perceptron outputs 0 when $a = 0$. Why is it called the decision boundary? In theory, the **boundary**:

$$w^T x + b = 0$$

Is **not assigned to any class**, it's the **limit** between them. Exactly on the boundary ($a = 0$), the model *is indifferent*, because **geometrically** that point is the **separator**, not really part of any region (see Figure 2, page 49, to visualize this concept). However, in practice, the $\leq$ sign in the perceptron decision rule is just a **tie-breaking rule**, otherwise we wouldn't know what to output when $a = 0$. But for geometry and theory, we're interested in **where the switch happens**, so we call the exact set of points the **decision boundary**.

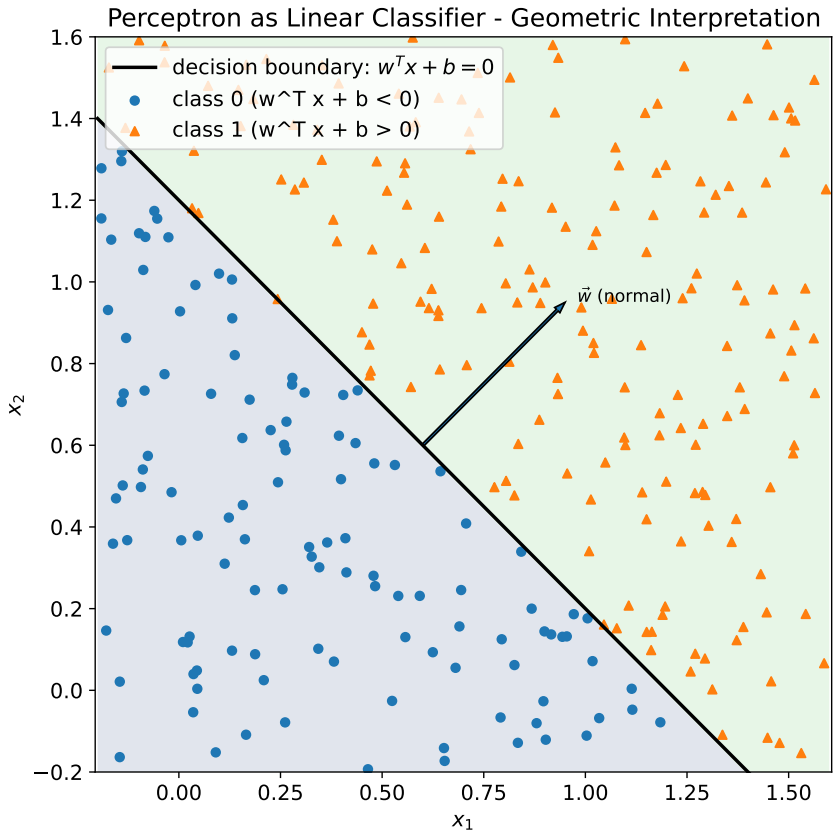


Figure 2: A 2D example of a perceptron as a linear classifier. The line represents the decision boundary where $w^T x + b = 0$. Points on one side of the line are classified as class 1 (green area, orange triangles, everything that satisfies $w^T x + b > 0$), while points on the other side are classified as class 0 (blue, $w^T x + b < 0$). The arrow indicates the **normal vector** $\vec{w}$, which is perpendicular to the decision boundary and points towards the class-1 side. The normal vector $\vec{w}$ points in the direction where the perceptron output increases.

Concept	**Meaning**
w	Defines the *direction* of the separating hyperplane.
b	Shifts the hyperplane from the origin.
$w^T x + b = 0$	Equation of the decision boundary (hyperplane).
$w^T x + b > 0$	Region classified as class 1.
$w^T x + b < 0$	Region classified as class 0.
$\vec{w}$	Normal vector to the decision boundary, indicating the direction of increasing output.
Limitation	Can only classify linearly separable data.

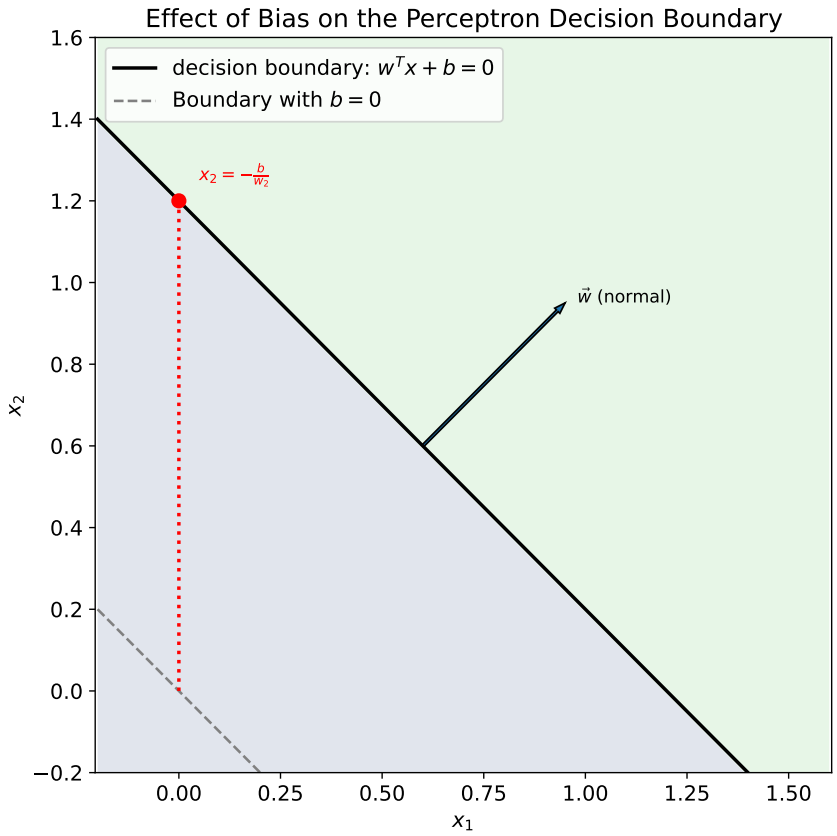


Figure 3: Geometric interpretation of the bias in a perceptron. The solid black line shows the decision boundary $w^T x + b = 0$ for $b = -1.2$, while the dashed gray line represents the case $b = 0$. The red dotted segment highlights the vertical shift of the intercept caused by the bias. The normal vector $\vec{w}$ is perpendicular to the boundary and points toward the region where the neuron output is 1 ($w^T x + b > 0$).

❓ If the bias is negative, why does the boundary shift upwards?
Imagine $w = [1, 1]$. Then $w^T x + b = x_1 + x_2 + b$. Without bias ($b = 0$), the boundary is:

$$x_1 + x_2 = 0$$

Is the **line through the origin** at a 45-degree angle. Now, if we **add** $b = -1.2$, the boundary becomes:

$$x_1 + x_2 - 1.2 = 0 \quad \Rightarrow \quad x_1 + x_2 = 1.2$$

This line is **shifted upwards** because for any given x_1, x_2 must be larger to satisfy the equation. Thus, a **negative bias** shifts the decision boundary **upwards**, while a **positive bias** would shift it **downwards**. In this case, for x_2 direction, the bias effectively **increases** the threshold that x_2 must reach to cross the boundary:

$$x_2 = -x_1 + 1.2$$

2.2.5 Boolean Operators & Linear Separability

Once we've seen that a perceptron can learn **logical functions** (like AND, OR), the next natural question is:

"*Can it learn **all** possible logical operators?*"

Short answer: **No**. And understanding why leads to the crucial idea of **linear separability**: the key limitation of the perceptron model.

Let's summarize the four fundamental binary logical functions (i.e., functions with two binary inputs and one binary output):

Operator	**Output = 1 when...**	**Linearly separable?**
AND	both inputs are 1	✔ Yes
OR	at least one input is 1	✔ Yes
NAND	at least one input is 0	✔ Yes
NOR	both inputs are 0	✔ Yes
XOR	exactly one input is 1	✖ No
XNOR	both inputs are the same	✖ No

Note that the first four operators (AND, OR, NAND, NOR) are all **linearly separable**, while the last two (XOR, XNOR) are **not**. But what does "linearly separable" mean in this context?

The game changer: *Linear Separability*

Definition 4: Linearly Separable

A dataset is **Linearly Separable** if there **exists** a straight line (in 2D), plane (in 3D), or **hyperplane** (in higher dimensions) that **perfectly divides** the **two classes of data points**. That is, all points of one class lie on one side, and all points of the other class lies on the opposite side.

Formally, given a dataset with two classes, it is linearly separable if there exist weights $w_1, w_2, \ldots, w_n$ and a bias b such that for every data point $(x_1, x_2, \ldots, x_n)$:

$$\begin{cases} w_1x_1 + w_2x_2 + \ldots + w_nx_n + b > 0 & \text{if the point belongs to Class 1} \\ w_1x_1 + w_2x_2 + \ldots + w_nx_n + b < 0 & \text{if the point belongs to Class 2} \end{cases}$$

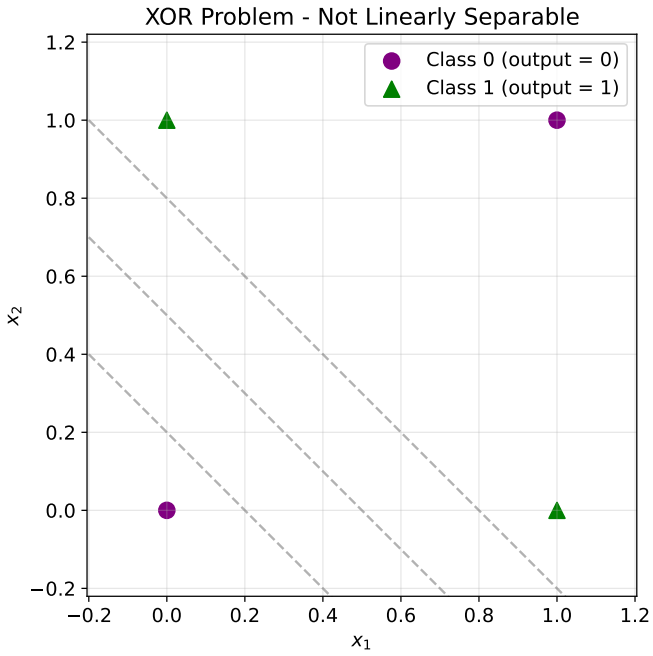
If such weights and bias exist, the dataset is linearly separable. Otherwise, no single perceptron can solve it (i.e., classify it correctly).

⚠ The XOR problem - The classic example of non-linear separability

Until now, we've seen that perceptrons can learn linearly separable functions like AND and OR. However, the linear separability limitation becomes evident when we consider some logical functions, such as XOR (exclusive OR). A little reminder of the XOR truth table:

x_1	x_2	XOR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	0

The XOR function outputs 1 only when exactly one of its inputs is 1. If we plot the input-output pairs of the XOR function on a 2D plane, we get the following points:



Here, the points are arranged in an "X" pattern:

- Class 1 points are at (0, 1) and (1, 0) (opposite corners).
- Class 0 points are at (0, 0) and (1, 1) (remaining corners).

No single straight line can separate the Class 1 points from the Class 0 points. We'd need *two lines* forming a region (a non-linear boundary). Hence, the XOR function is **not linearly separable**, and a single-layer perceptron cannot learn it.

In summary, the perceptron can only create **linear decision boundaries**, so:

- ✅ It perfectly models **linearly separable** problems (like AND, OR, simple threshold rules).
- ❌ If fails for **non-linearly separable** problems (like XOR, parity, circle-vs-ring, etc.).

This realization in the 1960s led to what's often called the "**AI winter**," as researchers recognized the limitations of single-layer perceptrons. However, this challenge also paved the way for the development of **multi-layer neural networks** (and backpropagation), which can overcome these limitations by creating complex, non-linear decision boundaries, combining multiple perceptrons in layers.

2.3 Feed-Forward Neural Networks (FNNs)

2.3.1 Architecture

After discovering that a single perceptron can only draw **one straight boundary**, researchers realized that we need **multiple layers** of neurons to build **non-linear decision surfaces**. That's how **Feed-Forward Neural Networks (FNNs)** were born.

Definition 5: Feed-Forward Neural Networks (FNNs)

A **Feed-Forward Neural Network (FNN)** is an artificial neural network where information **flows in one direction only**, from the input layer, through any hidden layers, to the output layer. There are no cycles or loops in the network.

$$x \rightarrow \text{Layer 1} \rightarrow \text{Layer 2} \rightarrow \ldots \rightarrow \text{Output Layer}$$

Each layer receives signals from the previous layer, processes them using weighted connections and activation functions, and passes the output to the next layer.

Structure of a Feed-Forward Network

An FNN is composed of:

1. **Input Layer**: one neuron per input feature (e.g., pixel value, sensor reading). Does not perform computation, it simply distributes inputs to the next layer.
2. **Hidden Layer(s)**: Contain neurons that each compute:

   $$a_j = f\left(w_j^T x + b_j\right)$$

   where w_j are the weights, b_j is the bias, x is the input vector from the previous layer, and f is the nonlinear activation function (sigmoid, tanh, ReLu, etc.). The index j identifies the specific neuron in the hidden layer. Each neuron learns different **intermediate features** of the data.
3. **Output Layer**: Produces the network's final result. Activation depends on the task, we mean for classification we often use **softmax** or **sigmoid**, while for regression we use a **linear** activation.

The connections between neurons are **weighted**:

- Each neuron in layer l is connected to **all** neurons in the previous layer $l-1$. This is called a **fully connected** or **dense** layer.
- Every connection has its own **weight**, which is learned during training. Every neuron also has a own **bias** term.
- During training, all these weights and biases are adjusted to minimize the difference between the predicted output and the actual target values.

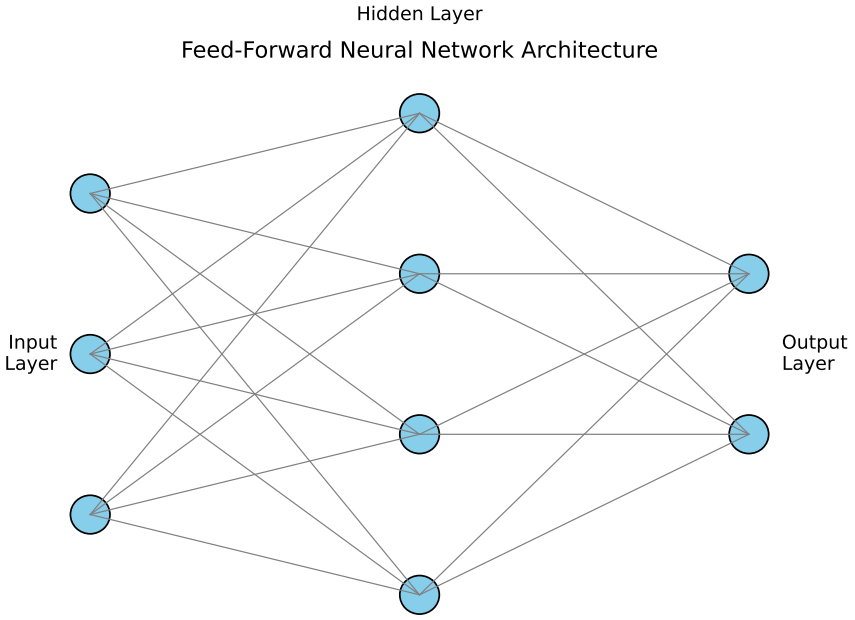


Figure 4: Architecture of a simple feed-forward neural network. Each layer is fully connected to the next one. Signals flow in one direction (input → hidden → output) without feedback connections.

FNNs can be represented as graphs based on this architecture:

- **Nodes** represent neurons.
- **Edges** represent weighted connections between neurons.

This graph representation helps visualize the network's structure and understand how information propagates through it.

Mathematically, for a layer l:

$$\begin{cases} a^{(l)} = f\left(W^{(l)} a^{(l-1)} + b^{(l)}\right) & \text{for hidden layers} \\ x^{(l)} = f\left(a^{(l)}\right) & \text{for output layer} \end{cases}$$

where:

- $a^{(l)}$ is the activation vector of layer l.
- $W^{(l)}$ is the weight matrix connecting layer $l-1$ to layer l.
- $b^{(l)}$ is the bias vector for layer l.
- f is the activation function.
- $x^{(l)}$ is the final output of the network.

This formalism allows us to **compute the output of the network given an input vector by sequentially applying these transformations layer by layer**.

How FNNs learn hierarchical features

Adding layers lets the network learn **hierarchical representations**:

- The first layers capture **simple patterns** (e.g., edges in images).
- Deeper layers combine these simple patterns into **more complex features** (e.g., shapes, objects).

This ability to build abstractions through depth is the essence of **Deep Learning**.

2.3.2 Activation Functions

Every neuron computes a **weighted sum** of its inputs and bias, called **Net Input** or **Activation Potential**:

$$a = w^T x + b \tag{11}$$

Up to this point, everything is **linear**. If we stopped here and used $y = a$ as the output, the neuron would just perform a *linear transformation*. So, we need to add some **non-linearity** to the neuron's output (a sort of *magic ingredient*) to allow the network to learn complex patterns. Without it, the entire neural network would collapse into a single linear operation: a big matrix multiplication. Remember that we come from the perceptron, which used a step function as activation and complex patterns, like XOR, could not be learned without non-linearity.

So we introduce the concept of **Activation Function**. A neuron takes the net input a computed above, and then applies an **activation function** $f(a)$ to produce its final output:

$$y = f(a)$$

The **Activation Function** defines **how the neuron "fires"**, i.e., how it transforms the raw input signal into an output that will be passed to the next layer.

Why do we really need activation functions?

Activation functions are what give neural networks their power and flexibility. Three main reasons why they are essential:

1. **To *break linearity***. Without it, the entire neural network would collapse into a single linear operation. Mathematically:

   $$\text{If } g(a) = a, \text{ then } g(W^{(2)} g(W^{(1)} x)) = W^{(2)} W^{(1)} x$$

   This is *still linear*, no matter how many layers we stack. We'd just have one big matrix multiplication. So, without $g(\cdot)$, the network couldn't model *curves*, *XOR*, *images*, *language*, or anything nonlinear.

2. **To allow *complex decision boundaries***. Think of a perceptron. With only linear operations, it can only separate data using a straight line (or plane, or hyperplane). By inserting a nonlinear $g(a)$, hidden neurons can **bend** the space (combine multiple lines to form curved boundaries). For example, a simple linear neuron can only separate classes with a single line, while nonlinear activations (like sigmoid or ReLU) allow the network to create complex, curved, multi-region separations.

3. **To *control output range***. Activation functions can also **squash** values:

Function	Formula	Output Range	Typical Use
Sigmoid	$\frac{1}{1+e^{-a}}$	$(0, 1)$	Binary classification output layers.
Tanh	$\frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$	$(-1, 1)$	Hidden layers in some networks.
Linear	a	$(-\infty, +\infty)$	Regression output layers.

That's why we use sigmoid/tanh in hidden layers or output layers for classification (to get probabilities between 0 and 1), while linear activations are used in regression tasks (to allow any real-valued output).

Exist many different activation functions, each with its own characteristics and use cases. The choice of activation function can significantly impact the performance and capabilities of a neural network. In the following, we will explore some of the most commonly used activation functions in neural networks.

In the next sections, we will mention the derivative result and the range of each activation function:

- During training, neural networks learn by **minimizing a loss function**, and this requires **backpropagation**, which is based entirely on **derivatives**. We will explain backpropagation later, but for now, here is a brief overview of how it works: (1) each neuron has parameters w_i (weights) and b (bias); (2) to adjust them, we compute how the **loss** changes if we slightly change each parameter; (3) mathematically, that's done through **gradients**, the derivatives of the loss with respect to the weights. When we apply the **chain rule** to compute these gradients, we get something like:
$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial a} \cdot \frac{\partial a}{\partial w_i}$$
Where $\frac{\partial y}{\partial a}$ is the derivative of the activation function $f(a)$. Therefore, having an activation function: **too small** $(= 0)$, means gradients vanish and the learning stops; **too large** gradients can cause instability. Hence, the derivative of the activation function is crucial for effective learning.

- The **range** of an activation determines what kind of outputs each neuron can produce, and this affects: (1) **how the next layer receives data**, and **how easy it is to train** the network. For example, if an activation function outputs values in a limited range (like between 0 and 1), it can help keep the network's outputs stable and prevent extreme values that could lead to numerical issues during training.

2.3.2.1 Linear

The **Linear Activation Function** is the simplest activation function, defined as:

$$f(a) = a \tag{12}$$

That means the neuron's output equals its input; there's no distortion or thresholding. So the neuron is just a **weighted sum** followed by nothing.

Intuitive interpretation

If all neurons in a network use $f(a) = a$, then every layer just performs a **linear transformation** of the input. Stacking **multiple linear layers doesn't add any expressive power**; the entire **network can be reduced to a single linear transformation**. In general, for a network with n layers, each represented by a weight matrix W_i, the overall transformation is:

$$f\left(W_n\left(W_{n-1}\left(\ldots W_2\left(W_1 x\right)\ldots\right)\right)\right) = \left(W_n W_{n-1}\ldots W_2 W_1\right) x$$

However, if the network only uses linear activation functions, then it simplifies to:

$$f\left(W_2\left(W_1 x\right)\right) = \left(W_2 W_1\right) x$$

Therefore, linear activation functions are rarely used in practice for hidden layers, as they cannot capture complex patterns in data. In other words, a purely linear network cannot learn anything more complex than a straight boundary; it is basically a big matrix multiplication because all the layers collapse into one.

Property	Description
Formula	$f(a) = a$
Derivative	$f'(a) = 1$
Range	$(-\infty, +\infty)$
Nonlinear?	✗
Typical use	Regression output layers (not hidden neurons)

When to use it

Even though a **linear activation** is useless inside hidden layers (because it doesn't add nonlinearity), it's still **important at the output layer** of certain models:

- **Regression problems**: when we want a real-valued output (like predicting house prices), a linear activation allows the network to produce any value in the range $(-\infty, +\infty)$. However, the linear activation is applied only at the output layer, while hidden layers use nonlinear activations to capture complex patterns.
- **Autoencoders or embedding layers**: sometimes the linear activation helps maintain continuous representations of data.

In summary, the **linear activation** keeps the output proportional to the input. It's mathematically simple and differentiable, but **does not allow the network to model nonlinear relationships**. Hence, not used in hidden layers.

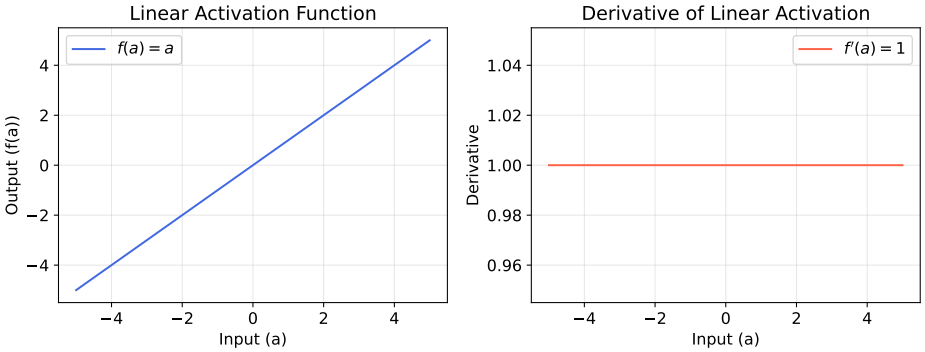


Figure 5: Linear activation $f(a) = a$ and its derivative. The function is the identity (a straight line, showing that the neuron outputs exactly what it receives), and its constant derivative $f'(a) = 1$ allows perfect gradient flow. However, being linear, it adds no expressive power to the network.

2.3.2.2 Sigmoid

The **Sigmoid Activation Function** (or **Logistic Activation Function**) is defined as:

$$f(a) = \frac{1}{1 + e^{-a}} \tag{13}$$

It "**squashes**" any real-valued **input** a **into a range** between **0 and 1**.

The Sigmoid converts its input into something that looks like a **smooth threshold**:

- ⇈ Large positive inputs a produce outputs close to 1;
- ⇊ Large negative inputs a produce outputs close to 0;
- ≈ Inputs a close to 0 produce outputs close to 0.5

It's often described as giving a "**firing probability**" to a neuron, mimicking how biological neurons activate gradually rather than with a hard step.

Graphically, the Sigmoid function is a smooth **S-shaped** curve (sigmoidal). It's **continuous** and **differentiable everywhere**. It has a gentle slope around 0 and saturates near the extremes (0 or 1).

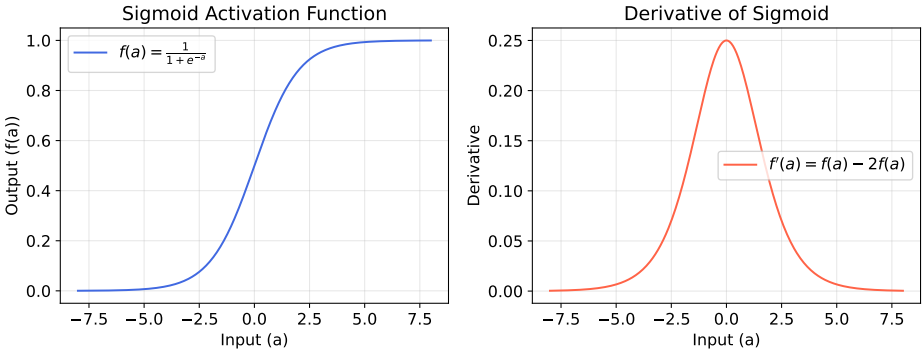


Figure 6: Sigmoid Activation Function and its derivative. The sigmoid introduces smooth nonlinearity and maps inputs into $(0, 1)$, but its derivative vanishes for large inputs, causing slow learning in deep networks.

The **derivative** tells us how sensitive the neuron's output is to changes in its input. For the Sigmoid function, the derivative is given by:

$$f'(a) = f(a) - 2f(a) = f(a) \cdot [1 - f(a)] \tag{14}$$

This means:

- When $f(a) \approx 0.5$, the derivative is maximized at 0.25, allowing for significant weight updates during training (**neuron is responsive**).
- When $f(a) \approx 0$ or $f(a) \approx 1$, the derivative approaches 0, leading to very small weight updates (**neuron is saturated** and gradients vanish).

  This **vanish gradient problem** makes deep networks with Sigmoid activations **hard to train**, as gradients become very small in earlier layers

during backpropagation. In other words, the Sigmoid function can cause **slow learning** in deep networks due to its saturating behavior at extreme input values.

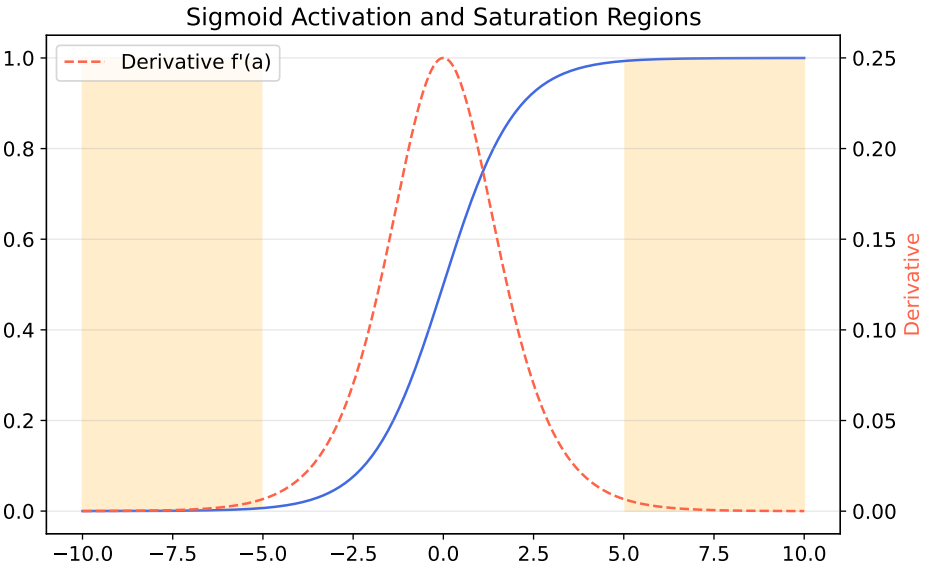


Figure 7: Sigmoid activation and saturation regions.

In figure 7 we can see the **vanish gradient problem** in action: when neurons saturate, their gradients vanish, making it hard for the network to learn from data during training.

- The blue curve shows the **sigmoid activation** $f(a)$. That smooth **S-shaped curve** (in blue) represents:

$$f(a) = \frac{1}{1 + e^{-a}}$$

  In the center (around $a = 0$), the output is about 0.5, and the curve is **steepest**; for large positive $a > 5$, the curve **flattens near 1**; for large negative $a < -5$, it **flattens near 0**. Those flat tails are the **saturation regions** (highlighted in orange). These regions mean that when the neuron receives very strong positive or negative inputs, its output doesn't change much anymore; it has reached its "max" or "min" activation.

- The orange curve are the parts of the curve where the output is **almost constant**:
  - On the left (for large negative a), the output is very close to 0 (saturated low);
  - On the right (for large positive a), the output is very close to 1 (saturated high).

  In those regions:

$$\frac{\partial f}{\partial a} = f'(a) \approx 0$$

So the neuron has **stopped responding**, even big changes in a cause almost no change in the output $f(a)$.

- The red curve shows the **derivative** $f'(a)$:

$$f'(a) = f(a) \cdot (1 - f(a))$$

The derivative is only significant in a **small central region** (roughly between -3 and 3). Outside this range, the derivative **drops to near zero**, indicating that the neuron is **saturated** and **not learning effectively**.

Property	**Value / Meaning**
Formula	$f(a) = \dfrac{1}{1+e^{-a}}$
Range	$(0, 1)$
Derivative	$f'(a) = f(a) \cdot (1 - f(a))$
Output interpretation	Probability or "firing strength".
Pros	Smooth, differentiable, bounded output, probabilistic interpretation.
Cons	Vanishing gradients for large a, outputs not zero-centered, computationally expensive.

When to use it

- **Output layer of binary classification** networks, where outputs represent probabilities:

$$\mathbb{P}(y = 1 \mid \mathbf{x}) = f(w^T \mathbf{x} + b)$$

- Historically used in hidden layers (in early networks), but now often replaced by ReLU or its variants due to vanishing gradient issues.

In summary, the Sigmoid activation function is essentially a **soft version** of the perceptron's step function:

$$\text{step: } f(a) = \begin{cases} 1 & \text{if } a \geq 0 \\ 0 & \text{if } a < 0 \end{cases} \quad \longrightarrow \quad \text{sigmoid: } f(a) = \frac{1}{1+e^{-a}}$$

So the sigmoid allowed neural networks to become **differentiable**, which made **gradient-based learning (backpropagation)** possible. However, its tendency to **saturate** and cause **vanishing gradients** has led to the adoption of alternative activation functions (like ReLU) in modern deep learning architectures.

2.3.2.3 Hyperbolic Tangent (tanh)

The **Hyperbolic Tangent (tanh) Activation Function**, commonly known as **tanh**, is defined as:

$$f(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{15}$$

And its derivative is:

$$f'(a) = 1 - \tanh^2(a) = 1 - f^2(a) \tag{16}$$

The tanh function maps input values to an output range between -1 and 1. It is a scaled version of the sigmoid function, centered around zero.

The tanh activation function looks **very similar to the sigmoid**, but it is **symmetric around zero**: outputs range from -1 to 1, which helps in centering the data and can lead to faster convergence during training. This makes it **zero-centered**, which is a big advantage.

- When the input is zero ($a = 0$), the output is also zero ($f(0) = 0$).
- For large positive inputs, the output approaches 1 ($f(a) \to 1$ as $a \to +\infty$).
- For large negative inputs, the output approaches -1 ($f(a) \to -1$ as $a \to -\infty$).

That means hidden neurons can have both positive and negative activations, which helps later layers learn faster because the data stays **balanced** around zero.

Why it's better than sigmoid

- **Range**: The tanh function outputs values between -1 and 1, while the sigmoid function outputs values between 0 and 1. This means that tanh is zero-centered, which can help with convergence during training.
- **Gradient around zero**: The derivative of the tanh function is ≈ 1 around zero, while the derivative of the sigmoid function is ≈ 0.25 around zero. This means that the tanh function has a steeper gradient[4] around zero, which can help with learning.
- **Saturates?** Both functions can saturate for large positive or negative inputs, leading to the vanishing gradient problem. However, because tanh is zero-centered, it can help mitigate this issue to some extent.
- **Training speed**: In practice, models using the tanh activation function often converge faster than those using the sigmoid function, especially in deep networks.

[4] A "steeper gradient" means that small changes in the input lead to larger changes in the output, which can help the model learn more effectively.

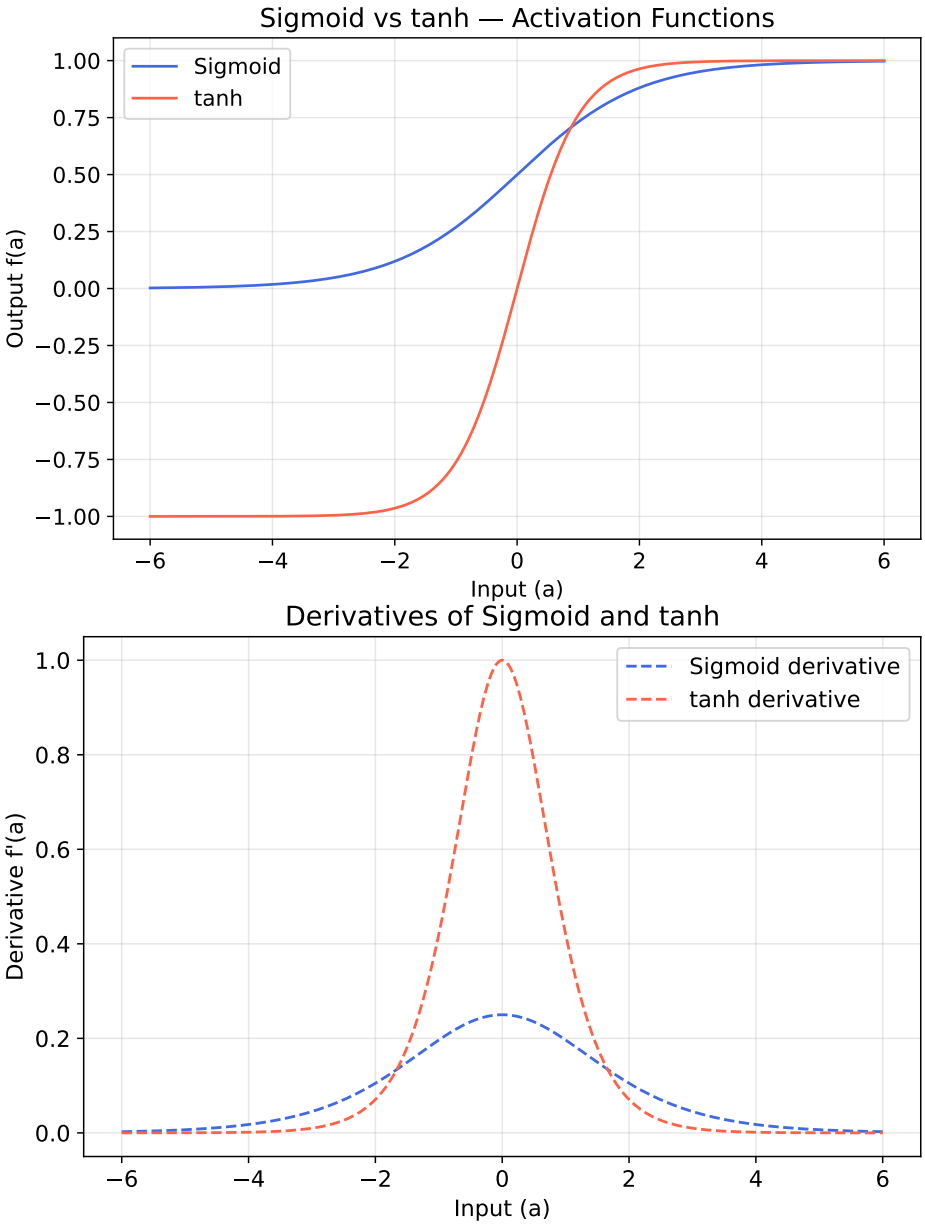


Figure 8: Hyperbolic Tangent (tanh) Activation Function and its Derivative.

The **zero-centered** output means activations can cancel each other out more easily, so the network doesn't get a constant "positive bias" in its gradient (a problem with sigmoid). This often leads to **faster convergence** during training.

When to use tanh

The tanh activation function is often preferred over the sigmoid function in hidden layers of neural networks, especially when the data is centered around zero. It is particularly useful in scenarios where:

- The **input** data is **normalized** to have a **mean of zero**.
- The **model requires faster convergence** during training.
- The **network is deep**, and the benefits of zero-centered activations help mitigate issues like vanishing gradients.

However, it's important to note that while tanh can be advantageous in many situations, it **still suffers from the vanishing gradient problem for very large or very small input values**. Therefore, in very deep networks, other activation functions like ReLU (Rectified Linear Unit) are often preferred.

2.3.3 Output Layer

The **output layer** is the *last* layer of the network, the one that produces the model's **final prediction**. Up to this point, the **hidden layers** have been learning to extract useful features (patterns, relationships, hierarchies). But the **output layer** translates all of that internal representation into the final, human-meaningful result. For example:

- A **continuous number** (e.g., house price) in **regression tasks** (e.g., predicting a numerical value).
- Or a **class label** (e.g., cat vs. dog) in **classification tasks** (e.g., categorizing images).

So, the **choice of activation function** in the output layer depends on the **type of output we want**. Exist several options:

- For **regression tasks**, where we want to predict a continuous value, we often use a **linear activation function** (or no activation function at all) in the output layer. This allows the network to produce a wide range of values.
- For **binary classification tasks**, where we want to classify inputs into two classes, we typically use the **sigmoid activation function** in the output layer. This squashes the output to a value between 0 and 1, which can be interpreted as a probability.
- For **multi-class classification tasks**, where we want to classify inputs into more than two classes, we often use the **softmax activation function** in the output layer. This produces a probability distribution over the classes, ensuring that the sum of the outputs equals 1.

The **design of the output layer** is crucial because it directly affects how well the network can perform its intended task. Choosing the appropriate activation function and structure for the output layer ensures that the network's predictions are meaningful and useful for the specific problem at hand.

2.3.3.1 Regression

In **regression problems**, we want the network to predict **a real-valued quantity**, something that can take *any* number, positive or negative. For example, predicting the price of a house based on its features (size, location, number of rooms, etc.) is a regression task. In this case, the **output layer** of the neural network typically consists of a **single neuron** that produces a **continuous output**, not categorical labels (we want a number, not a class like "expensive" or "cheap").

The output function

For regression tasks, we don't want to limit or distort the network's output. Therefore, the last layer simply uses a **linear activation** (page 59):

$$f(a) = a \quad \text{or equivalently} \quad y = w^T x + b$$

This means the output neuron just returns the raw weighted sum of its inputs, no squashing or thresholding.

If we used a **sigmoid** or **tanh** activation in the output layer, the output would be forced into $(0, 1)$ or $(-1, 1)$ ranges, respectively. This would be problematic for regression tasks where the target variable can take on a wide range of values. For example, if we're predicting house prices, we want the output to be able to represent any price, not just values between 0 and 1 (e.g., a house could cost $\$250,000$, which is far outside the range of a sigmoid output, or a temperature could be -10 degrees Celsius, which is outside the range of tanh). The **linear** activation allows any real number to be output, making it suitable for regression tasks.

Typical network setup for regression

A typical neural network for regression tasks has the following structure:

Component	Example
Hidden layers	Several, with nonlinear activations (e.g., ReLU, tanh).
Output layer	One neuron (for single output) with **linear activation**.
Loss function	**Mean Squared Error (MSE)** or **Mean Absolute Error (MAE)**.

Deepening: Mean Squared Error (MSE)

When our network predicts continuous values (like prices, temperatures, voltages, etc.), we need a way to measure **how far the predictions are from the real targets**. That's what a **loss function** does: it quantifies the prediction error. The **Mean Squared Error (MSE)** is the most

common one for regression tasks:

$$\text{MSE} = \frac{1}{N} \cdot \sum_{i=1}^{N} (y_i - t_i)^2 \tag{17}$$

Where:

- N is the number of data points (samples).
- y_i is the predicted value for the i-th sample.
- t_i is the true (target) value for the i-th sample.

The term $(y_i - t_i)^2$ is the **error** (difference between prediction and truth), and we **square** it to make all errors positive (avoid cancellation) and to penalize **larger mistakes more strongly** (e.g., an error of 10 counts 100 times more than an error of 1). Then we **average** over all samples to get the mean error per prediction.

So MSE measures how "spread out" our predictions are around the true values. A **lower MSE** means our model is doing a better job at predicting the continuous target variable.

Example 4: Example of MSE calculation

Suppose we have the following regression model:

Sample	t_i	y_i	$y_i - t_i$	Squared Error
1	2	3	+1	1
2	5	4	−1	1
3	6	8	+2	4
4	3	2	−1	1

To compute the MSE:

$$\text{MSE} = \frac{1}{4} \cdot (1 + 1 + 4 + 1) = \frac{1}{4} \cdot 7 = 1.75$$

So the Mean Squared Error for this model is 1.75, indicating the average squared difference between the predicted and true values.

About derivations, it is important to note that MSE is **differentiable**, which is crucial for training neural networks using gradient-based optimization methods (we will cover this in detail later). The derivative of MSE with respect to the predictions y_i is:

$$\frac{\partial \,\text{MSE}}{\partial \, y_i} = \frac{2}{N} \cdot (y_i - t_i) \tag{18}$$

So, the weight updates are proportional to how wrong each prediction is. It means, large errors produce larger gradients, leading to bigger

adjustments in the weights during training, which helps the model learn more effectively.

In summary, MSE tells us **how far off our predictions are on average**. It's like saying "*how wrong am I, squared and averaged?*" The squaring heavily punishes big mistakes, making MSE ideal when we care about precision in regression tasks.

Deepening: Mean Absolute Error (MAE)

The **Mean Absolute Error** measures the **average absolute distance** between predicted and true values:

$$\text{MAE} = \frac{1}{N} \cdot \sum_{i=1}^{N} |y_i - t_i| \tag{19}$$

Where:

- N is the number of data points (samples).
- y_i is the predicted value for the i-th sample.
- t_i is the true (target) value for the i-th sample.

Unlike MSE, which *squares* the difference, MAE simply takes the **absolute value** of the error. That means:

- Every error contributes proportionally to its magnitude (no squaring).
- Large errors don't explode quadratically, they contribute linearly.

So MAE measures the **average size of the mistakes**, regardless of direction.

Example 5: Example of MAE calculation

Using the same predictions as before (from Example on page 69), to compute the MAE:

$$\text{MAE} = \frac{1}{4} \cdot (1 + 1 + 2 + 1) = \frac{1}{4} \cdot 5 = 1.25$$

So the Mean Absolute Error for this model is 1.25, indicating the average absolute difference between the predicted and true values. Compared to MSE, MAE gives a more direct sense of the average error magnitude without squaring.

Regarding derivations, the MAE is **not differentiable** at points where the prediction equals the target (i.e., $y_i = t_i$) because of the absolute

value function. However, we can use the **subgradient** for optimization:

$$\frac{\partial |x|}{\partial x} = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ \text{undefined (but taken as 0)} & \text{if } x = 0 \end{cases} \tag{20}$$

So gradient updates from MAE are **constant in magnitude**. They don't depend on how far the prediction is from the truth. That's why MAE can converge slower but more robustly, especially in the presence of outliers (which can heavily skew MSE).

In summary, MAE tells us **how many units off my predictions are on average**, while MSE punishes larger errors more severely:

- **MSE** tries to **minimize variance**, forces the model to avoid large mistakes aggressively.
- **MAE** tries to **minimize average error**, focuses on overall robustness.

If our data has **outliers** (e.g., occasional very wrong samples), MAE is less distorted by them because it doesn't exaggerate their impact.

2.3.3.2 Binary Classification

In **binary classification**, the task is to decide **two possible outcomes**, for example *spam* vs *not spam* in email filtering, or *disease* vs *no disease* in medical diagnosis. The **output** of the network is typically a single neuron that produces a value between 0 and 1, representing the **probability** of one of the classes:

$$\mathcal{P}(y = 1 \mid x) \in [0, 1]$$

That is, "*how likely is this input to belong to class 1?*" The other class's probability can be derived as:

$$\mathcal{P}(y = 0 \mid x) = 1 - \mathcal{P}(y = 1 \mid x)$$

The output function

At the output layer, we typically have:

- **1 neuron**, because we only need one value (the probability of class 1).
- The **activation function** is usually the **sigmoid function** (or sometimes the **tanh function**), which maps any real-valued number into the range (0, 1), making it suitable for probability estimation.

The **sigmoid function** is defined as:

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

Or the **tanh function**:

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

Despite tanh outputting values in the range $(-1, 1)$, it can be scaled to (0, 1) for probability interpretation:

- $f(a) > 0 \Rightarrow$ class 1
- $f(a) < 0 \Rightarrow$ class 0

It's sometimes preferred due to its zero-centered output, which can help with optimization. However, the **sigmoid** function is **more commonly used** in binary classification tasks.

Typical network setup for binary classification

Component	Example
Hidden layers	Several, with nonlinear activations (e.g., ReLU, tanh).
Output layer	One neuron (for single output) with **sigmoid activation**.
Loss function	**Binary Cross-Entropy (log loss)**.

Deepening: Binary Cross-Entropy (BCE, Log Loss)

The goal in **binary classification** is to predict the *probability* that an input belongs to class 1, given by our network's sigmoid output:

$$\hat{y} = f(a) = \frac{1}{1+e^{-a}} \quad \in (0,1)$$

The true label t is:

- 1 if the sample belongs to class 1,
- 0 if it belongs to class 0.

The **Binary Cross-Entropy (BCE, Log Loss)** loss function measures the difference between the predicted probabilities $\hat{y}$ and the true labels t. It is defined as:

$$\text{BCE}\,(t,\hat{y}) = L = -\frac{1}{N}\cdot\sum_{i=1}^{N}\left[t_i\cdot\ln\left(\hat{y}_i\right)+(1-t_i)\cdot\ln\left(1-\hat{y}_i\right)\right] \tag{21}$$

Let's understand each term:

- N is the number of samples in the dataset.
- t_i is the true label for sample i (0 or 1, since it's binary classification).
- $\hat{y}_i$ is the predicted probability for sample i (output of the sigmoid).
- $\ln\left(\hat{y}_i\right)$ **penalizes** the model when it **predicts a low probability for the true class** (when $t_i = 1$).
- $\ln\left(1-\hat{y}_i\right)$ **penalizes** the model when it **predicts a high probability for the false class** (when $t_i = 0$).
- If the true label is 1 ($t_i = 1$), the loss simplifies to $-\ln\left(\hat{y}_i\right)$. The model is **penalized** when it predicts a **small** probability for class 1.
- If the true label is 0 ($t_i = 0$), the loss simplifies to $-\ln\left(1-\hat{y}_i\right)$. The model is **penalized** when it predicts a **large** probability for class 1.

So, **the closer the prediction is to the truth, the smaller the loss**.

Example 6: BCE Calculation Example

Let's consider a simple example with 4 samples:

Sample	True Label (t)	Predicted Probability ($\hat{y}$)
1	1	0.9
2	0	0.2
3	1	0.4
4	0	0.6

Now, we calculate the BCE loss for each sample:

- Sample 1: $[1 \cdot \ln(0.9) + (1-1) \cdot \ln(1-0.9)] = \ln(0.9) \approx -0.105$
- Sample 2: $[0 \cdot \ln(0.2) + (1-0) \cdot \ln(1-0.2)] = \ln(0.8) \approx -0.223$
- Sample 3: $[1 \cdot \ln(0.4) + (1-1) \cdot \ln(1-0.4)] = \ln(0.4) \approx -0.916$
- Sample 4: $[0 \cdot \ln(0.6) + (1-0) \cdot \ln(1-0.6)] = \ln(0.4) \approx -0.916$

Finally, we compute the average BCE loss over all samples:

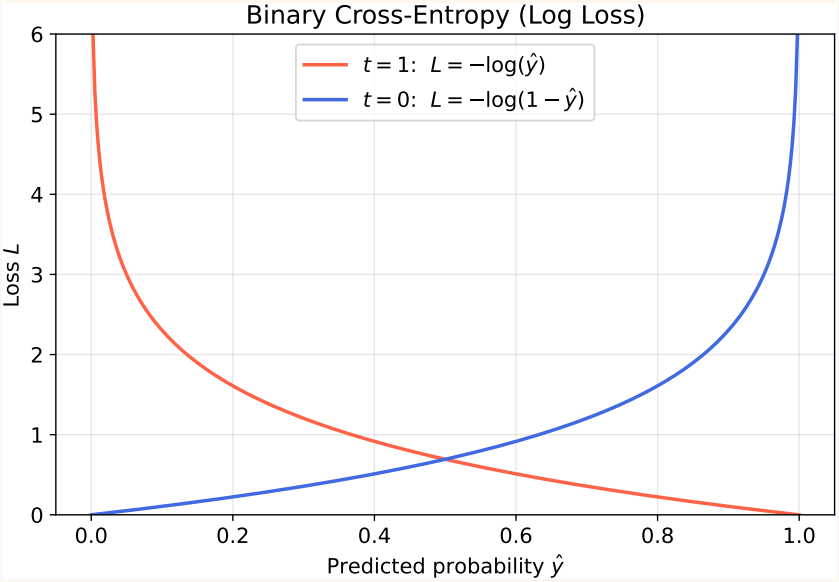
$$L = -\frac{1}{4}(-0.105 - 0.223 - 0.916 - 0.916) = -\frac{-2.16}{4} = 0.54$$

So, the BCE loss for this example is approximately **0.54**. This indicates that the model's predictions are not very accurate, as a lower loss value indicates better performance.

Cross-Entropy comes from **information theory**. It measures the **difference between two probability distributions**:

- The true distributions of the labels (0 or 1).
- The predicted distributions from the model (probabilities between 0 and 1).

Minimizing BCE is equivalent to **maximizing the likelihood** of our data under the model's predictions. So we're training the network to output probabilities that match the true labels as closely as possible.



The loss is **asymmetric**: wrong confident predictions get punished exponentially.

- Red curve ($t = 1$): Loss is low when predicted probability $\hat{y}$ is close to 1 (correct and confident), and high when $\hat{y}$ is close to 0 (incorrect).
- Blue curve ($t = 0$): Loss is low when predicted probability $\hat{y}$ is close to 0 (correct and confident), and high when $\hat{y}$ is close to 1 (incorrect).

Finally, BCE is **differentiable**, which is essential for training neural networks using gradient-based optimization methods. Its derivative with respect to a (the input to the sigmoid) is:

$$\frac{\partial L}{\partial a} = \hat{y} - t = f(a) - t \tag{22}$$

This derivative is used in backpropagation to update the network's weights during training.

In summary, **Binary Cross-Entropy** is the standard loss function for binary classification tasks in neural networks, effectively measuring the discrepancy between predicted probabilities and true binary labels, and guiding the training process to improve model performance. In simple words, it asks: "*how surprise would I be if the model's predicted probability were true?*" The less surprised (closer to 1 for correct class), the smaller the loss; the more surprised (model confident but wrong), the larger the penalty.

2.3.3.3 Multi-Class Classification

When we want the network to choose **one label among many**, for example to classify images of handwritten digits (0, 1, 2, . . . , 9), we need the model to output a **vector of probabilities**, one for each possible class. For example, if the model is 70% sure that the image is a 3, 20% sure it is an 8, and 10% sure it is a 5, the output vector should be:

$$\hat{y} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0.7 \\ 0 \\ 0.1 \\ 0 \\ 0.2 \\ 0 \\ 0 \end{bmatrix}$$

Where each entry corresponds to the predicted probability of each class (from 0 to 9). To achieve this, exists two main techniques: **one-hot encoding** for the labels and the **softmax activation function** for the output layer.

How we represent targets: One-Hot Encoding

One-Hot Encoding is a simple way to represent **categorical variables** (things that take one of several discrete values, like *color*, *day of week*, or *class label*) in a numerical format that a neural network can understand.

The problem it solves. Neural networks work only with **numbers**, not words or symbols. So if our categories are, for example, *cat*, *dog*, and *bird*, we can't feed them directly into the network. We must convert each category into a **numeric vector**.

How it works. The naïve approach would be to assign each category a unique integer (e.g., *cat* = 0, *dog* = 1, *bird* = 2). But this is **misleading**, because the network would think that *bird* (2) is somehow "bigger" or "twice" a "dog" (1). That numerical relationship is meaningless and these categories have no inherent order. Instead, we create a **binary vector** for each class (category), where:

- The position corresponding to that class is set to 1 ("**hot**").
- All other positions are set to 0 ("**cold**").

So for our example with three categories, the one-hot encoded vectors would be:

- *cat* $\rightarrow [1, 0, 0]$
- *dog* $\rightarrow [0, 1, 0]$
- *bird* $\rightarrow [0, 0, 1]$

Each vector is called **one-hot vector** because exactly **one element is "hot"** (1) and all others are "cold" (0).

How we get probabilities: Softmax Activation Function

The **Softmax Activation Function** takes a vector of arbitrary real numbers (called *logits*) and turns it into a **probability distribution**, i.e. a vector of positive numbers that **sum to 1**:

$$\text{softmax}(a_i) = \frac{e^{a_i}}{\sum_{j=1}^{K} e^{a_j}} \tag{23}$$

Where:

- a_i is the i-th **element of the input vector** (logits).
- K is the total **number of classes**.
- e is the base of the natural logarithm (neperian constant).

Intuition. Each neuron in the output layer produces a **score**: a real number that can be positive, negative, or large. Softmax converts these scores into **relative probabilities** that express how confident the network is about each class:

- Large $a_i \rightarrow$ large $e^{a_i} \rightarrow$ **high** probability.
- Small $a_i \rightarrow$ small $e^{a_i} \rightarrow$ **low** probability.

The exponential function e^{a_i} magnifies differences between scores, so the biggest score gets *much more weight*, but every class still receives a small share.

Example 7

Suppose the output layer of a neural network produces the following logits for a 3-class classification problem:

$$a = \begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix}$$

To convert these logits into probabilities using the softmax function, we first compute the exponentials:

$$e^a = \begin{bmatrix} e^{2.0} \\ e^{1.0} \\ e^{0.1} \end{bmatrix} \approx \begin{bmatrix} 7.389 \\ 2.718 \\ 1.105 \end{bmatrix}$$

Next, we sum these exponentials:

$$S = 7.389 + 2.718 + 1.105 \approx 11.212$$

Finally, we compute the softmax probabilities:

$$\text{softmax}(a) = \begin{bmatrix} \frac{7.389}{11.212} \\ \frac{2.718}{11.212} \\ \frac{1.105}{11.212} \end{bmatrix} \approx \begin{bmatrix} 0.659 \\ 0.242 \\ 0.099 \end{bmatrix}$$

Thus, the output probabilities for the three classes are approximately 65.9%, 24.2%, and 9.9%, respectively. The network is most confident that the input belongs to class 1.

Softmax acts like a "**competition**" between neurons:

- Each output neuron tries to **"win"** by having the highest score.
- The exponentials amplify the differences, making the highest score dominate.
- The normalization (dividing by the sum) ensures all probabilities add up to 1.

This is why it's called "softmax": it produces a **soft** version of the **maximum** function, where the highest score gets the most weight, but all classes still receive some probability (unlike a hard max which would assign 100% to the highest and 0% to all others).

🛠 Putting it all together

In a **K-class classification** problem, the network's final layer has:

- **K output neurons**, one for each class.
- Each neuron produces a **logit** (a_i) an unnormalized score.
- The **softmax function** converts these logits into a **probability distribution** over the K classes:

$$\hat{y}_i = \text{softmax}(a_i) = \frac{e^{a_i}}{\sum_{j=1}^{K} e^{a_j}}$$

So the network outputs a probability distribution over classes, all y_i are between 0 and 1, and they sum to 1. However, to train the network effectively, we also need a suitable loss function that works well with this setup (about training, we will discuss it later, but for now, let's focus on the loss function). This is where **Categorical Cross-Entropy (CCE)** comes into play.

The **Categorical Cross-Entropy (CCE)** loss function measures how close the predicted probability distribution $\hat{y}$ is to the true one-hot distribution $\mathbf{t}$:

$$\text{CCE}(\mathbf{t}, \hat{y}) = -\sum_{i=1}^{K} t_i \cdot \log(\hat{y}_i) \tag{24}$$

Because only the true class has $t_i = 1$ (all others are 0), this simplifies to:

$$\text{CCE}(\mathbf{t}, \hat{y}) = -\log(\hat{y}_c) \tag{25}$$

Where:

- c is the **index of the true class**.
- y_c is the **predicted probability for the true class**.

This means CCE penalizes the model when it assigns a low probability to the true class, encouraging it to predict higher probabilities for the correct class during training.

Why Softmax and CCE work well together? The combination of Softmax and CCE is powerful because:

- **Softmax produces a valid probability distribution**, which is exactly what CCE needs to compute the loss.
- **CCE** focuses the learning on **maximizing the probability of the true class**, which aligns perfectly with the goal of classification tasks.
- The **gradients** computed from CCE with respect to the logits are well-behaved, making training more stable and efficient. The drivative of CCE combined with Softmax is the following:

$$\frac{\partial\ (\text{CCE})}{\partial\, a_i} = \hat{y}_i - t_i \tag{26}$$

  This means the gradient is simply the difference between the **predicted probability** and the **true label**, which is **easy to compute** and interpret.

This synergy makes Softmax + CCE the **standard choice for multi-class classification problems in neural networks**. It is a generalization of the Sigmoid + BCE setup used for binary classification, extending the same principles to handle multiple classes effectively.

2.3.4 Neural Networks as Universal Approximators

In 1989, Kurt Hornik, Maxwell Stinchcombe, and Halbert White published a seminal paper titled "Multilayer Feedforward Networks are Universal Approximators" [4]. This groundbreaking work established that Feed-Forward Neural Networks (FNNs) with at least one hidden layer and non-linear activation functions can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to any desired degree of accuracy, given sufficient neurons in the hidden layer:

> "*A single hidden layer feed-forward neural network with S-shaped activation functions can approximate any measurable function to any desired degree of accuracy on a compact set.*"

This theorem establishes the *theoretical power* of neural networks: given enough hidden neurons, an FNN can approximate **any continuous function** $f(x)$ over a bounded input domain. Let's define this more formally.

Theorem 1 (Universal Approximation Theorem)**.** *Let:*

$$f : \mathbb{R}^n \to \mathbb{R} \tag{27}$$

By any continuous function on a compact subset of $\mathbb{R}^n$ *(*$K \subset \mathbb{R}^n$*).*
Then, for any $\varepsilon > 0$*, there exists:*

- *A* ***single-hidden-layer neural network***
- *With* ***finite*** *number of neurons* J
- *And* ***non-linear activation*** $\sigma(\cdot)$ *(e.g., sigmoid, tanh, ReLU, etc.)*

Such that for all $x \in K$*:*

$$\left| f(x) - \sum_{j=1}^{J} w_j^{(2)} \cdot \sigma \cdot \sum_{i=1}^{n} \left(w_{ji}^{(1)} x_i + b_j \right) \right| < \varepsilon \tag{28}$$

Where:

- $w_{ji}^{(1)}$ *are the* ***weights*** *from* ***input layer to hidden layer****.*
- b_j *are the* ***biases*** *of the* ***hidden layer*** *neurons.*
- $\sum_{i=1}^{n} \left(w_{ji}^{(1)} x_i + b_j \right)$ *is the* ***input to the hidden layer*** *neuron* j*.*
- $\sigma(\cdot)$ *is the* ***non-linear activation function*** *applied at* ***hidden*** *layer neurons.*
- $w_j^{(2)}$ *are the* ***weights from hidden layer to output layer****.*
- *The output is the result of the neural network for input* x*.*

In simpler terms, any continuous can be represented by a neural network with just **one hidden layer**, if that layer has enough neurons and uses a non-linear activation function.

Why it works (intuition)

Each hidden neuron with non-linear activation acts like a **basis function** (similar to how sine and cosine functions can approximate any waveform in Fourier series). By combining enough of these basis functions (hidden neurons), the neural network can approximate complex functions by adjusting the weights and biases.

Imagine we have an unknown function $f(x) = \sin(x)$. And we want our neural network to **learn** this function. So, in other words, we want our neural network to approximate $f(x)$ as closely as possible ($\hat{f}(x) \approx f(x)$). Let's take a **single neuron** with a non-linear activation function (e.g., sigmoid):

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

If we plot this, it looks like an S-shaped curve (page 61):

- Almost 0 for large negative inputs.
- Almost 1 for large positive inputs.
- Smoothly transitions between 0 and 1 around input 0.

When we apply this neuron to a **linear combination of** x:

$$\sigma(w \cdot x + b)$$

We get a *shifted and stretched S-curve* along the x-axis. Now imagine we have **many hidden neurons**, each with their own weights and biases:

$$\hat{f}(x) = \sum_{j=1}^{J} w_j^{(2)} \cdot \sigma(w_j^{(1)} \cdot x + b_j)$$

Each neuron produces its own "bump" or "S-step" at a different location. When we **add them together**, those bumps **stack up and blend**, creating any curve shape we want. And that's the whole trick! Just like adding sinusoids can approximate any periodic signal (Fourier series), adding non-linear S-shaped functions can approximate any continuous curve. Note that non-linearity is crucial. If we used only linear activations and stacked them, the result would still be a linear function. This would collapse the network's expressive power.

Important note

The Universal Approximation Theorem guarantees that a neural network **can** approximate any continuous function, but it does not tell us **how to find** the right weights and biases to do so. In practice, training a neural network to approximate a specific function requires effective optimization algorithms (like gradient descent) and sufficient training data. Additionally, while a single hidden layer is theoretically sufficient, deeper networks (with more hidden layers) often learn more efficiently and generalize better in practice.

2.4 Learning and Optimization

Let's retrace what we've built so far step by step:

1. We started with the **historical context**, understanding why we want machines to "learn" like brains, transitioning from symbolic AI to data-driven learning.
2. Next, we explored the **Perceptron**, the simplest computational neuron, which introduced us to linear decision boundaries and Hebbian learning.
3. However, we also learned about the **limitations of the Perceptron**, particularly its inability to solve non-linear problems like XOR.
4. To overcome these limitations, we delved into **Feed-Forward Neural Networks (FNNs)**, discovering how multi-layer networks with hidden layers and nonlinear activations can model complex functions.
5. Finally, we touched on the **Universal Approximation Theorem**, which assures us that even a single hidden layer is theoretically sufficient to approximate any function (though in practice, deeper networks often perform better).

Now we know **what** the architecture can represent. But we haven't yet learned **how** to *find the right weights* that make it represent what we want. And that's *exactly* why this section begins.

After defining the structure of a neural network, we must **teach it** to perform a task, such as classifying images or predicting values. This teaching process is called **learning** or **training** (or simply *learning by optimization*). Think of the journey like this:

$$\text{Architecture} \rightarrow \text{Function Space} \rightarrow \text{Optimization} \rightarrow \text{Learning}$$

We've defined the **function space** (what kinds of functions the network can represent), and now, we must **search inside that space** for the specific function that matches our data. This search is done through **optimization algorithms** that adjust the network's weights based on the data we provide. So, in this section, we'll answer three big questions:

1. ***How does a neural networks learn?*** (page 83) By comparing predictions with known targets (supervised learning).
2. ***How do we measure "how wrong" it is?*** (page 86) Through *loss functions* (some of which we have already encountered in the output layer design).
3. ***How do we improve it?*** (page 90) Through *optimization algorithms* like gradient descent and (later) backpropagation.

2.4.1 Supervised Learning and Training Dataset

This section introduces the **formal setup** of *Supervised Learning* in the context of neural networks. It defines **what data to use**, **what we want the network to learn**, and **how we measure learning success**. It's the *conceptual skeleton* that the later mathematical tools (loss, gradient descent, backpropagation) will stand on.

What is Supervised Learning?

Supervised Learning is a **machine learning paradigm** where the algorithm learns **from examples that include both the input and the correct output**. In other words, it learns **under supervision** from labeled data.

The basic idea is simple. We give the model a set of **training examples**:

$$\mathcal{D} = \{(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)\}$$

Formally, a **dataset** is:

$$\mathcal{D} = \left\{ (x_i, t_i) \right\}_{i=1}^{N} \tag{29}$$

Where:

- x_i is the **input data** (e.g., an image, temperature readings, pixels, sensor values, etc.).
- t_i is the **target output** (the *label* or *ground truth* we want the model to predict).
- N is the total number of training examples.

The model (in our case, a neural network) tries to learn a **function** $f(\cdot)$ such that:

$$f(x_i) \approx t_i \quad \text{for all } i = 1, 2, \ldots, N$$

For all examples in the training set. In other words, to find a function $f(x)$ that not only fits the training data but also **generalizes** well to unseen data (i.e., it can correctly predict outputs for new inputs not in the training set). Formally, a **model** is a function:

$$g(x; w) \text{ with parameters } w \tag{30}$$

Where:

- $g(\cdot; w)$ is the model (neural network) with parameters w (weights and biases).
- The goal is to find the optimal parameters w^* such that:

$$w^* = \arg\min_w E(w) \tag{31}$$

  Where $E(w)$ is a **loss function** that measures how far predictions $g(x_i; w)$ are from the true targets t_i across the training set.

The method is called **supervised** because the learning process is **guided**: each input x comes with the **correct answer** t. The network uses it to know whether it was right or wrong, and to adjust its weights accordingly.

Independently by the paradigm used (supervised, unsupervised, reinforcement learning), with **Training** we mean the **process of adjusting weights w so that the network reproduces the mapping between inputs and outputs seen in the data**. This is done by **minimizing a loss function** that quantifies the difference between the network's predictions and the true targets in the training dataset.

Neural Networks as a Parametric Model

In general, a neural network can be written as a **parametric function**:

$$y\left(x;w\right)=g\left(x,w\right) \tag{32}$$

Where:

- x is the input features (data).
- w is the set of parameters (weights and biases in all layers) of the network.
- $y\left(x;w\right)$ is the output of the network (the prediction for input x given parameters w).
- ; indicates that y depends on both x and w.
- $g(\cdot,\cdot)$ represents the entire computation performed by the neural network (all layers, activations, etc.). See above equation 30.

We want $y\left(x_n;w\right)$ to be as close as possible to the target t_n for each training example (x_n,t_n) in the dataset $\mathcal{D}$. This is achieved by **optimizing the parameters** w to minimize a **loss function** that measures the discrepancy between predictions and targets across all training examples. Formally, find parameters w^* that minimize a loss function $E(w)$:

$$w^*=\arg\min_w E\left(w\right) \tag{33}$$

Where $E(w)$ quantifies the error between $y\left(x_n;w\right)$ and t_n for all training examples. The loss function measures how wrong the model is across all training examples:

$$E\left(w\right)=\sum_{n=1}^{N}\ell\left(t_n,y\left(x_n;w\right)\right) \tag{34}$$

Where $\ell(\cdot,\cdot)$ is a loss function that quantifies the error for a single example (e.g., Mean Squared Error for regression, Cross-Entropy Loss for classification).

The power of this setup

This framework allows us to covers a wide range of tasks:

- **Regression**: Predicting continuous values (e.g., house prices, temperature).
- **Classification**: Assigning inputs to discrete categories (e.g., spam vs. not spam, image recognition).
- **Function Approximation**: Learning complex mappings from inputs to outputs.

By defining the problem in terms of inputs, targets, and a loss function, we can apply various optimization algorithms (like gradient descent) to train the neural network effectively. So, we must care only about:

- The **structure of the network** (layers, activation functions).
- The **choice of loss function** (depends on the task).
- The **optimization algorithm** to minimize the loss.

This abstraction makes neural networks a versatile tool for many machine learning problems.

Example 8: Analogy

Imagine a student (the network) learning to solve math exercises.

- **Inputs** x_n are the exercises given to the student by the teacher.
- **Targets** t_n are the correct answers provided by the solution book.
- **Network** $g(x, w)$ is the student's method of solving the exercises, which depends on their current knowledge (parameters w).
- **Loss function** $E(w)$ measures how many answers the student got wrong compared to the solution book.
- **Optimization** (training) is the process of the student studying and adjusting their methods (updating w) to minimize mistakes on future exercises.

Over time, with enough practice (training examples), the student improves their ability to solve new exercises correctly (generalization). So training means making fewer mistakes over time by adjusting reasoning (weights).

2.4.2 Error Minimization and Loss Function (SSE)

To teach a neural network, we need a way to **measure how wrong** it is. That measure is the **error (loss) function**. Once defined. we can **minimize** it by adjusting the weights, and the process **is** the essence of learning.

Definition 6: Error Function

Given a **training set**:

$$\mathcal{D} = \left\{ (x_n, t_n) \right\}_{n=1}^{N} \tag{35}$$

And the network's predictions:

$$y_n = g\left(x_n; w\right) \tag{36}$$

The **Error Function** $E(w)$ measures the total discrepancy between all predictions and their true targets:

$$E(w) = \sum_{n=1}^{N} \text{Loss}(t_n, y_n) \tag{37}$$

Where $\text{Loss}\left(\cdot\right)$ is the per-sample difference between the predicted and actual value.

Definition 7: Loss Function

A **Loss Function** (sometimes called **error** or **Cost Function**) is a **mathematical function that quantifies how wrong a model's predictions are** compared to the true (target) values. In other words:

$$\text{Loss}(t, y) = \text{scalar measure of discrepancy between } t \text{ and } y \tag{38}$$

Where t is the true target value, and y is the model's prediction. The loss function **outputs a single number representing how bad the prediction is**; lower values indicate better predictions. During training/learning, the network tries to **minimize** this loss by adjusting its weights, to reduce its mistakes.

It is strictly related to the **Error Function** $E(w)$, which aggregates the loss over the entire training set to give a total measure of how well the model is performing. Indeed, for a single training example (x_n, t_n), we have:

$$L_n = \text{Loss}\left(t_n, g\left(x_n; w\right)\right) \tag{39}$$

Where L_n is the loss for sample n, t_n is the true target, and $g\left(x_n; w\right)$ is the model's prediction for input x_n with weights w. And the total error

function over all N samples is:

$$E(w) = \sum_{n=1}^{N} L_n = \sum_{n=1}^{N} \text{Loss}\left(t_n, g\left(x_n; w\right)\right)$$

So, the **loss function measures the error for one sample**, while the **Error Function sums these losses over the entire dataset** to give a total error measure.

The simplest and most classic choice: SSE

Exists many choices for the loss function. The simplest and most classic is the **Sum of Squared Errors (SSE)**. In early neural networks (and still often in regression problems), the **Sum of Squared Errors (SSE)** was the standard loss function:

$$E(w) = \sum_{n=1}^{N} \text{Loss}(t_n, y_n) = \sum_{n=1}^{N} \left[t_n - g\left(x_n; w\right)\right]^2 \tag{40}$$

- t_n is the true target for sample n.
- $g\left(x_n; w\right)$ is the network's prediction for input x_n.

Similar to the Mean Squared Error (MSE, page 69), the SSE squares the differences to makes all errors **positive** (so under- and over-predictions both count) and **emphasizes large errors** (penalizes them more heavily). Also, squaring makes the error function **differentiable**, which is crucial for optimization algorithms like gradient descent.

Minimizing the SSE means finding the weights w that make the network's predictions as close as possible to the true targets across the entire training set. Formally, learning becomes finding the set of weights w^* that minimize the total error:

$$w^* = \arg\min_{w} E(w)$$

Each weight w_i acts like a small knob controlling part of the network's behavior. We tweak these knobs slightly so that the network's predictions move closer to the true outputs. When all knobs are adjusted such that $E(w)$ is as small as possible, the network has **learned** the function.

Geometric Interpretation

Minimizing the error means **finding a point in parameter space w where the error surface $E(w)$ reaches its minimum**. We can visualize $E(w)$ as a **landscape**: valleys represent low error (good predictions), and hills represent high error (bad predictions). The learning process is like navigating this landscape to find the lowest valley, which corresponds to the optimal weights w^* that minimize the error.

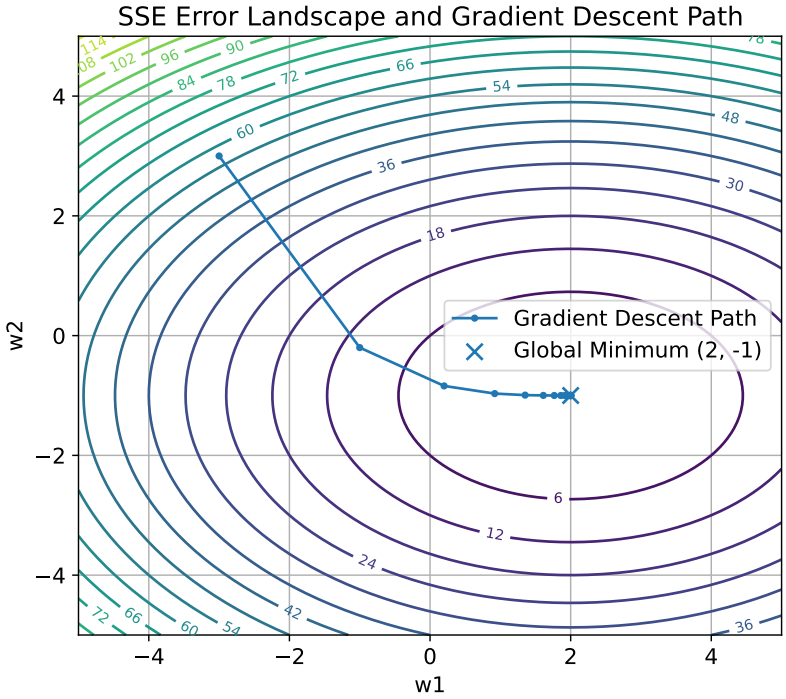


Figure 9: How the **Sum of Squared Errors (SSE)** behaves as a function of the **model parameters (weights)**, and how **gradient descent** moves step by step toward the minimum error point. The x-axis w_1 and y-axis w_2 represent two weights (parameters) of the model/ Each contour line shows **all combinations of weights** (w_1, w_2) that produce the **same total error** $E(w_1, w_2)$. The loss function we use:

$$E\left(w_1, w_2\right) = \left(w_1 - 2\right)^2 + 2\left(w_2 + 1\right)^2$$

Those small points connected by a line represent the **path that gradient descent follows** over time. Starting from an initial guess, each step moves downhill toward lower error, reaching the minimum point where the error is lowest (point $(2, -1)$ in this case). In a real network, the number of weights isn't just two but can be thousands or millions, making the error surface a high-dimensional landscape. We can't visualize that directly, but this contour map is an **analogy** to help understand how optimization algorithms like gradient descent navigate the error surface to find the best weights.

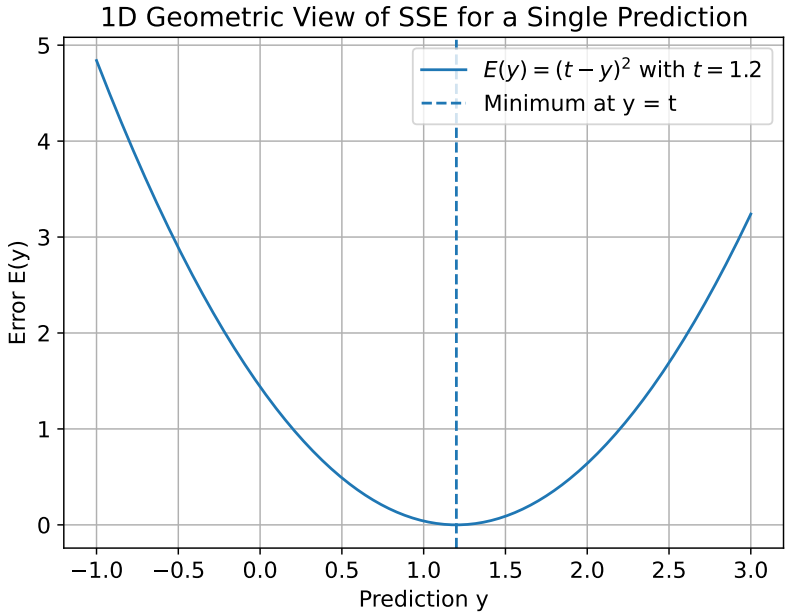


Figure 10: The **error function** for **a single training example** using the **Sum of Squared Errors (SSE)** loss: $E(y) = (t - y)^2$, where t is the true target and y is the model's prediction. The x-axis represents the model's prediction y (all possible output values the model could predict for this input sample), and the y-axis shows the corresponding error $E(y)$ (how wrong the model would be for each possible prediction). This graph shows **how the error changes** as the model output moves away from the correct answer. It is a **parabola**, because the error grows quadratically as we move away from the correct value. At $y = t$, the error is zero (the model predicts perfectly) because $E(t) = (t - t)^2 = 0$. As y moves away from t, the difference grows, and squaring makes it **positive and larger**. The dashed vertical line marks the **minimum point** where the prediction perfectly matches the target ($y = t$, zero error).

Relation to Statistical Foundations

The squared error has a nice **probabilistic interpretation**. If we assume that the target values t_n are generated by a model with **Gaussian noise**:

$$t_n = g(x_n; w) + \epsilon_n, \quad \epsilon_n \sim \mathcal{N}(0, \sigma^2)$$

Then minimizing the sum of squared errors (SSE) is equivalent to **Maximum Likelihood Estimation (MLE)** of the weights w. In other words, by minimizing SSE, we are finding the weights that make the observed data most probable under the assumed Gaussian noise model. So the SSE is not arbitrary choice; it has a solid statistical basis when the noise in the data is Gaussian. In the section Maximum Likelihood Estimation (MLE) (page 103), we will explore this connection in more detail.

2.4.3 Gradient Descent Basics

We've defined the learning goal:

$$w^* = \arg\min_w E(w)$$

Where $E(w)$ is our **error (or loss) function**, for instance:

$$E(w) = \sum_{i=1}^{N} (t_n - g(x_n; w))^2$$

The idea of **gradient descent** is to find this minimum *iteratively*, by moving the parameters w step by step in the direction that **reduces** the error.

The Concept of the Gradient and the Key Idea of Gradient Descent

Let's start simple. For a function of one variable $E(w)$, the **derivative** $\frac{\partial E}{\partial w}$ at a point w tells us the slope of the function:

- If positive $\rightarrow$ $E(w)$ increases as w increases.
- If negative $\rightarrow$ $E(w)$ decreases as w increases.
- If zero $\rightarrow$ $E(w)$ is flat (local minimum or maximum).

In higher dimensions (e.g., multiple weights $w_1, w_2, \ldots, w_d$), we generalize the derivative to a **vector** of partial derivatives:

$$\nabla E(w) = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_d} \end{bmatrix}$$

This vector, the **gradient** of E at w, points in the direction of **steepest ascent** of the function (the **direction** in which $E(w)$ **increases the most rapidly**).

So, if we want to **minimize** $E(w)$, we should move in the direction of **steepest descent**, which is the **opposite direction** of the gradient, i.e., $-\nabla E(w)$. That's why it's called **gradient descent**!

Definition 8: Gradient Descent

Gradient Descent is an **iterative optimization algorithm** used to find the set of parameters (weights) that **minimize a loss function**. In simple words, it is the process by which a neural network *learns* by **repeatedly adjusting its weights in the direction that reduces**

the loss the most.

Formally, let w be the vector of weights, and $E(w)$ be the loss function. The **gradient** with respect to the weights is given by $\nabla E(w)$:

$$\nabla E(w) = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_k} \end{bmatrix} \tag{41}$$

This vector points in the **direction of steepest increase** of $E(w)$. To *minimize* the loss, we go in the **opposite direction** of the gradient. That gives the update rule for the weights, known as **Core Learning Rule**:

$$w_{k+1} = w_k - \eta \nabla E(w_k) \tag{42}$$

Where:

- w_k is the weight vector at iteration k.
- $\nabla E(w_k)$ is the gradient (slope) of the error function at w_k.
- η is the **learning rate** (step size), a small positive scalar that controls the step size.
- w_{k+1} is the updated weight vector after taking a step in the direction of steepest descent.

It is called Core Learning Rule because it is the **fundamental principle underlying how neural networks learn from data by adjusting their weights to minimize error**. Everything that comes next (like backpropagation) are all *variants or extensions* of this exact rule. They all keep this same pattern:

new param = old param−(some learning rate)×(some form of gradient)

The only difference is *how* the gradient or learning rate is computed or adjusted. It is a sort of **DNA of learning** in neural networks.

The Neural Network Case

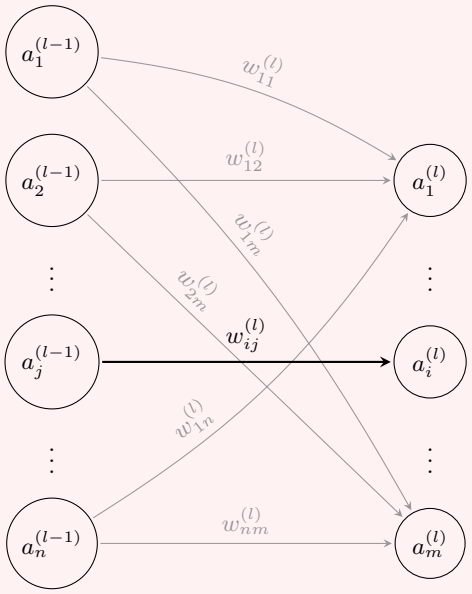
The previous definition is **general**, and it doesn't care whether w is one number, a list or a tensor of weights inside a neural network. However, sice we are in the context of neural networks, let's introduce some notation specific to them.

In a neural network, w isn't just one parameter vector (one weight vector), but rather a **collection of all weights and biases across all layers**:

$$w = \left\{ w_{ij}^{(l+1)}, w_{ij}^{(l+2)}, \ldots, w_{ij}^{(L)}, b_i^{(l+1)}, b_i^{(l+2)}, \ldots, b_i^{(L)} \right\} \tag{43}$$

Where:

- L is the **total number of layers in the network**.
- Each $w_{ij}^{(l)}$ is the weight connecting neuron j in layer $l-1$ to neuron i in layer l:



- Each $b_i^{(l)}$ is the bias for neuron i in layer l.

The **gradient** $\nabla E(w)$ then contains the partial derivatives of the loss function with respect to **each individual weight and bias** in the network:

$$\nabla E(w) = \left\{ \frac{\partial E}{\partial w_{ij}^{(l)}}, \frac{\partial E}{\partial b_i^{(l)}} \right\} \tag{44}$$

The **core learning rule** still applies, but now we update **each weight and bias** individually:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial E}{\partial w_{ij}^{(l)}} \quad \text{and} \quad b_i^{(l)} \leftarrow b_i^{(l)} - \eta \frac{\partial E}{\partial b_i^{(l)}} \tag{45}$$

This is how **gradient descent** is applied in the context of training neural networks.

How it works (intuitively)

Intuitively, gradient descent works as follows:

1. Start from an **initial guess** for the weights w_0 (often random).
2. Compute the **gradient** $\nabla E(w_0)$ of the loss function at the current weights:

$$\nabla E(w_0) = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_d} \end{bmatrix}_{w=w_0}$$

3. Do a step **against** that gradient to update the weights using the **core learning rule**:

$$w_{k+1} = w_k - \eta \nabla E(w_k) \quad \Rightarrow \quad w_1 = w_0 - \eta \nabla E(w_0)$$

   Then, the new weights w_1 should yield a **lower loss** $E(w_1) < E(w_0)$. If we were on a neural network, we would update **all weights and biases** similarly:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial E}{\partial w_{ij}^{(l)}} \quad , \quad b_i^{(l)} \leftarrow b_i^{(l)} - \eta \frac{\partial E}{\partial b_i^{(l)}}$$

4. Repeat until: the gradient becomes small (close to zero), or we reach a maximum number of iterations.

Convergence Speed and False Positives. The speed at which gradient descent converges to the minimum depends on:

- The **shape** of the loss function (e.g., steepness, curvature). It strongly affects how gradient descent behaves:
  - **Convex surface** (like a bowl): Gradient descent always converges there (since there is a **single global minimum**). However, if η is too large, it may oscillate around the minimum.
  - **Non-convex surface** (like many hills and valleys): There may be **multiple local minima** and **saddle points**. Gradient descent might:
    - Get stuck in a **local minimum** (the derivative is zero, but it is not the global optimum).
    - Oscillate in flat regions.
    - Move very slowly along narrow valleys.
- The **learning rate** η.

  The **learning rate** η is crucial: it controls **how big a step** we take each iteration.

- η too **small** → **Learning is very slow** because we take tiny steps (many iterations needed).
- η too **large** → **May overshoot the minimum** and even diverge (loss increases instead of decreasing).
- Choosing a good η is often done via **experimentation** or using techniques. However, if η is chosen well, gradient descent can efficiently find a good set of weights that minimize the loss function.

Don't worry if this seems abstract now; we'll later learn that **adaptive optimizers** (like ***momentum***, ***Adam***, etc.) are more robust ways to handle this.

- The **initial weights** w_0. In **convex** functions (like a parabola), there is a global minimum, and *gradient descent* will converge to it. In **non-convex** functions (like many neural network loss landscapes), there may be multiple local minima, and gradient descent may converge to one of them depending on the starting point. So in practice, we often run gradient descent multiple times with different initial weights to find a good solution.

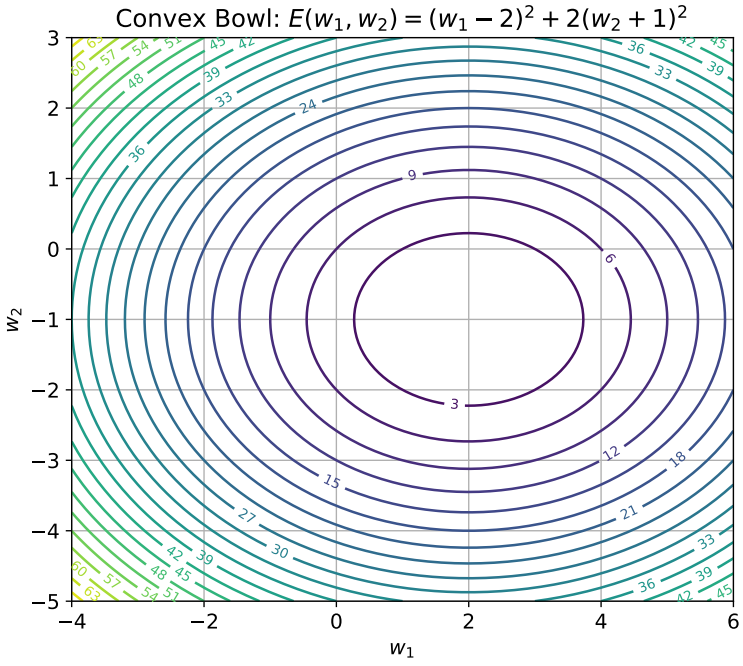


Figure 11: Example of a **convex quadratic surface** (bowl-shaped). The contour lines (ellipses) represent levels of constant error $E(w_1, w_2)$. There is a **single global minimum** at the center of the bowl ($w_1^* = 2, w_2^* = -1$), and outer ellipses represent higher error values. This is the ideal scenario for gradient descent, as it will always converge to the global minimum regardless of the starting point (similar to Figure 9, page 88).

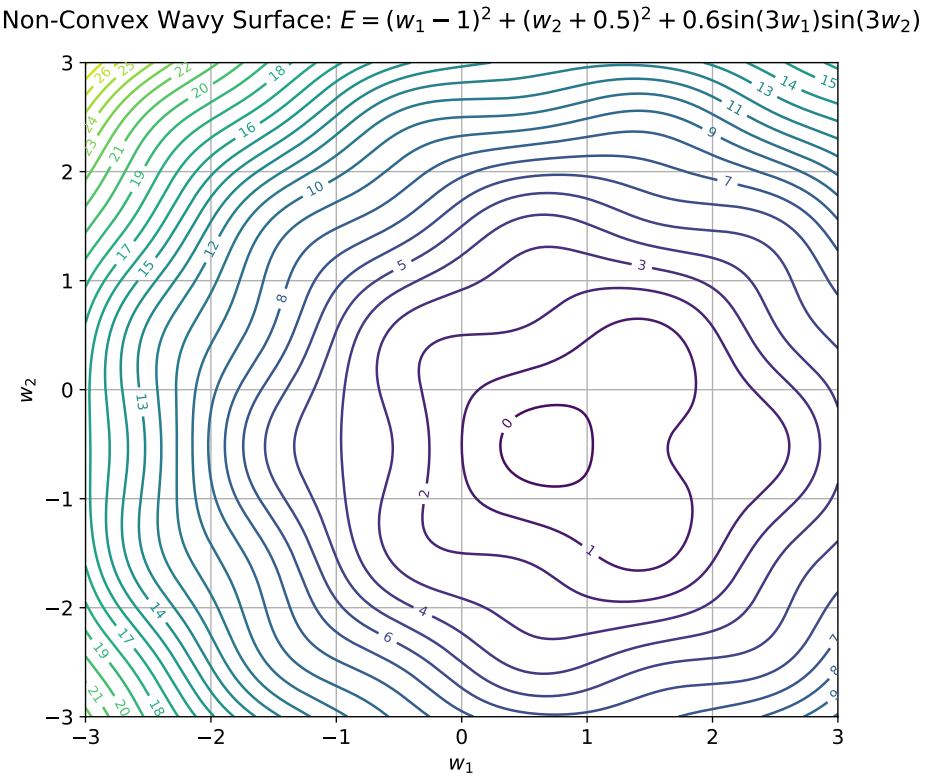


Figure 12: Example of a **non-convex surface** with multiple local minima and saddle points. The loss function $E(w_1, w_2)$ has two components: (1) the **quadratic bowl** that pushes weights toward the global minimum, and (2) the **sinusoidal term** that introduces *oscillations* in the surface. Those oscillations create **small waves** (bumps and dips). Each dip (a small valley) is a *local minimum*, and each bump is a *local maximum* or ridge. This is what happens in **non-linear models** like deep neural networks, where the composition of many layers and activations makes the error surface very complex (and non-convex). There are thousands or millions of such local minima, making optimization challenging. Gradient descent may get trapped in one of these local minima instead of finding the global minimum. However, in practice, many local minima have **similar performance**, so this isn't always bad.

2.4.4 Backpropagation (Conceptual Introduction)

Why Backpropagation exists?

The **problem** we face is computational:

- Gradient descent (page 90) at each iteration requires the computation of **all partial derivatives** of the loss function E with respect to **all weights and biases** in the network. Formally, we need to compute (introduced in section 2.4.3, page 92):

$$\nabla E(w) = \left\{ \frac{\partial E}{\partial w_{ij}^{(l)}}, \frac{\partial E}{\partial b_i^{(l)}} \right\}$$

- In a network with thousands of weights, each weight influences the output *indirectly* through multiple layers.

Computing those derivatives by brute force (finite differences or manual application of the chain rule) would be **computationally expensive** and **inefficient**. We need a more efficient way to compute these gradients.

Backpropagation Concept

Backpropagation (or **backward propagation of errors**) is an efficient **algorithm** used to **compute the gradients** of the loss function with respect to all weights and biases **in a neural network**. It leverages the **chain rule of calculus** in a systematic, efficient way to reuse computations and propagate errors backward through the network. It is analogous to hashmaps in programming, where intermediate results are stored and reused to avoid redundant calculations.

Remark: The Chain Rule

The **Chain Rule** is a fundamental rule in calculus used to compute the derivative of a composite function.
Suppose we have a **function inside another function**:

$$y = f\left(g(x)\right)$$

We want to know how y changes when we slightly change x. In other words, we want to find the derivative $\frac{\partial y}{\partial x}$. The chain rule tells us that we can break this down into two parts:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial x} \tag{46}$$

This means that to find out how y changes with respect to x, we first find out how y changes with respect to g (the inner function), and then multiply that by how g changes with respect to x. It's literally "follow the chain" of dependencies.
Let's see a quick example. Let's say:

$$y = f\left(g(x)\right) = (2x+3)^2$$

We can identify:

- Inner function: $g(x) = 2x + 3$
- Outer function: $f(g) = g^2$

Now, we compute the derivatives:

- $\frac{\partial f}{\partial g}\left(g^2\right) = 2g$
- $\frac{\partial g}{\partial x}\left(2x + 3\right) = 2$

Now, applying the chain rule:

$$\frac{\partial y}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x} = 2g \cdot 2 = 4g$$

Finally, substituting back $g(x)$:

$$\frac{\partial y}{\partial x} = 4(2x + 3) = 8x + 12$$

Conceptually, Backpropagation works during the training phase of a neural network. **During training phase**, there are **two complementary flows** of information:

- **Forward Pass** (Input $\rightarrow$ Output): The purpose is to **measure how good the current weights are at predicting the target outputs**. The input data is passed through the network layer by layer to compute the output. During this phase, the activations of each neuron are computed and stored for later use. Formally, compute predictions $y(x; w)$ for input x and weights w, and save intermediate activations $a^{(l)}$ for each layer l. Here, $a^{(l)}$ represents the activations at layer l:

  $$a^{(l)} = f\left(W^{(l)}a^{(l-1)} + b^{(l)}\right) = f(z^{(l)})$$

  where f is the activation function, $W^{(l)}$ are the weights, and $b^{(l)}$ are the biases at layer l.

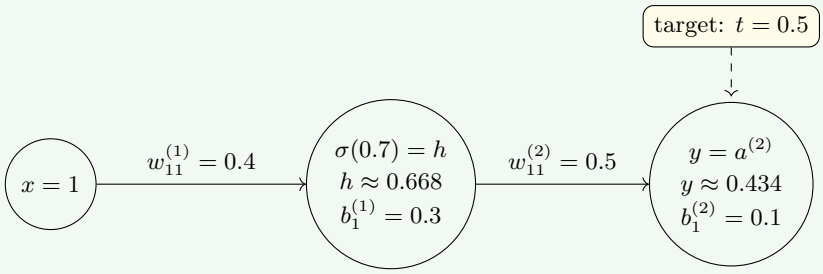
- **Backward Pass** (Output $\rightarrow$ Input): The purpose is to **know how to change weights to reduce the loss**. The error (the difference between the predicted output and the actual target) is propagated backward through the network. During this phase, the gradients of the loss function with respect to each weight and bias are computed using the chain rule, utilizing the stored activations from the forward pass. Formally, compute gradients $\frac{\partial E}{\partial w_{ij}^{(l)}}$ and $\frac{\partial E}{\partial b_i^{(l)}}$ for all weights and biases using the chain rule. The stored activations $a^{(l)}$ are used to compute these gradients efficiently.

Hence the name **backpropagation**: the error is propagated backward through the network to compute gradients efficiently. In the next pages, we will see a

detailed example of how backpropagation works step by step. Formally deriving the backpropagation equations will be covered in later sections.

Example 9: Numerical Example of Backpropagation Concept

We'll train a **1-hidden-layer neural network** to learn the function $y = x$ for one input sample $x = 1$ and target $t = 0.5$. Visually, the network looks like this:



The network architecture is as follows:

- Input layer: 1 neuron, with notation $a^{(0)} = x = 1$.
- Hidden layer: 1 neuron, with a sigmoid activation function:

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

- Output layer: 1 neuron, with notation $y = w_2 \cdot h + b_2$. Where y is the output, w_2 is the weight connecting hidden to output layer, and b_2 is the bias of the output layer. The activation function is linear:

$$g^{(2)}(a) = a$$

The weights and biases are initialized as follows (randomly chosen for this example):

- $w_{11}^{(1)} = 0.4$ (weight from input to hidden layer)
- $b_1^{(1)} = 0.3$ (bias of hidden layer)
- $w_{11}^{(2)} = 0.5$ (weight from hidden to output layer)
- $b_1^{(2)} = 0.1$ (bias of output layer)

The learning rate η is set to 0.1.

Backpropagation Step 1: Forward Pass. In this step, we compute the activations of the hidden and output layers given the input $a^{(0)} = x = 1$.

1. **Compute hidden layer activation** (net input). The neuron has the index $i = 1$ in layer $l = 1$ since it's the first hidden layer (equation 11, page 57):

$$a_1^{(1)} = w_{11}^{(1)} \cdot a^{(0)} + b_1^{(1)} = 0.4 \cdot 1 + 0.3 = 0.7$$

The $a_1^{(1)}$ calculated above is the weighted sum before activation. Now, this value is passed through the sigmoid activation function:

$$h = g^{(1)}\left(a_1^{(1)}\right) = g^{(1)}(0.7) = \sigma(0.7) = \frac{1}{1+e^{-0.7}} \approx 0.668$$

2. **Compute output layer activation**. Again, using equation 11 (page 57), we compute the net input to the output neuron:

$$a_1^{(2)} = w_{11}^{(2)} \cdot h + b_1^{(2)} = 0.5 \cdot 0.668 + 0.1 \approx 0.434$$

But now, Since the output layer uses a linear activation function, the output is:

$$y = g^{(2)}(a_1^{(2)}) = a_1^{(2)} \approx 0.434$$

3. **Compute the error**. The predicted output from the forward pass is $y \approx 0.434$. The error can now be computed using the target $t = 0.5$. For example, using Mean Squared Error (MSE):

$$E = \frac{1}{2}(t-y)^2 = \frac{1}{2}(0.5 - 0.434)^2 \approx 0.0022$$

This error quantifies how far the network's prediction is from the target.

At the end of this step, we have stored (cached) some intermediate values needed for the backward pass:

- **Input activation**: $a^{(0)} = 1$
- **Linear combo to hidden layer**: $a_1^{(1)} = w_{11}^{(1)} \cdot a^{(0)} + b_1^{(1)} = 0.7$
- **Hidden layer activation**: $h = g^{(1)}\left(a_1^{(1)}\right) = \sigma(0.7) \approx 0.668$
- **Linear combo to output layer**: $a_1^{(2)} = w_{11}^{(2)} \cdot h + b_1^{(2)} \approx 0.434$
- **Output activation (identity)**: $y = g^{(2)}\left(a_1^{(2)}\right) \approx 0.434$
- **Error**: $E \approx 0.0022$

Backpropagation Step 2: Backward Pass. In this step, we compute the gradients of the loss function with respect to each weight and bias using backpropagation. We use the Mean Squared Error (MSE, page 68) loss function:

$$E = \frac{1}{2}(t-y)^2$$

Where t is the target output ($t = 0.5$) and y is the predicted output from the forward pass ($y \approx 0.434$). We will compute the gradients layer by layer, starting from the output layer and moving backward to the hidden layer. Also, we will **denote the error term for each layer as** $\delta^{(l)}$ and we propagate it backward through the network (using the cached values from the forward pass!). Simply put, we need to find $\nabla E(w)$:

$$\nabla E(w) = \left\{ \frac{\partial E}{\partial w_{ij}^{(l)}}, \frac{\partial E}{\partial b_i^{(l)}} \right\} = \left\{ \frac{\partial E}{\partial w_{11}^{(2)}}, \frac{\partial E}{\partial b_1^{(2)}}, \frac{\partial E}{\partial w_{11}^{(1)}}, \frac{\partial E}{\partial b_1^{(1)}} \right\}$$

Where:

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial w_{ij}^{(l)}} \qquad \frac{\partial E}{\partial b_i^{(l)}} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial b_i^{(l)}}$$

However, to avoid recomputing terms, we define the error term $\delta^{(l)}$ for layer l as:

$$\delta^{(l)} = \frac{\partial E}{\partial a_i^{(l)}} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial a_i^{(l)}}$$

This allows us to express the gradients more compactly:

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \delta^{(l)} \cdot a_j^{(l-1)} \qquad \frac{\partial E}{\partial b_i^{(l)}} = \delta^{(l)}$$

These equations may be unfamiliar now, but they will make sense in future sections when we derive them formally. For now, let's treat them as given formulas for computing the gradients using the error terms $\delta^{(l)}$. Now, we compute $\delta^{(l)}$ for each layer starting from the output layer and moving backward to the hidden layer. The steps are as follows:

1. **Compute the error at the output layer**:

$$\begin{aligned} \frac{\partial E}{\partial b_i^{(l)}} = \frac{\partial E}{\partial b_1^{(2)}} = \delta^{(2)} &= \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial a_1^{(2)}} \\ &= \frac{\partial}{\partial y} \left(\frac{1}{2}(t-y)^2 \right) \cdot 1 \\ &= -(t-y) \cdot 1 \\ &= -(0.5 - 0.434) = -0.066 \end{aligned}$$

   It represents how much the output y deviates from the target t. Here we used the output activation function value y from the **cached values**.

2. **Compute gradients for weights and biases in the output layer**:

$$\underbrace{\frac{\partial E}{\partial w_{11}^{(2)}}}_{\frac{\partial E}{\partial w_{ij}^{(l)}}} = \delta^{(2)} \cdot \underbrace{a_1^{(1)}}_{a_j^{(l-1)}} = \delta^{(2)} \cdot h \approx -0.066 \cdot 0.668 \approx -0.044$$

It shows how much the weight $w_{11}^{(2)}$ should be adjusted to reduce the error. It will be used to update the weight during the optimization step. Again, we used the hidden layer activation ($h \approx 0.668$) from the **cached values**. Now, also compute the gradient for the bias that is obtained directly from the previously step (also **cached**):

$$\frac{\partial E}{\partial b_i^{(l)}} = \frac{\partial E}{\partial b_1^{(2)}} = \delta^{(2)} \approx -0.066$$

3. Compute the error at the hidden layer:

$$\delta^{(1)} = \delta^{(2)} \cdot w_{11}^{(2)} \cdot g'^{(1)}\left(a_1^{(1)}\right)$$

where $g'^{(1)}(a)$ is the derivative of the sigmoid function:

$$g'^{(1)}(a) = \sigma(a) \cdot (1 - \sigma(a))$$

Thus,

$$g'^{(1)}(0.7) = \sigma(0.7) \cdot (1 - \sigma(0.7)) \approx 0.668 \cdot (1 - 0.668) \approx 0.222$$

Where we used the **cached values** of the activation of the hidden layer $\sigma(0.7) \approx 0.668$ from the forward pass. Therefore,

$$\delta^{(1)} \approx -0.066 \cdot 0.5 \cdot 0.222 \approx -0.0073$$

4. Compute gradients for weights and biases in the hidden layer:

$$\frac{\partial E}{\partial w_{11}^{(1)}} = \delta^{(1)} \cdot a^{(0)} \approx -0.0073 \cdot 1 \approx -0.0073$$

$$\frac{\partial E}{\partial b_1^{(1)}} = \delta^{(1)} \approx -0.0073$$

Step 3: Update Weights and Biases. Finally, we update the weights and biases using the computed gradients and the learning rate $\eta = 0.1$ (all the gradients are **cached** from the backward pass):

- Update weight from hidden to output layer:

$$w_{11}^{(2)} \leftarrow w_{11}^{(2)} - \eta \cdot \frac{\partial E}{\partial w_{11}^{(2)}} \approx 0.5 - 0.1 \cdot (-0.044) \approx 0.5044$$

- Update bias of output layer:

$$b_1^{(2)} \leftarrow b_1^{(2)} - \eta \cdot \frac{\partial E}{\partial b_1^{(2)}} \approx 0.1 - 0.1 \cdot (-0.066) \approx 0.1066$$

- Update weight from input to hidden layer:

$$w_{11}^{(1)} \leftarrow w_{11}^{(1)} - \eta \cdot \frac{\partial E}{\partial w_{11}^{(1)}} \approx 0.4 - 0.1 \cdot (-0.0073) \approx 0.40073$$

- Update bias of hidden layer:

$$b_1^{(1)} \leftarrow b_1^{(1)} - \eta \cdot \frac{\partial E}{\partial b_1^{(1)}} \approx 0.3 - 0.1 \cdot (-0.0073) \approx 0.30073$$

After this single iteration of forward and backward passes, the weights and biases have been updated to better approximate the target function $y = x$ for the input $x = 1$. Repeating this process over many iterations and samples will allow the network to learn the desired mapping.

2.5 Maximum Likelihood Estimation (MLE)

In the previous sections, we discussed how to *compute* gradients (backpropagation) and *optimize* weights (gradient descent) in neural networks. But **why** do we minimize a loss function in the first place? What's the **statistical justification** for this approach? Is there a **probabilistic interpretation** of learning in neural networks? And why is it important?

We've already seen that when we train a neural network, we *minimize a loss function* $E(w)$ (page 84). MLE gives a **statistical justification** for that loss: it's equivalent to **maximizing the probability** of observing our data given the model parameters. In other words, **MLE turns learning into a probability problem**.

Definition 9: Maximum Likelihood Estimation (MLE)

Let our model have parameters θ (or w in neural networks), and our dataset be:

$$\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$$

We assume that each data point is drawn from a probability distribution that depends on those parameters:

$$p\left(x_n \mid \theta\right)$$

If all samples are **i.i.d.** (independent and identically distributed) then the probability of the entire dataset is:

$$p\left(\mathcal{D} \mid \theta\right) = \prod_{n=1}^{N} p\left(x_n \mid \theta\right)$$

This is called the **likelihood function**:

$$L\left(\theta\right) = p\left(\mathcal{D} \mid \theta\right) = \prod_{n=1}^{N} p\left(x_n \mid \theta\right) \tag{47}$$

Then the **Maximum Likelihood Estimation (MLE)** is the value of θ that maximizes this likelihood:

$$\hat{\theta}_{\text{MLE}} = \arg\max_{\theta} L\left(\theta\right) = \arg\max_{\theta} p\left(\mathcal{D} \mid \theta\right) \tag{48}$$

In practice, we often maximize the **log-likelihood** instead:

$$\hat{\theta}_{\text{MLE}} = \arg\max_{\theta} \log L\left(\theta\right) = \arg\max_{\theta} \log p\left(\mathcal{D} \mid \theta\right) \tag{49}$$

Because the logarithm is a monotonic function, maximizing the log-likelihood is equivalent to maximizing the likelihood itself. Also, the logarithm allows us to turn products into sums, which are easier to work with.

What does i.i.d. mean?

The acronym **i.i.d.** stands for *independent and identically distributed*. It's a **fundamental assumption** in statistics that simplifies how we model data. When we say that our samples are i.i.d., we mean two things:

1. **Independent**. Each observation (data point) does **not depend** on the others. Formally:

$$p(x_1, x_2, \ldots, x_N) = \prod_{n=1}^{N} p(x_n)$$

   So, knowing x_1 tells us nothing about x_2, and so on. Each is separate, independent draw from the underlying process. For example, if we toss a coin 10 times, each toss is independent of the others, assuming the coin is fair and the tosses don't influence each other.

2. **Identically Distributed**. All samples come from the **same probability distribution**. Formally:

$$x_n \sim p(x \mid \theta) \quad \forall n$$

   This means they are generated by the **same model parameters** (θ) (e.g., same mean and variance in a Gaussian). For example, each coin toss has the same probability ($P(\text{Head}) = 0.5$).

So, when we say:

$$x_1, x_2, \ldots, x_N \overset{\text{are i.i.d.}}{\sim} p(x \mid \theta)$$

We mean that each sample x_n is an independent draw from the **same** distribution underlying probability parameterized by θ. This is the **standard assumption** when deriving the Likelihood:

$$p(\mathcal{D} \mid \theta) = \prod_{n=1}^{N} p(x_n \mid \theta)$$

Without the i.i.d. assumption, we couldn't factor the joint probability into a product of individual probabilities, making the analysis much more complex.

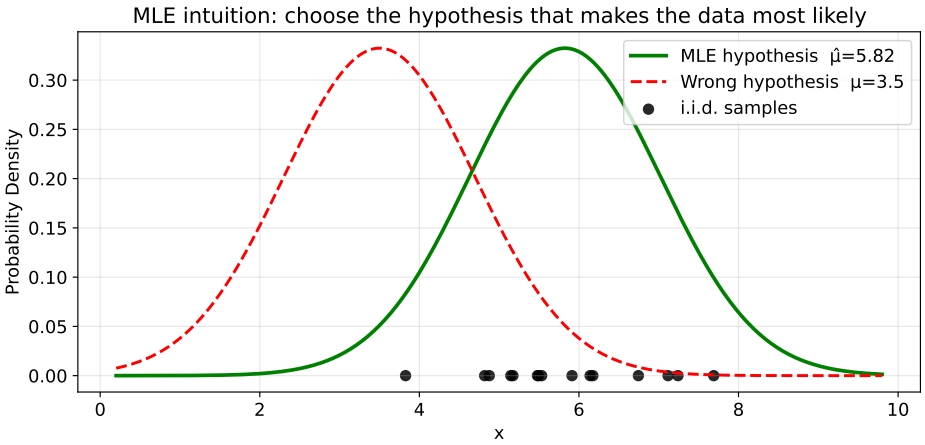
Core Idea of MLE

The core idea of MLE is to find the parameter values that make the **observed data most probable** under our assumed statistical model. By maximizing the likelihood function, we identify the parameters that best explain the data we have collected. Let's illustrate this with a simple example.

Imagine a **Gaussian distribution** (bell curve) centered around some unknown true mean $\hat{\mu}$. Each black point below the curve represents one **sample** (observation): $x_1, x_2, \ldots, x_N$. These samples:

- Come from the **same distribution** $\mathcal{N}(\mu, \sigma^2)$, so they are **identically distributed**. It means they share the same mean μ and variance σ^2.
- Are drawn **independently** from each other, so they are **independent**.

If we were to repeat the sampling many times, we'd get different sets of samples, but all coming from *the same bell curve*.



Now, MLE asks: "*which value of μ would make these observed points most likely under the Gaussian model?*" If we shift the curve too far left or right, the points no longer lie near its center, reducing their likelihood. The "best" μ is the one centering the curve on the data cloud, called $\hat{\mu}_{\text{MLE}}$. This $\hat{\mu}_{\text{MLE}}$ is the **Maximum Likelihood Estimate** of the mean.

Let's derive the MLE for the mean of a Gaussian distribution step-by-step.

1. **Model assumption**. We assume our data points are i.i.d. samples from a Gaussian distribution:

   $$x_1, x_2, \ldots, x_N \sim \mathcal{N}(\mu, \sigma^2)$$

   Where σ^2 is known, and μ is the parameter we want to estimate. Each point has probability density:

   $$p\left(x_n \mid \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_n - \mu)^2}{2\sigma^2}\right)$$

   The **probability density** is a function that describes the likelihood of a random variable taking on a specific value. For continuous variables, it indicates how dense the probability is around that value.

2. **Likelihood of the dataset**. The likelihood of the entire dataset, assuming i.i.d. samples, is:

   $$L\left(\mu\right) = p\left(\mathcal{D} \mid \mu, \sigma^2\right) = \prod_{n=1}^{N} p\left(x_n \mid \mu, \sigma^2\right)$$

3. **Log-likelihood**. We take the logarithm of the likelihood to simplify calculations:

   $$\log \ell\left(\mu\right) = -\frac{N}{2} \log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2} \sum_{n=1}^{N} \left(x_n - \mu\right)^2$$

   Derivative with respect to μ. We differentiate the log-likelihood with respect to μ because we want to find the value of μ that maximizes it (and

the gradient tells us where the maximum is):

$$\begin{aligned}\frac{\partial \ell}{\partial \mu} &= -\frac{1}{2\sigma^2} \cdot \frac{\partial}{\partial \mu} \sum_{n=1}^{N} (x_n - \mu)^2 \\ &= -\frac{1}{2\sigma^2} \cdot \sum_{n=1}^{N} 2\,(\mu - x_n) \\ &\Rightarrow \frac{1}{\sigma^2} \cdot \sum_{n=1}^{N} (x_n - \mu)\end{aligned}$$

4. **Set derivative to zero**. To find the maximum, we set the derivative equal to zero and solve for μ, because at the maximum point, the slope of the function is zero:

$$\begin{aligned}0 &= \frac{1}{\sigma^2} \cdot \sum_{n=1}^{N} (x_n - \mu) \\ &\Rightarrow \sum_{n=1}^{N} x_n - N\mu = 0 \\ &\Rightarrow N\mu = \sum_{n=1}^{N} x_n \\ &\Rightarrow \mu = \frac{1}{N} \cdot \sum_{n=1}^{N} x_n\end{aligned}$$

Thus, the MLE for the mean of a Gaussian distribution is the **sample mean**:

$$\hat{\mu}_{\text{MLE}} = \frac{1}{N} \cdot \sum_{n=1}^{N} x_n \tag{50}$$

This derivation shows how MLE provides a principled way to estimate model parameters by maximizing the likelihood of the observed data. In this case, it leads us to the intuitive result that the best estimate for the mean of a Gaussian is simply the average of the observed samples. In simple terms, Gaussian is the geometric intuition behind the MLE of the mean.

Applying MLE to Neural Networks

In neural networks we do something that looks like this:

$$\text{Find } w^* = \arg\min_{w} E\,(w)$$

Where $E(w)$ is the **loss function** (e.g., Mean Squared Error, Binary Cross-Entropy). MLE gives the **statistical justification** for this optimization: minimizing these loses is equivalent to maximizing the likelihood of the observed data under the model defined by the neural network with parameters w. So, **MLE defines *what* loss we should minimize**, and **gradient descent** plus **backpropagation** define ***how* to minimize it**.

1. **Assume a probabilistic model for our targets**. We assume each training example (x_n, t_n) was generated by an unknown process:

$$t_n \sim p\left(t_n \mid x_n, w\right)$$

Where:

- x_n is the input (features).
- t_n is the target (label).
- w are the model parameters (weights of the neural network).
- $p\left(t_n \mid x_n, w\right)$ is the probability of observing target t_n given input x_n and model parameters w.

2. **Apply the Maximum Likelihood principle**. We want to find weights that make our observed dataset as probable as possible:

$$\hat{w}_{\text{MLE}} = \arg\max_w p\left(\mathcal{D} \mid w\right) = \arg\max_w \prod_{n=1}^{N} p\left(t_n \mid x_n, w\right)$$

Taking the logarithm:

$$\hat{w}_{\text{MLE}} = \arg\max_w \sum_{n=1}^{N} \log p\left(t_n \mid x_n, w\right)$$

Or equivalently:

$$\hat{w}_{\text{MLE}} = \arg\min_w E(w)$$

Where:

$$E(w) = -\sum_{n=1}^{N} \log p\left(t_n \mid x_n, w\right)$$

Is the **negative log-likelihood loss**, in neural networks often called simply the **loss function** (used in training).

3. **Depending on the probabilistic assumptions, we get different loss functions**. For example:

- For Regression tasks, the probabilistic model is often Gaussian:

$$t_n \sim \mathcal{N}\left(y\left(x_n, w\right), \sigma^2\right)$$

With logarithm of the likelihood:

$$-\frac{1}{2\sigma^2} \cdot \sum_{n=1}^{N} \left(t_n - y\left(x_n, w\right)\right)^2$$

This leads to the **Mean Squared Error (MSE)** loss (page 69):

$$E(w) = \frac{1}{N} \cdot \sum_{n=1}^{N} \left(t_n - y\left(x_n, w\right)\right)^2 \qquad \underbrace{w^* = \arg\min_w E(w)}_{\text{optimize weights}}$$

- For Binary Classification tasks, the probabilistic model is often Bernoulli:

$$t_n \sim \text{Bernoulli}\left(y\left(x_n, w\right)\right)$$

With logarithm of the likelihood:

$$\sum_{n=1}^{N} \left[t_n \log y + (1 - t_n) \log\left(1 - y\right)\right]$$

This leads to the **Binary Cross-Entropy (BCE)** loss (page 73):

$$E(w) = -\frac{1}{N} \cdot \sum_{n=1}^{N} \left[t_n \cdot \ln\left(y\left(x_n, w\right)\right) + (1 - t_n) \cdot \ln\left(1 - y\left(x_n, w\right)\right)\right]$$

$$\underbrace{w^* = \arg\min_w E(w)}_{\text{optimize weights}}$$

- For Multi-Class Classification tasks, the probabilistic model is often Categorical:

$$t_n \sim \text{Categorical}\left(\text{softmax}(y\left(x_n, w\right))\right)$$

With logarithm of the likelihood:

$$\sum_{n=1}^{N} \sum_{c=1}^{C} t_{n,c} \log y_c$$

This leads to the **Categorical Cross-Entropy (CCE)** loss (page 79):

$$E(w) = -\frac{1}{N} \cdot \sum_{n=1}^{N} \sum_{c=1}^{C} t_{n,c} \cdot \ln\left(y_c\left(x_n, w\right)\right)$$

$$\underbrace{w^* = \arg\min_w E(w)}_{\text{optimize weights}}$$

So if we assume **Gaussian noise**, the MLE leads to MSE; if we assume **Bernoulli labels**, it leads to binary cross-entropy; and if we assume **Categorical labels**, it leads to categorical cross-entropy. This shows how **the choice of loss function is directly tied to our probabilistic assumptions about the data**.

In summary, MLE provides a **statistical foundation** for training neural networks by showing that minimizing common loss functions is equivalent to maximizing the likelihood of the observed data under appropriate probabilistic models. This connection helps us understand **why** we use certain loss functions and guides us in choosing the right one based on the nature of our data and task.

Deepening: How to Choose the Error Function?

Now that we understand where loss functions come from via MLE, the next question is: **How do we choose the right loss function for a given problem?** The choice depends on the **type of task** (regression vs. classification) and the **underlying probabilistic assumptions** about the data.

1. **Linear**. If our model is linear and data are separable, we can use **Perceptron Loss**. This loss focuses on maximizing the margin between classes.
2. **Regression Tasks**. If we're predicting continuous values, we often assume Gaussian noise in the targets. This leads us to use the **Mean Squared Error (MSE)** loss, which corresponds to maximizing the likelihood under a Gaussian model.
3. **Binary Classification Tasks**. If we're classifying inputs into two classes, we typically model the targets as Bernoulli-distributed. This results in using the **Binary Cross-Entropy (BCE)** loss, which maximizes the likelihood under a Bernoulli model.
4. **Multi-Class Classification Tasks**. For problems with more than two classes, we assume a Categorical distribution for the targets. This leads us to use the **Categorical Cross-Entropy (CCE)** loss, which maximizes the likelihood under a Categorical model.

In practice, it's essential to align our choice of loss function with our assumptions about the data generation process. This ensures that our training procedure is statistically sound and that we're optimizing for the most appropriate objective given our specific problem.

2.6 Perceptron Learning Algorithm

After understanding how modern feed-forward neural networks are trained (through **gradient descent**, **backpropagation**, and **MLE**), we now return to the **first learning rule ever proposed** for neural models: the **Perceptron Learning Algorithm** (PLA), introduced by Frank Rosenblatt in 1957.

Why study this old algorithm after learning about modern techniques?

Because it is actually the **ancestor** of the entire modern training framework we just saw (gradient descent, error minimization, MLE, backpropagation). In simple terms, backpropagation didn't appear out of nowhere: it is a generalization of the Perceptron rule. Rosenblatt's 1957 perceptron was the **first algorithm** that *learned from data autonomously*. It introduced three ideas that survive today in modern deep learning:

1. **Weighted sum of inputs → decision**. The perceptron **computes a weighted sum of its inputs and applies a threshold to decide its output** (0 or 1). This is the basis of all neural networks. We have already formalized this as:
$$a = w^T x + b$$

2. **Error-based correction**. The perceptron **updates its weights based on the error it makes on each training example**. This is the core idea of learning from data, which we have seen in modern networks through loss functions and gradient descent.

3. **Iterative improvement**. The perceptron learning **algorithm iteratively adjusts its weights over multiple passes through the training data**, refining its decision boundary. This iterative process is fundamental to modern training algorithms, including stochastic gradient descent.

Every deep learning model, even a transformer, still rests on these same principles.

Remark: What is an algebraic hyperplane?

In $\mathbb{R}^d$ a (linear) **Hyperplane** is the set of points $x \in \mathbb{R}^d$ that satisfy a single linear equation of the form:

$$w^T x + w_0 = 0$$

Where:

- $w \in \mathbb{R}^d$ is the **normal vector** (perpendicular to the hyperplane).
- w_0 is the **bias/offset** shifting the hyperplane from the origin (or b in our notation).
- The **signed distance** of any point x from the hyperplane is given

by:

$$d(x, \Pi) = \frac{w^T x + w_0}{\|w\|}$$

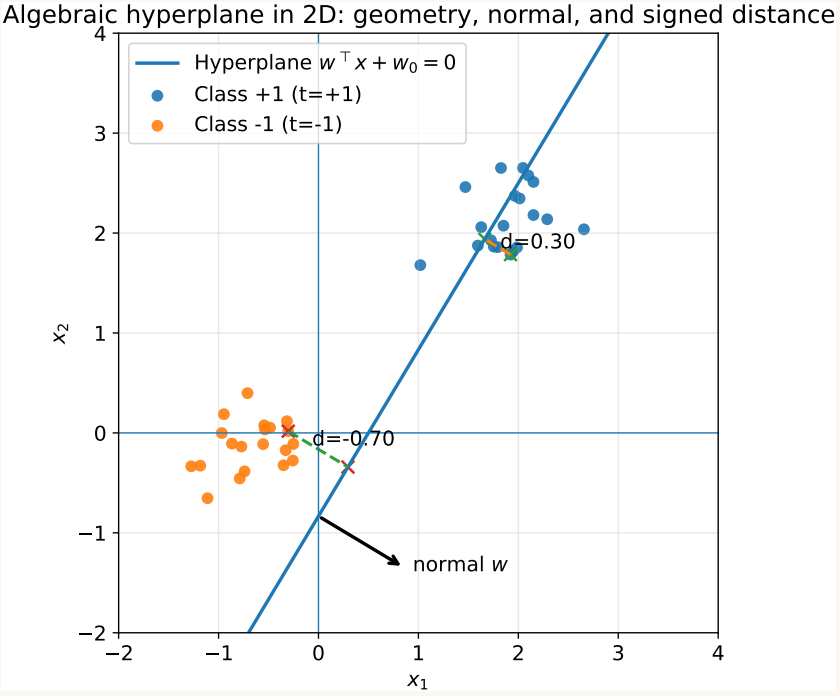
Its **sign** tells the side of the hyperplane; its **magnitude** is the perpendicular distance.

For a linear classifier (perceptron), we assign class by the sign:

$$\text{class}(x) = \text{sign}\left(w^T x + w_0\right)$$

In 2D, a hyperplane is a line (see below). In the following figure, the hyperplane separates two classes of points in $\mathbb{R}^2$. Two example points are projected perpendicularly onto the hyperplane to illustrate their distances.

Algebraic hyperplane in 2D: geometry, normal, and signed distance



The perceptron learning algorithm is designed for **binary classification tasks** where the data is **linearly separable**. It iteratively adjusts the weights of the perceptron based on the errors it makes on the training data, aiming to find a hyperplane that separates the two classes. So, we want to derive an **error function** that expresses how *wrong* the perceptron is, such that we can later minimize it using a gradient-based rule.

1. **Start from the *algebraic hyperplane***. For a linear classifier, the decision boundary is a**hyperplane**:

$$w^T x + w_0 = 0$$

Where:

- w is the normal vector to the plane (its orientation).
- w_0 is the bias term (offset from the origin).
- x is an input vector (sample).

Every point x_i can lie:

- **Above the plane** if $w^T x_i + w_0 > 0$ (positive side, class $+1$).
- **Below the plane** if $w^T x_i + w_0 < 0$ (negative side, class -1).
- **On the plane** if $w^T x_i + w_0 = 0$ (neutral).

The value calculated by $w^T x_i + w_0$ is called the **activation**, **algebraic distance** or **net input** (page 57):

$$a_i = w^T x_i + w_0$$

2. **Encode class labels as $t_i \in \{+1, -1\}$**

   - If a sample belongs to the positive class, set $t_i = +1$.
   - If it belongs to the negative class, set $t_i = -1$.

   Then the **predicted class** (the output of the perceptron) is given by:

$$\hat{y}_i = \text{sign}\,(a_i) = \text{sign}\left(w^T x_i + w_0\right)$$

3. **Determine whether a sample is correct or misclassified**. We multiply the predicted *algebraic distance* by the true label t_i:

$$t_i \cdot a_i \Rightarrow t_i \cdot \left(w^T x_i + w_0\right)$$

   This product indicates correctness:

Case	Expression	Meaning
Correctly classified	$t_i \left(w^T x_i + w_0\right) > 0$	Sample lies on the correct side of the hyperplane.
Misclassified	$t_i \left(w^T x_i + w_0\right) < 0$	Sample lies on the wrong side.
Exactly on the boundary	$t_i \left(w^T x_i + w_0\right) = 0$	Neutral (rare case).

   So this single product already *encodes correctness* of the classification.

4. **Define the *set of misclassified samples***. We now define the set of misclassified samples M as:

$$M = \left\{i \mid t_i \left(w^T x_i + w_0\right) < 0\right\}$$

   These are the indices of samples that are on the wrong side of the hyperplane (i.e., misclassified).

5. **Measure "how wrong" those samples are**. Each misclassified point x_i with $i \in M$ lies at a **signed distance** from the hyperplane given by:

$$d_i = \frac{w^T x_i + w_0}{|w|}$$

Since the point is misclassified, $t_i \cdot \left(w^T x_i + w_0\right) < 0$, so the signed distance is **negative** (i.e., it's on the wrong side). To simplify, we can just use the **activation** $a_i = w^T x_i + w_0$ as a measure of error (ignoring the normalization by $|w|$). This simplification isn't a problem because it only scales the error, not its direction.

6. **Build an error function using those distances**. To quantify the total error made by the perceptron, we sum the negative activations of all misclassified samples. So, we want a scalar function $D\left(w, w_0\right)$ that:

   - Is **positive** when there are misclassifications.
   - Is **zero** when all samples are correctly classified.
   - Increases when misclassified points are further from the decision boundary.

   Simply summing the negative activations of misclassified points achieves this:

$$D\left(w, w_0\right) = -\sum_{i \in M} t_i \cdot \left(w^T x_i + w_0\right)$$

   Where:

   - $w^T x_i + w_0$ is the algebraic distance (activation) of sample i from the hyperplane.
   - t_i is the true label ($+1$ or -1) that flips the sign for misclassified points (flipping it makes it negative when misclassified).
   - The negative sign in front ensures that D is positive when there are misclassifications (since $t_i \cdot \left(w^T x_i + w_0\right) < 0$ for misclassified points).
   - The sum over $i \in M$ aggregates the errors from all misclassified samples.

   So this function is **large when many points are misclassified or far away**, and **zero when all points are correctly classified**.

7. **Objective: Minimize this error function**. The perceptron learning algorithm aims to find weights w and bias w_0 that minimize the error function $D\left(w, w_0\right)$. Formally:

$$\min_{w, w_0} D\left(w, w_0\right) = \min_{w, w_0} \left(-\sum_{i \in M} t_i \cdot \left(w^T x_i + w_0\right) \right)$$

Minimizing D will push $t_i \cdot \left(w^T x_i + w_0\right)$ to be **positive** for all samples, meaning all points will be **correctly classified**.

Now that we have defined the error function $D(w, w_0)$ that quantifies how wrong the perceptron is, we can proceed to derive the **Perceptron Learning Algorithm** by finding a way to update the weights w and bias w_0 to minimize this error. This will involve computing the gradients of D with respect to w and w_0, and using these gradients to iteratively adjust the parameters in the direction that reduces the error. In simple terms, we have developed the recipe (the error function) and now we will derive the cooking instructions (the learning rule).

1. **Compute the gradient**. We'll compute the **partial derivatives** with respect to w and w_0:

$$\begin{aligned}\frac{\partial D}{\partial w} &= -\sum_{i \in M} t_i \cdot \frac{\partial \left(w^T x_i + w_0\right)}{\partial w} \\ &= -\sum_{i \in M} t_i \cdot x_i\end{aligned}$$

$$\begin{aligned}\frac{\partial D}{\partial w_0} &= -\sum_{i \in M} t_i \cdot \frac{\partial \left(w^T x_i + w_0\right)}{\partial w_0} \\ &= -\sum_{i \in M} t_i \cdot 1\end{aligned}$$

2. **Apply gradient descent**. Gradient descent updates parameters **in the direction opposite to the gradient** to minimize the error:

$$w^{(\text{new})} = w^{(\text{old})} - \eta \cdot \frac{\partial D}{\partial w}$$

$$w_0^{(\text{new})} = w_0^{(\text{old})} - \eta \cdot \frac{\partial D}{\partial w_0}$$

Where η is the learning rate (a small positive constant). Substituting the partial derivatives calculated earlier:

$$w^{(\text{new})} = w^{(\text{old})} - \eta \cdot \left(-\sum_{i \in M} t_i \cdot x_i\right) = w^{(\text{old})} + \eta \cdot \sum_{i \in M} t_i \cdot x_i$$

$$w_0^{(\text{new})} = w_0^{(\text{old})} - \eta \cdot \left(-\sum_{i \in M} t_i\right) = w_0^{(\text{old})} + \eta \cdot \sum_{i \in M} t_i$$

Trying to optimize over all misclassified points at once. Here, we are happy to see that the updates **add contributions from all misclassified points**, pushing the weights and bias in the direction that reduces their error (i.e., moves *all* misclassified points to the correct side of the hyperplane $\sum_{i \in M} t_i \cdot x_i$ and $\sum_{i \in M} t_i$).

However, this approach can be computationally expensive for large datasets since it requires summing over all misclassified points in each update. Furthermore, there is no randomness, which can lead to slow convergence and getting stuck in suboptimal solutions. To address these issues, we can add a pinch of **stochasticity** to the learning process.

3. **Make it *stochastic***. The word "stochastic" means "*involving randomness or uncertainty*". It comes from the Greek "*stochastikos*", meaning "*able to guess*" or "*randomly determined*". So a **stochastic process** is one that includes random variables or random choices; instead, a **deterministic process** always behaves the same way for the same inputs (no randomness). In neural networks, **stochastic** means that we don't **compute the gradient** using *all* training data at once, but rather we approximate it **using a *subset*** (or even a single sample). This introduces a bit of *random noise* in each update, but makes learning faster and more dynamic.

   In perceptron learning algorithm, computing the whole sum over all misclassified points M can be expensive and inefficient, especially for large datasets. Instead, we can update **one misclassified point at a time**:

$$w^{(\text{new})} = w^{(\text{old})} + \eta \cdot t_i \cdot x_i$$
$$w_0^{(\text{new})} = w_0^{(\text{old})} + \eta \cdot t_i$$

   - $t_i \cdot x_i$ are the points in the direction that will move x_i to the correct side of the hyperplane.
   - $\eta > 0$ is the **learning rate** (small step size).

   This approach is called **Stochastic Gradient Descent (SGD)** (or **Batch Gradient Descent** with batch size 1, where the **batch size is the number of samples used to compute the gradient at each step**) because we use a single (or a few) random samples to approximate the gradient, rather than the full dataset.

   This makes the learning process more efficient and allows the perceptron to adapt quickly to new data because each update is based on the most recent misclassification and not the entire dataset at once. See the Figure 13 for an illustration of stochastic vs full-batch updates (page 116).

4. **Interpret geometrically**. We can interpret the updates geometrically:

   - **Case 1: Point correctly classified**. If $t_i \cdot \left(w^T x_i + w_0\right) > 0$, no update is made.
   - **Case 2: Point misclassified**. If $t_i \cdot \left(w^T x_i + w_0\right) < 0$, then update using:

$$w \leftarrow w + \eta \cdot t_i \cdot x_i$$
$$w_0 \leftarrow w_0 + \eta \cdot t_i$$

     This pushes the hyperplane **towards the misclassified point**, reducing its error $D\left(w, w_0\right)$.

   Intuitively:

   - If $t_i = +1$ (positive class), the point is misclassified, so we **add** x_i to w to move the hyperplane closer to w.
   - If $t_i = -1$ (negative class) and the point is misclassified, we **subtract** x_i from w to move the hyperplane away from w.

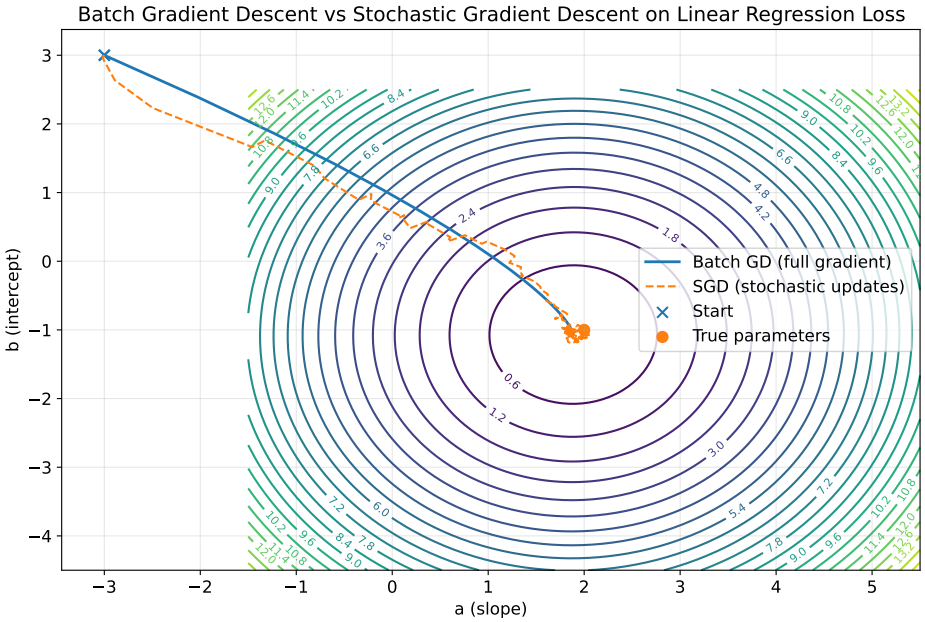


Figure 13: An illustration example of stochastic updates vs full-batch updates. The contours are the **MSE loss** for a simple linear regression in parameter space (a, b). **Batch Gradient Descent** follows a smooth path (solid line) using the **full dataset gradient** each step, while **Stochastic Gradient Descent** (dashed line) takes a more erratic path using **individual sample gradients**, leading to faster but noisier convergence.

```
// Init
Initialize w, w_0 = small random values
Set learning rate η > 0
// Training loop
while not converged:
    for each training sample (x_i, t_i):
        // Compute activation
        a_i = w^T x_i + w_0
        // If misclassified update weights and bias
        if t_i · a_i ≤ 0:
            w ← w + η · t_i · x_i
            w_0 ← w_0 + η · t_i
```

Listing 1: Perceptron Learning Algorithm

2.7 Summary

This section starts with the **simplest neuron** (the perceptron, page 38) and ends with the **first learning algorithm** (the Perceptron Learning Rule), showing how these early ideas evolved into the modern **feed-forward networks** that can learn complex, non-linear patterns.

1. **Where we started: the Perceptron model** (page 38). At the beginning of the chapter, we see historical context: the perceptron as the first trainable neural network model, invented by Frank Rosenblatt in 1957. It introduced the idea of adjusting weights based on errors to learn from data:
$$y = \text{sign}\left(w^T x + b\right)$$
This was the **first artificial neuron**, a linear classifier with a hard threshold activation function. However, it could only solve linearly separable problems and had limitations (e.g., XOR problem).

2. **Multilayer networks (FNNs)**. To overcome this limitation, we introduced **hidden layers**, **differentiable activations** (sigmoid, tanh), and **continuous outputs**. Now the model can approximate **any continuous function**, not just linear boundaries. This is what transforms the perceptron into a **Feed-Forward Neural Network (FNN)**.

3. **How do we train these networks?** That lead to:

   - **Gradient descent**: an optimization algorithm to minimize the loss function by iteratively updating weights in the direction of the steepest descent.
   - **Backpropagation**: an efficient way to compute gradients for all weights in the network using the chain rule of calculus (a sort of cache mechanism to avoid redundant calculations).
   - **Maximum Likelihood Estimation (MLE)**: a statistical framework to derive loss functions (e.g., cross-entropy for classification, mean squared error for regression) based on the likelihood of the observed data given the model parameters.

   We saw how modern NNs *learn*, layer by layer, using data and gradients.

4. **Return to the perceptron learning algorithm**. Once we understood *how modern networks learn*, we revisited the **Perceptron Learning Algorithm** as a simple case of these principles. It uses a Stochastic Gradient Descent (SGD) approach to update weights based on individual training examples.

3 Neural Networks and Overfitting

3.1 Universal Approximation Theorem

During the 1980s, researchers were trying to **understand the theoretical power** of neural networks. Before this period, many scientists were skeptical about the capabilities of neural networks: "*could neural networks really learn any kind of relationship between inputs and outputs, or were they limited to simple functions?*". In 1989-1991, a series of papers by:

- **George Cybenko (1989)**: *"Approximation by superpositions of a sigmoidal function"* [1];
- **Kurt Hornik (1991)**: *"Approximation Capabilities of Multilayer Feedforward Networks"* [3].

Proved rigorously that: even a **single hidden layer** feed-forward neural network with enough neurons and a non-linear activation (like sigmoid or tanh) can approximate **any continuous function** on a compact domain of $\mathbb{R}^n$ to any desired degree of accuracy. This result gave **mathematical legitimacy** to neural networks, showing that they are not just *pattern machines*, but theoretically *universal function approximators*.

Formal Statement

Let's formalize the theorem.

Theorem 2 (**Universal Approximation Theorem**)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a **continuous** function defined on a **compact set** $K \subset \mathbb{R}^n$.*

Then, for any smaller number $\varepsilon > 0$, there exists a neural network function $F(x)$ of the form:

$$F(x) = \sum_{j=1}^{m} \alpha_j \cdot g\left(w_j^T x + b_j\right) \tag{51}$$

Such that:

$$|F(x) - f(x)| < \varepsilon \quad \forall x \in K \tag{52}$$

Where:

- $g(\cdot)$ *is a **nonlinear**, **continuous**, **bounded activation function** (e.g., sigmoid, tanh);*
- α_j, w_j, b_j *are the network parameters (weights and biases);*
- m *is the number of neurons in the hidden layer.*
- ε *is the desired approximation accuracy.*

*In other words, with **enough hidden neurons**, a simple feedforward neural network can represent **any function**, no matter how complex.*

Intuition

The Universal Approximation Theorem tells us that **neural networks are incredibly powerful function approximators**. Even with just a **single hidden layer**, they can learn to represent **any continuous function** to an arbitrary degree of accuracy, as long as we provide enough neurons. This is because the non-linear activation functions allow the network to **combine simple building blocks** (the outputs of individual neurons) into **complex structures** that can capture intricate patterns in the data. However, it's important to note that while the theorem guarantees the existence of such a network, it does not provide a practical way to find the right architecture or parameters, nor does it address issues like **overfitting** or **generalization** to unseen data.

Geometric Intuition. Geometrically, each neuron in the hidden layer defines a **non-linear "bump"** or "ridge" in the input space. By combining enough of these bumps (weighted by α_j), the network can **shape its output** to follow any desired surface. Visually, if we had a 2D function $f(x_1, x_2)$, each hidden neuron adds a small deformation to the surface. Stacking many of them yields a highly flexible model:

$$f(x_1, x_2) \approx \sum_{j=1}^{m} \alpha_j \cdot g\left(w_{j1} \cdot x_1 + w_{j2} \cdot x_2 + b_j\right)$$

So, neural networks are **universal sculptors** of mathematical functions, capable of molding their outputs to fit any continuous shape we desire.

Practical Implications

The theorem tells us **existence**, not **constructability** (it says a perfect network *exists*, but not how to find its weights efficiently). In practice, we rely on **training algorithms (like backpropagation)**, which may get stuck in local minima or underfit/overfit the data. Also, even though a single layer is theoretically enough, **deep networks** (many layers) cam approximate the same function with *fewer neurons* and *more efficient representations*. This is why *deep learning* became the dominant paradigm.

Ockham's Razor and Model Simplicity

The idea dates back to **William of Ockham (c. 1285-1349)**, a Franciscan friar and philosopher who formulated a logical and methodological principle still fundamental to science and machine learning "Entia non sunt multiplicanda praeter necessitatem" (entities must not be multiplied beyond necessity). In essence, prefer the **simplest explanation** that fits the data. This became known as **Ockham's Razor**, where "razor" metaphorically represents the act of **cutting away unnecessary complexity**.

In the context of Neural Networks and Deep Learning, Ockham's Razor suggests that when building models, we should **favor simpler architectures** that adequately capture the underlying patterns in the data without introducing unnecessary complexity. In other words, among all models that can explain the

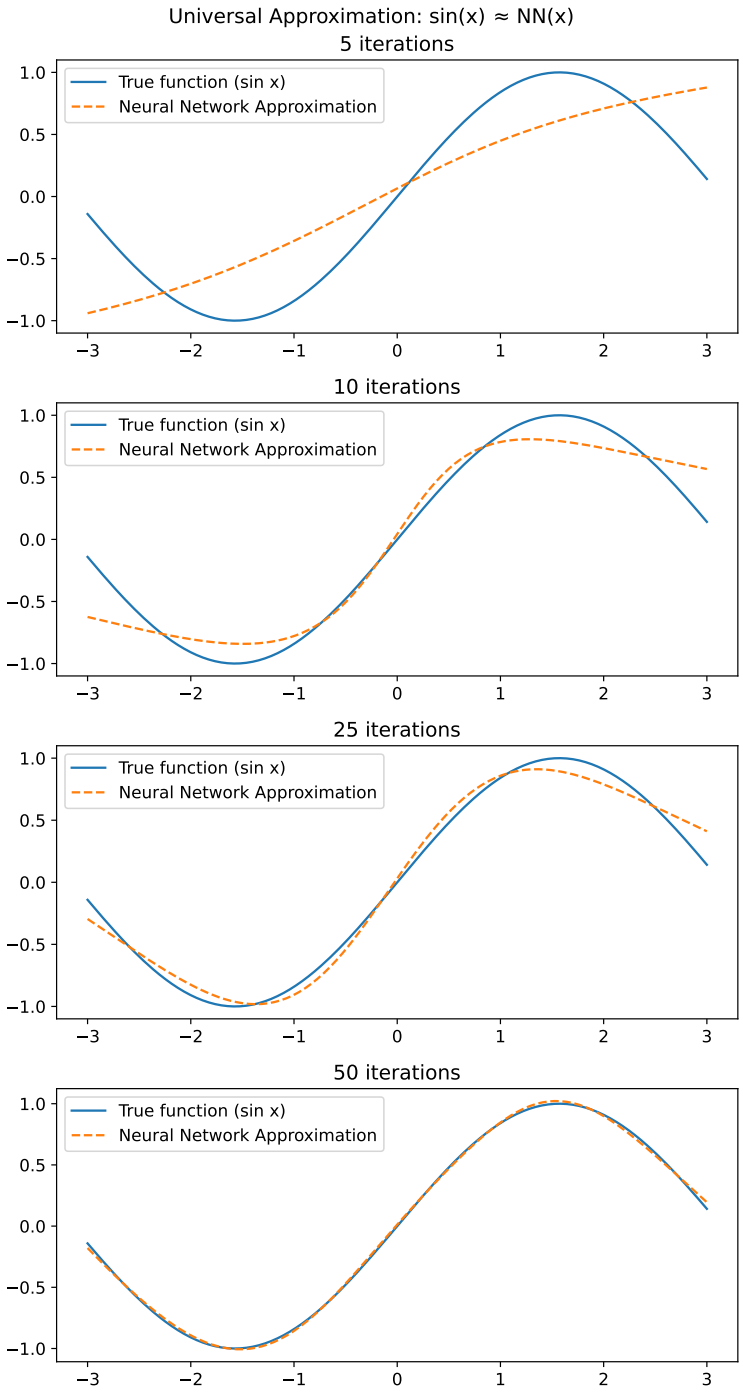


Figure 14: This grid of four plots illustrates how a simple 1-hidden layer network (with 20 neurons) can approximate a nonlinear function, such as a sine wave. This simple 1-hidden layer neural network learns to approximate $\sin(x)$ even though we never told it what "sine" is. Also, this example shows how increasing the number of iterations allows the network to better fit the target function.

training data, choose the one with the **lowest complexity** that still generalizes well to unseen data. Because neural networks are universal approximators, they can model **almost anything**, including true underlying patterns and random noise. But this flexibility is dangerous: a too-powerful model may **memorize** the training data rather than **learn patterns**, leading to **overfitting**. However, a model that is too simple may **underfit**, failing to capture important structures in the data. Thus, Ockham's Razor guides us to find a **balance** between simplicity and complexity, aiming for models that are just complex enough to capture the true patterns without overfitting.

Remark: What is Overfitting?

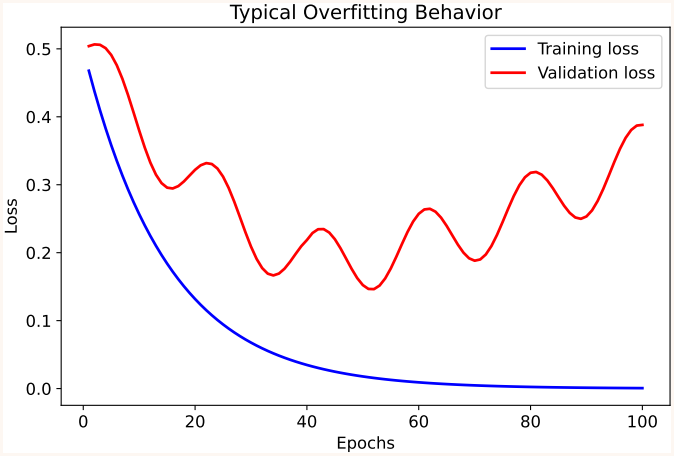
Overfitting occurs when a **model learns** not just the **true patterns** in the data, **but also the random noise**. It's like memorizing answer for an exam instead of understanding the subject; we do well on the questions we saw before (training data), but fail on new ones (test data). Imagine fitting a curve to data points:

Fit Type	Description
Underfitting	The model is too simple (e.g., a straight line when the pattern is curved), it misses important structure.
Good Fit	The model captures the true pattern without being too complex.
Overfitting	The model bends and twists to go exactly through every training point, even if those points contain random noise.

Our model is overfitting when we observe the following:

- **Training error**: very *low*
- **Test error**: *high*

The model performs **too well** on the training set because it's **memorized it**, not generalized it. Usually, we can spot overfitting by plotting training and test errors over time:



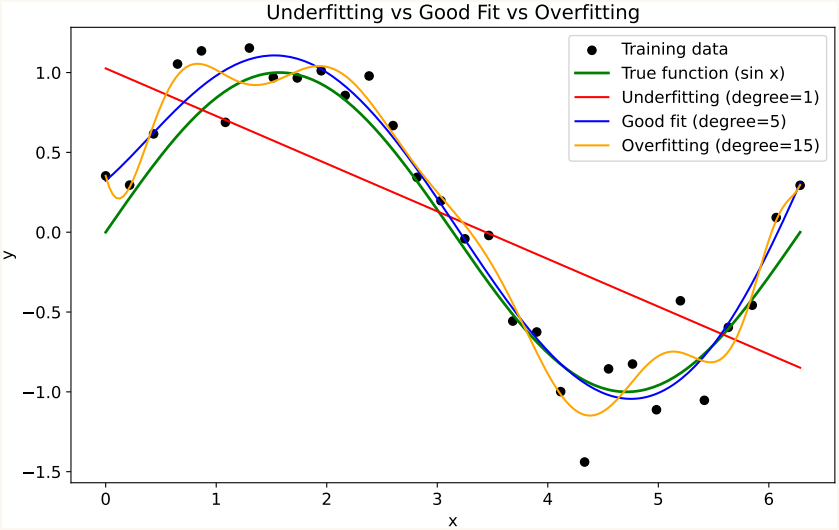
Remark: What is Underfitting?

Underfitting occurs when a model is **too simple** to capture the underlying structure or patterns of the data. It performs poorly not only on unseen (test) data but **also on the training data**. Formally, if $f(x)$ is the true function we want to learn, and $\hat{f}$ is our model's prediction, underfitting means that:

$$\hat{f}(x) \text{ cannot approximate } f(x) \text{ even on the training set}$$

So, the **training error remains high**, and naturally the **test error** will be high as well.

We can think of underfitting as using a **too rigid model** for a complex relationship. For example, trying to fit a **straight line** to data that follows a **sinusoidal** curve, or training a neural network with **too few neurons/layers**, so it can't represent the non-linearities of the data. The model simply **doesn't have enough capacity** (parameters, complexity, expressiveness) to learn the data's structure.



In a neural network context, underfitting can occur when:

- It has **too few neurons** or **layers** to represent the data completely.
- The **training time** is too short → not enough gradient updates to learn the patterns.
- The **learning rate** is too high → never converges properly.
- Strong **regularization** (e.g., large weight decay, dropout) limits learning capacity.
- Features are **not informative** (bad processing, missing normalization).

3.2 Model Complexity

Machine learning, and neural networks in particular, are based on an **inductive assumption**: "if a model performs well on a **large and representative set of training examples**, then it will also perform well on **unseen examples** drawn from the same distribution". This is called **Inductive Hypothesis** (or **Inductive Bias**) because it assumes that the patterns learned from the training data will generalize to new data.

In other words, we **trust that patterns learned from the training data generalize** to future data, as long as the data are **independent and identically distributed (i.i.d.)**, and the model captures the **true underlying structure** rather than random noise.

Formally, if:

$$E_{\text{train}} = \mathbb{E}_{(x,t)\sim D_{\text{train}}}\left[(f(x;w)-t)^2\right]$$

is the expected training error, and

$$E_{\text{test}} = \mathbb{E}_{(x,t)\sim D_{\text{test}}}\left[(f(x;w)-t)^2\right]$$

is the expected test error, where D_{train} and D_{test} are drawn from the same distribution, **then the inductive hypothesis assumes** $E_{\text{train}} \approx E_{\text{test}}$ when:

- $D_{\text{train}} \approx D_{\text{test}}$ (i.i.d. data)
- The model has learned **general rules** rather than memorizing specific examples (i.e., it has not overfitted)

If that **assumption breaks** (e.g., due to *too high complexity*, *data shift*, or *small dataset*), **generalization fails**, leading to poor performance on unseen data.

How can we quantify model complexity?

The **complexity** of a neural network is determined by several factors, including:

- The **number of parameters** (weights and biases) in the network: More parameters generally mean higher complexity.
- The **depth** of the network (number of layers): Deeper networks can capture more complex patterns.
- The **nonlinearities** introduced by activation functions: More complex activation functions can increase the model's capacity to learn intricate patterns.
- The **regularization** applied. Regularization is a technique used to reduce model complexity and prevent overfitting.

As complexity increases, the model's **capacity** to fit the data grows.

Not too complex, not too simple: how to find the right balance?

Finding the right model complexity is crucial for good generalization. This involves balancing:

- **Underfitting**: When the model is too simple to capture the underlying patterns in the data, leading to high bias and poor performance on both training and test data.
- **Overfitting**: When the model is too complex and captures noise in the training data, leading to low training error but high test error (high variance).

This balance is often referred to as the **Bias-Variance Trade-off**, a mathematical intuition that helps explain the relationship between model complexity, bias, and variance:

$$E_{\text{test}} = E\left[\left(y - \hat{f}(x)\right)^2\right] = \underbrace{(\text{Bias}^2)}_{\text{Systematic error}} + \underbrace{\text{Variance}}_{\text{Sensitivity error}} + \underbrace{\text{Noise}}_{\text{Irreducible error}} \tag{53}$$

Where:

- E_{test} is the expected test error.
- y is the true output.
- $\hat{f}(x)$ is the model's prediction.
- **Bias** is the systematic error introduced by approximating a real-world problem with a simplified model. High bias can cause the model to miss relevant relations between features and target outputs (underfitting). For example, a linear model trying to fit a highly nonlinear relationship will have high bias.
- **Variance** is the sensitivity error due to fluctuations in the training data. High variance can cause the model to model the random noise in the training data rather than the intended outputs (overfitting). For example, a very deep neural network with many parameters may fit the training data perfectly but perform poorly on unseen data.
- **Noise** is the irreducible error inherent in the data itself, which cannot be eliminated by any model. For example, measurement errors or inherent randomness in the data generation process contribute to noise.

The goal is to find a model complexity that minimizes the total error, which is the sum of bias and variance, paying attention to underfitting (high bias, low variance) and overfitting (low bias, high variance). In practice, the **inductive hypothesis is what makes machine learning possible**, **model complexity decides whether this hypothesis *holds* or *breaks***, and the **bias-variance trade-off provides a framework to understand and manage this balance**.

Definition 1: Inductive Hypothesis

The **Inductive Hypothesis** (or **inductive bias**) is the fundamental assumption that **a model that performs well on the training data will also perform well on unseen data**, proved that both come from the **same underlying distribution**.

In mathematical terms:

$$E_{\text{test}} \approx E_{\text{train}} \quad \text{if} \quad D_{\text{train}} \sim D_{\text{test}} \tag{54}$$

Where E_{train} and E_{test} are the expected training and test errors, respectively, and D_{train} and D_{test} are the training and test data distributions; the $\sim$ symbol indicates that both datasets are drawn from the same distribution.

This assumption underlies *all* machine learning: without it, no matter how low the training error is, we would have no reason to believe the model will generalize to new data.

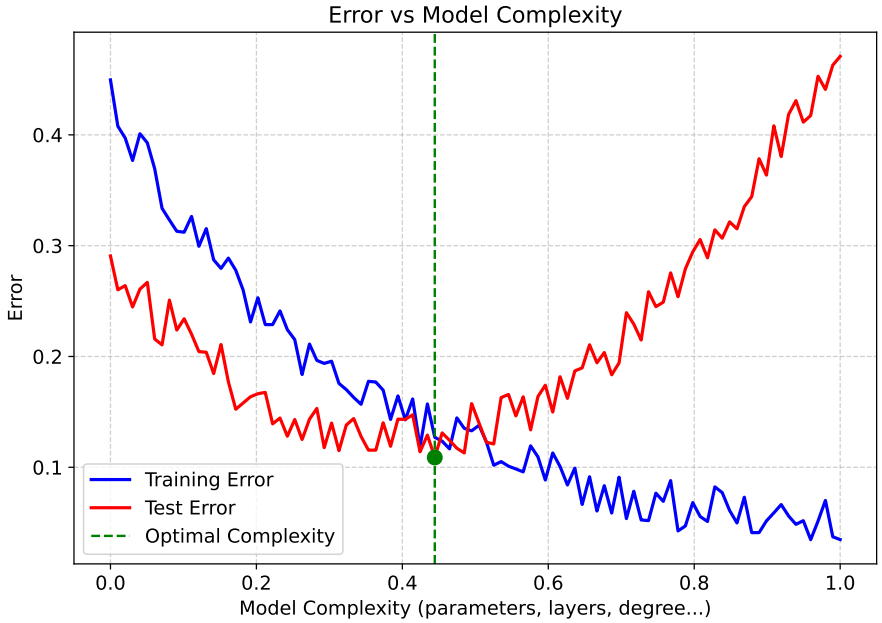


Figure 15: This plot illustrates the **Model Complexity vs. Error Curve**, showing how **training** and **test** errors evolve as **model complexity** increases, giving the classic U-shaped curve for test error. Initially, as model complexity increases, both training and test errors decrease, indicating better fit to the data. However, beyond a certain point (the optimal complexity), the test error starts to increase due to overfitting, while the training error continues to decrease. The optimal model complexity is where the test error is minimized, balancing bias and variance effectively.

3.3 Measuring Generalization

When we train a neural network, we typically monitor the **training loss** (i.e., how well the model predicts the training data). However, a **low training error** does *not* necessarily mean that our model is generalizing well to unseen data.

⚠ **The model has *seen* the training data.** The model's parameters were **directly optimized** to minimize that same error. So it's like asking a student to re-solve the same exercises used in the exam preparation. Success there tells us *nothing* about their ability to handle new ones.

$$E_{\text{train}} = \text{Empirical Risk (on known data)}$$

$$E_{\text{test}} = \text{True Risk (on unseen data)}$$

⚠ **Optimism bias.** Since the model *learned from* those points, the estimate of performance on that data is **optimistically biased**, it always looks better than reality.

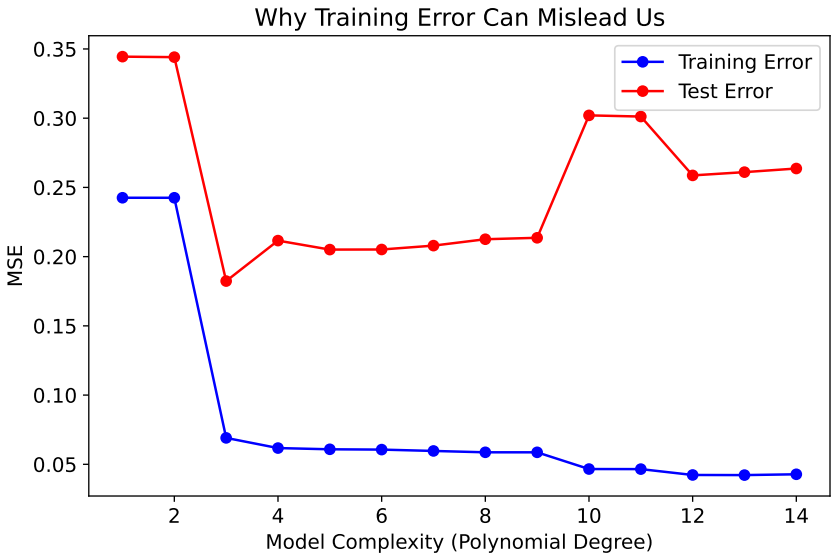
$$E_{\text{train}} \leq E_{\text{test}}$$

Always true for flexible models that can overfit the training data. The more complex the model, the stronger the bias (the model can fit noise, artificially lowering training error E_{train}).

⚠ **Example.** Imagine fitting a very flexible polynomial to noisy data:

- With degree 1 (linear), training error is high (underfitting).
- With degree 15, training error goes to zero, but the curve oscillates widly. Test error on unseen data is huge (overfitting).

That's why we cannot rely on training error E_{train} to assess generalization. It **doesn't measure generalization**, only memorization of training data.



How to measure generalization?

To correctly **measure generalization**, we must evaluate on data the model **has never seen during training**. This is done by **dataset splitting**. We divide the available dataset D into disjoint subsets:

Set	Purpose	Size
Training set	Learn model parameters (weights, biases).	During training.
Validation set	Tune hyperparameters, monitor overfitting.	During training.
Test set	Asses final generalization performance.	After training.

The goal is:

$$\begin{aligned} \text{Training data} &\rightarrow \text{Model fitting} \\ \text{Validation data} &\rightarrow \text{Model selection} \\ \text{Test data} &\rightarrow \text{Model assessment} \end{aligned}$$

This ensures that the **test error** E_{test} is an **unbiased estimate** of the model's true generalization performance on unseen data.

Okay, but how to split the data?

The most common approach is **Random Subsampling** (or Hold-Out Method):

1. Randomly **shuffle** the **dataset** D.
2. Randomly **split** our **dataset** into three disjoint subsets, for example:
   - 70% for training,
   - 15% for validation,
   - 15% for testing.
3. **Train** on the training set.
4. **Tune** on the validation set.
5. **Evaluate** once on the test set.

Because of the randomness in sampling, we often repeat the split several times (with different seeds) and **average** the results to reduce bias. This process is, of course, automated using libraries like `scikit-learn` (Python), not done manually.

Use Stratified Sampling. Imagine we have a dataset for binary classification with 90% of class A and 10% of class B. That means 90% of our data are Class A and only 10% are Class B (**class imbalance**). If we randomly split our dataset into training and test sets (say, 80% training, 20% test), there's a **risk** that one of the splits contains *almost no examples of Class B*. To avoid this, we use **Stratified Sampling**, which ensures that **each subset (train, validation and test) preserves the same class properties as the original dataset**. In our example, both training and test sets would have approximately 90% of Class A and 10% of Class B, maintaining the class distribution.

3.4 Terminology Clarifications

In the context of neural networks and deep learning, certain terms are often used interchangeably or may have nuanced meanings depending on the context. Here are some clarifications on commonly used terminology:

- **Training Dataset**. Let's start from the whole thing we have available to us. The **training dataset** (sometimes called the *available dataset*) is the complete collection of samples we can access for building and evaluating our model. It contains all our labeled examples:

$$\mathcal{D} = \{(x_i, t_i)\}_{i=1}^{N}$$

  But we will **not** train our model on all of them at once. We'll split them into smaller subsets with different purposes.

- **Training Set**. The **training set** is the portion of the data used to **fit the model parameters** (i.e., to adjust the weights and biases so that the network learns patterns). It is used during backpropagation and gradient descent. The loss computed on this set is called the **training loss** and drives weight updates. Performance on this set tells us if the model is *learning*, but **not if it generalizes well**.

$$E_{\text{train}} = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \left(t_i - f\left(x_i; w\right)\right)^2$$

  We *can* monitor the training loss over epochs to see if the model is converging, but it's not sufficient to decide if the model is "good" (we'll soon use validation for that).

- **Validation Set**. The **validation set** is used to **evaluate and tune** the model *during* training, without directly affecting the weights. It's like a "preview" of how the model will perform on new data. The main purposes of the validation set are:
  - Selecting **hyperparameters** (like learning rate, number of neurons, regularization, dropout rate, etc.).
  - Performing **early stopping** (detecting overfitting by monitoring validation loss).
  - Comparing different model architectures.

  We typically compute a **validation loss** or **validation accuracy** after each epoch:

$$E_{\text{val}} = \frac{1}{N_{\text{val}}} \sum_{i=1}^{N_{\text{val}}} \left(t_i - f\left(x_i; w\right)\right)^2$$

  When validation error starts increasing while training error decreases, it's a sign of **overfitting**.

- **Test Set**. The **test set** is used *only once*, at the very end, to obtain an **unbiased estimate of the models generalization performance**. It acts as a *simulation of the real world*: the model has never seen these

samples during training or validation. No gradient updates or hyperparameter tuning should be done based on test set performance. Its purpose is **final assessment** only.

$$E_{\text{test}} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \left(t_i - f\left(x_i; w\right)\right)^2$$

After we've looked at the test performance, we should not go back and tune hyperparameters, otherwise the test set stops being "unseen" data, and our estimate becomes optimistically biased.

- **Golden Rule**: Always keep the test set completely separate until the very end. Use training and validation sets for model development, and only use the test set for final evaluation. So, never use validation/test data to update model weights.

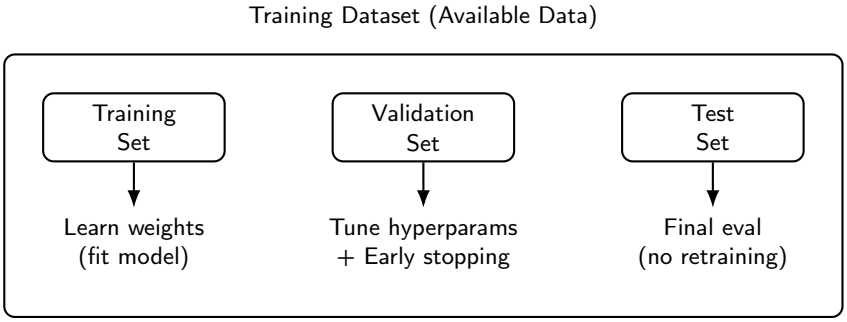


Figure 16: Visualization of the relationships between training, validation, and test sets within the overall training dataset.

3.5 Cross-Validation Techniques

Cross-Validation is the use of the **training dataset** to both: train the model (*parameter fitting* and *model selection*), and **estimate its error on new data**. In other words, we don't need a separate external test dataset for every trial, we can reuse the available data intelligently (e.g., part for training, part for checking generalization).

Main Idea. Instead of holding out a single fixed portion (like in the simple *hold-out* method), cross-validation systematically **rotates** which samples are used for training and which for validation. Each sample eventually acts as validation data once and training data many times. Thus, we can:

- **Train multiple models** on different training subsets;
- **Validate** each on the complementary subset;
- **Average the validation errors** to get a more robust estimate of the model's true performance on unseen data.

It is especially useful when the dataset is limited, as it maximizes the use of available data for both training and validation. However, other techniques (like hold-out, LOOCV, K-Fold) are specific implementations of cross-validation with different trade-offs in terms of bias, variance, and computational cost. In the following, we explore some of these techniques.

3.5.1 Hold-Out Validation

The **Hold-Out Validation** (or **Hold-Out Method**) is the simplest form of cross-validation. It consists of dividing the available dataset into **two or three subsets**, each serving a specific role in training, tuning, and evaluating the model. This method provides a first, practical way to test the **inductive hypothesis**, that a model performing well on unseen data will also generalize to future examples (page 123).

How does it work?

In the typical **three-way split**, we start from the **training dataset**, i.e., all the data we can use to develop the model, we then split it into disjoint subsets:

- **Training Set**: This subset is used to **parameter fitting**, i.e., to train the model by adjusting its weights based on the input-output pairs.
- **Validation Set**: This subset is used to **model selection**, i.e., to evaluate different model configurations (e.g., architectures, hyperparameters) and select the best one based on its performance on this set.
- **Test Set** (hold-out): This subset is used to **model assessment**, i.e., to provide an unbiased evaluation of the final model's performance on unseen data after training and validation are complete.

The steps are as follows:

1. **Divide** the available dataset into training and validation (and possibly test) sets.
2. **Train** the neural networks using only the training set.
3. **Validate** periodically on the validation set to monitor generalization:
   - Tune hyperparameters (e.g., learning rate, architecture) based on validation performance.
   - Apply *early stopping* if validation error starts to increase (indicating overfitting).
4. **Assess** the final model on the hold-out test set (data not seen during training) to estimate its true generalization error.

The hold-out validation gives an estimate of how well the model performs on *unseen data* by simulating future inputs using the reserved portion of the dataset. It's essentially a "**miniature deployment test**" performed before real-world use.

⚠ Risks and Limitations

While hold-out validation is straightforward and easy to implement, it has some limitations. The main risk is that **hold-out validation can be biased** depending on how the data are split:

1. **Non-representative sampling**
   - The validation set may not accurately reflect the data distribution.
   - The estimated generalization error may be too optimistic or too pessimistic.
2. **Small datasets**
   - If we hold out too many samples, there are too few left for training.
   - If we hold out too few, the validation estimate becomes noisy and unreliable.
3. **Unbalanced classes (classification case)**
   - If the classes are not represented equally in the training and validation sets, the model may not learn to generalize well across all classes.

   ✔ **Solution:** use **stratified sampling** (page 127) to maintain class proportions in each subset.
4. **Single split variability**
   - A different random split can yield a different result.
   - The error estimate depends too much on "which data" ended up in the validation set.

   ✔ **Solution:** Later, we will see more robust techniques (like **K-Fold Cross-Validation**) that mitigate this issue by averaging results over multiple splits.

? When to use Hold-Out Validation?

Hold-out validation is most appropriate when:

- ✔ When the **dataset is large enough** to afford separate training, validation and test sets without sacrificing training data. With large enough we mean at least a few thousand samples.
- ✔ When we want **fast model evaluation** without the computational overhead of more complex cross-validation methods (only one training phase).
- ✔ When small fluctuations in the validation split are not expected to significantly affect the model selection process.

3.5.2 Leave-One-Out Cross-Validation (LOOCV)

Leave-One-Out Cross-Validation (LOOCV) is a special case of *K-Fold Cross-Validation* (next section) where the number of folds K is equal to the number of samples N in the dataset. It means that each data point acts **once** as validation data, and $N-1$ times as part of the training set.

How does it work?

1. Given a dataset with N samples:

$$\mathcal{D} = \{(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)\}$$

2. For each sample $i = 1, 2, \ldots, N$:

   - Train the model on all samples except the i-th one:

$$\mathcal{D}_{\text{train}}^{(i)} = \mathcal{D} \setminus \{(x_i, t_i)\}$$

   - Validate (test) the model on the i-th sample:

$$\mathcal{D}_{\text{val}}^{(i)} = \{(x_i, t_i)\}$$

3. Collect all validation errors E_i from each iteration, and compute the **average error**:

$$\hat{E}_{\text{LOOCV}} = \frac{1}{N} \cdot \sum_{i=1}^{N} E_i$$

   This average error $\hat{E}_{\text{LOOCV}}$ gives an **almost unbiased estimate** of the model's generalization error.

Risks and Limitations

While LOOCV maximizes data usage for training and provides a nearly unbiased error estimate, it has some drawbacks:

1. **Computational Cost**. We must train the model N times (once for each sample), which can be **prohibitively expensive** for large datasets or deep neural networks.

2. **High variance in models**. Each training set differs by only one sample, so the models can be **very similar**. This can lead to high variance in the validation errors, making the average error estimate less stable.

3. **Not practical for deep learning**. Neural networks often require thousands of training iterations to converge, making LOOCV impractical for large-scale problems.

When to use LOOCV?

LOOCV has an **unbiased estimate** (each data point serves as validation once, so every sample influences the estimate equally) and an **efficient use of data** (almost all samples are used for training in each iteration, ideal when data are scarce). Thanks to these properties, LOOCV is particularly useful when:

- ✔ When N is **small** (e.g., a few hundred samples or less).
- ✔ When we want an **almost unbiased** generalization error estimate.
- ✔ When computation time is **not a concern** (e.g., simple models).

LOOCV approximates the *true expected error* E_{test} and it's nearly unbiased because each sample plays both roles, training and validation, but it tends to have **high variance** because each training set is almost identical, leading to similar models. However, for modern deep learning, it's mostly of **theoretical interest**, a conceptual benchmark rather than a practical tool.

3.5.3 K-Fold Cross-Validation

K-Fold Cross-Validation divides the available dataset into K equally (or nearly equally) sized subsets, called *folds*. The model is trained and validated K times, each time using a different fold as the **validation set**, and the remaining $K-1$ folds as the **training set**. After completing all K rounds, the K validation errors are **averaged** to estimate the model's overall generalization performance.

How does it work?

1. Given a dataset with N samples:

$$\mathcal{D} = \{(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)\}$$

2. Split the dataset $\mathcal{D}$ into K *folds* (disjoint subsets):

$$\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \ldots \cup \mathcal{D}_K$$

   Where each fold $\mathcal{D}_k$ contains approximately $\frac{N}{K}$ samples:

$$|\mathcal{D}_k| \approx \frac{N}{K}, \quad \text{for } k = 1, 2, \ldots, K$$

3. For each fold $k = 1, 2, \ldots, K$:

   - Train the model on the other $K-1$ folds (i.e., all folds except the k-th one, mathematically $\mathcal{D} \setminus \mathcal{D}_k$):

$$\mathcal{D}^{(k)}_{\text{train}} = \bigcup_{\substack{j=1 \\ j \neq k}}^{K} \mathcal{D}_j$$

   - Validate (test) the model on the k-th fold:

$$\mathcal{D}^{(k)}_{\text{val}} = \mathcal{D}_k$$

   - Compute the validation error $\hat{e}_k$ on the validation set $\mathcal{D}^{(k)}_{\text{val}}$.

4. Collect all validation errors E_k from each iteration, and compute the **average error**:

$$\hat{E}_{\text{K-Fold}} = \frac{1}{K} \cdot \sum_{k=1}^{K} \hat{e}_k$$

   This average error $\hat{E}_{\text{K-Fold}}$ provides an estimate of the model's generalization error.

Differently from LOOCV, K-Fold Cross-Validation allows us to choose a smaller K:

- **5-Fold Cross-Validation** is the most commonly used value, balancing bias and variance in the error estimate.

- **10-Fold Cross-Validation** is also popular, especially in scenarios where more data is available, providing a slightly lower bias at the cost of increased computational time.
- **Stratified K($K = N$)-Fold Cross-Validation** equivalent to LOOCV, where each fold contains exactly one sample. Unbiased but computationally expensive.
- **2 or 3-Fold Cross-Validation** can be used for very large datasets where computational efficiency is a concern, but may lead to higher variance in the error estimate.

⚠ Limitations and ✔ Advantages

Compared to Hold-Out, it uses the entire dataset more efficiently because each sample is used for validation once and for training $K - 1$ times (lower bias). Also, **compared to LOOCV**, it is much cheaper computationally (only K trainings instead of N).

✔ **Efficient data usage**: all samples contribute to both training and validation.

✔ **Reduced bias**: the averaged approximates the expected generalization error better than a single split.

✔ **Stability**: more reliable than hold-out because the specific data division matter less.

✖ **Computational cost**: the model is trained K times, still heavier than a single hold-out validation. For deep neural networks, this can be impractical.

✖ **Data leakage risk**: all processing (normalization, scaling, etc.) must be **recomputed inside each fold**, otherwise information from the validation folds can leak into the training folds.

✖ **Variance in small datasets**: if K is too small, each fold may not represent the full data distribution well.

3.5.4 Nested Cross-Validation

When we train a neural network (or any machine learning model), we actually perform **two separate activities**:

- **Model selection**: choose the *best configuration* (e.g., hyperparameters, architecture) based on performance on a validation set.
- **Model assessment**: estimate the *true generalization ability* of the final chosen model on unseen data.

⚠ But here lies a **problem**: in most cross-validation setups, **we reuse the same data for both tasks**. That means we **peek** at our test data while tuning our model, leading to **optimistic bias** in our performance estimates.

❓ **Why that's a problem (hidden optimism)?** Let's say we're testing 10 different neuronal network architectures and we run **5-fold cross-validation** ($K = 5$) to see which performs best.

1. We choose the one with the lowest average validation error across the 5 folds.
2. Then we report that same average validation error as "the model's performance".

⚠ But this is **optimistically biased**, because:

- We **used** the validation data to pick the best model.
- Therefore, the validation score no longer represents unseen data, because the model (and our choice) are **tuned** to those particular folds we used (even if indirectly).

So we get a number that looks great, but it's **too optimistic**, it doesn't reflect how the model would perform on truly unseen data.

✅ **There is no more optimism with Nested Cross-Validation**

Nested Cross-Validation is a technique that **separates model selection from model assessment** by introducing **two layers of cross-validation loops**:

- An **outer loop** for **model assessment** (unseen test data). It evaluates *how good that chosen configuration* really is on truly unseen data.
- An **inner loop** for **model selection** (tuning hyperparameters). It decides *which configuration* performs best (uses only training data of that outer fold).

So now, when we report the average outer test error, it reflects **real generalization**, not the performance on data we optimized for.

Example 1: Nested Cross-Validation Analogy

Think of it like preparing for an exam:

- We take **practice tests** to decide the best study method (this is the **inner loop, model selection**).
- Then, we take the **final exam** to measure how well our preparation really worked (this is the **outer loop, model assessment**).

If we report our *practice test* scores as our *final exam* result, we'll be unrealistically confident, exactly what happens without nested cross-validation.

How does Nested Cross-Validation work?

1. **Outer Loop (Model Assessment)**:
   - Split the entire dataset into K **outer folds**:
   $$\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K\}$$
   - For each outer fold k:
     - Reserve fold $\mathcal{D}_k^{\text{outer}}$ for **testing**.
     - Use the remaining $K-1$ folds as the **training set** for model selection in the inner loop:
     $$\mathcal{D}_{\text{train}}^{\text{outer}} = \bigcup_{\substack{j=1 \\ j \neq k}}^{K} \mathcal{D}_j$$
2. **Inner Loop (Model Selection)**:
   - Inside that outer training data $\mathcal{D}_{\text{train}}^{\text{outer}}$, perform another K-fold cross-validation, called M**-fold cross-validation**:
   $$\mathcal{D}_{\text{train}}^{\text{outer}} = \left\{\mathcal{D}_1^{\text{inner}}, \mathcal{D}_2^{\text{inner}}, \ldots, \mathcal{D}_M^{\text{inner}}\right\}$$
   To **tune hyperparameters** (e.g., learning rate, number of layers, etc.). The letter M is used to distinguish it from the outer loop's K.
   - Select the configuration θ_k^* that gives the **lowest inner validation error averaged** over the M inner folds:
   $$\theta_k^* = \arg\min_{\theta} \left(\frac{1}{M} \cdot \sum_{m=1}^{M} \text{Error}\left(\mathcal{D}_m^{\text{inner, val}}; \theta\right) \right)$$
3. **Model Evaluation**:
   - Retrain the model on **all inner folds combined** using the selected configuration θ_k^*:
   $$\mathcal{D}_{\text{train}}^{\text{inner}} = \bigcup_{m=1}^{M} \mathcal{D}_m^{\text{inner}}$$

- Evaluate it on the **outer test fold** $\mathcal{D}_k^{\text{outer}}$ to get the test error for that outer fold:

$$\text{Test Error}_k = \text{Error}\left(\mathcal{D}_k^{\text{outer}}; \theta_k^*\right)$$

And store the test error for that outer fold:

$$e_k = \text{Test Error}_k$$

4. **Average over outer folds**. After completing all K outer folds, compute the overall performance estimate:

$$\text{Overall Test Error} = \hat{E}_{\text{nested}} = \frac{1}{K}\sum_{k=1}^{K} e_k$$

This gives an **unbiased estimate** of the model's generalization performance.

Limitations and Advantages

- ✔ Provides an **unbiased generalization estimate**, even after hyperparameter tuning.
- ✔ Ensures a clear **separation between selection and assessment**.
- ✔ Uses **all data efficiently** across different folds.
- ✘ **Computationally expensive**: Each outer folds contains a full inner cross-validation loop, so the model is trained $K \times M$ times. For deep neural networks, this can be prohibitive.
- ✘ **Complex implementation**: Careful bookkeeping is required to manage data splits and model configurations across nested loops. In other words, it's easy to make mistakes if not implemented carefully.

Some typical choices are $K = 5$ for the outer loop and $M = 3$ for the inner loop, balancing computational cost and reliable estimates. For more accurate estimates, $K = 10$ and $M = 5$ can be used, but at a higher computational cost. For small datasets, nested cross-validation is particularly beneficial to avoid overfitting during model selection, and a common choice is $K = 10$ and $M = 5$.

3.6 Preventing Overfitting

So far we've seen:

- **Overfitting**: when the model fits both the signal *and* the noise in the training data, leading to poor generalization on unseen data.
- **Underfitting**: when the model is too simple to capture the underlying patterns in the data, resulting in poor performance on both training and test sets.
- **Cross-Validation**: how to detect when a model generalizes poorly using techniques like Hold-Out Validation, Leave-One-Out Cross-Validation (LOOCV), K-Fold Cross-Validation, and Nested Cross-Validation.

But detecting overfitting is only half the battle. The next crucial step is to implement strategies to **prevent or limit overfitting during training** and enhance the model's ability to generalize well to new, unseen data. This section presents the **three main families of solutions** that modern deep learning uses to *control complexity* and *improve generalization*.

The Core Problem: Neural networks have enormous representational power, by the *Universal Approximation Theorem* (page 118), they can approximate any continuous function. However, if they have **too many parameters**, and **too little data** (or data with noise), they will **memorize** rather than **learn**. Preventing overfitting is thus about **introducing constraints or checks** that force the model to extract **essential structure** from the data, not incidental details.

Overview of Prevention Techniques. In this section, we will explore three main strategies to prevent overfitting in neural networks:

1. **Training control**: **Early Stopping technique** (page 142). This method stops training when validation error starts increasing.
2. **Model complexity control**: **Regularization (Weight Decay, L2) technique** (page 151). This method adds a penalty when weights grow too large, because large weights often indicate memorization.
3. **Randomization / Model averaging**: **Dropout technique** (page 160). This method randomly deactivates neurons during training, forcing the network to learn redundant representations that generalize better.

Each of these reduce the **effective capacity** of the network in a different way. Either by limiting how long it trains, how larger weights can grow, or how tightly neurons depend on each other.

How these techniques relate to the Training Curve

Similar to the curve of **Model Complexity vs. Error** shown in Figure 15, these techniques aim to find the optimal point where the model is complex enough to capture the underlying patterns in the data, but not so complex that it overfits. The **Training vs. Validation Error Curve** plot (shown in Figure 17) illustrates how training and validation errors evolve during training. The goal of these techniques is to keep the model in the region where both training and validation errors are low, avoiding the point where validation error starts to increase due to overfitting.

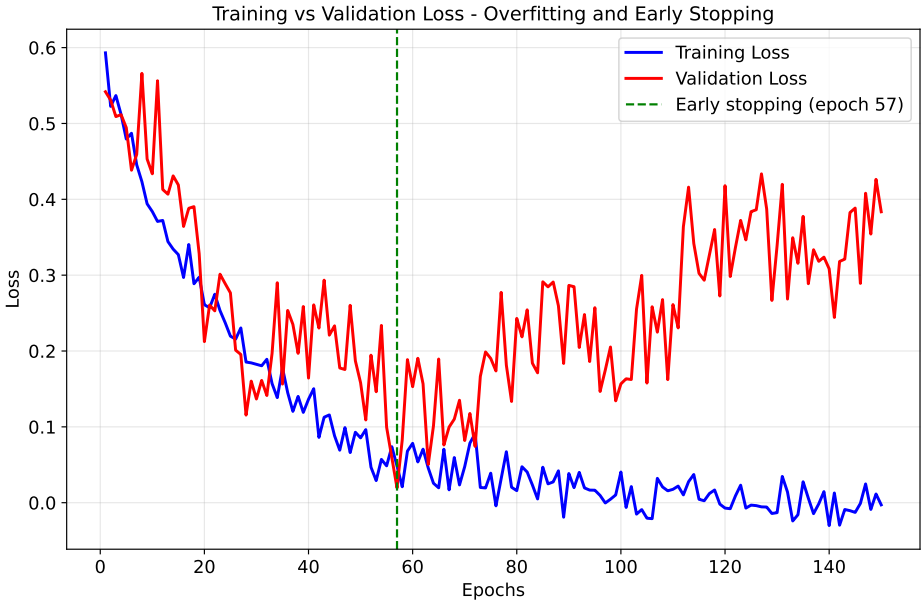


Figure 17: This plot illustrates the **Training vs. Validation Error Curve** during the training of a neural network. Initially, both training and validation errors decrease as the model learns from the data. However, after a certain point (epoch 57), the validation error starts to increase while the training error continues to decrease, indicating that the model is beginning to overfit the training data. The optimal stopping point is where the validation error is minimized, which can be achieved using techniques like Early Stopping.

Furthermore, all **these techniques operationalize the Ockham's Razor principle** we saw earlier: "prefer the simplest model that explains the data well" (page 119). By constraining the model's capacity in various ways, we encourage it to focus on the most salient features of the data, leading to better generalization and performance on unseen data.

3.6.1 Early Stopping

During training, the **training error** keeps decreasing as the model learns to fit the training data better and better. However, at some point, the **validation error** starts to increase again, indicating that the model is beginning to overfit the training data (as we saw in Figure 17 on page 141). That's the signature of **overfitting**: the model is still improving on the training data, but getting worse on unseen data.

The first and simplest method to prevent overfitting is **early stopping**. The idea is very straightforward: we stop the training process **right before** the validation error starts to rise.

Monitoring Validation Error

To apply early stopping, we need to monitor both:

- The **training error/loss** $E_{\text{train}}(k)$ (measures how well the network fits the training data).
- The **validation error/loss** $E_{\text{val}}(k)$ (measures how well it generalizes to unseen data).

Where k is the current training iteration (or epoch). So, during training, at each iteration (epoch) k, we look for the **epoch k_{ES}** (Early Stopping iteration) that minimizes the validation error:

$$E_{\text{val}}(k) \text{ is minimal } \implies k = k_{ES}$$

Once we find k_{ES}, we stop training and use the model parameters (weights and biases) from that iteration for our final model.

Stopping Criteria (iteration k_{ES})

Definition 2: Early Stopping

Early Stopping is a regularization technique that prevents overfitting by **monitoring the validation error during training** and **halting the learning process when the error begins to increase**.

The stopping iteration k_{ES} is defined as the epoch where the validation error $E_{\text{val}}(k)$ is minimized. Formally:

$$k_{ES} = \underset{k}{\operatorname{argmin}}\, E_{\text{val}}(k) \tag{55}$$

It marks the point of **best generalization**; beyond it, the network starts fitting noise rather than signal.

By interrupting training at k_{ES}, the model parameters remain close to their optimal generalizing values, providing an **online estimate of the true generalization error**. With "*online estimate*", we mean that

we can assess the model's performance on unseen data without needing a separate test set at this stage; "*online*" refers to the fact that this evaluation happens during the training process itself.

However, simply stopping at the first sign of validation error increase can be problematic due to random fluctuations (noise) in the validation curve. So, in practice, we implement a more robust strategy. We usually use a **patience window** to avoid stopping too early due to random noise in the validation curve. The **Patience Window** (or **Patience Parameter**) defines **how many epochs** the training process should **wait after the last improvement in validation error** before deciding to stop. In other words, we don't stop immediately when $E_{\text{val}}(k)$ increases once; instead, we wait for a few epochs to see if it improves again (since small fluctuations can occur due to noise).

Example 2: Early Stopping with Patience Window

Imagine this simplified validation loss curve during training (table values):

Epoch	Validation Loss	Improvement?
...	...	...
10	0.40	✔ improvement
11	0.38	✔ improvement
12	0.37	✔ improvement
13	0.375	✘ slightly worse
14	0.373	✔ improvement
15	0.376	✘ worse again
16	0.379	✘ worse
17	0.382	✘ worse

Suppose we set a **patience window of 2 epochs**, the algorithm will stop after **epoch 17**, because it waited for 2 epochs after the last improvement (epoch 14) and saw no new decrease in validation loss. If we had set a patience of 4 epochs, it would have waited until epoch 19 before stopping.

⚠ **Be careful: it's a heuristic!** Early stopping with a *patience window* is **not mathematically guaranteed** to find the optimal epoch k_{ES}; we might stop a few epochs **too early** or **too late**, depending on noise, random initialization, learning rate, and other hyperparameters. In formal terms, it doesn't *guarantee* the true optimum, but it approximates a **local minimum in the generalization curve**, which is what we really want in practice. The local minimum corresponds to a model that generalizes well without overfitting, even if it's not the absolute best possible model.

Online Estimation of Generalization Error

The Early Stopping method can be viewed as an **online estimation of the true generalization error**, because we continuously track the validation loss as the network learns, and the shape of that curve tells us when the model begins to overfit. So, we use validation data to **approximate the true test error dynamically** without needing to retrain multiple times. Thus, early stopping is like a **built-in regularizer**: it doesn't change the loss function, but limits training to the "sweet spot" where generalization is maximal. Where "online" means that this estimation happens during the training process itself, rather than after training is complete.

Limitations and Advantages

- ✔ **Simple & efficient**: no modification to loss or architecture
- ✔ **Automatic**: modern frameworks can monitor and stop automatically.
- ✔ **Improves generalization** without additional parameters.
- ✘ Requires a **validation set**, which reduces training data slightly.
- ✘ Works best when validation loss is smooth (less noisy).
- ✘ The "patience" value and monitoring metric must be chosen carefully.

3.6.2 Hyperparameter Tuning

After understanding **Early Stopping** (which stops training at the right moment), we now focus on **choosing the right model itself**; that is, deciding *how big*, *how deep*, *how fast*, and *how regularized* the network should be. This process is called **Hyperparameter Tuning**, and it's a **crucial step to control overfitting and improve generalization**.

Definition 3: Hyperparameter Tuning

Hyperparameter Tuning is the process of **selecting the best model configuration**, that is, the combination of hyperparameters (such as the number of layers, neurons, learning rate, regularization, strength, etc.). That yields the **lowest validation error** and thus the best generalization ability.

Unlike **parameters** (weights and biases), which are learned automatically during training, **hyperparameters** control *how* learning occurs and *how complex* the model can be.

The optimal configuration:

$$\theta^* = \arg\min_{\theta_i} E_{\text{val}}(\theta_i) \tag{56}$$

is chosen by **comparing validation errors** across candidate models, often within a cross-validation framework to reduce bias.

⚖ Parameters vs. Hyperparameters

The definition above highlights the distinction between **parameters** and **hyperparameters**:

- **Parameters**: These are the internal **weights and biases** of the neural network that are **learned during training through optimization algorithms** like gradient descent. They directly influence the model's predictions, so they determine the function learned by the network.
- **Hyperparameters**: These are **external configurations set before training begins**. They include choices like the:
  - **Number of layers**: This determines the depth of the network. More layers can capture more complex patterns but may also lead to overfitting.
  - **Number of neurons per layer**: This controls the width of the network. More neurons can model more complex functions but increase the risk of overfitting.
  - **Learning rate**: This is a crucial hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated.

- **Batch size**: This defines the number of training examples utilized in one iteration. Smaller batch sizes can provide a more accurate estimate of the gradient but take longer to train. We have seen this in the **Stochastic Gradient Descent** (*batch gradient descent*, item 2, page 114)
- **Regularization strength** γ: This controls the amount of regularization applied to the model to prevent overfitting. Higher values impose a stronger penalty on large weights. We will see this in future sections.
- **Dropout rate**: This defines the fraction of neurons to drop during training to prevent overfitting. A higher dropout rate means more neurons are ignored during each training iteration. Also this will be covered in future sections.

Hyperparameters are set **manually** by the user or **automatically** through hyperparameter optimization techniques (e.g., grid search, random search, Bayesian optimization). They control *how* the model learns and *how complex* it can become, thus affecting its ability to generalize to unseen data.

Why is Hyperparameter tuning important? And why does it matter for overfitting?

Hyperparameter tuning is essential for several reasons, especially in the context of overfitting. It determines the **model's capacity to learn from data** and its ability to generalize to unseen examples.

Hyperparameter	**Low Value**	**High Value**
Number of layers/neurons	Underfitting (too simple)	Overfitting (too complex)
Learning rate	Slow convergence	Unstable or overfitting
Regularization strength γ	Weak penalty, overfit	Too strong, underfit
Dropout rate	Too little regularization	Too much, underfit

Table 1: Impact of Hyperparameter Values on Model Performance.

In geneal, **too low values** of hyperparameters can lead to **underfitting**, where the model is too simple to capture the underlying patterns in the data. Conversely, **too high values** can lead to **overfitting**, where the model learns the training data too well, including its noise and outliers, resulting in poor generalization to new data. So tuning them correctly is essential to reach the **sweet spot** of generalization.

Hyperparameter Tuning Algorithm

The **Hyperparameter Tuning Algorithm** is not a real algorithm per se, but rather a **conceptual framework** for selecting the best hyperparameters based on validation performance. The input are:

- A **set Θ of possible hyperparameter configurations**:

$$\Theta = \{\theta_1, \theta_2, \ldots, \theta_M\}$$

  Where each θ_i is a specific combination of hyperparameter values (e.g., number of layers, learning rate, regularization strength, etc.), and M is the total number of configurations to evaluate.

- A **training dataset** $\mathcal{D}_{\text{train}}$ used to train the model for each hyperparameter configuration.

- A **validation dataset** $\mathcal{D}_{\text{val}}$ used to evaluate the model's performance for each configuration. Here we assume that our dataset is large enough to be split into training and validation sets. If not, cross-validation techniques can be employed to make the most of limited data (hold-out method page 131, k-fold page 135 or leave-one-out cross-validation page 133, or nested cross-validation page 137).

The procedure is as follows:

1. **Define candidate configurations**. Choose which hyperparameter combinations to evaluate. Each configuration θ_i should specify values for all relevant hyperparameters. For example, number of layers, number of neurons per layer, learning rate, regularization strength, etc.

2. **For each configuration $\theta_i \in \Theta$**:

   (a) **Train** a neural network using $\mathcal{D}_{\text{train}}$ with the hyperparameters specified by θ_i. This involves initializing the model, performing forward and backward passes, and updating weights according to the chosen optimization algorithm (we will see this in future sections).

   (b) **Evaluate** its performance on the **validation set** $\mathcal{D}_{\text{val}}$ to compute the validation error $E_{\text{val}}(\theta_i)$. This error metric could be mean squared error, cross-entropy loss, accuracy, etc., depending on the task. If using cross-validation, average the validation errors across all folds to get a robust estimate.

3. **Compare validation errors**. Identify the configuration that gives the **lowest validation error**:

$$\theta^* = \arg\min_{\theta_i \in \Theta} E_{\text{val}}(\theta_i)$$

4. **Select the best model**. Keep the model trained with the optimal hyperparameters θ^*, or retrain it from scratch using the **union of training** and **validation data** with the chosen hyperparameters to maximize the data available for learning.

5. **(optional) Final test**. Evaluate the chosen model on a **separate test set** $\mathcal{D}_{\text{test}}$ to estimate its generalization performance on unseen data.

The output of this procedure is the **optimal hyperparameter configuration** θ^* that minimizes the validation error, along with the **trained model** that can be used for predictions on new data, and an **estimate of its generalization performance** (from validation or test set).

How to choose candidate hyperparameter configurations?

In practice, candidate hyperparameter configurations are not chosen manually because this would be too time-consuming and inefficient. Instead, we use **automatic search methods** to explore the hyperparameter space effectively.

Once we define the set of hyperparameters we want to tune, the following methods **automatically explore** different configurations and identify which one minimizes the **validation error** (or maximizes validation accuracy, depending on the task). The three classical methods are:

1. **Grid Search** (Exhaustive Search on a Discrete Grid). This method evaluates the model on **all possible combinations** of a predefined set of hyperparameter values. For example, we define a **grid** for each hyperparameter:

   $$\eta \in \{0.001, 0.01, 0.1\} \qquad \gamma \in \{0.01, 0.1, 1.0\} \qquad \text{layers} \in \{2, 3, 4\} \qquad \dots$$

   Then the algorithm trains a model for **every combination** of these values:

   $$\Theta = \{(\eta, \gamma, \text{layers}, \dots) \mid \eta \in \{\dots\}, \gamma \in \{\dots\}, \text{layers} \in \{\dots\}, \dots\}$$

   After training and evaluating each configuration, the one with the lowest validation error is selected (minimum E_{val}).

   Advantages
   - Simple and systematic.
   - Parallelizable and easy to implement.

   Limitations
   - Computationally expensive (combinations grow exponentially with more hyperparameters). Also, parallelizable only if enough resources are available.
   - Many evaluations wasted on unimportant regions of the hyperparameter space.
   - Works well only with a *few* hyperparameters or *coarse grids*.

2. **Random Search** (Stochastic Sampling of Hyperparameter Space). Instead of evaluating all combinations, this method **randomly samples** hyperparameter configurations from specified distributions. For example,

we define ranges or distributions for each hyperparameter:

$$\eta \sim \text{Uniform}(0.001, 0.1)$$
$$\gamma \sim \text{LogUniform}(0.01, 1.0)$$
$$\text{layers} \sim \text{DiscreteUniform}\{2, 3, 4\}$$
$$\dots$$

Then the algorithm samples N configurations randomly:

$$\Theta = \{\theta_1, \theta_2, \dots, \theta_N\}$$

After training and evaluating each sampled configuration, the one with the lowest validation error is selected. This method is **much more efficient than grid search**, because typically only a few hyperparameters significantly impact performance.

✔ **Advantages**

- ✔ Covers the search space more efficiently.
- ✔ Works better for high-dimensional spaces.
- ✔ Can easily add or extend hyperparameters.

⚠ **Limitations**

- ✖ Still blind, don't use information from previous trials.
- ✖ May miss the best configuration by chance.

3. **Bayesian Optimization** (Probabilistic Model-Based Search, Learning from Past Trials). The idea is simple yet powerful. Model the function:

$$f(\theta) = E_{\text{val}}(\theta)$$

As an **unknown function** (black box) and use **probabilistic reasoning** to decide which hyperparameters to test next. The steps are:

(a) Use previous evaluations to build a **probabilistic surrogate model** (e.g. a Gaussiano Process, Tree Parzen Estimator, etc.) that approximates the relationship between hyperparameters and validation error ($f(\theta)$).

(b) Define an **acquisition function** (e.g. Expected Improvement, Upper Confidence Bound, etc.) that quantifies the potential benefit of evaluating a new configuration based on the surrogate model. It balances **exploration** (trying uncertain areas) and **exploitation** (focusing on promising areas).

(c) Choose the next configuration θ_{next} to evaluate by maximizing the acquisition function (i.e., the configuration that is expected to yield the most improvement).

(d) Now, train the model with θ_{next}, evaluate its validation error $E_{\text{val}}(\theta_{\text{next}})$, and update the surrogate model with this new data point.

(e) Repeat steps 2-4 until a stopping criterion is met (e.g., a maximum number of evaluations or convergence).

This method is not a naive search; it **learns from past evaluations** to make informed decisions about which hyperparameters to test next, leading to more efficient optimization.

✔ **Advantages**

- ✔ **Much fewer evaluations** needed for good performance.
- ✔ Adapts search intelligently (guided by prior results).
- ✔ Suitable for **expensive models** (deep networks, large datasets).

⚠ **Limitations**

- ✘ Implementation complexity (requires probabilistic modeling).
- ✘ Needs a meaningful continuous search space (discrete hyperparameters can be tricky).
- ✘ Slower per iteration (but far fewer iterations needed).

These methods follow the same underlying goal: **minimize the validation error** by efficiently exploring the hyperparameter space

$$\theta^* = \arg\min_{\theta_i \in \Theta} E_{\text{val}}(\theta_i)$$

But they differ in ***how* they select candidate configurations** to evaluate, balancing exploration and exploitation in different ways.

3.6.3 Weight Decay (L2 Regularization)

The **Weight Decay** (or **L2 Regularization**) is a widely used technique to prevent overfitting in neural networks by adding a penalty term to the loss function that discourages large weights. This method helps to keep the model simpler and more generalizable by constraining the magnitude of the weights.

The key idea behind weight decay is to penalize overly large weights to prevent the network from overfitting training noise. When we train a neural network, we minimize a **loss function** (e.g. MSE regression or cross-entropy for classification):

$$E_{\text{train}}(w) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}\left(y_i, f\left(x_i; w\right)\right)$$

If we let the optimization freely minimize this error, the network might use **very large weights** to fit every detail, creating a complex,oscillating decision surface that produces low training error but poor generalization to unseen data (i.e., **overfitting**). To counter this, we add a **penalty on the magnitude of weights** to the loss function:

$$E_{\text{reg}}(w) = E_{\text{train}}(w) + \frac{\gamma}{2}\sum_{q} w_q^2$$

Where:

- w_q is the q-th weight in the network.
- $\gamma > 0$ is the regularization parameter that controls the strength of the penalty (sometimes called **weight decay coefficient**). A larger γ encourages smaller weights, leading to a simpler model (more bias, less variance), while a smaller γ allows for more complex models (less bias, more variance).

This makes the optimization prefer **small, smooth weights**, producing smoother mappings from inputs to outputs, which helps in generalization.

In other words, the optimizer now minimizes both **error** and **weight energy**. This discourages the model from relying too heavily on any single feature or neuron, promoting a more distributed representation that is less likely to overfit the training data (the network learns gentler transformations, **simpler hypotheses**).

Definition 4: Weight Decay (L2 Regularization)

Weight Decay, or **L2 Regularization**, is a technique that prevents overfitting by adding a **penalty on the magnitude of the network's weights** to the loss function. The regularized loss function is:

$$E_{\text{reg}}(w) = E_{\text{train}}(w) + \frac{\gamma}{2}\sum_{q} w_q^2 \tag{57}$$

Where:

- $E_{\text{train}}(w)$ is the original training loss (e.g., MSE or cross-entropy).
- w_q are the weights of the neural network.
- $\gamma > 0$ is the regularization parameter controlling the strength of the penalty.

By discouraging large weights, the model learns smoother mappings and achieves better generalization, effectively limiting its complexity.

⚖ Relation to Early Stopping

Early Stopping limited training time to prevent overfitting (page 142), while **weight decay directly limits the magnitude of parameters**. Both methods implement the Ockham's Razor principle by favoring simpler models that generalize better, but act at different stages:

- **Early Stopping stops training** when validation error starts to rise, so before weights can grow too large.
- **Weight Decay continuously penalizes** large weights during training, keeping them small throughout the process.

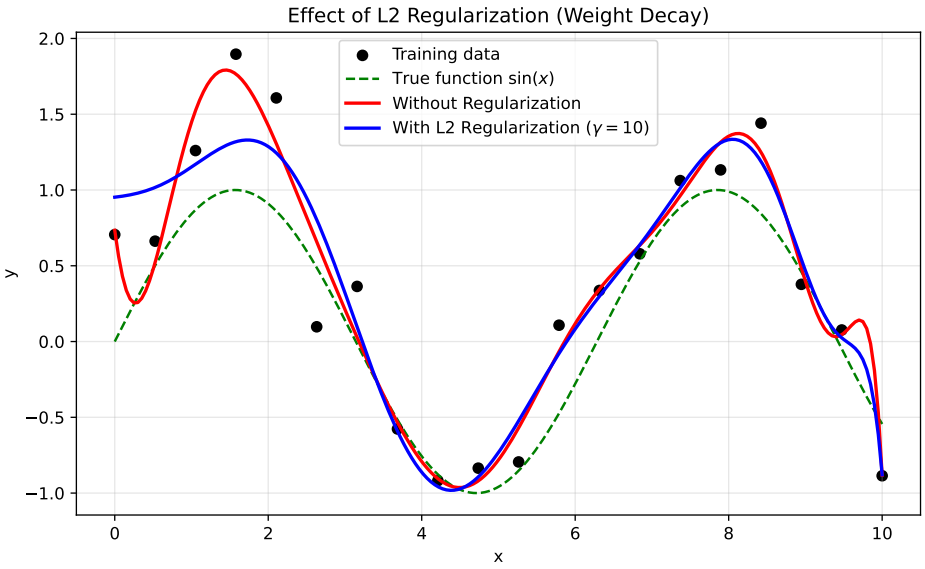


Figure 18: A simulation of polynomial regression with two different fits: one without regularization (large, oscillating weights, overfitting the data) and one with L2 regularization (smaller weights, smoother fit). The L2 regularized model generalizes better to unseen data.

Bayesian Interpretation (MAP vs MLE)

When we train a neural network, we're finding parameters w that best explain the data $D = \{x_i, y_i\}$, where x_i are inputs and y_i are outputs. There are two main approaches to estimate these parameters:

- **Maximum Likelihood Estimation (MLE)** (page 103): choose the parameters w that **maximize the probability of the data given the model**:

  $$w_{\text{MLE}} = \arg\max_w P(D \mid w)$$

  Or equivalently, minimize the negative log-likelihood (which corresponds to minimizing the training error $E_{\text{train}}(w)$):

  $$w_{\text{MLE}} = \arg\min_w -\log P(D \mid w) \equiv \arg\min_w E_{\text{train}}(w)$$

  That's just the **training loss** we usually minimize (like MSE or cross-entropy).

  ✔ MLE fits the data as best as possible.

  ✖ But it can lead to overfitting, especially with complex models and limited data. This is because MLE doesn't penalize complex parameters.

- **Maximum A Posteriori Estimation (MAP)**: instead of trusting data blindly, we also include **prior beliefs** about what weights are likely. Let's explain this step by step. When we use **MLE**, we're saying: "*we don't know anything about the parameters w; just find the ones that make the data as likely as possible*". That's pure **data fitting**. But what if we *do* have priori knowledge? For example, we might believe that weights shouldn't be too large (to avoid overfitting), and most weights should be closer to zero (the network should be simple). Then we can express this belief **probabilistically**, by giving each possible value of w a **prior probability** $P(w)$ that reflects **how plausible we think that value is before seeing any data**. Here, MAP combines this belief with the observed data.

  We want the **posterior probability** of the weights given the data:

  $$P(w \mid D) = \frac{P(D \mid w)\, P(w)}{P(D)}$$

  - $P(D \mid w)$ is the **likelihood** of the data given weights (same as in MLE).
  - $P(w)$ is the **prior** probability of weights (our belief about weights before seeing data).
  - $P(w \mid D)$ is the **posterior** probability of weights given the data (what we believe about weights after seeing data).
  - $P(D)$ is the evidence (normalizing constant).

The **Maximum A Posteriori (MAP)** estimation means: choose the weights w that **maximize the posterior probability** $P(W \mid D)$:

$$w_{\text{MAP}} = \arg\max_w P(w \mid D)$$

Using Bayes' rule, we can rewrite this as:

$$w_{\text{MAP}} = \arg\max_w \frac{P(D \mid w)\, P(w)}{P(D)} = \arg\max_w P(D \mid w)\, P(w)$$

Since $P(D)$ is constant with respect to w, we can ignore it in the optimization. Taking logs, we get:

$$w_{\text{MAP}} = \arg\max_w \left[\log P(D \mid w) + \log P(w)\right]$$

Where the **first term** fits the data (like MLE), and the **second term** adds a *regularization effect* (our prior belief about weights).

To **connect this to weight decay**, we need to choose a specific prior $P(w)$. A common choice is a **Gaussian prior** centered at zero:

$$P(w) = \prod_q \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{w_q^2}{2\sigma^2}\right)$$

This prior says that we believe weights are likely to be small (close to zero), with variance σ^2 controlling how strongly we believe this. Taking the log of this prior gives:

$$\log P(w) = -\sum_q \frac{w_q^2}{2\sigma^2} + \text{constant}$$

Ignoring constants, the MAP objective becomes:

$$w_{\text{MAP}} = \arg\max_w \left[\log P(D \mid w) - \sum_q \frac{w_q^2}{2\sigma^2}\right]$$

Or equivalently, minimizing the negative log-posterior:

$$w_{\text{MAP}} = \arg\min_w \left[-\log P(D \mid w) + \sum_q \frac{w_q^2}{2\sigma^2}\right]$$

This is exactly the same as minimizing the regularized loss function with weight decay:

$$E_{\text{reg}}(w) = E_{\text{train}}(w) + \frac{\gamma}{2}\sum_q w_q^2$$

Where $\gamma = \dfrac{1}{\sigma^2}$. Thus, **weight decay can be interpreted as MAP estimation with a Gaussian prior on weights**. This Bayesian perspective shows that weight decay not only helps prevent overfitting but also incorporates prior beliefs about model simplicity into the learning process.

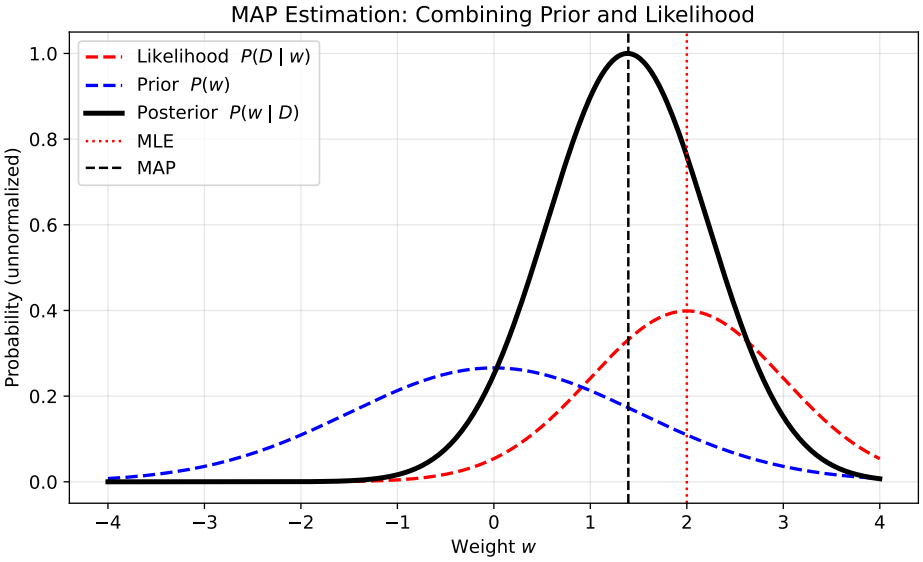


Figure 19: Bayesian interpretation of Weight Decay (L2 Regularization): MLE focuses solely on fitting the data, while MAP incorporates prior beliefs about weights, leading to more generalizable models. We have simulated a one-dimensional weight w: the **likelihood** $P(D \mid w)$ is peaked around some data-fitted value (like MLE); the **prior** $P(w)$ is a Gaussian centered at zero (we believe small weights are more likely); the **posterior** $P(w \mid D)$ combines (product) both, resulting in a peak that balances data fit and weight size.

Gaussian Priori Explanation

What is a prior over weights? When we train a model, we want to find good values for all weights w_q. If we follow a **Bayesian approach**, we say: each weight w_q is a random variable, and before seeing data we have a belief about how likely different values are. This belief is encoded in a **prior distribution**.

Choosing a Gaussian Prior. A natural, simple choice for this prior is a **Gaussian (normal) distribution** centered at zero:

$$P(w_q) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{w_q^2}{2\sigma^2}\right)$$

Intuitively:

- The mean is zero, so small weights are more probable than large ones.
- The variance σ^2 controls how strongly we believe this:
  - A small σ^2 means we strongly believe weights should be close to zero (less flexibility).
  - A large σ^2 means we allow for larger weights (more flexibility).

Assuming independence between weights, the joint prior over all weights is:

$$P(w) = \prod_q P(w_q) = \prod_q \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{w_q^2}{2\sigma^2}\right)$$

So the prior over all weights is a multivariate Gaussian with diagonal covariance. Taking the log of this prior gives:

$$\log P(w) = \sum_q \log P(w_q) = -\frac{1}{2\sigma^2} \sum_q w_q^2 + \text{constant}$$

Combine with likelihood (training error.) The MAP objective (as derived earlier) is:

$$w_{\text{MAP}} = \arg\max_w \left[\log P(D \mid w) + \log P(w)\right]$$

Equivalently, minimizing the negative log-posterior:

$$E_{\text{reg}}(w) = -\log P(D \mid w) - \log P(w)$$

Replace $-\log P(D \mid w)$ with the **training loss** $E_{\text{train}}(w)$, and substitute the log prior:

$$E_{\text{reg}}(w) = E_{\text{train}}(w) + \frac{1}{2\sigma^2} \sum_q w_q^2 + \text{constant}$$

Define $\gamma = \frac{1}{\sigma^2}$, we get:

$$E_{\text{reg}}(w) = E_{\text{train}}(w) + \frac{\gamma}{2} \sum_q w_q^2$$

This is exactly the weight decay regularization term! Thus, assuming a Gaussian prior on weights leads directly to L2 regularization in the MAP framework.

Deepening: What is a Multivariate Gaussian with Diagonal Covariance?

If we have a **single weight** w_1, we can describe our belief about it as a **univariate Gaussian**:

$$P(w_1) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(w_1 - 0)^2}{2\sigma^2}\right)$$

That's just the classic bell curve centered at zero, spreading depending on σ^2.

Now suppose we have **two weights** w_1 and w_2. We could model their joint belief as a **2D Gaussian distribution**:

$$P(w_1, w_2) = \frac{1}{2\pi\,|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}^T \Sigma^{-1} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}\right)$$

Where Σ is the **covariance matrix**. This tells us **how the two weights vary together**.

The covariance matrix Σ looks like this:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{bmatrix}$$

Where:

- σ_1^2 and σ_2^2 are the variances of w_1 and w_2.
- ρ is the correlation coefficient between w_1 and w_2.
  - If $\rho = 0$, the weights are **independent** (no correlation). This means changing w_1 doesn't affect w_2.
  - If $\rho \neq 0$, the weights are **correlated** (changing one affects the other). This means if w_1 increases, w_2 might also tend to increase (if $\rho > 0$) or decrease (if $\rho < 0$).

When we assume the weights are **independent**, we set $\rho = 0$. This simplifies the **covariance matrix to a diagonal matrix**, because the off-diagonal terms (which represent correlations) become zero:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & 0 & 0 & \dots \\ 0 & \sigma_2^2 & 0 & \vdots \\ 0 & 0 & \sigma_3^2 & \vdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Then the multivariate Gaussian **factorizes** into independent 1D Gaussians for each weight:

$$P(w) = \prod_q \frac{1}{\sqrt{2\pi\sigma_q^2}} \exp\left(-\frac{w_q^2}{2\sigma_q^2}\right)$$

Each weight has its own Gaussian prior, and they don't influence each other. That's exactly what we assumed: each w_q has its own $\mathcal{N}\left(0, \sigma^2\right)$ prior, and all weights are independent.

Finally, when the covariance is diagonal ($\rho = 0$ and equal for all weights), we get:

$$\Sigma = \sigma^2 I$$

Where I is the identity matrix. This means that the log prior simplifies to:

$$-\log P\left(w\right) = \frac{1}{2\sigma^2} w^T w = \frac{1}{2\sigma^2} \sum_q w_q^2$$

Which is exactly the L2 regularization term we use in weight decay! So the assumption of a *multivariate Gaussian with diagonal covariance* is what mathematically justifies the standard **weight decay formula**.

How do we choose the right regularization strength γ?

Different values of γ change how much we penalize large weights:

- $\gamma = 0$ means no regularization (just MLE, prone to overfitting).
- γ small means weak regularization (allows larger weights, more complex models).
- γ large means strong regularization (forces weights to be small, simpler models).

So we must find the **optimal** $\gamma = \gamma^*$ that gives the **best generalization**. The idea is to split our data into:

- A **training set** for fitting the weights w.
- A **validation set** for evaluating performance for each candidate γ.

Practically:

1. Choose a range of candidate γ values, for example:

$$\left|10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10\right|$$

2. For each candidate γ:
   - Train the neural network with regularized loss:

$$E_{\text{reg}}(w) = E_{\text{train}}(w) + \frac{\gamma}{2} \sum_q w_q^2$$

   - Compute validation error on the validation set:

$$E_{\text{val}}(w) = E_{\text{val}}(w) + \frac{\gamma}{2} \sum_q w_q^2$$

3. Pick the γ^* that gives the lowest validation error:

$$\gamma^* = \arg\min_{\gamma} E_{\text{val}} \tag{58}$$

4. Finally, retrain the network **on all data** using the selected γ^* to get the final model.

This cross-validation approach ensures we select a regularization strength that balances fitting the training data well while maintaining good generalization to unseen data.

3.6.4 Dropout (Stochastic Regularization)

Even with weight decay and early stopping, large networks can still **overfit** because **neurons learn to co-adapt** too much. For example, neuron A might learn to rely on neuron B being active to make its predictions. If B overfits, then A will also overfit. We need a way to **force independence** among neurons, to make the network robust to missing or noisy signals.

The Idea. During training, we **randomly "drop" neurons** (i.e., set their output to 0) with a certain probability p, independently at each training iteration.

- Each forward pass uses a **different random subnetwork**.
- The model can't rely on any specific neuron always being present, so it must learn **redundant representations**.

Without dropout, the activation of neuron j in layer l is:

$$h_j^{(l)} = g\left(\sum_i w_{ij}^{(l)} h_i^{(l-1)} + b_j^{(l)}\right)$$

- $h_i^{(l-1)}$ are the activations from the previous layer.
- $w_{ij}^{(l)}$ are the weights connecting neuron i in layer $l-1$ to neuron j in layer l.
- $b_j^{(l)}$ is the bias term for neuron j in layer l.
- $g(\cdot)$ is the activation function (e.g., tanh, sigmoid).

Dropout introduces a random **mask** $m_j^{(l)}$ for each neuron, sampled from a **Bernoulli distribution** with parameter p (the probability of **keeping** the neuron). Formally:

$$m_j^{(l)} \sim \text{Bernoulli}(p) \Rightarrow m_j^{(l)} = \begin{cases} 1 & \text{with probability } p \text{ neuron is kept} \\ 0 & \text{with probability } 1-p \text{ neuron is dropped} \end{cases} \tag{59}$$

Then we define the **new (masked) activation** as:

$$\tilde{h}_j^{(l)} = m_j^{(l)} \cdot h_j^{(l)} \tag{60}$$

- $h_j^{(l)}$ is the activation of neuron j in layer l.
- $m_j^{(l)}$ is a dropout mask (binary mask, 1 = keep neuron, 0 = drop neuron).
- p is the probability of **keeping** a neuron (typically $p = 0.5$ for hidden layers, $p = 0.8 - 0.9$ for input layer).
- $\tilde{h}_j^{(l)}$ is the **post-dropout activation** of neuron j in layer l.

Definition 5: Dropout

Dropout is a regularization technique where, *during training*, **each neuron is randomly dropped** (set to zero) with probability $1-p$, independently of other neurons. This **prevents co-adaptation of neurons** and encourages the network to learn robust features.

Formally, for neuron j in layer l:

$$m_j^{(l)} \sim \text{Bernoulli}(p) \quad \Rightarrow \quad m_j^{(l)} = \begin{cases} 1 & \text{with probability } p \\ 0 & \text{with probability } 1-p \end{cases}$$

$$\tilde{h}_j^{(l)} = m_j^{(l)} \cdot h_j^{(l)}$$

Where $\tilde{h}_j^{(l)}$ is the post-dropout activation.

What happens during training vs testing

During **training**, we apply dropout as described above, randomly dropping neurons with probability $1-p$. This forces the network to learn robust features that do not rely on any specific neuron. Formally, during training:

$$\tilde{h}_j^{(l)} = m_j^{(l)} \cdot h_j^{(l)}$$

During **testing**, we do not apply dropout. Instead, we use the full network (i.e., **all neurons are active**) and **scale the activations (or weights) by the probability** p to account for the fact that fewer neurons were active during training. This ensures that the output distribution remains consistent between training and testing. Formally, during testing:

$$h_j^{(l)} \leftarrow p \cdot h_j^{(l)} \tag{61}$$

This is called also **Weight Scaling Rule**.

Example 3: Dropout in a Simple Neural Network

Suppose a layer has a 5 neurons with activations:

$$h^{(l)} = [0.4, 0.7, 0.3, 0.1, 0.9]$$

And we set dropout probability $p = 0.6$ (i.e., 60% chance to keep each neuron), and we randomly sample masks:

$$m^{(l)} = [1, 0, 1, 0, 1]$$

During training, the post-dropout activations are:

$$\tilde{h}^{(l)} = m^{(l)} \cdot h^{(l)} = [0.4, 0, 0.3, 0, 0.9]$$

During testing, we use the full activations scaled by p (each neuron's activation is multiplied by 0.6):

$$h^{(l)} \leftarrow p \cdot h^{(l)} = [0.24, 0.42, 0.18, 0.06, 0.54]$$

3.7 Tips & Tricks

Once we know the loss function, the optimization rule (gradient descent/back-propagation), and the regularization techniques (weight decay, dropout, etc.), we still face a **practical problem**: "*even if our model and data are correct, why does training sometimes fail, diverge, or get stuck?*". This last section focuses on the **engineering side** of deep learning, the small design decisions that make or break our model's ability to learn effectively.

In this section, we will cover six practical "tricks" to improve neural network training:

1. **Activation Function Saturation** (page 163): Sigmoid and Tanh saturate for large inputs, then gradients vanish (go to zero). We'll see the **zero-gradient problem** and why modern networks use non-saturating functions.

2. **ReLu and Variants** (page 166): the most widely used activation today. Simple, fast, avoids saturation, but comes with its own issues (dead neurons). Variants like **Leaky ReLu**, **ELU**, and **GELU** help mitigate these problems.

3. **Weight Initialization** (page 170): setting initial weights correctly is *critical* for stable gradient propagation. The **Xavier (Glorot)** and **He** initializations balance signal variance between layers to avoid vanishing/-exploding gradients.

4. **Batch Normalization** (page 174): normalizes intermediate activations to keep each layer's input distribution stable across training. This speeds up convergence and allows for higher learning rates.

5. **Mini-Batch Training** (page 179): instead of computing gradients on the full dataset or single sample, we use small batches for better convergence and generalization. It's the practical compromise between **SGD** (Stochastic Gradient Descent) and **full-batch gradient descent**.

6. **Learning Rate Scheduling** (page 182): the learning rate controls the "step size" in optimization. Adaptive or decaying schedules (e.g., **Momentum**, **RMSProp**, **Adam**) ensure stable and efficient learning.

3.7.1 Activation Function Saturation

Problem: During backpropagation (page 96), the **gradient** must flow backward through all layers. If it becomes **very small** (≈ 0) in some layer, early layers **stop learning**. This is known as the **Vanishing Gradient Problem** (or **Zero-Gradient Problem**) and it happens when the **activation function saturates**.

What does "saturation" mean?

An **activation function** takes a neuron's input (weighted sum $z = w^T x + b$) and produces a non-linear output $g(z)$. For instance:

- The **sigmoid** function $g(z) = \frac{1}{1+e^{-z}}$
- The **tanh** function $g(z) = \tanh(z)$

Now, both are **S-shaped** (see Figure 6, page 61), tey flatten for large positive or negative z values:

z range	sigmoid $g(z)$	tanh $g(z)$	Derivative $g'(z)$
Very negative	≈ 0	≈ -1	≈ 0
Near zero	≈ 0.5	≈ 0	≈ 0.25 (sigmoid), ≈ 1 (tanh)
Very positive	≈ 1	≈ 1	≈ 0

When $|z|$ is large, the **output saturates** (flattens) and the **derivative** $g'(z)$ becomes **very small** (≈ 0).

And why zero gradient is a disaster?

Let's review the backpropagation. In any neural network trained with gradient descent, each parameter (weight or bias) is updated according to the **core learning rule** (Equation 45, page 92):

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial E}{\partial w_{ij}^{(l)}} \quad \text{and} \quad b_i^{(l)} \leftarrow b_i^{(l)} - \eta \frac{\partial E}{\partial b_i^{(l)}}$$

Where:

- η is the learning rate.
- $\frac{\partial E}{\partial w_{ij}^{(l)}}$ and $\frac{\partial E}{\partial b_i^{(l)}}$ are the gradients of the loss E with respect to the weights and biases.

Each derivative is computed using the **chain rule** in backpropagation. For a neuron i in layer l, we have:

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial w_{ij}^{(l)}} \qquad \frac{\partial E}{\partial b_i^{(l)}} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial b_i^{(l)}}$$

Where:

- y is the output of the network.
- $a_i^{(l)} = g\left(z_i^{(l)}\right)$ is the activation of neuron i in layer l.
- $z_i^{(l)} = \sum_k w_{ik}^{(l)} a_k^{(l-1)} + b_i^{(l)}$ is the weighted input to neuron i in layer l.

The critical term here is $\frac{\partial y}{\partial a_i^{(l)}}$, which involves the derivative of the activation function $g'\left(z_i^{(l)}\right)$. If the activation function **saturates**, its derivative becomes almost zero: $g'\left(z_i^{(l)}\right) \approx 0$. This leads to:

$$\frac{\partial E}{\partial w_{ij}^{(l)}} \approx 0 \quad \text{and} \quad \frac{\partial E}{\partial b_i^{(l)}} \approx 0$$

As a result, the weight and bias updates become negligible:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot 0 = w_{ij}^{(l)} \quad \text{and} \quad b_i^{(l)} \leftarrow b_i^{(l)} - \eta \cdot 0 = b_i^{(l)}$$

So the weight and bias remain unchanged, effectively **halting learning** in that layer (neuron effectively **stops learning**).

Also, the **problem is not local**, it **propagates backward**! For a neuron in layer $l-1$, its gradient depends on the next layer:

$$\delta_j^{(l-1)} = g'(z_j^{(l-1)}) \sum_i w_{ij}^{(l)} \delta_i^{(l)}$$

If the next layer has small derivatives ($g'(z_j^{(l-1)}) \approx 0$) due to saturation, then $\delta_j^{(l-1)}$ also becomes very small, causing the same issue in layer $l-1$. This **cascading effect** can lead to **vanishing gradients** throughout the network, because **gradients shrink exponentially** as they are propagated backward through multiple layers with saturated activations.

Solutions?

In future sections, we will explore various strategies to mitigate the vanishing gradient problem caused by activation function saturation, including:

- Use activation functions that do not saturate easily, such as **ReLU** (page 166).
- **Properly initialize weights** to keep activations in the non-saturated region (page 170).
- Use normalization techniques like **Batch Normalization** to maintain stable activation distributions (page 174).
- Employ architectures like **Residual Networks** that facilitate gradient flow.

Deepening: Exploding Gradient Problem

Not all problems with gradients are about them becoming too small. Sometimes, they can become excessively large, leading to instability during training.
The **Exploding Gradient Problem** is the opposite of the vanishing gradient problem. It occurs when **gradients grow exponentially during backpropagation and become numerically unstable** (very large values), causing very large updates to the weights and divergence of the training process.

How does it happen? It typically occurs when:

- Weights are initialized with very large values.
- The network is very deep, and the product of derivatives during backpropagation leads to large gradients.

If each layer amplifies the gradient by a factor greater than 1, the gradients can grow exponentially as they are propagated backward through the layers. For example, if each layer multiplies the gradient by a factor of 2, after n layers, the gradient will be multiplied by 2^n, leading to extremely large values. With only 50 layers, this results in a factor of $2^{50} \approx 1.13 \times 10^{15}$, which is numerically unstable. So a tiny gradient (e.g., 10^{-10}) can become a huge value (e.g., 10^5) after backpropagating through 50 layers.

In practice: Symptoms we might see during training include:

- The loss suddenly becomes **NaN or Inf** after a few epochs.
- The network weights contain extremely large values.
- The gradient norm is enormous (hundreds or thousands).
- Training oscillates wildly or diverges completely.

This often happens in **deep fully connected networks** with bad initialization or **RNNs (Recurrent Neural Networks)** because gradients are multiplied through time steps.

Solutions? In future sections, we will explore strategies to mitigate the exploding gradient problem, including:

- **Gradient Clipping:** Limit the maximum value of gradients during backpropagation.
- **Proper Weight Initialization:** Use techniques like Xavier or He initialization to keep gradients stable (page 170).
- **Use of Normalization Layers:** Such as Batch Normalization to stabilize activations and gradients (page 174).
- **Architectural Changes:** Such as using LSTM or GRU units in RNNs to control gradient flow.

3.7.2 ReLU and Variants

Definition 6: ReLU Activation Function

The **Rectified Linear Unit (ReLU)** is defined as:

$$g(z) = \max(0, z) \tag{62}$$

And its derivative:

$$g'(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases} \tag{63}$$

So it's a **piecewise linear** function that outputs the input directly if it is positive; otherwise, it outputs zero. No exponential, no sigmoid, just a straight cutoff at zero.

Why does it work? The powerful simplicity of the ReLU

The ReLU activation function has become the default choice for many neural network architectures due to its simplicity and effectiveness. Here are some reasons why ReLU works so well:

- **Non-saturating**: for $z > 0$, the derivative is constant (1), which helps mitigate the vanishing gradient problem that plagues sigmoid and tanh activations (page 163).
- **Computationally cheap**: ReLU is simply a thresholding at zero, which is computationally efficient compared to sigmoid or tanh functions that require expensive exponentials.
- **Sparse activation**: In practice, many neurons output zero, leading to a sparse representation that can be beneficial for learning and generalization.
- **Fast convergence**: Empirically, networks using ReLU tend to converge faster during training compared to those using traditional activation functions.
- **Biological motivation**: ReLU is inspired by the behavior of real neurons, which are either activated (firing) or not (silent), resembling the ReLU activation pattern.

So is it the perfect activation function? Not quite...

Despite its advantages, the **biggest drawback** of ReLU is the **"dead ReLU" problem**. The **Dead ReLU Problem**, or **Dying ReLU Problem**, occurs when a **significant portion of neurons in a neural network become inactive and only output zero for any input**. This happens when the weights are updated in such a way that the input to the ReLU activation function is always negative, causing the neuron to output zero and effectively "die".

When a neuron dies, it stops learning because its gradient is zero (since the derivative of ReLU is zero for inputs less than or equal to zero). This can lead to a situation where a large number of neurons in the network are inactive, reducing the model's capacity to learn complex patterns in the data.

In simple terms, if a neuron's input z becomes **negative for all samples**, its output will always be zero, then the gradient will also be zero, and the neuron will be marked as dead (i.e., it will not contribute to learning anymore). This often happens with: (1) large negative biases, (2) too high learning rates, or (3) poor weight initialization (with poor we mean that many neurons start in the negative region). Hence the ned for **ReLU variants** that maintain some gradient even for negative inputs.

ReLU Variants to the Rescue

To address the dead ReLU problem, several variants of the ReLU activation function have been proposed. Here the four most popular ones:

1. **Leaky ReLU**: a variant that **allows a small slope**, called α, **for negative inputs** (since the standard ReLU has a slope of 0 for negative inputs). It is defined as:

$$g(z) = \begin{cases} z, & \text{if } z > 0 \\ \alpha z, & \text{if } z \leq 0 \end{cases} \tag{64}$$

   where α is a small constant (e.g., 0.01). This creates a small, non-zero gradient when the unit is inactive, which helps to keep the neuron active during training. It is the simplest and most naïve attempt to solve the dead ReLU problem.

   Pros
   - ✔ Fixes dead neurons by allowing a small gradient when $z \leq 0$.
   - ✔ Keeps non-saturating property for positive inputs.

   Cons
   - ✖ Slight computational overhead compared to standard ReLU (but still cheap).
   - ✖ The choice of α is arbitrary and may require tuning.
   - ✖ Still linear for negative inputs, which may not capture complex patterns.

2. **Parametric ReLU (PReLU)**: similar to Leaky ReLU, but instead of using a fixed small slope α for negative inputs, **PReLU learns the slope parameter** during training. It is defined as:

$$g(z) = \begin{cases} z, & \text{if } z > 0 \\ az, & \text{if } z \leq 0 \end{cases} \tag{65}$$

   where a is a learnable parameter. This allows the model to adaptively learn the best slope for negative inputs based on the data. Useful when different neurons may benefit from different slopes.

Pros

- ✔ Learns the slope for negative inputs, potentially improving performance.
- ✔ Retains non-saturating property for positive inputs.

Cons

- ✘ Introduces additional parameters to learn, increasing model complexity.
- ✘ Slightly more computationally expensive than standard ReLU.

3. **Exponential Linear Unit (ELU)**: a smoother variant that **exponentially approaches a negative value** for negative inputs, defined as:

$$g(z) = \begin{cases} z, & \text{if } z > 0 \\ \alpha\left(e^z - 1\right), & \text{if } z \leq 0 \end{cases} \tag{66}$$

where α is a positive constant that controls the value to which ELU saturates for negative inputs. ELU has a smooth curve for negative inputs, which can help with learning (see Figure 20 page 169).

Pros

- ✔ Smooth gradient for negative inputs (no sharp corner at zero).
- ✔ Keeps mean activations close to zero, better convergence.
- ✔ Reduces bias shift, improving learning dynamics.

Cons

- ✘ More computationally expensive due to the exponential function.
- ✘ The choice of α may require tuning.

4. **Scaled Exponential Linear Unit (SELU)**: a **self-normalizing** activation function that **scales the output** to maintain a mean of zero and unit variance. It is defined as:

$$g(z) = \lambda \cdot \begin{cases} z, & \text{if } z > 0 \\ \alpha\left(e^z - 1\right), & \text{if } z \leq 0 \end{cases} \tag{67}$$

where λ and α are **predefined constants** (typically $\lambda \approx 1.0507$ and $\alpha \approx 1.6733$). Introduced with **Self-Normalizing Neural Networks** (Klambauer et al., 2017), SELU is designed to **keep the activations normalized throughout the network**, which can lead to faster convergence and improved performance.

Pros

- ✔ Self-normalizing properties help maintain stable activations.
- ✔ Reduces the need for batch normalization (since activations are kept normalized).
- ✔ Improves convergence speed and performance.

Cons

- ✘ More computationally expensive due to the exponential function.
- ✘ Requires careful weight initialization to maintain self-normalizing properties.

Activation	Formula	$g'(z<0)$	$g'(z>0)$	Notes
ReLU	$\max(0,z)$	0	1	Fast, simple, may die.
Leaky ReLU	$\max(\alpha z, z)$	α	1	Prevents dead neurons.
PReLU	$\max(az, z)$	a (learned)	1	Adaptive slope.
ELU	$\begin{cases} z, & z>0 \\ \alpha(e^z-1), & z\le 0 \end{cases}$	αe^z	1	Smooth, mean close to 0.
SELU	$\lambda \cdot \begin{cases} z, & z>0 \\ \alpha(e^z-1), & z\le 0 \end{cases}$	$\lambda\alpha e^z$	λ	Self-normalizing.

Table 2: Summary of ReLU, Leaky ReLU, PReLU, ELU, and SELU.

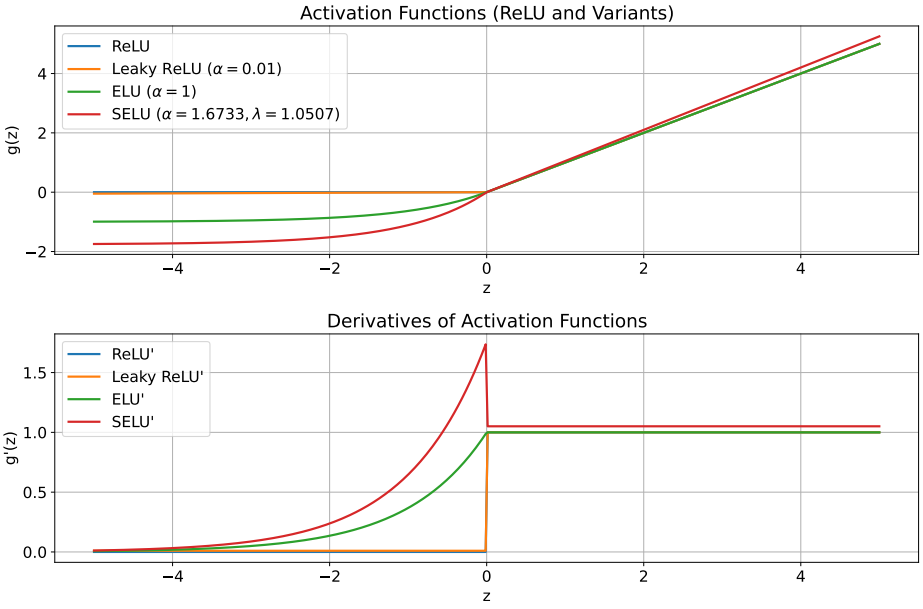


Figure 20: Comparison of ReLU and its variants: Leaky ReLU, PReLU, and ELU. Classic ReLU and Leaky ReLU are almost identical, with the Leaky having a slight slope for negative inputs. PReLU adapts its slope during training, whereas ELU provides a smooth transition for negative inputs. And SELU scales the output to maintain a mean of zero and unit variance.

3.7.3 Weight Initialization

When we initialize a neural network, we want activations and gradients to keep a stable scale across layers. Otherwise:

- If weights are **too small**, signals shrink layer after layer, leading to **vanishing gradients** (vanish gradient problem, page 163).
- If weights are **too large**, signals blow up exponentially, leading to **exploding gradients** (explode gradient problem, page 165).

Hence, initialization must **preserve the variance** of signals *forward* and *backward* through the network.

Forward-pass variance analysis

Consider one neuron:

$$z_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} h_i^{(l-1)} + b_j^{(l)} \quad \text{where} \quad h_i^{(l-1)} = g\left(z_i^{(l-1)}\right)$$

Where $h_i^{(l-1)}$ are the **activations from the previous layer**, $w_{ji}^{(l)}$ are the **weights**, $b_j^{(l)}$ is the **bias**, and $g(\cdot)$ is the **activation function**. Assume:

- **Inputs** $h_i^{(l-1)}$ are i.i.d. (independent and identically distributed, page 104) with:
$$\underbrace{\mathbb{E}\left[h_i^{(l-1)}\right]}_{\text{mean}} = 0 \quad \text{and} \quad \underbrace{\text{Var}\left[h_i^{(l-1)}\right]}_{\text{variance}} = v_h$$
- **Weights** $w_{ji}^{(l)}$ are i.i.d. with:
$$\mathbb{E}\left[w_{ji}^{(l)}\right] = 0 \quad \text{and} \quad \text{Var}\left[w_{ji}^{(l)}\right] = v_w$$
- **Bias** $b_j^{(l)}$ with:
$$\mathbb{E}\left[b_j^{(l)}\right] = 0 \quad \text{and} \quad \text{Var}\left[b_j^{(l)}\right] = v_b$$

Then, the variance of $z_j^{(l)}$ is:

$$\text{Var}\left[z_j^{(l)}\right] = n_{l-1} \cdot v_w \cdot v_h + v_b$$

Where n_{l-1} is the number of neurons in layer $l-1$ (the previous layer). To keep the variance stable across layers, we want:

$$\text{Var}\left[z_j^{(l)}\right] = \text{Var}\left[h_i^{(l-1)}\right] = v_h \quad \Rightarrow \quad n_{l-1} \cdot v_w \cdot v_h + v_b = v_h$$

Assuming v_b is small, we get:

$$n_{l-1} \cdot v_w \cdot \cancel{v_h} + \cancel{v_b} = \cancel{v_h} \quad \Rightarrow \quad v_w = \frac{1}{n_{l-1}} \tag{68}$$

This means we should **initialize weights with variance** $v_w = \dfrac{1}{n_{l-1}}$ **to preserve forward-pass variance**.

Backward-pass variance analysis

During backpropagation, the gradient with respect to activations $\delta_i^{(l)} = \frac{\partial E}{\partial z_i^{(l)}}$ obeys:

$$\delta_i^{(l)} = g'\left(z_i^{(l)}\right) \sum_{j=1}^{n_{l+1}} w_{ji}^{(l+1)} \delta_j^{(l+1)}$$

Where $g'(\cdot)$ is the derivative of the activation function, and $\delta_j^{(l+1)}$ are the gradients from the next layer. Assume:

- **Gradients** $\delta_j^{(l+1)}$ are i.i.d. with:

$$\mathbb{E}\left[\delta_j^{(l+1)}\right] = 0 \quad \text{and} \quad \text{Var}\left[\delta_j^{(l+1)}\right] = v_\delta$$

- **Weights** $w_{ji}^{(l+1)}$ are i.i.d. with:

$$\mathbb{E}\left[w_{ji}^{(l+1)}\right] = 0 \quad \text{and} \quad \text{Var}\left[w_{ji}^{(l+1)}\right] = v_w$$

- **Activation derivatives** $g'\left(z_i^{(l)}\right)$ are i.i.d. with:

$$\mathbb{E}\left[g'\left(z_i^{(l)}\right)\right] = 0 \quad \text{and} \quad \text{Var}\left[g'\left(z_i^{(l)}\right)\right] = v_{g'}$$

Then, the variance of $\delta_i^{(l)}$ is:

$$\text{Var}\left[\delta_i^{(l)}\right] = n_{l+1} \cdot v_w \cdot v_\delta \cdot v_{g'}$$

Where n_{l+1} is the number of neurons in layer $l+1$ (the next layer). To keep the variance stable across layers, we want:

$$\text{Var}\left[\delta_i^{(l)}\right] = \text{Var}\left[\delta_j^{(l+1)}\right] = v_\delta \quad \Rightarrow \quad n_{l+1} \cdot v_w \cdot v_\delta \cdot v_{g'} = v_\delta$$

Assuming v_δ is non-zero, we get:

$$n_{l+1} \cdot v_w \cdot \cancel{v_\delta} \cdot v_{g'} = \cancel{v_\delta} \quad \Rightarrow \quad v_w = \frac{1}{n_{l+1} \cdot v_{g'}} \tag{69}$$

This means we should **initialize weights with variance** $v_w = \frac{1}{n_{l+1} \cdot v_{g'}}$ **to preserve backward-pass variance**.

☆ The Xavier (Glorot) and He (Kaiming) Initialization

To satisfy both forward and backward variance preservation, we can combine equations (68) and (69). Exists two popular initialization schemes:

- **Xavier (Glorot) Initialization**. **Glorot & Bengio (2010)**, two researchers from the University of Montreal, proposed a balanced compromise between forward and backward variance preservation. They combined the two equations by averaging the number of neurons in the previous and next layers:

$$\text{Var}\left[w_{ji}^{(l)}\right] = v_w = \frac{2}{n_{l-1} + n_l} \tag{70}$$

  - n_{l-1} is the number of input units **from** the previous layer **to** layer l.
  - n_l is the number of output units **from** layer l **to** the next layer.

  It works well for activation functions like **tanh** or **sigmoid**, where both positive and negative activations are symmetric and can saturate easily. Hence, Xavier initialization keeps activations within a "safe" non-saturated range, reducing the change of vanishing gradients (or exploding gradients).

- **He (Kaiming) Initialization**. **He et al. (2015)**, researchers from Microsoft Research, proposed an initialization scheme specifically designed for **ReLU** and its variants. He proofed that with ReLU activations, only about half the neurons are active at a time (since ReLU outputs zero for negative inputs). Therefore, the effective variance of activations halves:

$$\text{Var}\left[h^{(l)}\right] = \frac{1}{2} \cdot \text{Var}\left[z^{(l)}\right]$$

  Where $h^{(l)}$ are the activations after ReLU, and $z^{(l)}$ are the pre-activation values. To compensate for this reduction, **He et al. (2015)** proposed to double the variance of weights:

$$\text{Var}\left[w_{ji}^{(l)}\right] = v_w = \frac{2}{n_{l-1}} \tag{71}$$

  This ensures that activations after the ReLU maintain a unit variance and gradients remain stable during backpropagation.

In summary, both initialization methods were designed to **control the variance** of: activations during the **forward pass** and gradients during the **backward pass**. They make sure that, *on average*:

$$\text{Var}\left[z^{(l)}\right] \approx \text{Var}\left[z^{(l-1)}\right] \quad \text{and} \quad \text{Var}\left[\delta^{(l)}\right] \approx \text{Var}\left[\delta^{(l+1)}\right]$$

So they explicitly avoid both:

- ✗ $\text{Var} < 1$ (vanishing gradients)
- ✓ $\text{Var} = 1$ (stable)
- ✗ $\text{Var} > 1$ (exploding gradients)

The goal is a **critical regime** (Var = 1) where the signal neither dies nor explodes. But, remember that Xavier and He initialization prevent both vanishing and exploding gradients only at the **start of training**. During training, weights are updated, and the variance can drift away from the ideal value. Therefore, other techniques like **batch normalization** (page 174) are often used in conjunction to maintain stable activations and gradients throughout training.

Initialization	Recommended for	Weight Variance
Xavier (Glorot)	Sigmoid / Tanh	$\frac{2}{n_{in} + n_{out}}$
He (Kaiming)	ReLU / Leaky ReLU	$\frac{2}{n_{in}}$

Table 3: Summary of popular weight initialization schemes. n_{in} is the number of input units to the layer, and n_{out} is the number of output units from the layer.

In practice, these initialization schemes significantly improve training stability and convergence speed, especially in deep networks. Modern deep learning frameworks (like `TensorFlow` and `PyTorch`) implement these initializations by default when creating layers.

3.7.4 Batch Normalization

When training deep networks, **each layer's input distribution keeps changing** as previous layers update their weights. This phenomenon is called **internal covariate shift**.

In machine learning, the **Internal Covariate Shift** (or simply **Covariate Shift**) happens when the **distribution or input data** changes between training and testing. For example, we train a model to detect cats using bright studio photos; then we test it on dark or outdoor photos. The model struggles because $P_{train}(X) \neq P_{test}(X)$, even though the task (detect cats) remains the same. That's the **classic covariate shift** problem: a change in the *input feature distribution* that forces the model to adapt to new data patterns. In a deep neural network, every layer has its own *inputs*, and those inputs come from the **previous layer's outputs**, which are *constantly changing* as training proceeds. So, while the external dataset stays the same, the **internal data** (the activations flowing between layers) keeps shifting its distribution during training. This is what we call **internal covariate shift**, and it is called "*internal*" because it happens *within* the network itself, not just between training and testing datasets.

In simple terms, every time earlier layers update their weights, the *input distribution* to later layers changes. This means that from the perspective of layer l:

- At iteration 1, its input $z^{(l)}$ might have mean $= 0$ and variance $= 1$.
- At iteration 100, the same input might have mean $= 3$ and variance $= 10$.

So layer l is constantly trying to adapt to a **moving target** distribution, its own input is unstable.

Why is internal covariate shift a problem? Neural networks assume (implicitly) that the distribution of each layer's inputs stays within a "reasonable" range. But if these distributions shift too much:

- Neurons may enter the **saturated region** of activation functions (like sigmoid or tanh), leading to **vanishing gradients**.
- Gradient flow becomes unstable, making optimization **slower** and more **difficult**.
- Convergence slows down dramatically, requiring **smaller learning rates** and careful initialization.

In other words, internal covariate shift makes training **harder and slower**, because every layer must re-learn how to operate each time the layers before it change.

Solution: Batch Normalization

Since covariate shift arises from changing input distributions, the solution is to **stabilize these distributions**. This is where **Batch Normalization (BN)** comes in. Introduced by Sergey Ioffe and Christian Szegedy in 2015 [5], BN fixes this problem by **re-centering and re-scaling** each layer's input at every training step, so that:

$$\mathbb{E}\left[z^{(l)}\right] = 0 \quad \text{and} \quad \text{Var}\left[z^{(l)}\right] = 1 \tag{72}$$

Where $z^{(l)}$ is the input to layer l, $\mathbb{E}$ is the expectation (mean), and Var is the variance. That means the input distribution to each layer remains stable, even as earlier layers' weights keep evolving. So layer l can focus on learning *useful transformation*, not on constantly re-adjusting to a changing scale or mean.

Formally, let's denote the pre-activations of a neuron as $z_i^{(l)}$ for the i-th example in a batch of size m. Batch Normalization performs the following steps during training:

1. **Compute the *batch mean***:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} z_i^{(l)} \tag{73}$$

   Where μ_B is the mean of the pre-activations over the batch.

2. **Compute the *batch variance***:

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}\left(z_i^{(l)} - \mu_B\right)^2 \tag{74}$$

   Where σ_B^2 is the variance of the pre-activations over the batch.

3. ***Normalize* each activation**:

$$\hat{z}_i^{(l)} = \frac{z_i^{(l)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{75}$$

   Where $\hat{z}_i^{(l)}$ is the normalized activation, and ε is a small constant added for numerical stability, preventing division by zero. After this step, the normalized activations $\hat{z}_i^{(l)}$ have mean 0 and variance 1 across the batch. But, since this step **destroys** the original mean and variance information of z, we need to add a way to **restore** it if needed.

4. ***Scale and shift* with learnable parameters**:

$$y_i^{(l)} = \gamma \hat{z}_i^{(l)} + \beta \tag{76}$$

   Where $y_i^{(l)}$ is the final output of the Batch Normalization layer, and γ and β are learnable **parameters that allow the network to restore the original distribution if needed**.

If Batch Normalization forces every layer to have mean 0 and variance 1, how can the two parameters γ and β possibly recover the *original* distribution? Isn't the information lost?
Normalization helps stability, but what if a neuron *needs* its activations to be, say: shifted up (mean = 2) or spread wider (variance = 9)? If we stop at pure normalization we'd **limit the representational power** of the layer, because every neuron would have to operate under the same fixed scale 1 and mean 0. That's where the learnable parameters γ and β come in:

- γ (scale) allows the network to **stretch or compress** the normalized activations, effectively controlling the variance.
- β (shift) allows the network to **move** the normalized activations up or down, effectively controlling the mean.

This lets the network **relearn any affine transformation** of the normalized values.

Example 4: Restoring Original Distribution with γ and β

Suppose before normalization a neuron had:

$$\mu_{\text{orig}} = 2, \quad \sigma_{\text{orig}} = 3$$

After normalization, the activations have:

$$\hat{z}_i^{(l)} = \frac{z_i - 2}{3} \quad \Rightarrow \quad \mathbb{E}[\hat{z}] = 0, \quad \text{Var}[\hat{z}] = 1$$

Then, if the network learns:

$$\gamma = 3, \quad \beta = 2$$

The final output becomes:

$$y_i^{(l)} = 3 \cdot \hat{z}_i^{(l)} + 2 = 3 \cdot \frac{z_i - 2}{3} + 2 = z_i$$

Thus, the original distribution is perfectly restored!

So the full Batch Normalization transformation can be summarized as:

$$y_i^{(l)} = \text{BN}\left(z_i^{(l)}\right) = \gamma \cdot \frac{z_i^{(l)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} + \beta \tag{77}$$

When to apply Batch Normalization?

During training, Batch Normalization is applied **inside the forward pass of the network**, right after the linear transformation (affine operation) and before the activation function. Formally, for layer l:

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

Batch Normalization acts as:

$$\text{BN: } \hat{z}^{(l)} = \frac{z^{(l)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \quad \text{then} \quad y^{(l)} = \gamma \hat{z}^{(l)} + \beta$$

And then:

$$a^{(l)} = g\left(y^{(l)}\right) \quad \text{where } g \text{ is the activation function (e.g., ReLU, sigmoid)}$$

So the order is typically **Linear** → **Batch Norm** → **Activation**.

What about during inference (testing)?

During **inference (testing)**, the model process **one example at a time** (or a few), not large batches. Therefore, computing "batch statistics" doesn't make sense anymore because a single sample doesn't have a meaningful mean or variance. Instead, we use **running estimates** (a.k.a. *moving averages*) of mean and variance that were computed during training. Specifically:

1. **Running averages during training**. As we train, we maintain running estimates of the mean and variance for each layer:

$$\mu_{\text{running}} = (1 - \alpha)\mu_{\text{running}} + \alpha\mu_B \tag{78}$$

$$\sigma^2_{\text{running}} = (1 - \alpha)\sigma^2_{\text{running}} + \alpha\sigma_B^2 \tag{79}$$

   Where α is a small constant (e.g., 0.1) that controls the update rate. These running estimates capture the overall distribution of activations over the entire training set.

2. **At inference (testing)**. At inference, Batch Normalization uses these **"frozen" running estimates** instead of batch statistics:

$$y_i^{(l)} = \gamma \cdot \frac{z_i^{(l)} - \mu_{\text{running}}}{\sqrt{\sigma^2_{\text{running}} + \varepsilon}} + \beta$$

   These are constants, no mini-batch dependence, no randomness. This ensures that the model's behavior is consistent and stable during inference.

Benefits of Batch Normalization

- **Stabilizes training**: Keeps activations in non-saturating range, reducing vanishing/exploding effects.
- **Allows higher learning rates**: Because activations are normalized, large updates won't blow up the model.
- **Regularization effect**: Adds small noise (due to batch statistics), improving generalization.
- **Reduces sensitivity to initialization**: Training becomes less dependent on perfect Xavier/He settings.
- **Improves convergence speed**: Layers adapt faster because their inputs are more predictable.

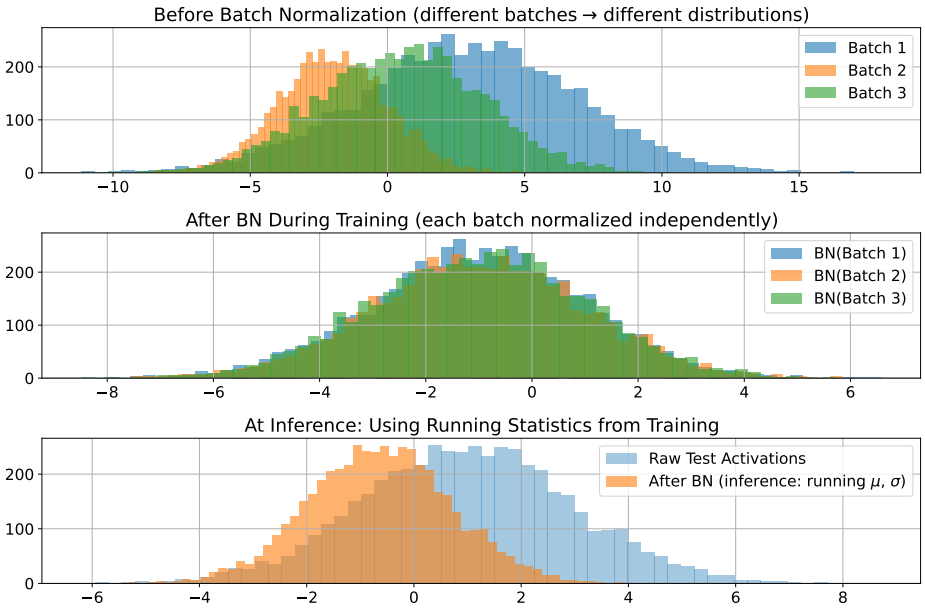


Figure 21: Batch Normalization normalizes layer inputs to stabilize training. Before BN, each training batch has a very different distribution (different mean & variance, top plot). After BN (during training), each batch is independently normalized to roughly the same mean and variance — stable inputs for the next layer (middle plot). At inference (testing), the model uses the running averages of mean and variance collected during training, producing a fixed and consistent normalization for new unseen data (bottom plot).

3.7.5 Mini-Batch Training

When we train a neural network, we want to minimize the **empirical risk**:

$$E(w) = \frac{1}{N} \sum_{n=1}^{N} L\left(f\left(x_n; w\right), t_n\right)$$

- N is the number of training samples
- L is the loss function
- $f(x_n; w)$ is the model's prediction for input x_n with parameters w (outputs)
- t_n is the true target for input x_n

Theoretically, the gradient for updating weights should be computed over the entire dataset, over **all** N **samples**:

$$\nabla_w E(w) = \frac{1}{N} \sum_{n=1}^{N} \nabla_w L\left(f\left(x_n; w\right), t_n\right)$$

But in practice, especially with large datasets, this is computationally expensive and inefficient.

Solution: Mini-Batch Gradient Descent

Instead of using all N samples to compute the gradient, we use a smaller subset of the data called a **mini-batch**. In general, the mini-batch technique is one of a set of batching techniques. The main batch techniques are:

- **Full-Batch Gradient Descent (BGD)**: uses the entire dataset to compute the gradient. It is the method that we introduced when we covered gradient descent, as it is the most general (it uses all the data, page 90). However, it allows to get the exact gradient, but it is computationally **expensive** for large datasets and can lead to **slow convergence**.
- **Stochastic Gradient Descent (SGD)**: uses a single sample to compute the gradient at each iteration. We introduced it when we covered stochastic gradient descent (page 115) because we wanted to show how randomness can help in optimization. It is computationally **efficient** and can lead to **faster convergence**, but the gradient estimate is **noisy** and can lead to **unstable updates**.
- **Mini-Batch Gradient Descent (MBGD)**: uses a small subset of the dataset (mini-batch) to compute the gradient at each iteration. It is a compromise between BGD and SGD, **balancing computational efficiency and gradient accuracy**. It is widely used in practice due to its effectiveness in training deep neural networks.

The **Mini-Batch Gradient Descent (MBGD)** algorithm works as follows:

1. Shuffle the training dataset to ensure randomness.
2. **Divide** the dataset into **mini-batches of size** m (where $m \ll N$, much smaller than N).
3. For **each mini-batch** $B = \{x_1, x_2, \ldots, x_m\}$:
   (a) Compute the **gradient of the loss function** with respect to the weights using only the samples in the mini-batch:
   $$\nabla_w E_B(w) = \frac{1}{m} \sum_{i=1}^{m} \nabla_w L\left(f\left(x_i; w\right), t_i\right)$$
   (b) **Update the weights** using the computed gradient:
   $$w \leftarrow w - \eta \nabla_w E_B(w)$$
   where η is the learning rate.
4. Repeat until convergence or for a fixed number of epochs.

⚖ The trade-off: Stability vs. Speed

Choosing the right mini-batch size is crucial:

Property	Small ($1 \leq m \leq 32$)	Large ($256 \leq m \leq 8192$)
Computation per step	Very low	Very high
Update frequency	Very high	Very low
Gradient noise	High, helps generalization	Low, risk of sharp minima
Convergence	Fast but noisy	Smooth but slower learning per sample
GPU parallelism	Inefficient	Efficient

So **too small batches** can lead to **noisy updates** and **unstable training**, while **too large batches** can lead to **slow convergence** and **poor generalization**. A common practice is to start with a moderate batch size (e.g., 32, 64, or 128) and adjust based on the model's performance and available computational resources.

Impact on convergence and generalization

Mini-batch training affects both convergence and generalization of neural networks:

- **Convergence**. The stochasticity (small random fluctuations in gradient direction) helps the optimizer: escape shallow or sharp local minima, explore a larger portion of the loss landscape, and converge faster in expectation.
- **Generalization**. Mini-batch noise acts as a **regularizer**, preventing the model from memorizing training data perfectly (and thus overfitting).
  - Too large a batch → almost deterministic gradient → risk of converging to **sharp minima** → poor generalization.
  - Smaller batches → noisier gradients → exploration of **flatter minima** → better generalization.

3.7.6 Learning Rate Scheduling

The **learning rate** η is one of the most important hyperparameters in training neural networks. It controls how big the weight updates are at each step of the optimization process:

$$w_{k+1} = w_k - \eta \nabla_w E(w_k)$$

If η is too **large**, training may **diverge** (overshoot minima). If it's too **small**, training is **slow** and can get stuck in **local minima** or **plateaus**. So we need ways to make the learning rate **adapt intelligently over time**.

⚖ Fixed vs. Adaptive Learning Rates

There are four main strategies for setting the learning rate during training:

- **Fixed Learning Rate** (baseline). The learning rate η remains **constant** throughout training. So **gradient descent (GD)** or **stochastic gradient descent (SGD)** uses the same η at every iteration:

  $$w_{k+1} = w_k - \eta \nabla_w E(w_k)$$

  Works fine for simple convex problems (for example Figure 11, page 94), but for deeper networks, loss landscapes are complex, and a fixed η can lead to **slow convergence** or divergence.

- **Learning Rate Scheduling (decay)**. The learning rate η is **reduced over time** according to a predefined schedule. Typical schedules include:

  - **Step Decay**: reduce η by a factor (e.g., halve it) every s epochs:

    $$\eta_t = \eta_0 \cdot \text{drop}^{\lfloor t \div s \rfloor} \tag{80}$$

    Where η_0 is the initial learning rate, t is the epoch number, and drop is the factor by which to reduce η.

  - **Exponential Decay**: reduce η by a factor **every** epoch:

    $$\eta_t = \eta_0 \cdot e^{-\lambda t} \tag{81}$$

    Where λ is the decay rate and t is the epoch number.

  - **1/t Decay**: reduce η according to the inverse of the epoch number:

    $$\eta_t = \frac{\eta_0}{1 + \lambda t} \tag{82}$$

    Where λ is the decay rate and t is the epoch number. It is a classic stochastic approximation rule.

  - **Cosine Annealing**: vary η according to a cosine function:

    $$\eta_t = \frac{\eta_0}{2} \cdot \left(1 + \cos\left(\frac{t}{T}\pi\right)\right) \tag{83}$$

    Where T is the maximum number of epochs. This allows for periodic "*restarts*" of the learning rate, because the cosine function oscillates (i.e., the interval $[0, \pi]$ can be repeated multiple times during training).

In practice, **learning rate is set to a relatively high value** initially to allow for rapid learning, and then **gradually decreased** to allow for fine-tuning as training progresses.

- **Momentum: adding inertia to updates**. Since plain SGD (Stochaastic Gradient Descent) may zig-zag in narrow valleys of the loss surface, **momentum** accelerates learning by accumulating a **velocity vector** in directions of persistent reduction in loss:

$$v_t = \alpha v_{t-1} - \eta \nabla_w E\left(w_t\right) \tag{84}$$

Then weights are updated using this velocity:

$$w_{t+1} = w_t + v_t \tag{85}$$

Where v_t is the velocity at time t, $\alpha \in [0, 1)$ is the momentum coefficient (typically around 0.9), and η is the learning rate.

- **Adaptive Learning Rate Methods**. These methods adjust the learning rate for each parameter individually based on historical gradients:
  - **RMSProp (Root Mean Square Propagation)**. Adapts the step using a *running average of squared gradients*:

$$\begin{aligned} s_t &= \rho s_{t-1} + (1-\rho) g_t^2 \\ w_{t+1} &= w_t - \frac{\eta}{\sqrt{s_t + \varepsilon}} g_t \end{aligned} \tag{86}$$

Where s_t is the running average of squared gradients, ρ is the decay rate (typically around 0.9), g_t is the gradient at time t, and ε is a small constant to prevent division by zero. If gradients are large, then the denominator increases, and the step size decreases, and vice versa. This helps **stabilize updates and allows for larger learning rates**.

  - **Adam (Adaptive Moment Estimation)**: combines momentum (mean of gradients) and RMSProp (variance of gradients) by maintaining both a velocity vector and an exponentially decaying average of squared gradients:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\ w_{t+1} &= w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} \end{aligned} \tag{87}$$

Where:

* m_t is the first moment (mean) estimate,

- v_t is the second moment (uncentered variance) estimate,
- β_1 and β_2 are decay rates for the moment estimates (typically $\beta_1 = 0.9$, $\beta_2 = 0.999$),
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates,
- ε is a small constant to prevent division by zero, typically around 10^{-8}.

Adam **adapts quickly**, **handles noisy gradients**, and **works well with mini-batches**. However, sometimes **generalizes worse** than SGD with momentum (flatter minima are less explored).

In practice, we can take a hybrid approach:

1. **Start with Adam** for general-purpose training, especially with complex architectures or noisy data.
2. **Switch to SGD with Momentum** for final fine-tuning to potentially achieve better generalization.
3. **Learning rate scheduling** (cosine, step, or warm restarts) is used even *with* Adam; we can adapt the *global* η while Adam adapts per-parameter rates.

4 Recurrent Neural Networks (RNNs)

In the previous sections, we have explored Feed-Forward Neural Networks (page 54), which are well-suited for tasks where inputs and outputs are fixed in size and independent of each other. So far, every model we have discussed processes data in a **static** manner: they took a fixed-size input vector $\mathbf{x}$ and produced a fixed-size output vector $\mathbf{y}$, **without considering any temporal or sequential dependencies** (no history, no sequence, no memory).

However, many real-world applications involve sequential data, where the order of the data points matters, such as time series analysis, natural language processing, and speech recognition. To handle such sequential data, we need a different type of neural network architecture known as **Recurrent Neural Networks (RNNs)**. They are models designed to handle **sequences** of data by maintaining a **hidden state** that captures information about previous time steps, allowing them to exhibit temporal dynamic behavior.

- **Sequence Modeling**: RNNs are specifically designed to model sequences of data, making them suitable for tasks where the order of inputs matters.
- **Memoryless vs Memory-based models**: Unlike FNNs, which are memoryless and process each input independently, RNNs have a form of memory that allows them to retain information from previous inputs in the sequence. We will explore some models that exhibit this property, such as Autoregressive Models, Dynamical Systems, Hidden Markov Models (HMMs), and RNNs themselves.
- **Recurrent Neural Networks (RNNs)**: RNNs are a class of neural networks that have connections that form directed cycles, allowing information to persist over time. This enables them to capture temporal dependencies in sequential data. We will delve into the architecture of RNNs, recurrent connections, and hidden states.
- **Backpropagation Through Time (BPTT)**: Training RNNs involves a specialized version of backpropagation called Backpropagation Through Time (BPTT). We will discuss how BPTT works and how it allows RNNs to learn from sequential data.
- **Vanishing and Exploding Gradients**: RNNs can suffer from vanishing and exploding gradient problems during training, which can hinder their ability to learn long-term dependencies. We will explore these issues and discuss techniques to mitigate them.
- **Long Short-Term Memory (LSTM)**: To address the limitations of standard RNNs, we will introduce Long Short-Term Memory (LSTM) networks, which are a type of RNN designed to capture long-term dependencies more effectively.

In simple terms, if a feed-forward neural network maps a function from input to output, a recurrent neural network maps a sequence of inputs to a sequence of outputs, taking into account the temporal dependencies between the inputs. This makes RNNs particularly powerful for tasks involving sequential data, where the context provided by previous inputs is crucial for making accurate predictions.

4.1 Sequence Modeling

⚖ Static vs Dynamic Datasets

Until now, all the models we have seen (Perceptron, Feed-Forward, etc.) assumed **static datasets**, i.e., each input $\mathbf{x}_n$ is **independent** from the others:

$$\mathbf{x}_n = [x_1, x_2, \dots, x_I] \xrightarrow{\text{produces}} \hat{y}_n = f(\mathbf{x}_n; \theta)$$

This means that the network **does not know any notion of time** or order of the inputs. Each sample is processed as a separate point in the dataset. However, **many real-world problem are sequential**, for example:

- Speech recognition (audio signal is a sequence of sound waves)
- Stock price prediction (time series data)
- Text generation (sequence of words or characters)
- Human motion capture (sequence of body poses over time)

These require **dynamic datasets**, where:

$$\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_T]$$

Represents **a temporal sequence** rather than independent samples. Here, each observation $\mathbf{x}_t$ depends on the past:

$$\mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots)$$

Thus, a model must **remember or integrate information** across time.

🕐 From Fixed Inputs to Temporal Sequences

Now that we have established the difference between *static* and *dynamic* datasets, *how can we adapt our models to process sequences?* In a **feed-forward neural network (FNN)**, the computation is:

$$\hat{y} = f(\mathbf{x}; \mathbf{W})$$

Where:

- $\mathbf{x} \in \mathbb{R}^I$ is a fixed-size input vector where I is the number of features.
- $\mathbf{W}$ are the model parameters (weights and biases).
- $f(\cdot)$ is a non-linear function (the neural network activation functions).
- $\hat{y}$ is the output prediction.

Every sample is processed **independently**, the network "sees" no link between sample $\mathbf{x}_n$ and $\mathbf{x}_{n+1}$. But in sequential data, we have **temporal sequences**:

$$\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_T]$$

Each vector $\mathbf{x}_t$ is not isolated and depends on what came before. A key point to note is that the **current observation x_t only has meaning when considered in the context of the past**. For example, in language modeling, the word "bank" can refer to a financial institution or the side of a river, depending on the preceding words. Therefore, to effectively model sequences, we need architectures that can **capture temporal dependencies** and **retain information** from previous time steps. To handle sequences, we define a **temporal model**:

$$\hat{y}_t = f\left(x_t, x_{t-1}, \ldots, x_0\right) \tag{88}$$

Where:

- $\hat{y}_t$ is the output prediction at time t.
- $f(\cdot)$ is a function that considers ("remembers") the current input x_t and all previous inputs $x_{t-1}, \ldots, x_0$.

In vector form, this corresponds to an **ordered sequence of inputs and outputs**:

$$\mathbf{x} = [x_0, x_1, \ldots, x_T] \quad \mathbf{Y} = [y_0, y_1, \ldots, y_T]$$

Why fixed Networks fail. The first attempt to fix this problem could be to use a **fixed feed-forward network** that reuse knowledge from one input to the next. However, this approach has several limitations:

- There is no explicit mechanism to store **context** or **memory** of previous inputs.
- Parameters are shared across time, but **activations do not carry over** from one time step to the next.
- Most importantly, the model is explicitly **memoryless** and cannot capture temporal correlations beyond the current input.

If we forced a static Neural Network to handle a sequence, we'd need to **concatenate all time steps** into a single giant input vector:

$$\mathbf{x} = [x_0, x_1, \ldots, x_T]$$

Which explodes in size and loses flexibility (variable-length sequences can't be handled).

Idea behind Sequence Modeling. We introduce the concept of a **Temporal Dimension** and **State Propagation**. At each time step t:

1. The model takes x_t as input.
2. It combines it with **internal state** h_{t-1} (the memory from the previous time step).
3. It produces both:
   - Output: y_t
   - New hidden state:

$$h_t = f_h\left(x_t, h_{t-1}\right) \tag{89}$$

Where h_t captures information from all previous inputs up to time t, and y_t is the prediction based on the current input and the accumulated context.

This recursive definition is the core idea of **Recurrent Neural Networks (RNNs)**: a single model unrolled through time, where the same parameters are used at each time step, but the hidden state allows information to flow across time. It is recursive because the output at time t depends on the output at time $t-1$ through the hidden state, then $t-2$, and so on.

Handling Time Dimension: Temporal Data Representation

The theory behind sequence modeling requires us to rethink how we represent data. Instead of a single-fixed input space, we now have a **spatio-temporal space** that includes both **Feature Dimension** I and **Time Dimension** T. So instead of a single flat vector $\mathbf{x}$, we have a **sequence**:

$$\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_T] \in \mathbb{R}^{I \times (T+1)}$$

This means that at every discrete time t, we observe:

$$\mathbf{x}_t = [x_{t,1},\ x_{t,2},\ \ldots,\ x_{t,I}]$$

Where $x_{t,i}$ is the i-th feature at time t, and $\mathbf{x}_t$ is a vector of features measured at that time step t. Each vector is **not independent** but part of a sequence where the order matters, and this is the essence of **sequential modeling**. We can express this dependency as:

$$P\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \ldots, \mathbf{x}_0\right) \tag{90}$$

This conditional probability captures the idea that the current observation $\mathbf{x}_t$ depends on all previous observations in the sequence.

Types of models

There are **different modeling philosophies** for handling memory in sequence data. We can group them into two big categories:

- Memoryless Models. These models **ignore the internal notion of time**, and treat previous inputs as just *extra inputs*. Two models we will see in the next sections fall into this category:
  - **Autoregressive (AR) Models**: Predict the current output based on a fixed number of previous inputs.
  - **Feed-Forward Networks with Time Delays**: Use a sliding window of past inputs as additional features.
- Models with Memory. These models **explicitly maintain an internal state** that captures information from previous time steps. The main models we will see in the next sections are:
  - **Linear Dynamical Systems (LDS)**: Use linear transformations to update the hidden state over time.

- **Hidden Markov Models (HMMs)**: Use probabilistic transitions between hidden states to model sequences.
- **Recurrent Neural Networks (RNNs)**: Use non-linear functions to update the hidden state, allowing for complex temporal dependencies.

4.2 Memoryless Models

4.2.1 Autoregressive (AR) Models

The **Autoregressive (AR) Model** is one of the oldest approach to handle temporal sequences, especially used in **signal processing**, **finance**, and **time-series forecasting**. Its principle is simple: the current value of a sequence depends on a finite number of its **past values**. In other words, the model **uses the past to predict the future**, but *without* any hidden memory structure.

Definition 1: Autoregressive (AR) Model

For a scalar time-series x_t, an **Autoregressive (AR) Model**, or simply $AR(p)$ model of order p, is defined as:

$$\hat{x}_t = a_1 x_{t-1} + a_2 x_{t-2} + \ldots + a_p x_{t-p} + \varepsilon_t \tag{91}$$

Where:

- $a_1, a_2, \ldots, a_p$ are the model parameters (weights) that determine the influence of past values.
- p is the number of **delay taps** (how back in time we look).
- ε_t is a white noise error term, accounting for randomness or unexplained variations.

If we arrange data as vectors, we can express the AR model in vector form:

$$\mathbf{x}_t = [x_t, x_{t-1}, x_{t-2}, \ldots, x_{t-p+1}]^T$$

Thus, the AR model can be compactly written as:

$$\hat{x}_t = \mathbf{a}^T \mathbf{x}_{t-1} + \varepsilon_t \tag{92}$$

Where $\mathbf{a} = [a_1, a_2, \ldots, a_p]^T$. Or equivalently, for multivariate data:

$$\hat{\mathbf{x}}_t = \mathbf{A}_1 \mathbf{x}_{t-1} + \mathbf{A}_2 \mathbf{x}_{t-2} + \ldots + \mathbf{A}_p \mathbf{x}_{t-p} + \boldsymbol{\varepsilon}_t$$

Example 1: Autoregressive (AR) Model

Suppose we want to predict the next day's temperature:

$$T_{\text{today}} = 0.7 \cdot T_{\text{yesterday}} + 0.2 \cdot T_{2 \text{ days ago}} + 0.1 \cdot T_{3 \text{ days ago}}$$

This means that the most recent day (yesterday) has the highest influence on today's temperature, while the influence decreases for days further back in time. The model linearly mixes these past temperatures to make a prediction.

⚠ Properties and Limitations

- **Linear limitation**: The AR model assumes a linear relationship between past and current values, which **may not capture complex patterns in data** (e.g., if signal rising, slow down).
- **Finite memory (fixed window)**: It only considers a fixed number of past values (determined by p), **ignoring longer-term dependencies**.
- **Stationary assumption**: AR models typically assume that the underlying time series is stationary, meaning its statistical properties do not change over time. This can be a limitation when dealing with real-world data that may exhibit trends or seasonal patterns.
- **Deterministic, no hidden representation**: AR models do not maintain a hidden state or memory of past inputs beyond the fixed window, which can limit their ability to model complex temporal dependencies.

That's why we move toward **Feed-Forward extensions** and eventually **Recurrent Neural Networks**, which can introduce non-linearities and maintain hidden states for better temporal modeling.

4.2.2 Feed-Forward Extensions: TDNNs

Autoregressive models are **linear**:

$$\hat{x}_t = a_1 x_{t-1} + a_2 x_{t-2} + \ldots + a_p x_{t-p}$$

They work only if the relationship between past and future is *linear*. But real-world signals (speech, finance, human motion, etc.) are **nonlinear** and complex. The idea is to replace the simple linear combination with a **Feed-Forward Neural Network (FNN)** that can learn nonlinear dependencies.

We still use the **sliding window** of the last p time steps as input:

$$\mathbf{x}_t = [x_{t-1}, x_{t-2}, \ldots, x_{t-p}]^T$$

But instead of computing a weighted sum, we feed it into a neural network:

$$\hat{y}_t = f(\mathbf{x}_t; \mathbf{W}, \mathbf{b})$$

Where f is the neural network function parameterized by weights $\mathbf{W}$ and biases $\mathbf{b}$, and it can include **one or more hidden layers** with nonlinear activation functions (ReLU, tanh, etc.).

Definition 2: Feed-Forward Time-Delay Neural Network

A **Feed-Forward Time-Delay Neural Network (TDNN)** is a neural network architecture designed for **sequence modeling**, where the **input** consists of a **fixed-size window of past time steps**, and the **network learns to predict future values based on this context**. TDNNs can capture nonlinear relationships in temporal data by utilizing multiple layers and nonlinear activation functions.
Mathematically, a TDNN can be represented as:

$$\begin{aligned} h^{(1)} &= \sigma\left(\mathbf{W}^{(1)} \underbrace{[x_{t-1}, x_{t-2}, \ldots, x_{t-p}]}_{\mathbf{x}_t} + \mathbf{b}^{(1)}\right) \\ h^{(2)} &= \sigma\left(\mathbf{W}^{(2)} h^{(1)} + \mathbf{b}^{(2)}\right) \\ &\vdots \\ \hat{y}_t &= \mathbf{W}^{(L)} h^{(L-1)} + \mathbf{b}^{(L)} \end{aligned} \tag{93}$$

Where:

- σ is a **nonlinear activation function** (e.g., ReLU, tanh).
- $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the weights and biases for layer l.
- L is the total number of layers in the network.
- $h_t^{(l)}$ represents the hidden layer activations at layer l and time t.

Each time step produces its own prediction based on the **previous** p **samples**.

Example 2: TDNN for Time Series Prediction

Suppose we want to predict whether the **temperature** will rise or fall tomorrow. The relation may depend on nonlinear combinations, for example: "if temperature has been rising for 2 days **and** humidity is decreasing, then likely it will rise again". Such behavior cannot be captured by linear AR coefficients, but a neural network can **learn** this rule through nonlinear layers.

⚖ Properties and Limitations

- ✔ **Nonlinear**: TDNNs can model complex, nonlinear relationships in temporal data. This allows them to capture patterns that linear models cannot.
- ✘ **Still memoryless**: TDNNs do not have an internal state that persists across time steps. **Each prediction is made solely based on the fixed-size input window**, without retaining information from previous predictions. It is the same limitation as AR models.
- ✘ **Deterministic**: TDNNs produce a single output for a given input window, without modeling uncertainty or variability in predictions. In other words, no stochastic transition or hidden states. This can be a limitation in scenarios where uncertainty is important (e.g., weather forecasting).
- ✘ **Static structure**: The architecture of a TDNN is fixed once defined, meaning it cannot dynamically adjust its structure based on the input sequence length or complexity. Network must be retrained or resized for different sequence lengths.
- ✔ **Parallelizable**: Each window can be processed independently, allowing for efficient training and inference on modern hardware (GPUs).

So TDNNs are a step up from linear AR models, adding **nonlinearity** and **learned feature extraction**, but they **still lack the ability to maintain an internal state or model uncertainty over time**. For tasks requiring memory of past states or probabilistic predictions, more advanced architectures like Recurrent Neural Networks (RNNs) or Long Short-Term Memory (LSTM) networks are needed.

4.3 Models with Memory

4.3.1 Hidden State Dynamics and Outputs

So far, our models only saw **a sliding window** of past data. But what if the dependency extends over *hundreds* of time steps? Keeping all past inputs becomes impossible. Instead of feeding all the history explicitly, we can **summarize** it in a **hidden state vector** $\mathbf{h}_t$. That state:

- ✔ Evolves over time;
- ✔ Captures everything important about the past;
- ✔ Is sufficient to predict the present or future.

$$\text{Past Inputs } (x_0, x_1, \ldots, x_{t-1}) \xrightarrow{\text{compressed into}} \mathbf{h}_{t-1}$$

At each time step, the model receives a new input x_t and updates its hidden state $\mathbf{h}_t$ based on the previous state $\mathbf{h}_{t-1}$ and the new input x_t:

$$\mathbf{h}_t = F\left(\mathbf{h}_{t-1}, \mathbf{x}_t\right)$$

Finally, the model produces an output $\mathbf{y}_t$ based on the current hidden state $\mathbf{h}_t$:

$$\mathbf{y}_t = G\left(\mathbf{h}_t\right)$$

Since we are in a **dynamical system** setting, F and G are special functions:

- F is the **state transition function**, which defines how the hidden state evolves over time;
- G is the **output function**, which defines how the outputs are generated from the hidden states.
- $\mathbf{h}_t$ is the **hidden state** at time step t;
- $\mathbf{y}_t$ is the **output** at time step t.
- $\mathbf{x}_t$ is the **input** at time step t.

The functions F and G can be **deterministic or stochastic**, **linear or nonlinear**.

Example 3: Linear Dynamical System (LDS)

If both state transition function F and output function G are linear, we have a **Linear Dynamical System (LDS)**:

$$\begin{cases} \mathbf{h}_t = \mathbf{A}\mathbf{h}_{t-1} + \mathbf{B}\mathbf{x}_t + \omega_t \\ \mathbf{y}_t = \mathbf{C}\mathbf{h}_t + \nu_t \end{cases}$$

Where:

- $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are system matrices defined by the model;
- ω_t is the process noise (usually Gaussian);

- ν_t is the observation noise (usually Gaussian).

This system is widely used in control theory and signal processing:

- Evolves linearly in state space;
- Has memory through the matrix $\mathbf{A}$, which defines how past states influence the current state.
- And is **stochastic**, so uncertainty is tracked explicitly.

If the hidden state $\mathbf{h}_t$ can't be observed directly, we can estimate it using the **Kalman filter**. If this example is unfamiliar, don't worry; we will cover it in more detail later. It serves here to illustrate the concept of models with memory using hidden states.

Where is the Memory?

The memory in these models is encapsulated in the **hidden state vector** $\mathbf{h}_t$. This vector serves as a **summary** of all past inputs and states, allowing the model to retain information over time without needing to store the entire history explicitly. So it is a sort of *model's memory* of past events, which influences its current and future behavior. This state can be represented conditionally as (90, page 188):

$$P\left(\mathbf{h}_t \mid \mathbf{h}_{t-1}, \mathbf{h}_{t-2}, \ldots, \mathbf{h}_0\right)$$

But since $\mathbf{h}_{t-1} \sim P\left(\mathbf{h}_{t-1} \mid \mathbf{h}_{t-2}, \ldots, \mathbf{h}_0\right)$, we can simplify this to:

$$P\left(\mathbf{h}_t \mid \mathbf{h}_{t-1}\right) \approx P\left(\mathbf{h}_t \mid \mathbf{h}_{t-1}, \mathbf{h}_{t-2}, \ldots, \mathbf{h}_0\right)$$

This property is known as the **Markov Property** and is fundamental in the design of models with memory, because it allows us to focus on the most recent state $\mathbf{h}_{t-1}$ to predict the next state $\mathbf{h}_t$, rather than needing to consider the entire history of states. It lightens the computational load and simplifies the modeling process because we only need to keep track of the current state, not the entire sequence of past states.

Definition 3: Markov Property

A process satisfies the **Markov Property** if "the *future* depends **only** on the *present*, and **not** directly on the *past* before it". Formally:

$$P\left(h_t \mid h_{t-1}, h_{t-2}, \ldots, h_0\right) = P\left(h_t \mid h_{t-1}\right) \tag{94}$$

So, once we know the **current state** h_{t-1}, the **past states** $h_{t-2}, \ldots, h_0$ provide no additional information about the **future state** h_t. All relevant information from the past is **summarized** inside the current state h_{t-1}.

4.3.2 Linear Dynamical Systems and the Kalman Filter

A **Linear Dynamical System (LDS)** is a *state-space model* that represents how a system evolves over time using **linear equations**. It assumes that there's a hidden internal state $\mathbf{h}_t$ that evolves dynamically and generates the observed outputs $\mathbf{y}_t$. It's the mathematical backbone behind: signal tracking, control systems, robotics, and time-series forecasting (before RNNs took over). Formally, an LDS is defined by two main equations:

$$\text{State Equation:} \quad \mathbf{h}_t = \mathbf{A}\mathbf{h}_{t-1} + \mathbf{B}\mathbf{x}_t + \mathbf{w}_t \tag{95}$$

$$\text{Observation Equation:} \quad \mathbf{y}_t = \mathbf{C}\mathbf{h}_t + \mathbf{v}_t \tag{96}$$

- $\mathbf{h}_t$ is the **hidden** (latent) **state** at time t (a continuous-valued vector, linearly evolving over time).
- $\mathbf{x}_t$ is the **input** (control) at time t.
- $\mathbf{y}_t$ is the **observed output** at time t.
- $\mathbf{A}$ is the **state transition matrix** that defines how the hidden state evolves.
- $\mathbf{B}$ is the **control input matrix** that defines how inputs affect the hidden state.
- $\mathbf{C}$ is the **observation matrix** that maps the hidden state to the observed output.
- $\mathbf{w}_t$ and $\mathbf{v}_t$ are **process** and **observation noise**, typically assumed to be Gaussian with zero mean.

All matrices ($\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$) are constant over time, making the system linear and time-invariant. Furthermore, the **noise** terms $\mathbf{w}_t$ and $\mathbf{v}_t$ are usually modeled as **Gaussian** distributions.

⚠ State Estimation Problem

The *state estimation problem* poses a particular challenge in linear dynamical systems. Let's start with the basic situation. We have a **system that evolves in time**, like:

$$h_t = Ah_{t-1} + Bx_t + w_t$$

And we can **observe** something related to it:

$$y_t = Ch_t + v_t$$

But here's the catch: we **don't get to see the hidden state h_t directly**. Instead, we only observe y_t, which is a noisy version of h_t. So the **State Estimation Problem** is: *Given the observations $y_1, y_2, \ldots, y_t$ up to time t, how can we estimate the hidden state h_t?*

? Mathematically, the equation looks solvable, so *why* can't we just compute h_t directly? The **fundamental issue** is that we **don't know the initial state** h_0. Without knowing h_0, we can't accurately compute h_t for any

$t > 0$. This uncertainty about the initial state propagates through time, making it impossible to determine the exact value of h_t just from the equations.

Why not just set $h_0 = 0$ or 1 and move on? After all, it's the start and there's no past. While it's true that we can choose an initial value for h_0, this choice is essentially a **guess**. The problem is that this **guess may be far from the true initial state**, leading to significant errors in estimating h_t as time progresses. The uncertainty in h_0 means that our estimates of h_t will always carry some level of uncertainty, which is why we need methods like the Kalman filter to help us estimate the hidden states more accurately over time. For example, in Figure 22, we see how a bad initial guess can lead to poor estimates if the initial uncertainty is low.

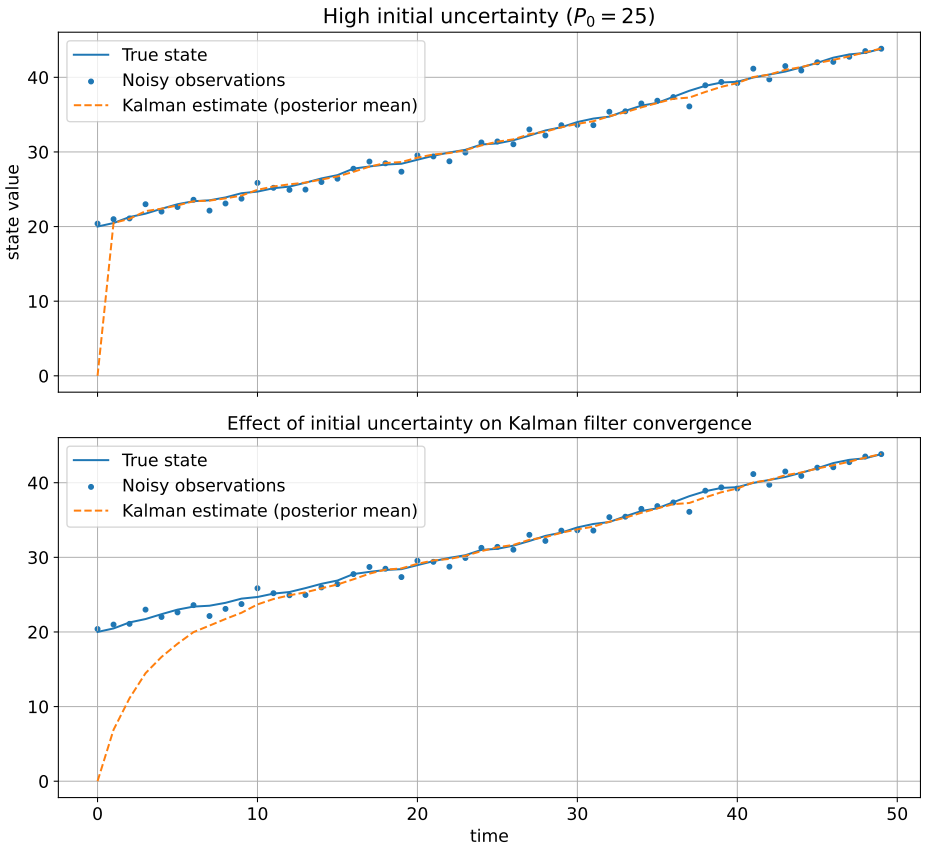


Figure 22: A comparison of Kalman filter performance with different initial uncertainty levels. The top plot shows the filter's ability to recover the true state when starting with high uncertainty about the initial state ($P_0 = 25$). The bottom plot illustrates how low initial uncertainty ($P_0 = 0.25$) can lead to poor estimates when the initial guess is incorrect, causing the filter to trust the wrong initial state too much.

Solution: The Kalman Filter

To tackle the state estimation problem in linear dynamical systems, we use the **Kalman Filter**. The Kalman filter is an **algorithm** that provides an **efficient way to estimate the hidden state $\mathbf{h}_t$ over time, even when we don't know the initial state**. The idea is simple. If we don't have the initial state, then we give our **initial guess a certain degree of uncertainty**, which can be high if we are very unsure about it. Then, as we receive new observations $\mathbf{y}_t$ (at every time step t), we **update our estimate** of the hidden state using a two-step process:

1. **Prediction Step**. Predict the new state and uncertainty based on the previous estimate:

$$\begin{cases} \hat{h}_{(t\,|\,t-1)} = A\,\hat{h}_{(t-1\,|\,t-1)} + B\,x_t \\ P_{(t\,|\,t-1)} = A\,P_{(t-1\,|\,t-1)}\,A^T + Q \end{cases} \tag{97}$$

   - $\hat{h}_{(t\,|\,t-1)}$ is the **predicted state estimate** at time t given observations up to time $t-1$.
   - $P_{(t\,|\,t-1)}$ is the **predicted estimate covariance** (uncertainty) at time t.
   - Q is the process **noise covariance matrix**.

   At time $t = 0$, we set:

$$\begin{cases} \hat{h}_{(t\,|\,t-1)} = A\,\hat{h}_{(t-1\,|\,t-1)} + B\,x_t \\ P_{(t\,|\,t-1)} = A\,P_{(t-1\,|\,t-1)}\,A^T + Q \end{cases} \xrightarrow{t=0} \begin{cases} \hat{h}_{(0\,|\,-1)} = \mu_0 \\ P_{(0\,|\,-1)} = P_0 \end{cases}$$

   Where μ_0 is our **initial guess for the state**, and P_0 is the **initial uncertainty** (covariance) **associated with that guess**.

2. **Update Step**. When the new observation $\mathbf{y}_t$ arrives, we update our state estimate and uncertainty:

$$\begin{cases} K_t = P_{(t\,|\,t-1)}C^T\left(C\,P_{(t\,|\,t-1)}\,C^T + R\right)^{-1} \\ \hat{h}_{(t\,|\,t)} = \hat{h}_{(t\,|\,t-1)} + K_t\left(y_t - C\,\hat{h}_{(t\,|\,t-1)}\right) \\ P_{(t\,|\,t)} = (I - K_t\,C)\,P_{(t\,|\,t-1)} \end{cases} \tag{98}$$

   - K_t is the **Kalman gain**, which determines how much we trust the new observation versus our prediction.
   - R is the **observation noise covariance matrix**.
   - $\hat{h}_{(t\,|\,t)}$ is the **updated state estimate** at time t *after* incorporating the observation.
   - $P_{(t\,|\,t)}$ is the **updated estimate covariance** (uncertainty) at time t.

   At time $t = 0$, we set:

$$\begin{cases} K_0 = P_{(0\,|\,-1)}C^T\left(C\,P_{(0\,|\,-1)}\,C^T + R\right)^{-1} \\ \hat{h}_{(0\,|\,0)} = \hat{h}_{(0\,|\,-1)} + K_0\left(y_0 - C\,\hat{h}_{(0\,|\,-1)}\right) \\ P_{(0\,|\,0)} = (I - K_0\,C)\,P_{(0\,|\,-1)} \end{cases}$$

⚠ Why can't we just set a big uncertainty and let the Kalman filter fix everything fast?

Setting a large initial uncertainty (P_0) allows the Kalman filter to be **more responsive to new measurements initially**, as it indicates that we are very unsure about our initial state estimate. However, this approach has its **drawbacks**:

- **Overfitting to Noise**: A very high initial uncertainty can lead the filter to **overfit to the noise in the early measurements**, resulting in erratic estimates that do not accurately reflect the true state.
- **Slower Convergence**: While a high uncertainty allows for rapid adjustments, it can also cause the filter to **take longer to converge** to a stable estimate, especially if the measurements are noisy. Due to the high uncertainty, the **filter may oscillate significantly** before settling down.
- **Instability**: In some cases, an excessively high initial uncertainty can lead to **numerical instability in the filter's calculations**, particularly in the computation of the Kalman gain.

Therefore, while a large initial uncertainty can be beneficial in certain scenarios, it is crucial to balance it with the characteristics of the system and the expected noise levels to ensure optimal performance of the Kalman filter. Let's see how different settings of process noise (Q) and observation noise (R) affect the Kalman filter's performance:

- **High Process Noise Q and Low Observation Noise R**: The filter tends to trust the measurements more, leading to rapid adjustments in the state estimate. This can be beneficial when the system is highly dynamic, but it may also result in **overfitting to noisy measurements**.

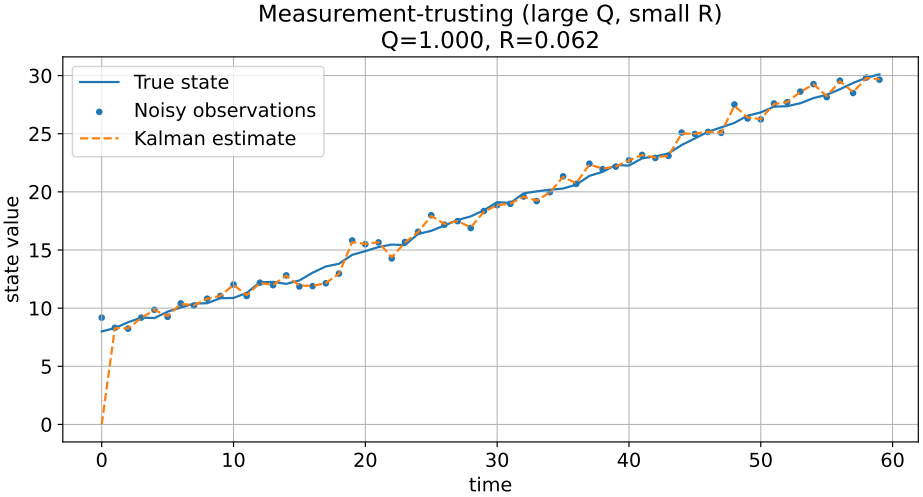


Figure 23: Kalman filter performance with high process noise (Q) and low observation noise (R). The filter quickly adapts to measurements but may overfit to noise.

✅ **Balanced Process and Observation Noise (Q and R)**: The filter strikes a balance between trusting the model predictions and the measurements. This setting is often ideal for many applications, as it allows the filter to adapt to changes while maintaining stability.

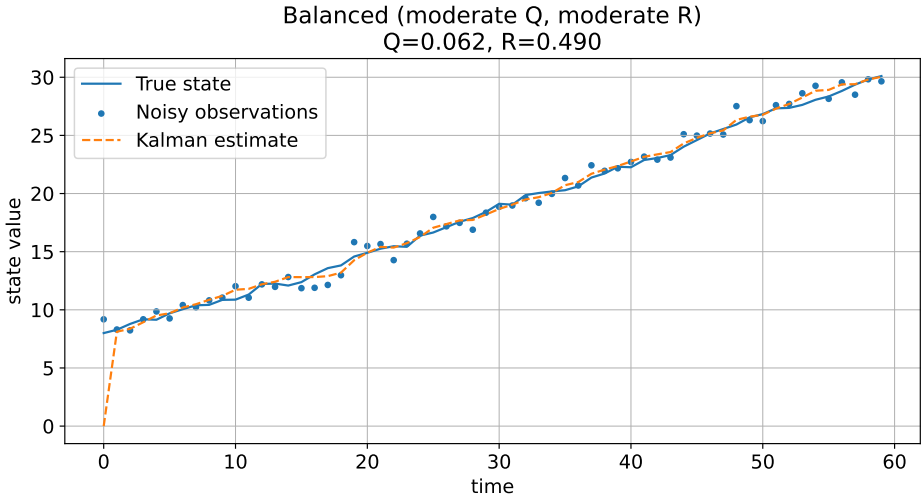


Figure 24: Kalman filter performance with balanced process and observation noise (Q and R). The filter effectively balances predictions and measurements.

⚖️ **Low Process Noise Q and High Observation Noise R**: The filter relies more on the model predictions, leading to smoother estimates. This is useful when the system is relatively stable, but it may result in slower adaptation to actual changes in the state.

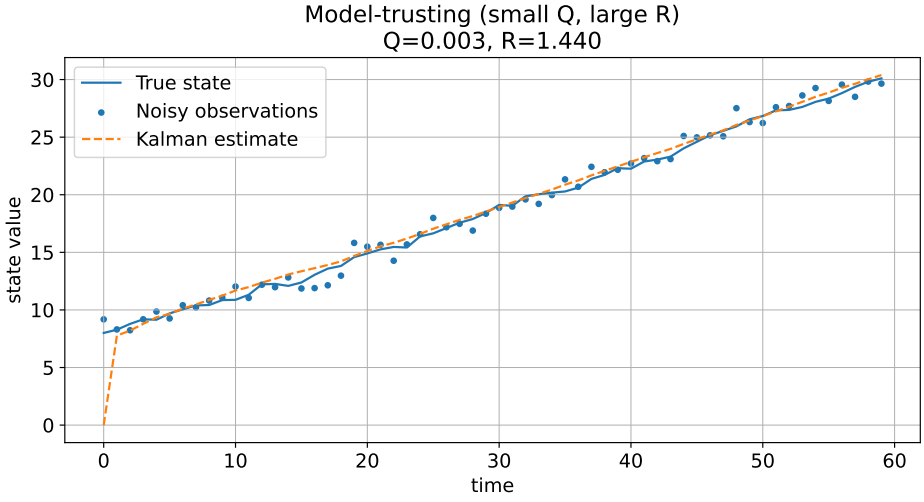


Figure 25: Kalman filter performance with low process noise (Q) and high observation noise (R). The filter produces smoother estimates but may adapt slowly to changes.

- **Kalman Gain K_t Behavior**: The Kalman gain adjusts dynamically based on the noise characteristics. A higher gain indicates more trust in the measurements, while a lower gain indicates more trust in the model predictions.

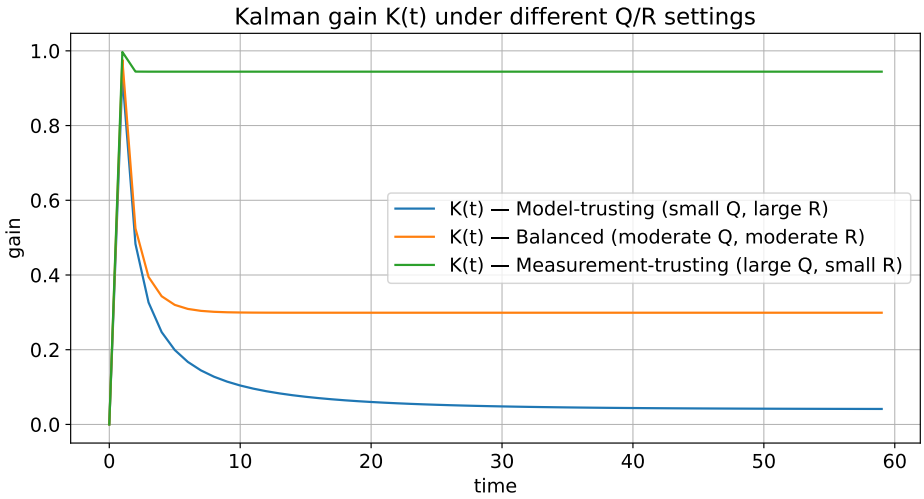


Figure 26: Kalman gain (K_t) behavior under different noise covariance settings. The gain reflects the filter's trust in measurements versus predictions.

4.3.3 Hidden Markov Models (HMMs)

In a **Linear Dynamical System (LDS)** like the Kalman filter, the hidden state h_t was **continuous** (e.g., a vector of real numbers that evolves over time). But in many real problems, the system evolves through **discrete modes or categories**, not continuous quantities. For example:

- A speaker pronouncing *phonemes* (basic sound units) in speech recognition.
- A user switching *behaviors* (e.g., browsing, purchasing) in activity recognition.
- Weather being *sunny*, *rainy*, or *cloudy* in weather modeling (not continuous temperature values).

So instead of continuous state vectors $h_t \in \mathbb{R}^n$, we have **discrete hidden states** $h_t \in \{1, 2, \ldots, K\}$ where K is the number of possible hidden states. This leads us to **Hidden Markov Models (HMMs)**, which are probabilistic models for sequences with discrete hidden states.

Definition 4: Markov Chain

A **Markov Chain** is a **mathematical model** that *implements* the Markov property (page 195). A Markov chain consists of:

- A **finite or countable set of states**.
- A **transition matrix**:

$$A = [a_{ij}], \quad a_{ij} = P(s_t = j \mid s_{t-1} = i)$$

- An **initial state distribution**:

$$\pi_i = [\pi_i], \quad \pi_i = P(s_0 = i)$$

A Markov chain **uses the Markov property** to describe how the state changes over time.

Example 4: Weather Model as a Markov Chain

Consider a simple weather model with three states: *Sunny*, *Rainy*, and *Cloudy*. The transition matrix A could be:

$$A = \begin{bmatrix} 0.8 & 0.1 & 0.1 \\ 0.4 & 0.5 & 0.1 \\ 0.3 & 0.3 & 0.4 \end{bmatrix}$$

This means that if today is Sunny, there is an 80% chance that tomorrow will also be Sunny, a 10% chance of Rainy, and a 10% chance of Cloudy.

The initial state distribution π could be:

$$\pi = \begin{bmatrix} 0.5 \\ 0.3 \\ 0.2 \end{bmatrix}$$

Indicating a 50% chance of starting in Sunny, 30% in Rainy, and 20% in Cloudy. This Markov chain can be used to model and predict weather patterns over time.

Definition 5: Hidden Markov Model (HMM)

A **Hidden Markov Model (HMM)** is a **probabilistic dynamical system** where:

- The system has a sequence of **hidden states**:

  $$s_1, s_2, \ldots, s_T, \quad s_t \in \{1, 2, \ldots, K\}$$

  Which evolve over time according to a **Markov chain**.
- At each time step, the system produces an **observation** y_t whose distribution depends **only on the current hidden state**.

An HMM is fully defined by the triplet:

$$\lambda = (\pi, A, B) \tag{99}$$

Where:

- **Initial state distribution** π:

  $$\pi_i = P(s_1 = i) \tag{100}$$

- **State transition probabilities** A:

  $$A = [a_{ij}] \qquad a_{ij} = P(s_t = j \mid s_{t-1} = i) \tag{101}$$

- **Emission probabilities** B:

  $$B = [b_j(k)] \qquad b_j(k) = P(y_t = k \mid s_t = j) \tag{102}$$

So the Hidden Markov Model defines a *probabilistic process* (stochastic process) with **two parallel processes** happening simultaneously:

1. **Hidden state sequence**. This is a sequence of **hidden states**:

   $$S = (s_1, s_2, \ldots, s_T) \quad s_t \in \{1, 2, \ldots, K\}$$

   These states represent what is *really happening* in the system, but we **cannot observe them directly** (state estimation problem). Instead, they are **latent variables** that we need to infer from the observations.

This layer evolves according to the **Markov Chain**:

$$P(s_t \mid s_{t-1}) = a_{s_{t-1},s_t}$$

- **Hidden** because we do not observe these states directly.
- **Markov** because the next state depends only on the current state (Markov property).
- **Chain** because it is a sequence of states evolving over time.

⚠ Markov Assumption on the Hidden States: First-Order Markov Property

The Markov Chain assumes the **first-order Markov property** for the hidden states:

$$P(s_t \mid s_1, s_2, \ldots, s_{t-1}) = P(s_t \mid s_{t-1})$$

This means that the **next hidden state** depends **only on the current one**, not on the entire history of past states. This assumption simplifies the model and makes computations tractable, because the entire history before s_{t-1} is irrelevant once current state s_{t-1} is known. Without this assumption, the model would become significantly more complex and harder to work with (**exponentially many parameters**) because we would need to consider all previous states to predict the next one:

$$\text{Without Markov Assumption: } \Rightarrow P(s_t \mid s_1, s_2, \ldots, s_{t-1})$$

2. **Observation sequence**. This is a sequence of **observed data**:

$$Y = (y_1, y_2, \ldots, y_t) \quad y_T \in \mathcal{O}$$

These are the **actual measurements**, the data we *can* see. Each observation is generated **from the current hidden state only**:

$$P(y_t \mid s_t) = b_{s_t}(y_t)$$

This is called the **emission probability**, or **emission model**, because the hidden state *emits* an observation according to this distribution.

- **Observation** because these are the data we actually see.
- **Generated** because they are produced by the hidden states.
- **Not Markov** because observations do not have the Markov property; they depend on the hidden states.

⚠ Emission Independence Assumption

This is the second crucial assumption of Hidden Markov Models. It states that the **observation at time t depends only on the current hidden state s_t**:

$$P(y_t \mid s_t, s_{t-1}, \ldots, y_{t-1}) = P(y_t \mid s_t)$$

This means that once we know the hidden state s_t, the observation y_t is conditionally independent of all previous observations and states. In simple terms, the **observation** at time t (what we really see) depends **only on the hidden state** at time t (what is really happening), and **not on previous states or previous observations**. For example, if the hidden state is "Rainy", the probability of seeing "Umbrella" at time t does not depend on yesterday's weather, yesterday's observation, or any earlier time steps (last week, last month, etc.). The hidden state already contains all needed information. Without this assumption, the model would become significantly more complex and harder to work with, as we would need to consider dependencies on all previous states and observations:

$$\text{Without Emission Independence: } \Rightarrow P(y_t \mid s_t, s_{t-1}, \ldots, y_{t-1})$$

Instead, the emission independence assumption allows us to **simplify the model and focus on the relationship between the current hidden state and the current observation**:

$$P(y_t \mid s_t) = b_{s_t}(y_t)$$

Why do we need both assumptions for an HMM to work?

Because without these assumptions, the probability of a sequence would involve **all states and all observations interacting** with each other directly. That would make modeling impossible, inference impossible and learning impossible (because the number of parameters would explode combinatorially). Because **hiddent transitions** and **observations** are two independent kinds of dependencies, we can **factor the joint probability** of the entire sequence into manageable parts. Given the two assumptions, the full joint probability of the hidden states and observations can be expressed as:

$$P(S, Y) = P(s_1) \cdot \prod_{t=2}^{T} P(s_t \mid s_{t-1}) \cdot \prod_{t=1}^{T} P(y_t \mid s_t) \tag{103}$$

- The **first term** $P(s_1)$ represents the probability of starting in the initial hidden state.
- The **first product** multiplies all the state transition probabilities together, capturing how the hidden states evolve over time according to the Markov chain.
- The **second product** multiplies all the observation emission probabilities together, capturing how each observation is generated from the corresponding hidden state.

Obviously, this factorization would be **impossible** without the two assumptions, because we would have to consider all possible dependencies between states and observations across time steps.

Example 5: HMM Analogy: Weather and Activities

Imagine we are trying to infer the weather (hidden states) based on someone's activities (observations). The hidden states could be:

- Sunny
- Rainy
- Cloudy

The observations could be:

- Going for a walk
- Carrying an umbrella
- Staying indoors

The HMM would model how the weather changes over time (Markov chain) and how each type of weather influences the likelihood of different activities (emission probabilities). By observing the activities, we can infer the most likely sequence of weather conditions. For example, if we see someone carrying an umbrella, we might infer that it is likely Rainy. On the other hand, if we see them going for a walk, it might indicate Sunny weather.

The model is **fully specified** by the triplet $\lambda = (\pi, A, B)$:

- **Initial State Distribution** π (where we start). It tells us the **probability that the process starts in each hidden state**:

$$\pi_i = P(s_1 = i)$$

For example, in a weather model, π could represent the probabilities of starting the day as Sunny, Rainy, or Cloudy:

$$\pi = [0.6, 0.3, 0.1]$$

Where there is a 60% chance of starting Sunny, 30% Rainy, and 10% Cloudy. If we don't know the initial state, we can assume a uniform distribution:

$$\pi_i = \frac{1}{K} \quad \forall i$$

But in real tasks, π matters a lot (especially for short sequences).

- **State Transition Matrix** A (how the hidden states evolve over time). It is a $N \times N$ matrix where each entry a_{ij} represents the **probability of transitioning from hidden state** i **to hidden state** j:

$$A = [a_{ij}] \qquad a_{ij} = P(s_t = j \mid s_{t-1} = i)$$

This is the **Markov Chain** component of the HMM, defining how likely it is to move between different hidden states at each time step.

For example, in a weather model, A could represent the probabilities of transitioning between Sunny, Rainy, and Cloudy states:

$$A = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.4 & 0.5 & 0.1 \\ 0.3 & 0.3 & 0.4 \end{bmatrix}$$

Where $a_{12} = 0.2$ means there is a 20% chance of transitioning from Sunny to Rainy.

- **Emission Probability Matrix** B (given a hidden state, how likely are the observations). It describes the **probability of each observation given a hidden state**. It is a sort of "lookup table" (or mapping) that tells us how likely each observation is for each hidden state:

$$B = [b_j(k)] \qquad b_j(k) = P\left(y_t = k \mid s_t = j\right)$$

It encodes the **emission model**: hidden states **emit** observations according to probability distributions.

For example, in a weather model, B could represent the probabilities of different activities given the weather states:

$$B = \begin{bmatrix} 0.8 & 0.1 & 0.1 \\ 0.2 & 0.7 & 0.1 \\ 0.5 & 0.2 & 0.3 \end{bmatrix}$$

Where the rows correspond to hidden states (Sunny, Rainy, Cloudy) and the columns correspond to observations (Walk, Umbrella, Indoors). For instance, $b_1(1) = 0.8$ means that if the weather is Sunny, there is an 80% chance of going for a walk.

So the HMM triplet $\lambda = (\pi, A, B)$ completely defines the model, allowing us to define where the hidden state **starts** (π), how it **evolves** over time (A), and how it **produces observations** (B).

The Three Fundamental Problems of HMMs

Hidden Markov Models are powerful, but to use them effectively, we need to solve three fundamental problems:

1. **Evaluation Problem**. Given an HMM $\lambda = (\pi, A, B)$ and an observation sequence $Y = (y_1, y_2, \ldots, y_T)$, compute the probability of the observation sequence:

$$P\left(Y \mid \lambda\right)$$

This tells us how likely the observed data is under the model. It is useful for model comparison and likelihood estimation.

Solution: Forward Algorithm. The **Forward Algorithm** is not covered here, but it efficiently computes this probability using dynamic programming. A general idea is to recursively compute the probabilities of being in each hidden state at each time step, given the observations up to that point.

2. **Decoding Problem**. Given an HMM $\lambda = (\pi, A, B)$ and an observation sequence $Y = (y_1, y_2, \ldots, y_T)$, find the most likely sequence of hidden states $S = (s_1, s_2, \ldots, s_T)$ that could have generated the observations:

$$\hat{S} = \arg\max_S P(S \mid Y, \lambda)$$

This helps us infer the hidden states from the observed data (e.g., determining the weather sequence from activity observations).

Solution: Viterbi Algorithm. The **Viterbi Algorithm** is explained in detail later.

3. **Learning Problem**. Given an observation sequence $Y = (y_1, y_2, \ldots, y_T)$, estimate the HMM parameters $\lambda = (\pi, A, B)$ that best explain the observed data. This involves finding the model parameters that maximize the likelihood of the observations:

$$\lambda^* = \arg\max_\lambda P(Y \mid \lambda)$$

This allows us to train the HMM from data.

Solution: Baum-Welch Algorithm. The **Baum-Welch Algorithm** (a special case of the Expectation-Maximization, EM, algorithm) is used to iteratively estimate the HMM parameters by maximizing the likelihood of the observed data. It is not covered here, but it involves computing expected counts of transitions and emissions based on the current model, then updating the parameters accordingly.

The Viterbi Algorithm for the Decoding Problem

The Viterbi algorithm was introduced to address the decoding problem of HMMs. Here, we only explain this algorithm in detail because it is used to demonstrate the limitations of the HMM model and the need for more powerful models, such as RNNs.

First of all, we need to understand why the decoding problem is hard. In an Hidden Markov Model, the **hidden states** $s_1, s_2, \ldots, s_T$ are not directly observable. Instead, we only see the **observations** $y_1, y_2, \ldots, y_T$ that are generated from these hidden states. But often, what we really want is **the hidden states** themselves, because they represent the underlying process we are interested in (e.g., the actual weather conditions, the speaker's phonemes, etc.). So the **Decoding Problem** is about **finding the most likely sequence of hidden states given the observed data**:

$$\hat{S} = \arg\max_S P(S \mid Y, \lambda)$$

Where:

- $S = (s_1, s_2, \ldots, s_T)$ is a candidate sequence of hidden states.
- $Y = (y_1, y_2, \ldots, y_T)$ is the observed sequence.

- $\lambda = (\pi, A, B)$ are the HMM parameters.

Because there are **exponentially many** possible hidden state sequences (if there are K hidden states and T time steps, there are K^T possible sequences), we need an efficient algorithm to find the most likely one without enumerating all possibilities. The **Viterbi Algorithm** is the **efficient dynamic programming algorithm** that solves this problem in **polynomial time** ($O(T \cdot K^2)$) instead of exponential time ($O(K^T)$).

Intuition Behind the Viterbi Algorithm. Viterbi uses the fact that HMM probabilities factor nicely:

$$P(S, Y) = P(s_1) \cdot \prod_{t=2}^{T} P(s_t \mid s_{t-1}) \cdot \prod_{t=1}^{T} P(y_t \mid s_t)$$

This means that we do not explore all paths, but rather we only keep the **best partial path** to each state at each time step. At time t, for each possible state, keep the probability of the best path (highest probability) that ends in that state. So, at each time step, we need two main components:

- **The "best score" matrix** δ of size $K \times T$, where $\delta_j(t)$ represents the probability of the most likely path that ends in state j at time t:

$$\delta_j(t) = \max_{s_1, s_2, \ldots, s_{t-1}} P(s_1, s_2, \ldots, s_{t-1}, s_t = j, y_1, y_2, \ldots, y_t \mid \lambda) \tag{104}$$

- **The "argmax pointer" matrix** ψ of size $K \times T$, where $\psi_j(t)$ stores the index of the state at time $t-1$ that led to the best path to state j at time t:

$$\psi_j(t) = \arg\max_i [\delta_i(t-1) \cdot a_{ij}] \tag{105}$$

  Later, we will use this matrix to backtrack and reconstruct the most likely sequence of hidden states.

The **Viterbi algorithm** proceeds in four main steps:

1. **Initialization** $t = 1$. For each state j (where a hidden state can be):

$$\delta_j(1) = \pi_j \cdot b_j(y_1)$$

   Where π_j is the initial probability of starting in state j, and $b_j(y_1)$ is the emission probability of observing y_1 from state j. Also, initialize the pointer:

$$\psi_j(1) = 0$$

   That is, there is no previous state at time 1.

2. **Recursion for** $t = 2 \ldots T$. We compute matrices δ and ψ for each state j:

$$\delta_j(t) = \max_i [\delta_{t-1}(i) \cdot a_{ij}] \cdot b_j(y_t)$$

   This finds the best previous state i that leads to state j at time t, multiplies by the transition probability a_{ij} and the emission probability $b_j(y_t)$. We also update the pointer:

$$\psi_j(t) = \arg\max_i [\delta_{t-1}(i) \cdot a_{ij}]$$

   This stores the index of the best previous state.

3. **Termination**. After filling in the matrices, we find the probability of the best full path:

$$P^* = \max_j \delta_T(j)$$

And the final state of the best path:

$$s_T^* = \arg\max_j \delta_T(j)$$

4. **Backtracking**. Finally, we reconstruct the most likely sequence of hidden states by backtracking through the pointer matrix ψ:

$$s_t^* = \psi_{t+1}(s_{t+1}^*) \quad \text{for } t = T-1, T-2, \ldots, 1$$

This traces back from the final state s_T^* to the initial state s_1^*, using the pointers stored in ψ.

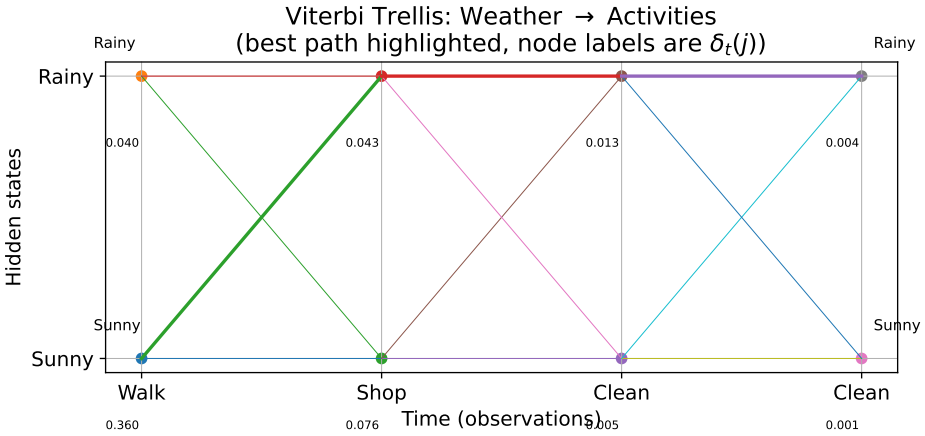


Figure 27: **Viterbi trellis diagram for a simple Weather to Activities HMM.** Each column is a time step (Walk, Shop, Clean, Clean), each row a hidden state (Sunny, Rainy). The node values ($\delta_t(j)$) are the Viterbi scores: the probability of the *best* path that ends in state j at time t. Thin edges show all possible transitions; the thick path highlights the most probable hidden sequence found by Viterbi: Sunny → Rainy → Rainy → Rainy.

4.3.4 Comparison to Deterministic Recurrent Systems

In this section, we will attempt to answer the following question: *How do classical dynamical models (e.g., Kalman filters and Hidden Markov Models) compare to deterministic recurrent systems (e.g., Recurrent Neural Networks)?* The answer will prepare us for the introduction of **neural-based recurrent architectures** in the next section.

First of all, we highlight the difference between **probabilistic** and **deterministic** models.

- **Probabilistic Dynamical Systems** (e.g., HMMs, Kalman filters) model the evolution of a system over time by **incorporating uncertainty and randomness**. They **use probability distributions** to represent the state of the system and the observations, allowing them to handle noise and uncertainty in the data effectively. In HMMs, for instance, the system transitions between hidden states based on probabilistic rules, and observations are generated from these states according to specific probability distributions.
- **Deterministic Recurrent Systems** (e.g., RNNs) model the evolution of a system over time by **using fixed rules or functions** to determine the next state based on the current state and input. They do not inherently account for uncertainty or randomness in their predictions. Instead, they rely on learned weights and activation functions to capture temporal dependencies in sequential data.

Let's make a more detailed comparison between these two types of models:

- **Probabilistic Dynamical Systems** are well-suited for tasks where uncertainty and noise are prevalent, such as speech recognition, natural language processing, and time series forecasting. They can effectively model the underlying stochastic processes that generate the observed data.

  ✔ **Key Properties**

  - State transitions are **probabilistic**, allowing the model to capture uncertainty in the system's evolution.
  - Observations are **probabilistic**, enabling the model to handle noisy data effectively.
  - They explicitly represent **uncertainty** in both state transitions and observations.
  - Inference is based on **probability distributions**, allowing for robust predictions in the presence of noise.
  - Decisions are based on **likelihoods** and **argmax** (e.g., Viterbi algorithm for HMMs) to find the most probable sequence of states.

  In practice, they are often used in scenarios where the data is noisy or incomplete, and where modeling uncertainty is crucial for accurate predictions.

- **Deterministic Recurrent Systems** excel in tasks that require learning complex temporal patterns and dependencies from large datasets, such as language modeling, machine translation, and video analysis. They can capture intricate relationships in sequential data through their learned representations. They are the ancestor of modern RNN architectures like LSTMs and GRUs. A deterministic recurrent system can be described by the following equations:

  $$h_t = f\left(h_{t-1},\, x_t\right)$$

  Where f is a **deterministic function** (no randomness).

  ✔ **Key Properties**

  - No noise terms.
  - No probability distributions.
  - Hidden state evolves deterministically.
  - Same input sequence, then same hidden state trajectory.
  - Learning (if any) is achieved via optimization of a loss function (e.g., MSE, cross-entropy) using gradient-based methods, not via probabilistic inference.

  This makes them computationally efficient and easier to implement, especially for large-scale problems, but less expressive for handling uncertainty; less suited for tasks where observations are noisy; harder to reason about probabilistically. In practice, they are often used in scenarios where large amounts of sequential data are available, and the focus is on learning complex temporal patterns rather than modeling uncertainty.

The main reason to transition from probabilistic dynamical systems to deterministic recurrent systems is that the latter can **learn complex temporal patterns and dependencies from large datasets**. This makes them more suitable for tasks such as language modeling, machine translation, and video analysis. RNNs can capture intricate relationships in sequential data through their learned representations, which is often more challenging for probabilistic dynamical systems that rely on predefined probabilistic rules and distributions. Additionally, RNNs can be trained end-to-end using gradient-based optimization techniques, allowing them to adapt to the specific characteristics of their training data. This flexibility and adaptability make RNNs a powerful choice for many modern deep learning applications.

4.4 Definition

4.4.1 What is a RNN?

Feed-Forward Networks (FFNs) cannot handle sequences because:

- They take only **fixed-size** input vectors.
- They have **no internal state**, so the **forget everything** once the forward pass is done.

To handle temporal data, we need **a model that keeps track of past information while processing the present**. This requires **memory**. RNNs introduce memory via **recurrent connections**, which allow information to persist across time steps.

Definition 6: Recurrent Neural Network (RNN)

A **Recurrent Neural Network (RNN)** is a neural architecture designed to process **sequences** by maintaining a **memory** of past inputs through **recurrent connections**. Formally:

$$h_t = f\left(W_x \cdot x_t + W_h \cdot h_{t-1} + b\right) \tag{106}$$

Where:

- x_t is the input at time step t.
- h_t is the hidden state (memory) at time step t (**memory** of the RNN).
- W_x and W_h are weight matrices for input-to-hidden and hidden-to-hidden connections, respectively.
- b is a bias vector.
- f is a non-linear activation function (e.g., tanh, ReLU).

The RNN produces an output y_t at each time step, which can be computed as:

$$y_t = g\left(W_y \cdot h_t\right) \tag{107}$$

Where:

- W_y is the weight matrix for hidden-to-output connections.
- g is an activation function for the output layer (e.g., softmax for classification).

What is a recurrent connection?

In a normal neural network layer, the output is computed as:

$$h = \sigma\left(Wx + b\right)$$

Where h is the output, x is the input, W is the weight matrix, b is the bias, and σ is an activation function. In a layer with **recurrent connections**, the output at time step t also depends on the output from the previous time step $t-1$:

$$h_t = \sigma\left(W_x \cdot x_t + W_h \cdot h_{t-1} + b\right)$$

So, we have a new term $W_h \cdot h_{t-1}$ that incorporates information from the previous time step, allowing the network to maintain a form of memory over time. It is called **recurrent connection** because the output at one time step is fed back into the network as input for the next time step, creating a loop in the network architecture (a link from the **previous hidden state** back into the computation of the **current** hidden state). This single recursive connection is what gives the RNN its "memory".

What is the hidden state?

The **hidden state** h_t in an RNN is a **continuous vector** that serves as the network's **memory** at time step t. It is updated at **every time step** based on the current input x_t and the previous hidden state h_{t-1}:

$$h_t = f\left(h_{t-1}, x_t\right) = \sigma\left(W_x \cdot x_t + W_h \cdot h_{t-1} + b\right)$$

This recovers the structure of **dynamical systems** (e.g., Kalman filters, HMMs) we saw before, but now the update function is **learned**, not fixed. It is more flexible and powerful, allowing the RNN to **capture complex temporal dependencies in the data** (nonlinear relationships, long-term dependencies, etc.).

Why this gives the RNN memory

Because every hidden state h_t contains:

- Information from h_{t-1} (the previous hidden state).
- Which in turn contains information from h_{t-2}.
- And so on, back to h_0.

So the RNN hidden state implicitly **stores a compressed summary of *all previous inputs***:

$$h_t = f\left(h_{t-1}, x_t\right) = f\left(f\left(h_{t-2}, x_{t-1}\right), x_t\right) = f\left(f\left(f\left(h_{t-3}, x_{t-2}\right), x_{t-1}\right), x_t\right) = \ldots$$

This recursive structure is the "memory mechanism" of RNNs, allowing them to **retain information over time** and make predictions based on the entire sequence history.

Hidden State as Distributed Memory

As we have seen, at each time step t, the RNN updates:

$$h_t = f\left(W_x \cdot x_t + W_h \cdot h_{t-1} + b\right)$$

- h_{t-1} contains **what the network remembers about the past**.
- x_t injects **new information from the present**.
- W_h decides **how strongly past memory is kept**.
- W_x decides **how strongly new input is incorporated**.

So the RNN is **carrying forward a summary of all previous inputs**. This summary is the hidden state vector h_t.

What does "distributed memory" mean? The memory is not stored in one neuron. Instead, memory is stored across many components of the hidden vector. Each component of h_t:

- ✗ Does not represent a human-interpretable concept.
- ✗ Does not store one specific piece of information.
- ✗ Is not binary (not like Hidden Markov Model discrete states).

Instead, **memory is encoded across *all dimensions* of the hidden state vector**. This is the same idea as "distributed representations" in deep learning: a concept is not stored in a single neuron, but rather it's stored in the *pattern* of activations and memory is encoded in many dimensions simultaneously.

The RNN hidden state is thus a distributed memory because its information is encoded across all its dimensions, and **each neuron extracts different aspects** of this memory through different learned weight projections.

Example 6: Hidden State Analogy

Consider each neuron in the hidden state vector h_t as a **musician** in an orchestra.

Each musician (**neuron**) has different instruments (**weights**), different skills (**non-linearities**), different interpretations (**activations**), and different roles in the ensemble (**dimensions of the hidden state**).

So although they all read the same sheet music (the **input sequence**), each musician contributes a unique sound (**feature**) to the overall performance (**hidden state**):

- The violinist (neuron 1) might focus on melody (*temporal patterns*).
- The percussionist (neuron 2) might emphasize rhythm (*short-term dependencies*).
- The bassist (neuron 3) might provide harmonic support (*long-term*

context).

- And so on for all musicians (neurons) in the orchestra (hidden state).

So, same input (sheet music), different roles (neurons), different extracted information (features from memory).

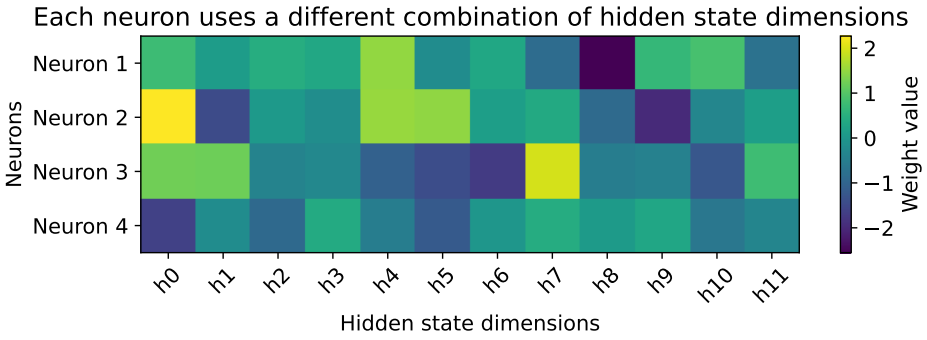


Figure 28: Heatmap with 4 different neurons (rows), each extracting different information from the same hidden state vector (columns). The colors indicate the weight magnitudes, showing how each neuron focuses on different parts of the hidden state to extract unique features. Greater weight magnitudes (lighter colors) indicate stronger influence from those hidden state dimensions. This illustrates the concept of **distributed memory** in RNNs, where each neuron captures different aspects of the overall memory encoded in the hidden state.

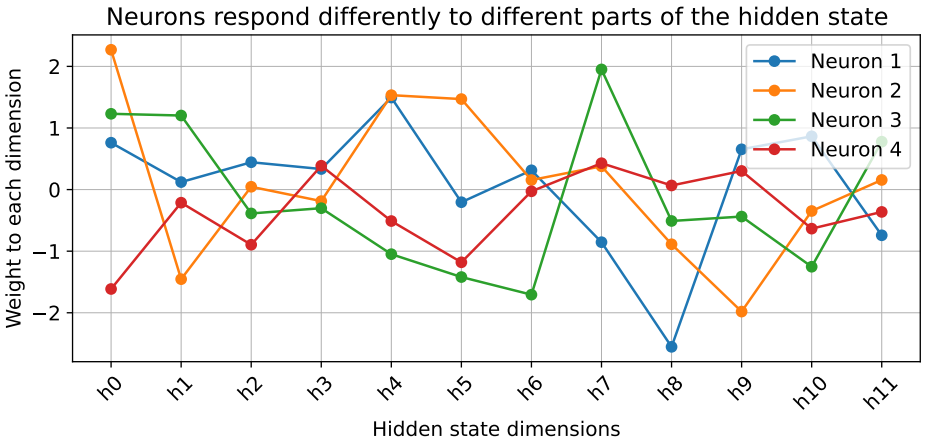


Figure 29: Line plot of 4 different neurons (lines), each extracting different information from the same hidden state vector (x-axis). The y-values represent the weight magnitudes, showing how each neuron focuses on different parts of the hidden state to extract unique features. So even though **all neurons receive the same hidden state vector** h, they *do different things with it*, they extract different aspects of the stored memory.

Why are there two matrices W_x and W_h in the RNN equation?

We will explore them in the next sections, but we can already clarify their roles here (conceptually). In the RNN equation:

$$h_t = f\left(W_x \cdot x_t + W_h \cdot h_{t-1} + b\right)$$

We have two different weight matrices:

- W_x **Input-to-Hidden Weight Matrix**. This matrix transforms the **current input vector** x_t into the hidden space. In other words, it answers the question: *"how does the current input influence the new memory state?"*. If x_t (the input) has dimension d_{in} and the hidden state has dimension d_h, then W_x (the input-to-hidden weight matrix) has shape (d_h, d_{in}). For example, if x_t is a word embedding of size 100 and the RNN hidden state size is 64, then W_x will be a matrix of shape $(64, 100) = 6400$ parameters.
- W_h **Hidden-to-Hidden Weight Matrix**. This matrix transforms the **previous hidden state** h_{t-1} into the new hidden space:

  $$W_h \cdot h_{t-1}$$

  This is the **core of memory** in an RNN, as it tells how past information influences the present, it recycles the previous hidden state into the new one and it is applied **at every time step**. If hidden size is d_h, then W_h has shape (d_h, d_h). It is **square** because it transforms hidden states into hidden states. For example, if the RNN hidden state size is 64, then W_h will be a matrix of shape $(64, 64) = 4096$ parameters.

In the next section, we will see how these matrices operate when the RNN is unrolled over time (i.e., when we visualize the RNN processing a sequence step by step). This will help clarify their distinct roles in shaping the RNN's memory dynamics.

4.4.2 Nonlinear Update Equations with Weights

As we have anticipated in (106) on page 213 of the definition, RNNs use **learned weight matrices** to transform the input and hidden states before applying a nonlinear activation function. The general form of the update equation is:

$$h_t = f\left(W_x \cdot x_t + W_h \cdot h_{t-1} + b\right)$$

But what does the equation really mean? There are two "sources" of information for the new hidden state h_t:

1. **Current input** $\rightarrow$ via W_x:

   $$W_x \cdot x_t$$

   This says "***what does the new observation*** x_t ***mean?***". It captures the information from the current input at time t. It is similar to weights in a normal feedforward neural network, transforming the input into a feature representation.

2. **Previous hidden state** $\rightarrow$ via W_h:

   $$W_h \cdot h_{t-1}$$

   This is the **memory contribution** and carries information from all past time steps (via h_{t-1}). It answers the question "***what do we remember from the past?***".

3. **Add bias and apply nonlinearity**:

   $$f(\cdot) = \tanh,\ \text{ReLU},\ \text{sigmoid},\ \ldots$$

   This introduces **nonlinearity**, making the system much more expressive than a linear dynamical system. The bias b allows shifting the activation function, enabling the model to learn more complex patterns.

An alternative notation that is often used in packages such as `PyTorch` and `TensorFlow` is:

$$h_t = \sigma\left(W_x \cdot x_t + W_h \cdot h_{t-1} + b\right)$$

Where σ is a generic nonlinear activation function (e.g., tanh, ReLU, sigmoid, etc.).

Why this equation is *nonlinear* and why it matters?

If the activations were removed:

$$h_t = W_x \cdot x_t + W_h \cdot h_{t-1}$$

We get a **Linear Dynamical System** (LDS), that is equivalent to Kalman filtering, and cannot learn complex temporal patterns. Also, memory decays or explodes geometrically over time, making it hard to capture long-term dependencies. Instead, adding the nonlinearity:

$$h_t = \tanh(\cdot)$$

Turns the system into a **universal nonlinear function approximator** for sequences.

What do the weight matrices W_x and W_h represent?

As we have anticipated on page 217, the RNN update equation contains two distinct weight matrices.

Every time step uses the same matrices W_x and W_h to process the input and hidden state. This parameter sharing allows the RNN to generalize across time steps, learning temporal patterns that are invariant to position in the sequence.

- W_x**: Input $\xrightarrow{\text{to}}$ Hidden weights**. "***How should the network interpret the current input*** x_t***?***". The role of this matrix is to **map** the **incoming signal** x_t into the **hidden state space**.
  - It **extracts** features from the *current* time step.
  - It **determines** what part of the current input is important.
  - It **transforms** x_t into a form that can be combined with memory.

  If the input is a word embedding, sensor reading, pixel row, etc., W_x learns how to process it. For example, if x_t is a word embedding like "pizza", $W_x \cdot x_t$ might activate "food topic" neurons in the hidden state.

  Mathematically, W_x has shape:

  $$W_x \in \mathbb{R}^{H \times D}$$

  Where D is the input dimension and H is the hidden state dimension.

- W_h**: Hidden $\xrightarrow{\text{to}}$ Hidden weights**. "***How should the network update its memory based on the past?***". This matrix controls how the **previous hidden state** affects the **next one**.
  - It tells the RNN **what to remember**, **what to forget**, and **how**.
  - It controls **how much** of **past information flows into the present**.
  - It defines **how memory evolves over time**.

  This is the **core** of recurrence: **the hidden state feeds back into itself through W_h**, allowing the network to build a temporal context. For example, if earlier in the sequence we saw "not", and now we see "bad", $W_h \cdot h_{t-1}$ helps interpret the meaning based on the negation remembered earlier.

  Mathematically, W_h has shape:

  $$W_h \in \mathbb{R}^{H \times H}$$

  Since it maps the hidden state back into itself. The H terms correspond to the number of hidden units.

Why two separate weight matrices?

Because the RNN must combine **two types of information** at each time step:

- **New information** (from x_t).
- **Past information** (from h_{t-1}).

They play different roles:

- W_x **processes the current input**, extracting relevant features: "*how to read the current input*".
- W_h **manages the memory**, determining how past states influence the present: "*how to transform the past memory*".

Together they form:

$$h_t = \text{memory update}\left(\underbrace{\text{input processing}\,(x_t)}_{\text{via } W_x}, \underbrace{\text{memory transformation}\,(h_{t-1})}_{\text{via } W_h}\right)$$

Where do W_x and W_h come from?

As we will see in the upcoming section on **Backpropagation Through Time (BPTT)**, these matrices are **initialized randomly** and then **learned from data** through gradient descent. During training, the RNN adjusts W_x and W_h to minimize the loss function, effectively learning how to interpret inputs and manage memory for the specific task at hand (e.g., language modeling, time series prediction, etc.). Over time, they converge to values that allow the RNN to capture complex temporal dependencies in the data.

4.4.3 Universal Computation Capability (Hava Siegelmann)

Recurrent Neural Networks (RNNs) with rational weights and sigmoidal activation functions are capable of simulating Turing machines, thus possessing universal computation capability. The idea comes from the work of Hava Siegelmann and Eduardo Sontag in the 1995s, who demonstrated that RNNs can perform computations equivalent to those of Turing machines, given sufficient time and resources. In other words, **a simple Recurrent Neural Network (RNN), with nonlinear activation, can simulate any computable function** (i.e., it is as powerful as a Turing machine).

What does "Universal Computation" mean?

The term "*universal computation*" refers to the **ability of a computational system to perform any computation that can be described algorithmically**. In the context of RNNs, it means that these networks can simulate any Turing machine, which is a theoretical model of computation that can solve any problem that is computable. This implies that RNNs can, in theory, learn and execute any algorithm or function that a Turing machine can, given enough time and resources.

Why is this important? Implications of Universal Computation Capability of RNNs

The universal computation capability of RNNs has several important implications:

- **Expressive Power**: RNNs can represent a wide range of functions and algorithms, making them versatile for various tasks, including language modeling, sequence prediction, and time series analysis.
- **Learning Complex Patterns**: RNNs can learn complex temporal patterns and dependencies in sequential data, which is crucial for tasks like speech recognition and natural language processing.
- **Theoretical Foundation**: The ability of RNNs to simulate Turing machines provides a strong theoretical foundation for their use in machine learning and artificial intelligence.
- **Limitations and Challenges**: While RNNs are theoretically capable of universal computation, practical limitations such as training difficulties, vanishing/exploding gradients, and computational resource constraints can hinder their performance in real-world applications.

Overall, the universal computation capability of RNNs highlights their potential as powerful computational models, capable of tackling a wide range of complex problems in various domains.

Intuition: Why RNNs are Turing Complete?

A machine is considered Turing complete if it can simulate a Turing machine, which means it can perform any computation that a Turing machine can, given enough time and resources. So, why are RNNs Turing complete? Consider:

- The hidden state h_t.
- The recurrence $h_t = f\left(W_x \cdot x_t + W_h \cdot h_{t-1}\right)$

The hidden state acts like a **memory tape** (continuous, high-dimensional) that evolves according to a **rule** feeding into itself indefinitely. This is exactly what a Turing machine does: it has a memory tape, a transition rule, and uses past state and current input to compute the next state. Thus, RNN can emulate read/write operations on a tape, transitions, state machines, counters, stacks, etc. With appropriate weights and activation functions, RNNs can implement the necessary operations to simulate any Turing machine, thereby achieving Turing completeness.

Caveat: Practical Limitations

While RNNs are theoretically Turing complete, practical limitations exist:

- ✗ RNNs cannot run infinite time steps in practice.
- ✗ They do not have infinite precision in weights and activations.
- ✗ They suffer from vanishing and exploding gradient problems during training.
- ✗ They cannot store long-term memory reliably.
- ✗ Training is extremely challenging for complex tasks.

These **limitations have led to the development of more advanced RNN architectures**, such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), which are designed to mitigate some of these challenges and improve the ability of RNNs to learn long-term dependencies in sequential data.

4.5 Backpropagation Through Time (BPTT)

Backpropagation Through Time (BPTT) is the algorithm used to train Recurrent Neural Networks (RNNs). It is simply **backpropagation** (page 96) **applied to the unrolled RNN**. However, an RNN is **not a standard feed-forward network** because it has **cycles** due to its recurrent connections:

$$h_{t-1} \to [\text{RNN cell}] \to h_t \to [\text{RNN cell}] \to h_{t+1} \to \dots$$

These cycles create **dependencies over time**, making the RNN **deep in time** rather than deep in space. So, the RNN cannot run backpropagation directly on its cyclic graph. The solution is to **unroll the RNN over time**, converting it into a **Directed Acyclic Graph (DAG)**. Once unrolled, we can perform backpropagation as usual, computing gradients for each time step and accumulating them to update the shared weights.

Once unrolled, we simply:

1. Run forward pass through all timesteps:

$$\text{compute } h_1, h_2, \dots, h_T$$

2. Compute the loss at each timestep and sum them:

$$E = \sum_{t=1}^{T} E_t$$

3. Run backpropagation through the unrolled network, computing gradients at each timestep and accumulating them:

$$\frac{\partial E}{\partial W_x} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W_x}, \quad \frac{\partial E}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W_h}, \quad \frac{\partial E}{\partial b} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial b}$$

4. Update the shared weights using the accumulated gradients.

That is Backpropagation Through Time in a nutshell. It is just plain backpropagation but applied **through time**, not space. But, **how do we unroll an RNN?** Let's explore that next.

4.5.1 RNN unrolling over U time steps

Imagine a normal Feed-Forward Neural Network (FNN):

$$x \to [\text{Layer 1}] \to [\text{Layer 2}] \to \dots \to [\text{Layer N}] \to y$$

Each layer transforms the signal and passes it forward to the next layer. This is **deep in space** because we have multiple layers stacked vertically (one on top of the other).

Now, instead of stacking layers in space (vertically), we stack the *same* layer **in time** (horizontally):

$$\underbrace{\text{RNN cell}}_{t=1} \rightarrow \underbrace{\text{RNN cell}}_{t=2} \rightarrow \ldots \rightarrow \underbrace{\text{RNN cell}}_{t=U}$$

That's all unrolling is. **Each cell is the same network**. It just receives **different inputs** (x_1, x_2, …, x_U) at each time step and the **previous hidden state** instead of the previous layer's output. So, an RNN is basically **a small neural network copied over and over along the time axis**, all copies share the same weights and each copy processes one timestep.

What is an RNN cell? A **cell** is the *tiny neural network* that computes:

$$h_t = f\left(W_x \cdot x_t + W_h \cdot h_{t-1} + b\right)$$

This little network contains:

- The weight matrix W_x that processes the **current input** x_t;
- The weight matrix W_h that processes the **previous hidden state** h_{t-1};
- The bias vector b;
- The activation function f (e.g., tanh or ReLU).

This is the **entire RNN**. A cell is a function:

$$(x_t,\, h_{t-1}) \mapsto h_t$$

That's it.

What does it mean that "all cells are the same"? It means:

- They use the **same weight matrices**, the same W_x, W_h, and b, for every time step.
- They are the **same function** applied repeatedly over time. The RNN does not create new parameters at each time step; it **reuses the same ones**.

In code, an RNN is just a **for loop** that applies the same function (the RNN cell) to each input in the sequence, passing along the hidden state:

```
h = h0  # initial hidden state
for t in range(U):
    # same RNN_cell function at each time step
    h = RNN_cell(x[t], h)
```

There is only **one copy** of W_x and W_h in memory, shared across all time steps. But the loop executes **multiple times**, so it *looks like* the function is repeated many times. In previous diagrams, we drew **multiple copies of the cell** to illustrate the unrolling over time, but in reality, it's **just the same cell being applied repeatedly**.

What does "unroll" mean?

It means we create **U copies** of the RNN cell, one per time step:

$$
\begin{aligned}
x_{t-3} \rightarrow & \underbrace{\text{RNN cell}}_{t-3} \rightarrow h_{t-3} \\
x_{t-2} \rightarrow & \underbrace{\text{RNN cell}}_{t-2} \rightarrow h_{t-2} \\
x_{t-1} \rightarrow & \underbrace{\text{RNN cell}}_{t-1} \rightarrow h_{t-1} \\
x_{t} \rightarrow & \underbrace{\text{RNN cell}}_{t} \rightarrow h_{t}
\end{aligned}
$$

Each cell processes its own input and previous hidden state, producing its own output hidden state. But **all cells share the same weights**. Unrolling is just a way to visualize how the RNN processes sequences over time. This means:

- In forward propagation → every timestep uses the same weights to compute its hidden state.
- In backpropagation → gradients from all time steps accumulate to update the *same* weights.
- In update rules → we average (or sum) gradients from all time steps to adjust the shared weights.

Why unroll an RNN? Unrolling helps us understand how the RNN processes sequences step-by-step. But, more importantly, unrolling is essential for **training** the RNN using *Backpropagation Through Time (BPTT)*. BPTT requires a DAG (Directed Acyclic Graph) structure to compute gradients, explicit connections between operations, and a clear path for gradients to flow backward through time. The RNN loop must be turned into a **long chain** before we can apply backpropagation.

Mathematical view of unrolling

For U time steps, the loss function E depends on the outputs at each time step:

$$
E = \sum_{t=1}^{U} E_t
$$

Where E_t is the **loss at time step** t. Each output h_t depends on the weights W_x, W_h, and b shared across all time steps. During backpropagation, we compute the gradients of the loss with respect to the weights:

$$
\frac{\partial E}{\partial W_x} = \sum_{t=1}^{U} \frac{\partial E_t}{\partial W_x}, \quad \frac{\partial E}{\partial W_h} = \sum_{t=1}^{U} \frac{\partial E_t}{\partial W_h}, \quad \frac{\partial E}{\partial b} = \sum_{t=1}^{U} \frac{\partial E_t}{\partial b}
$$

This means that the **total gradient for each weight matrix is the sum of the gradients from each time step**. This accumulation is crucial for

updating the shared weights during training. Each term depends not just on timestep t but via the recurrence on all previous timesteps:

$$h_{t-1}, h_{t-2}, \ldots, h_0$$

This is why we need to unroll the RNN: to explicitly see how each time step's output depends on all previous time steps, allowing us to compute gradients correctly through time. Also, this is what makes RNN training **deep in time** (many time steps) rather than deep in space (many layers).

In summary, when training:

1. **Choose U**, the number of past steps to backpropagate (full sequence or truncated).
2. **Create U replicas** of the RNN cell in computation graph.
3. **Forward pass**, compute:

   $$h_{t-U}, h_{t-U+1}, \ldots, h_t$$

   With shared weights.
4. **Backward pass**, compute gradients from t down to $t - U$:

   $$\frac{\partial E}{\partial W_x}, \quad \frac{\partial E}{\partial W_h}, \quad \frac{\partial E}{\partial b}$$

5. **Combine all gradients** for shared weights W_x, W_h, and b.
6. **Update** the single shared weight matrices using the combined gradients.

4.5.2 Shared weights across time

In the previous section, we saw how to unroll an RNN over U time steps, creating U copies of the RNN cell. Now, let's discuss a crucial aspect of RNNs: **shared weights across time**. Everything in BPTT, vanishing gradients, and even LSTMs depends on this idea, because it allows RNNs to generalize across different time steps in a sequence.

What "shared across time" means

When we unroll an RNN through time:

$$\begin{aligned} x_1 &\rightarrow [\text{cell}]_1 \rightarrow h_1 \\ x_2 &\rightarrow [\text{cell}]_2 \rightarrow h_2 \\ x_3 &\rightarrow [\text{cell}]_3 \rightarrow h_3 \\ &\vdots \quad\quad \vdots \\ x_U &\rightarrow [\text{cell}]_U \rightarrow h_U \end{aligned}$$

We *draw* multiple RNN cells. But all of them must use the **same parameters**:

- Same W_x (input-to-hidden weights);
- Same W_h (hidden-to-hidden weights);
- Same b (biases);

In the diagram they appear as:

$$\begin{aligned} W_x^{(1)}, W_h^{(1)} &\rightsquigarrow [\text{cell}]_1 \\ W_x^{(2)}, W_h^{(2)} &\rightsquigarrow [\text{cell}]_2 \\ W_x^{(3)}, W_h^{(3)} &\rightsquigarrow [\text{cell}]_3 \\ \vdots \quad\quad &\quad\quad \vdots \\ W_x^{(U)}, W_h^{(U)} &\rightsquigarrow [\text{cell}]_U \end{aligned}$$

But these are **not** different matrices! They are **copies** of the same matrices:

$$W_x^{(1)} = W_x^{(2)} = W_x^{(3)} = \cdots = W_x^{(U)} = W_x$$

$$W_h^{(1)} = W_h^{(2)} = W_h^{(3)} = \cdots = W_h^{(U)} = W_h$$

Why must weights be shared?

The RNN is a *single* function repeated over time. This means that the recurrence:

$$h_t = f\left(W_x x_t + W_h h_{t-1} + b\right)$$

It is the **same rule** applied at each time step t. If we allowed different weights at different time steps, the RNN would not be able to generalize across time,

and it would require a separate set of parameters for each time step, leading to an explosion in the number of parameters. Mathematically, this would mean:

$$\text{Weights not shared} \implies W_x^{(t)}, W_h^{(t)}, b^{(t)} \text{ for each } t = 1, 2, \ldots, U$$

Where the number of parameters grows linearly with the number of time steps U. This would make training infeasible for long sequences and prevent the model from learning temporal patterns that are consistent across time.

✔ Instead, by **sharing weights**, we ensure that the **RNN allows the network to learn consistent temporal dynamics**, because the **same operation is applied again and again over time**, just like the same function is applied to different inputs in a feedforward neural network. This weight sharing is fundamental to the success of RNNs in modeling sequential data.

What does sharing really mean during training?

When we unroll for U time steps, each replica accumulates gradients for the **same parameters**. So the gradient update is:

$$\frac{\partial E}{\partial W_h} = \sum_{t=1}^{U} \frac{\partial E_t}{\partial W_h^{(t)}}$$

But all these $W_h^{(t)}$ are the **same matrix** W_h. So we:

1. Accumulate all gradients across time;
2. Average (or sum) them;
3. Apply the update *once* to the single shared matrix W_h.

In other words, during backpropagation through time, we **compute gradients at each time step**, but since the weights are shared, we **combine these gradients to update the single set of parameters**. This ensures that learning is consistent across all time steps and allows the RNN to effectively capture temporal dependencies in the data.

Example 7: Sharing weights analogy

Imagine we have a single recipe (the RNN cell) that we use to bake multiple cakes (time steps). Each time we bake a cake, we follow the same recipe (shared weights). If we find that the cakes are too sweet, we adjust the recipe once (update shared weights) based on feedback from all the cakes we've baked, rather than changing the recipe for each individual cake. This way, all future cakes benefit from the improved recipe.

⚠ Why is this critical for vanishing/exploding gradients?

As we have seen, the vanishing (page 163) and exploding (page 165) gradient problems arise during backpropagation through time due to the repeated multiplication of very small or very large gradients. But, *why is RNN weight sharing critical here?*

The key point is that **the same weights are used at each time step**, so during backpropagation, the gradients are repeatedly multiplied by the same weight matrices W_h:

$$\begin{aligned} h_t &= f(W_h \cdot h_{t-1}) \\ h_{t-1} &= f(W_h \cdot h_{t-2}) \\ h_{t-2} &= f(W_h \cdot h_{t-3}) \\ &\vdots \quad\quad \vdots \\ h_1 &= f(W_h \cdot h_0) \end{aligned}$$

And during backpropagation, the gradients uses the **Jacobian** (derivative) of the hidden state with respect to the previous hidden state:

$$\frac{\partial h_t}{\partial h_{t-1}} = W_h^T \cdot f'(\cdot)$$

So going backwards from T to 0:

$$\frac{\partial h_T}{\partial h_0} = \prod_{t=1}^{T} \frac{\partial h_t}{\partial h_{t-1}} = \prod_{t=1}^{T} \left(W_h^T \cdot f'(\cdot)\right)$$

Because the same W_h is used at each time step, the gradients are repeatedly multiplied by the same matrix for T time steps. ⚠ **And here lies the problem**: since W_h is constant, if its eigenvalues are less than 1, the gradients will shrink exponentially (vanishing gradients); if they are greater than 1, the gradients will grow exponentially (exploding gradients). So, this entire phenomenon of vanishing/exploding gradients is **directly tied to the fact that weights are shared across time** in RNNs.

Remark: Why eigenvalues matter

The behavior of repeated matrix multiplications is governed by the eigenvalues of the matrix. If the eigenvalues of W_h are:

- **Less than 1**: The gradients shrink exponentially, leading to vanishing gradients.
- **Greater than 1**: The gradients grow exponentially, leading to exploding gradients.
- **Equal to 1**: The gradients remain stable, avoiding both vanishing and exploding issues.

But the real question is: ***Why eigenvalues matter when multiplying matrices repeatedly?*** Because **eigenvalues tell us how the matrix**

stretches or shrinks vectors when repeatedly applied.

What is an eigenvalue? For a square matrix W, a vector v is an eigenvector if:

$$Wv = \lambda v$$

Meaning that v does not change direction when multiplied by W, only its magnitude is scaled by λ (the eigenvalue).

- $\lambda > 1$: vector grows (stretches)
- $\lambda < 1$: vector shrinks (compresses)
- $\lambda = 1$: vector remains the same length
- $\lambda < 0$: vector flips direction

Thus **eigenvalues describe the scaling effect of a matrix**.

4.5.3 Training Algorithm Steps

The training algorithm for Backpropagation Through Time (BPTT) can be summarized in the following steps:

1. **Forward Pass (Unroll for U Time Steps)**. Unroll the RNN across time:

$$\begin{aligned} x_1 \rightarrow & \quad [\text{cell}] \quad \rightarrow h_1 \\ x_2 \rightarrow & \quad [\text{cell}] \quad \rightarrow h_2 \\ & \vdots \qquad\quad \vdots \\ x_U \rightarrow & \quad [\text{cell}] \quad \rightarrow h \end{aligned}$$

   Compute at each time step t:

   (a) The new **hidden state** h_t using the current input x_t and the previous hidden state h_{t-1}:

$$h_t = f\left(W_x \cdot x_t + W_h \cdot h_{t-1}\right)$$

   (b) The **output** y_t based on the hidden state h_t:

$$y_t = g\left(U \cdot h_t\right)$$

   Where g is the output activation function, and U is the output weight matrix that maps hidden states to outputs.

   (c) The **loss** E_t at each time step using the predicted output y_t and the true target $\hat{y}_t$:

$$E_t = \text{loss}\left(y_t, \hat{y}_t\right)$$

   The total loss over the sequence is computed as:

$$E = \sum_{t=1}^{U} E_t$$

2. **Backward Pass (Backprop Through Time)**. Backpropagate errors through the **unrolled network**, from $t = U$ down to $t = 1$, that mathematically is expressed as follows:

$$\frac{\partial E}{\partial \theta} = \sum_{t=1}^{U} \frac{\partial E_t}{\partial \theta}$$

   Where θ represents the shared weights: W_x, W_h, and U. At each timestep t compute:

   (a) The **gradient of the loss with respect to the hidden state** h_t:

$$\frac{\partial E}{\partial h_t} = \frac{\partial E_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} + \frac{\partial E}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial h_t}$$

   Depends on both E_t and the gradient from the next time step h_{t+1}. In other words, errors are propagated backward through time.

(b) The **gradients with respect to the weights**:

$$\frac{\partial E}{\partial W_x}, \quad \frac{\partial E}{\partial W_h}, \quad \frac{\partial E}{\partial U}$$

3. **Accumulate/Average Gradients Across Time**. Because weights are **shared across time**, each timestep contributes to their gradient. Accumulate these gradients:

$$\frac{\partial E}{\partial W_x} = \sum_{t=1}^{U} \frac{\partial E_t}{\partial W_x}, \quad \frac{\partial E}{\partial W_h} = \sum_{t=1}^{U} \frac{\partial E_t}{\partial W_h}, \quad \frac{\partial E}{\partial U} = \sum_{t=1}^{U} \frac{\partial E_t}{\partial U}$$

4. **Update Shared Weights**. Use the accumulated gradients to update the weights using an optimization algorithm (e.g., Stochastic Gradient Descent, Adam):

$$\begin{aligned} W_h &\leftarrow W_h - \eta \frac{\partial E}{\partial W_h} \\ W_x &\leftarrow W_x - \eta \frac{\partial E}{\partial W_x} \\ U &\leftarrow U - \eta \frac{\partial E}{\partial U} \end{aligned}$$

Where η is the learning rate (page 182). These updated weights will be used for the next training iteration.

5. **Repeat for Multiple Epochs**. Repeat the above steps for multiple epochs over the training dataset until convergence or until a stopping criterion is met (e.g., validation loss stops improving).

The BPTT training algorithm **unrolls the RNN** in time, performs a **forward and backward pass** across all unrolled steps, **accumulates gradients** for shared weights, and **updates them using gradient descent**.

4.5.4 Vanishing and Exploding Gradients Limitation

The vanishing and exploding gradient problem is the **core difficulty** of training "vanilla" RNNs, and the reason LSTMs and GRUs were invented. We saw some anticipation on page 229, but here, we will delve deeper into the mathematical details of this phenomenon.

Why do gradients vanish or explode in RNNs?

Because during Backpropagation Through Time (BPTT) algorithm, gradients must be passed all the way back through **T steps** (the length of the sequence). This means the gradient involves repeatedly multiplying by the same matrix:

$$\frac{\partial h_t}{\partial h_{t-1}} = W_h^T \cdot f'(\cdot)$$

Where $f'(\cdot)$ is the derivative of the activation function (like tanh or σ). So across T time steps, the gradient becomes:

$$\frac{\partial h_T}{\partial h_0} \propto \left(W_h^T\right)^T$$

Where the $\propto$ symbol indicates proportionality, ignoring the activation derivatives for simplicity. In simple terms, this means we are multiplying the weight matrix W_h **T times**. How we explained earlier, **repeated multiplication by the same matrix amplifies or shrinks signals exponentially**. That's the heart of the vanishing/exploding gradient problem in RNNs.

Most common case: Vanish Gradient

If the eigenvalues of W_h are **less than 1**:

$$|\lambda_i| < 1 \quad \forall i$$

Then as we multiply W_h repeatedly, the gradients **shrink exponentially**:

$$\left(W_h^T\right)^T \to 0 \quad \text{as } T \to \infty$$

This kills the gradient exponentially fast, making it nearly impossible for the model to learn long-term dependencies:

- Early time steps receive near-zero gradients;
- Long-term dependencies are impossible to learn;
- The RNN "forgets" anything older than a few time steps (e.g., 5-10);
- Training focuses only on very recent inputs.

This is why vanilla RNNs cannot learn long sequences effectively.

Less common case: Exploding Gradient

Conversely, if the eigenvalues of W_h are **greater than 1**:

$$|\lambda_i| > 1 \quad \text{for some } i$$

Then the gradients **grow exponentially**:

$$\left(W_h^T\right)^T \to \infty \quad \text{as } T \to \infty$$

This leads to unstable training, because the gradients:

- Blow up to very large values;
- Produce insane updates to the weights;
- Destabilize training;
- Result in NaN values and divergence.

In practice, **exploding gradients cause the loss curve to suddenly shoot to infinity**, making training impossible without special techniques.

Why exploding gradients are easier to fix

A simple technique called **Gradient Clipping** can effectively mitigate exploding gradients. **Gradient Clipping** is a method where we **limit the maximum value of the gradients** during backpropagation. If the gradient norm exceeds a certain threshold, we scale it down to that threshold. This prevents the gradients from becoming too large and destabilizing training. However, **vanishing gradients are much harder to fix**, which is why architectures like LSTMs and GRUs were developed to address this fundamental limitation of vanilla RNNs.

Mathematical Intuition

To understand this mathematically, consider a simple vanilla RNN with hidden state update:

$$h_t = f\left(W_h h_{t-1} + W_x x_t + b\right)$$

We care about **how much the loss at time** t:

$$E_t = \text{loss}\left(y_t, \hat{y}_t\right)$$

Depends on a **hidden state at an earlier time** $t-k$, because this dependence (the gradient) determines whether the model can assign credit to events k steps in the past ($k < t$). If that gradient vanishes or explodes, the network will struggle to learn long-range dependencies. In other words, the ability of the RNN to learn from past information hinges on the behavior of this gradient: if it becomes too small (vanishes), the model forgets; if it becomes too large (explodes), training becomes unstable.

Backpropagation Through Time (BPTT) algorithm tells us that:

$$\frac{\partial E_t}{\partial h_k} = \frac{\partial E_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{k+1}}{\partial h_k}$$

This expression shows that to compute the gradient of the loss at time t with respect to the hidden state at time k, we need to multiply the gradients at each intermediate time step from k to t. The **gradient starts at time t because that's where the loss is computed**, and we **backpropagate through each time step until we reach time k because that's the point in the past we are interested in** (e.g., to assign credit for events that happened k steps ago).

This can be compactly written as:

$$\frac{\partial E_t}{\partial h_k} = \frac{\partial E_t}{\partial h_t} \cdot \underbrace{\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}}_{\text{problem}}$$

So the gradient is a **product of Jacobians** from step k to step t. That product is where vanishing/exploding gradients arise. Since this term causes the gradient to either shrink or grow exponentially, we need to analyze it closely.

Analyzing the Jacobian Terms. Now, we take a closer look at each **Jacobian term** $\frac{\partial h_i}{\partial h_{i-1}}$, because they determine how the gradient behaves as we multiply them together. We can get each Jacobian term from:

$$a_i = W_h h_{i-1} + W_x x_i + b \quad \Rightarrow \quad h_i = \phi(a_i)$$

Where a_i is a **vector** (e.g., 100-dimensional if hidden size is 100), and h_i is obtained by applying the activation function ϕ (like tanh) element-wise to a_i:

$$h_i = \phi\left(a_i\right) \implies \begin{aligned} h_i^{(1)} &= \phi\left(a_i^{(1)}\right) \\ h_i^{(2)} &= \phi\left(a_i^{(2)}\right) \\ &\vdots \\ h_i^{(n)} &= \phi\left(a_i^{(n)}\right) \end{aligned}$$

So **each component of the vector is activated independently**. Using the **chain rule** of calculus (page 96) to understand how h_i changes when we slightly change h_{i-1}:

$$\frac{\partial h_i}{\partial h_{i-1}} = \frac{\partial h_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial h_{i-1}} = \frac{\partial \phi\left(a_i\right)}{\partial a_i} \cdot \frac{\partial a_i}{\partial h_{i-1}}$$

Where:

- $\frac{\partial a_i}{\partial h_{i-1}} = W_h$ is the weight matrix.

- $\frac{\partial h_i}{\partial a_i} = \frac{\partial \phi(a_i)}{\partial a_i}$ is a **diagonal matrix** with activation derivatives on the diagonal:

$$\frac{\partial h_i}{\partial a_i} = \begin{pmatrix} \phi'(a_i[1]) & 0 & 0 & \dots \\ 0 & \phi'(a_i[2]) & 0 & \dots \\ 0 & 0 & \phi'(a_i[3]) & \dots \\ \vdots & & & \ddots \end{pmatrix} = D_i = \operatorname{diag}(\phi'(a_i))$$

It is naturally a diagonal matrix because ϕ is applied **element-wise**:

$$h_i = \phi(a_i) = \begin{pmatrix} \phi(a_i[1]) \\ \phi(a_i[2]) \\ \phi(a_i[3]) \\ \vdots \\ \phi(a_i[n]) \end{pmatrix}$$

Putting it together:

$$\frac{\partial h_i}{\partial h_{i-1}} = D_i \cdot W_h$$

Analyzing the Product of Jacobians. Now, we will rewrite the product of Jacobians from time k to time t:

$$\frac{\partial E_t}{\partial h_k} = \frac{\partial E_t}{\partial h_t} \cdot \underbrace{\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}}_{\text{problem}}$$

Expanding the product:

$$\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} (D_i \cdot W_h) = D_t \cdot W_h \cdot D_{t-1} \cdot W_h \cdots D_{k+1} \cdot W_h$$

This product alternates between the diagonal matrices D_i (activation derivatives) and the weight matrix W_h. To understand whether this product vanishes or explodes, we need to analyze the **norms** of these matrices. **Why norms? Because the norm gives us a measure of the "size" of a matrix**, and repeated multiplication of matrices with norms less than 1 leads to vanishing gradients, while norms greater than 1 lead to exploding gradients (as discussed earlier).

So, we do **not** care about the *exact* gradient matrix. We care about something much simpler: ***does the gradient get bigger or smaller as it flows backwards in time?*** This means we need a **single number** that measures the "magnitude" of a vector or matrix. That number is the **norm**, which is a tool that tells us: if the gradient explodes (norm $\to \infty$), vanishes (norm $\to 0$), or stays the same (norm $\approx$ constant).

The exact gradient is impossible to compute symbolically because of the **non-linear** activations and **complex** interactions. But we don't need the exact value, we need to understand *its trend*. So, instead of computing it exactly, we compute an **upper bound** on its size using norms:

$$\left\|\frac{\partial E_t}{\partial h_k}\right\| = \left\|\frac{\partial E_t}{\partial h_t} \cdot \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}\right\|$$

We can use sub-multiplicative property of norms to separate this:

$$\text{sub-multiplicative property} \implies \|AB\| \leq \|A\| \cdot \|B\|$$

$$\left\|\frac{\partial E_t}{\partial h_k}\right\| \leq \left\|\frac{\partial E_t}{\partial h_t}\right\| \cdot \left\|\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}\right\|$$

And now, we focus on the second term:

$$\left\|\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}\right\|$$

This term determines whether the gradient vanishes or explodes as we back-propagate through time. Using sub-multiplicative property again:

$$\left\|\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}\right\| \leq \prod_{i=k+1}^{t} \left\|\frac{\partial h_i}{\partial h_{i-1}}\right\|$$

Recalling that:

$$\frac{\partial h_i}{\partial h_{i-1}} = D_i \cdot W_h$$

We substitute:

$$\left\|\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}\right\| \leq \prod_{i=k+1}^{t} \|D_i \cdot W_h\|$$

Using sub-multiplicative property one last time:

$$\|D_i \cdot W_h\| \leq \|D_i\| \cdot \|W_h\|$$

So we have:

$$\left\|\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}\right\| \leq \prod_{i=k+1}^{t} \|D_i\| \cdot \|W_h\|$$

Since $\|W_h\|$ is constant across time steps, we can factor it out:

$$\left\|\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}\right\| \leq \|W_h\|^{t-k} \cdot \prod_{i=k+1}^{t} \|D_i\|$$

Now, we analyze $\|D_i\|$. Since D_i is a diagonal matrix with activation derivatives on the diagonal, its norm is determined by the maximum absolute value of those derivatives:

$$\|D_i\| = \max_j \left|\phi'\left(a_i[j]\right)\right|$$

For common activation functions like tanh and σ (sigmoid):

- For tanh, the derivative $\phi'(z) = 1 - \tanh^2(z)$ has a maximum value of 1 (at $z = 0$, page 65).
- For σ (sigmoid), the derivative $\phi'(z) = \sigma(z) \cdot (1 - \sigma(z))$ has a maximum value of 0.25 (at $z = 0$, page 61).

We define γ_v as the **upper bound on the size of the recurrent weight matrix** W_h's contribution to the gradient:

$$\gamma_v = \|W_h\|$$

And we define γ'_h as the **upper bound on the size of the activation function derivatives**:

$$\gamma'_h = \max_j |\phi'(a_i[j])| \quad \equiv \quad \gamma'_h = \sup_z |\phi'(z)|$$

Where sup denotes the supremum (least upper bound) over all possible inputs z. For tanh and σ, we have:

$$\gamma'_h = \begin{cases} 1 & \text{for tanh} \\ 0.25 & \text{for } \sigma \end{cases}$$

Thus, we can bound $\|D_i\|$ by γ'_h:

$$\|D_i\| \leq \gamma'_h$$

Using this bound, we have:

$$\prod_{i=k+1}^{t} \|D_i\| \leq \gamma_h'^{t-k}$$

Hence, **for every timestep** i (using sub-multiplicative property):

$$\|D_i \cdot W_h\| \leq \|D_i\| \cdot \|W_h\| \leq \gamma'_h \cdot \gamma_v$$

And defining:

$$\gamma = \gamma'_h \cdot \gamma_v$$

We get:

$$\|D_i \cdot W_h\| \leq \gamma$$

Putting it all together:

$$\left\| \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} \right\| \leq \gamma^{t-k}$$

Therefore:

$$\left\| \frac{\partial E_t}{\partial h_k} \right\| \leq \left\| \frac{\partial E_t}{\partial h_t} \right\| \cdot \gamma^{t-k}$$

This inequality shows that the norm of the gradient $\dfrac{\partial E_t}{\partial h_k}$ depends exponentially on the term γ^{t-k}.

- If $\gamma < 1$, then $\gamma^{t-k} \to 0$ as $t - k \to \infty$, leading to **vanishing gradients**.
- If $\gamma > 1$, then $\gamma^{t-k} \to \infty$ as $t - k \to \infty$, leading to **exploding gradients**.
- If $\gamma = 1$, the gradient norm remains constant over time steps.

4.5.5 Dealing with Gradient Problems

We have seen and demonstrated how RNNs are prone to the vanishing and exploding gradient problems. To mitigate these issues, several techniques can be employed. Here, we introduce **three strategies** commonly used to address these gradient problems in RNNs:

- **(Simplest) Using ReLu activation functions**. ReLu (Rectified Linear Unit) activation functions (page 166) help mitigate the vanishing gradient problem due to their linear, non-saturating nature for positive inputs. Their derivative is:

$$\phi'(z) = 1 \quad \text{for } z > 0$$

  This means the activation derivative does **not shrink** the gradient, unlike sigmoid (max 0.25) or tanh (max 1) functions, which can lead to vanishing gradients over time steps. So with ReLu, the per-step Jacobian looks like:

$$\frac{\partial h_t}{\partial h_{t-1}} = D_t W_h \quad \text{where } D_t = \text{diag}(\phi'(z_t)) \text{ with } \phi'(z_t) \in \{0, 1\}$$

  This greatly reduces the risk of vanishing gradients, as the derivative is either 0 or 1, preventing exponential decay of gradients over time steps.

  - ✔ Suffer **less** from vanishing gradients.
  - ✘ Still can experience **exploding gradients**, as the weight matrix W_h can still lead to large values.
  - ✘ Still **struggle with long-term dependencies**, as ReLU does not inherently solve this issue.
  - ✔ **Train better than sigmoid/tanh** RNNs on many tasks.

  ReLU alone **cannot eliminate vanishing gradient** entirely, because the recurrent matrix W_h is still repeatedly multiplied. The LSTM architecture goes further with **gate-controlled linear gradients**.

- **Gradient Clipping**: limiting gradient magnitude (exploding vs. vanishing). When gradients exceed a certain threshold during backpropagation, they are **scaled down** to prevent instability. This is typically done by:

$$g \leftarrow \frac{g}{\|g\|} \cdot \tau \quad \text{if } \|g\| > \tau$$

  Where g is the gradient vector and τ is the threshold.

  - ✔ **Prevents exploding gradients**, stabilizing training.
  - ✘ **Does not solve vanishing gradients**, so long-term dependencies may still be difficult to learn.

- **Designing modules to retain long-term dependencies**. The core idea is to create an architecture that:
  - **Preserves** gradients over long ranges of time steps.
  - **Does not** repeatedly shrink them.
  - Behaves like an **information highway**.

– Propagates gradients without decay.

This is the entire reason LSTMs were invented. The LSTM (Long Short-Term Memory) architecture introduces **gates** that control the flow of information and gradients, allowing the network to maintain long-term dependencies effectively:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

And crucially:

$$\frac{\partial c_t}{\partial c_{t-1}} = f_t$$

Where f_t is the **forget gate**, which can be learned to be close to 1, allowing gradients to pass through many time steps without vanishing.

- ✔ **Effectively mitigates vanishing gradients**, enabling learning of long-term dependencies.
- ✔ **Also helps with exploding gradients**, as the gating mechanisms can regulate information flow.
- ✔ **Widely used** in practice for sequence modeling tasks.
- ✖ More complex and computationally intensive than standard RNNs.

In summary, while ReLU activation functions and gradient clipping can help mitigate gradient problems to some extent, the LSTM architecture provides a more robust solution for handling long-term dependencies in RNNs by effectively addressing both vanishing and exploding gradients through its gating mechanisms.

4.6 Long Short-Term Memory Network (LSTM)

Definition 7: Long Short-Term Memory Network (LSTM)

A **Long Short-Term Memory Network (LSTM)** is a special type of Recurrent Neural Network (RNN) architecture designed to **learn long-term dependencies** in sequential data by using an internal memory cell and a system of gates that control how information is stored, forgotten, and exposed.

It introduces a **memory cell state** C_t, a **hidden state** h_t, and three multiplicative **gates** (input, forget, output). These gates allow the network to:

- **Retain information for long periods**.
- **Protect memory from vanishing gradients**.
- **Selectively update and reset its internal memory** based on the input sequence.

The key feature of an LSTM is a **linear, self-recurrent connection** in the cell state (the Constant Error Carousel, CEC), which preserves information without being destroyed by nonlinear activations, allowing gradients to flow across many timesteps.

4.6.1 Architecture

Problem. Before LSTMs were introduced, (Hochreiter & Schmidhuber, 1997), training Recurrent Neural Networks (RNNs) was extremely difficult. The core issue was **not** model capacity (RNNs are Turing complete, Siegelmann & Sontag, 1995), but rather the **inability to learn long-term dependencies due to vanishing and exploding gradients** (page 239).

Solution. LSTMs were designed specifically to address this issue. The key innovation of LSTMs is the introduction of **memory cells** that can maintain information over long periods of time. Each memory cell contains three main components: the **input gate**, the **forget gate**, and the **output gate**. These gates regulate the flow of information into, out of, and within the memory cell.

How it works. Without going into the gate details, which will be covered in the next sections, the **solution to overcoming vanishing gradients lies in the cell state**, called **memory cell** C_t. A **Memory Cell** is a special **kind of RNN unit** that **maintains its state over time**, allowing it to store information for long periods. The cell state is updated at each time step t based on the input data and the previous cell state, but crucially, it can also **retain information without being overwritten**. This is achieved through the use of **gates** that control the flow of information into and out of the cell state. By carefully regulating these gates, LSTMs can **preserve gradients during backpropagation**, allowing them to learn long-term dependencies effectively.

Mathematically, the memory cell C_t is a linear self-loop (i.e., it can maintain its value over time)

$$C_t = C_{t-1} + (\text{some new information})$$

Or more generally:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{108}$$

- f_t is the **forget gate**, which determines how much of the previous cell state C_{t-1} to retain.
- i_t is the **input gate**, which controls how much new information $\tilde{C}_t$ to add to the cell state.
- $\tilde{C}_t$ is the **candidate cell state**, which is computed based on the current input and the previous hidden state.
- $\odot$ denotes **element-wise multiplication**.

The crucial part is the connection from C_{t-1} to C_t **can preserve information without squashing it through a non-linear activation function**, thanks to **Constant Error Carousel (CEC)**.

Insight. The **Constant Error Carousel (CEC)** within the LSTM memory cell is the key mechanism that allows LSTMs to overcome the vanishing gradient problem. Created by Sepp Hochreiter in 1997, the CEC is designed to maintain a constant error signal as it propagates through time, enabling the network to learn long-term dependencies effectively. It achieves this by allowing the cell state to **carry information across many time steps without being altered by non-linear activation functions**, which are typically responsible for diminishing gradients in standard RNNs. It does so through a **linear self-loop** in the cell state update equation, which allows the gradient to be preserved during backpropagation. This means that the error signal can flow back through many time steps without vanishing, enabling the LSTM to learn from long-term dependencies in the data.

Architecture Comparison: RNN vs LSTM

A classic RNN cell handles input x_t and previous hidden state h_{t-1} to produce the new hidden state h_t:

$$h_t = \phi\left(W_x \cdot x_t + W_h \cdot h_{t-1} + b\right) \tag{109}$$

where ϕ is a non-linear activation function (e.g., tanh or ReLU). Here, only the hidden state h_t is maintained over time, and the non-linear activation can lead to vanishing gradients. Every update compresses information through ϕ and there is **no mechanism to protect memory**. As a result, both information and gradients **degrade** over time, and the network suffers from **vanishing gradients**.

The LSTM cell takes the same inputs as an RNN cell x_t and h_{t-1}, but **crucially** also maintains:

- A **cell state** C_t that carries long-term information.

- Three **gates** (input, forget, output) that regulate information flow.

Thus, the **LSTM outputs** both:

- The **cell state** C_t, which can carry information across many time steps without being squashed by non-linearities:

$$C_{t-1} \xrightarrow{\text{linear, gated}} C_t$$

  It indicates what information to **remember** or **forget** over time. It is a long-term memory and acts like a **protected memory highway** (Constant Error Carousel) for gradients.

- The **hidden state** h_t, which is similar to the classical RNN, but modulated by the output gate:

$$h_{t-1} \xrightarrow{\text{gated + nonlinearity}} h_t$$

  It indicates what information to **output** at the current time step. It is a short-term representation.

The gates in the LSTM cell **control the flow of information** into and out of both the cell state and hidden state. By regulating what to remember, forget, and output, the LSTM can effectively manage long-term dependencies without suffering from vanishing gradients. The **cell state** C_t **provides a direct path for gradients to flow back through time**, preserving information over long sequences, while the hidden state h_t allows for dynamic representation of the current input. This architecture enables LSTMs to learn complex temporal patterns that standard RNNs struggle with.

4.6.2 Gates

Unlike classical RNN that only updates one hidden state h_t, an LSTM maintains **two internal states**: the **cell state** (long-term memory) C_t and the **hidden state** (short-term output) h_t. The cell state C_t flows horizontally through time with minimal interference. This is the *memory highway* or **constant error carousel (CEC)**. But the LSTM must decide:

- What new information to write.
- What old information to forget.
- What part of memory to expose.

To do this, it uses four components called **gates**. Each gate produces a vector of values between 0 and 1 (sigmoid), controlling how information flows through the network. The interaction of these gates creates an extremely flexible state machine inside the LSTM cell. The three gates and one candidate memory are defined as follows:

$$h_t = o_t \odot \tanh(C_t) \quad \text{where} \quad C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

1. **Forget gate** f_t: Decides what information to discard from the cell state. "***Which information stored in the cell state is no longer relevant?***"

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{110}$$

   - W_f: Weight matrix for the forget gate.
   - b_f: Bias vector for the forget gate.
   - σ: Sigmoid activation function.
   - $[h_{t-1}, x_t]$: Concatenation of previous hidden state and current input.

   This gate decides **how much of the previous cell state C_{t-1} should be kept**. It performs **selective deletion** of information from the long-term memory.

   ✔ $f_t \approx 1 \rightarrow$ Keep the information.

   ✖ $f_t \approx 0 \rightarrow$ Forget the information.

   ⚖ $f_t \approx 0.5 \rightarrow$ Partially forget the information.

   It is **needed** because **old context** sometimes becomes **irrelevant**. For example, in language modeling, once we move past a topic, we may want to forget its details: "*The capital of France is Paris.* ***By the way****, the capital of Germany is Berlin*". Here, the information about France can be forgotten when discussing Germany. It is needed to **prevent memory saturation** and **allow adaptation to new contexts**.

   ✔ **Vanishing Gradient.** The forget gate is deeply connected to the LSTM's trick for fighting the vanishing gradient problem. We define the

gradient flow through the cell state as:

$$\begin{aligned}\frac{\partial C_t}{\partial C_{t-1}} &= \frac{\partial}{\partial C_{t-1}}\left(f_t \odot C_{t-1} + i_t \odot \tilde{C}_t\right) \\ &= f_t \odot \frac{\partial C_{t-1}}{\partial C_{t-1}} + \frac{\partial (i_t \odot \tilde{C}_t)}{\partial C_{t-1}} \\ &= f_t + 0 \\ &= f_t\end{aligned}$$

This means that the gradient of the cell state at time t with respect to the previous cell state at time $t-1$ is exactly the forget gate f_t. If f_t is **close to 1**, the **gradient flows through unchanged**, preventing vanishing gradients. If f_t is **close to 0**, the **gradient is blocked**, allowing the network to forget irrelevant information. Thus, the forget gate **actively controls the gradient flows**, enabling the LSTM to maintain long-term dependencies while still being able to adapt and forget when necessary.

2. **Input gate** i_t: Decides what new information to add to the cell state. "***How much of the candidate memory*** $\tilde{C}_t$ ***should we store in the cell state?***"

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \tag{111}$$

Controls **how much the candidate memory will influence the cell state**. It protects the memory from random updates.

- ✗ If $i_t = 0 \rightarrow$ No new information is added, ignoring $\tilde{C}_t$.
- ✓ If $i_t = 1 \rightarrow$ All new information is added, fully adopting $\tilde{C}_t$.

This gate allows the model to be selective and **only incorporate relevant new information** into the cell state. Without it, the cell would overwrite its memory at every timestep, which would be catastrophic for long-term dependencies. The input gate is therefore a **filter** that protects memory from being updated too easily.

3. **Candidate cell state** $\tilde{C}_t$: Creates new candidate values that could be added to the cell state. It is extremely important because it determines **what new information is *proposed* to enter the long-term memory**.

$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{112}$$

This component generates **potential new information** to be added to the cell state, scaled by the input gate i_t. It is a *vector of new information* generated from the current input x_t and the previous short-term state h_{t-1}.

- It uses **tanh** to generate values between -1 and 1, **allowing both positive and negative updates** to the cell state. It also compresses information to a stable range and avoids uncontrolled growth.
- It is **modulated** later **by the input gate** i_t to determine how much of this candidate information should actually be written to the cell state.

We can think of $\tilde{C}_t$ as the **proposed update** to the cell state (long-term memory), which is then filtered by the input gate to ensure only relevant information is added. It works in tandem with the input gate:

- The candidate generates **what** to write.
- The input gate decides **how much** of that to actually write.

4. **Output gate** o_t: Decides what part of the cell state to output as the hidden state. "***Which part of this memory should we show as our output hidden state*** h_t***?***"

$$o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \tag{113}$$

This gate controls **what portion of the internal memory becomes visible in the hidden state** h_t.

✗ If $o_t = 0 \rightarrow$ No information from the cell state is exposed.

✓ If $o_t = 1 \rightarrow$ All information from the cell state is exposed.

This is important because not all information in the cell state is relevant for the current output. The output gate allows the LSTM to **selectively reveal** information based on the current context.

The **cell state update** C_t combines the effects of the forget and input gates:

$$C_t = \underbrace{f_t \odot C_{t-1}}_{\text{Forget old info}} + \underbrace{i_t \odot \tilde{C}_t}_{\text{Add new info}}$$

Here, the previous cell state C_{t-1} is scaled by the forget gate f_t, determining what to retain, while the candidate cell state $\tilde{C}_t$ is scaled by the input gate i_t, determining what new information to add. Finally, the **hidden state** h_t is computed by applying the output gate o_t to the activated cell state:

$$h_t = o_t \odot \tanh(C_t)$$

It is the **actual output of the LSTM cell** at time t, influenced by both the current cell state and the output gate.

- $\tanh(C_t)$ transforms the cell state into an output-friendly representation. This operator ensures bounded values (between -1 and 1), stability, good gradient behavior, and non-linearity.
- The output gate o_t filters that representation before exposing it. Allows the LSTM to **protect memory** while revealing only what is relevant

4.6.3 Lightweight Alternative: Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) were proposed by Cho et al. (2014) as a **simplified, faster, and lighter** recurrent architecture that retains much of the power of LSTMs but with fewer parameters and a simpler design.

Why GRU?

Long Short-Term Memory (LSTM) networks are powerful but can be computationally intensive due to their complex architecture involving multiple gates and memory cells:

- ✗ **Computationally heavy**, because of multiple matrix multiplications per time step.
- ✗ **Contain 3 gates + memory cell**, leading to more parameters to train.
- ✗ **Have many parameters (4 matrix multiplications per time step)**, which can lead to overfitting on smaller datasets.

GRUs address these issues by simplifying the architecture while still effectively capturing long-term dependencies in sequential data. Its goal is to:

- ✓ **Reduce computational complexity** by using fewer gates and parameters.
- ✓ **Maintain performance** comparable to LSTMs on many tasks.
- ✓ **Facilitate faster training** and inference times.
- ✓ **Simplify implementation** and tuning due to fewer hyperparameters.

They do this by merging some gates and eliminating the explicit memory cell found in LSTMs.

GRU Architecture

GRUs **do not have a separate cell state** C_t like LSTMs. Instead, they only maintain a **hidden state** h_t, which merges: short-term memory and long-term memory. GRUs use **two gates** to control the flow of information:

1. **Update Gate** z_t. Controls **how much of the past is kept** and **how much of the new information is added**.

$$\begin{bmatrix} i_t \\ f_t \end{bmatrix} \xrightarrow{\text{GRU}} \quad z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \tag{114}$$

   It **replaces both the input and forget gates** of the LSTM.

   - $z_t = 1$: use **only** new information, forget everything from the past (like forget gate closed, 0, and input gate open, 1).
   - $z_t = 0$: keep **only** past information, ignore new input (like forget gate open, 1, and input gate closed, 0).
   - $0 < z_t < 1$: balance between past and new information.

2. **Reset Gate** r_t. Controls **how much of the previous hidden state to consider when creating new candidate** information. This is similar to the input gate in LSTMs, but it directly influences the candidate hidden state. Similarly, it is computed as:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \tag{115}$$

   - r_t close to 0: ignore past hidden state, focus on new input.
   - r_t close to 1: consider past hidden state fully, influence of h_{t-1} is strong.

The number of gates is reduced from 3 in LSTMs to 2 in GRUs, **simplifying dramatically the architecture**. Also, GRUs do not have a separate memory cell; instead, they **maintain a single hidden state that combines both short-term and long-term memory** (h_t). This further reduces the number of parameters and computations required.

The **final hidden state** h_t in a GRU is computed as:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{116}$$

It interpolates between the previous hidden state, h_{t-1}, and the candidate hidden state, $\tilde{h}_t$, weighted by the update gate, z_t. Thus, the **update gate** decides how much past information to retain and how much new information to incorporate. Notably, it **is the only gate that directly influences the final hidden state**. Instead, the **reset gate** r_t **operates when computing the candidate hidden state** $\tilde{h}_t$:

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h) \tag{117}$$

Feature	**LSTM**	**GRU**
Gates	3 (input, forget, output)	2 (update, reset)
Memory Cell	Yes (C_t)	No (only hidden state h_t)
Hidden State	Separate from cell state	Combines short-term and long-term memory
Complexity	Higher	Lower
Parameters	More (4 matrix multiplications per time step)	Fewer (3 matrix multiplications per time step)
Training Speed	Slower	Faster
Performance	Strong on long sequences	Usually comparable to an LSTM, sometimes better on smaller datasets

Table 4: Comparison between LSTM and GRU architectures.

Final thoughts: Is GRU always better than LSTM?

In short, **no**, GRU is not the best RNN. In some scenarios, LSTMs may outperform GRUs. They **do not have the same expressive power** of LSTMs.

- **LSTMs Separate Long-Term and Short-Term Memory**. LSTM has two dedicated components (**cell state** C_t and **hidden state** h_t) to manage long-term and short-term dependencies separately. Instead, GRU merges them into a single vector. The separation in LSTMs allows to:
  - Preserve long-term memory in C_t **without exposing it**.
  - Use h_t for short-term, noisy, rapidly-changing representations.
  - Control what gets exposed to the next layer (via the output gate).

  This separation is **very powerful**, especially for tasks requiring deep context management, such as language modeling or machine translation.
- **LSTMs Have More Control (More Degrees of Freedom)**. LSTM has **three gates** (forget, input, output), while GRU has **two**. This gives LSTMs: more expressiveness, finer control of memory flow, and the ability to decouple "writing to memory" from "exposing memory". In some tasks, this extra structure provides: better accuracy, more stable long-term learning, and better handling of irregular or sparse signals. In summary, **with LSTM we can do a more nuanced manipulation of information**.
- **LSTMs Work Better on Very Long Sequences**. Empirically, on tasks requiring **very long-term dependencies** (hundreds or thousands of time steps), **LSTMs typically outperform GRUs**. Some examples:
  - Long text generation.
  - Speech sequences longer than several seconds.
  - Language models (pre-Transformer era).
  - Medical time series with sparse temporal relations.
  - Irregular biomechanical signals.

  Because the cell state in an LSTM is specifically designed for long-term memory, LSTMs **tend to retain information better**.

  GRUs can forget too aggressively because the update gate merges the input and forget gates. This coupling makes GRUs less flexible in controlling what to remember for long periods.
- **GRUs Can Underperform When Selective Visibility Matters**. LSTMs have an **output gate**:

$$h_t = o_t \odot \tanh(C_t)$$

  GRUs **do not**. This means **GRUs cannot selectively hide memory and it is critical in tasks where certain features need to stay internal and not be exposed to the next layer**.

4.6.4 Networks

So far, we have described the internal structure of **one LSTM cell**. But real sequence tasks require using this cell **repeatedly, one timestep at a time**. Thus, an **LSTM network** is simply **a chain of identical LSTM cells, each processing one element of the sequence, sharing all parameters**. With "identical", we mean that all cells have the same weights, same biases, same gates, and same equations. Only the **inputs** x_t and **internal states** (h_t, C_t) differ from cell to cell, depending on the timestep t.

The LSTM **unrolled across timesteps**:

- $x_1, x_2, \ldots, x_T$ are the inputs at each timestep;
- At each timestep t, the LSTM cell receives **current input** x_t and **previous states** (h_{t-1}, C_{t-1});
- It computes **new hidden state** h_t, **new cell state** C_t using the same equations, weights, and biases at each timestep, and produces **output** y_t (if relevant).

The key idea is **all cells share parameters, but each has its own internal states as time progresses**. This allows the network to **learn temporal patterns** in the data, as information can flow through the cell states across many timesteps, while the shared parameters ensure consistency in how information is processed at each step.

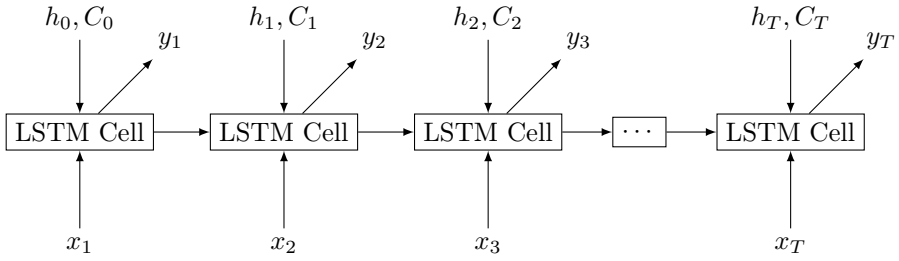


Figure 30: **LSTM network** (many-to-many architecture) **unrolled across timesteps**. Each LSTM cell processes one input at a time, sharing parameters but maintaining its own internal states. Outputs can be produced at each timestep if required.

What does the LSTM actually learn over time?

As it processes a sequence, the LSTM network **learns to update its cell state** C_t and **hidden state** h_t in a way that captures relevant information from the inputs over time. The LSTM network:

- ✔ **Accumulates information in the cell state**. This allows it to remember important features of the input sequence over long periods.
- ✔ **Selective memory updates**. Updates memory only when needed. The gates control when to forget old information and when to add new information, enabling the network to focus on relevant parts of the sequence.

- ✔ **Outputs predictions for each step**. The hidden state h_t can be used to make predictions at each timestep, allowing the network to generate outputs based on the learned temporal patterns.
- ✔ **Adapts context dynamically**. The LSTM can adjust its memory based on the context of the sequence, making it effective for tasks like language modeling, time series prediction, and more.

These capabilities allow LSTM networks to understand temporal relationships (e.g., sequences in text, speech, or time series data), dependencies across distant timesteps (e.g., long-term dependencies in sentences), evolving contexts (e.g., changing topics in a conversation), and patterns different lengths (e.g., variable-length sequences in natural language).

❓ How LSTMs are used in real models

LSTMs can be arranged in different **architectural patterns** depending on the task. Each pattern corresponds to how inputs and outputs are handled across time. These patterns apply not only to LSTMs, but also to GRUs, Transformers, and any sequential model. However, LSTMs are particularly well-suited for these patterns due to their ability to manage long-term dependencies. The main patterns are:

1. **Many-to-Many (Sequence-to-Sequence Output)**. **Output produced at *each* timestep**. This is the architecture shown on Figure 30; each LSTM cell processes a timestep and emits an output:

$$(x_1 \to h_1 \to y_1), \quad (x_2 \to h_2 \to y_2), \quad \ldots, \quad (x_T \to h_T \to y_T)$$

   #### ❓ When is this used?

   - **Part-of-Speech tagging**: Assigning a part of speech to each word in a sentence.
   - **Named Entity Recognition (NER)**: Identifying entities (like names, locations) in text.
   - **Frame-by-frame video classification**: Classifying each frame in a video sequence.
   - **Time-series forecasting**: Predicting future values at each time step.
   - **Speech recognition**: Transcribing spoken words into text in real-time.

   In this model, the input sequence and output sequence are of the same length, with each input element corresponding to an output element. For example, if the input is "John likes cats", the output could be "Noun Verb Noun" for part-of-speech tagging.

2. **Many-to-One (Sequence Classification)**. **Use the *final hidden state* h_T to classify the entire sequence**. The architecture looks like this:

$$h_T = \text{LSTM}(x_1, x_2, \ldots, x_T)$$

$$y = \text{softmax}\left(Wh_T + b\right)$$

When is this used?

- **Sentiment analysis**: Classifying the sentiment of a movie review (positive/negative).
- **Sequence classification**: Classifying DNA sequences or protein sequences.
- **Spam detection**: Classifying emails or messages as spam or not spam.
- **Audio classification**: Classifying audio clips into categories (e.g., music genre).

Here, the entire input sequence is summarized into a single vector (the final hidden state), which is then used for classification. For example, given a movie review, the model predicts whether the sentiment is positive or negative based on the entire text: "The movie was fantastic!" $\rightarrow$ "Positive". So, the **model uses *all* the timesteps** but **only the *final* state matter for classification**.

3. **One-to-Many (Sequence Generation / Decoder)**. **Start from a single input and generate a sequence**. For example:

$$h_0 = f(x) \quad \text{(embedding of a single input)}$$

$$y_1, y_2, \ldots, y_T = \text{LSTM}\left(h_0\right)$$

This is typically autoregressive:

- At step 1, input h_0 produces output y_1;
- At step 2, input is y_1 (or its embedding), producing y_2;
- This continues until the full sequence is generated.

When is this used?

- **Text generation**: Generating sentences or paragraphs from a prompt.
- **Image captioning**: Generating a descriptive caption for an image.
- **Music generation**: Composing music sequences from a seed note or chord.
- **Auto-regressive generation**: Generating sequences in tasks like language modeling.

Here, the **model starts with a single input** (like a prompt or seed, a vector) and **generates a sequence of outputs step by step**. For example, given the prompt "Once upon a time", the model might generate a full story.

4. **Encoder-Decoder (Seq2Seq) Architecture**. **Use one LSTM (encoder) to process the input sequence into a context vector, then another LSTM (decoder) to generate the output sequence from that vector**. This pattern is still used in modern Transformers. It is a **two-part LSTM system**:

(a) **Encoder**: Processes the input sequence $(x_1, x_2, \ldots, x_T)$ and produces a context vector (final hidden state) h_T:

$$h_T = \text{LSTM}_{\text{enc}}(x_1, x_2, \ldots, x_T)$$

(b) **Decoder**: Takes the context vector h_T and generates the output sequence $(y_1, y_2, \ldots, y_K)$:

$$y_1, y_2, \ldots, y_K = \text{LSTM}_{\text{dec}}(h_T)$$

When is this used?

- **Machine translation**: Translating sentences from one language to another (e.g., English to French).
- **Dialogue systems**: Generating responses in chatbots.
- **Summarization**: Creating concise summaries of longer texts.
- **Speech-to-text**: Converting spoken language into written text.
- **Question answering**: Generating answers based on a given context or passage.
- **Autoencoders for sequences**: Learning compressed representations of sequences.

Here, the **input and output sequences can have different lengths**. For example, translating "I love programming" (3 words) to "J'adore la programmation" (4 words). The encoder captures the meaning of the input sequence, and the decoder generates the corresponding output sequence.

5. **Many-to-Many with Different Lengths (e.g., CTC, translation)**. **Input and output sequences have different lengths, with outputs produced at each timestep**. This is common in tasks like speech recognition or translation where the number of input timesteps does not match the number of output timesteps. The architecture looks like this:

$$y_1, y_2, \ldots, y_K = \text{LSTM}(x_1, x_2, \ldots, x_T)$$

When is this used?

- **Speech recognition**: Converting audio signals into text, where the number of audio frames may differ from the number of words.
- **Machine translation**: Translating sentences where the source and target languages have different word counts.
- **Handwriting recognition**: Recognizing handwritten text from images, where the number of strokes may differ from the number of characters.

Here, the model processes the entire input sequence and produces an output sequence that may be of different length. Techniques like Connectionist Temporal Classification (CTC) are often used to align the input and output sequences during training.

LSTMs are flexible enough to be used in almost every sequential modeling configuration. Their structure allows handling variable-length sequences, producing or consuming sequences, compressing long contexts, and dynamically generating outputs based on learned temporal patterns.

4.6.5 Multi-layer LSTM

Up until now, we have analyzed **single-layer LSTMs**, in which one LSTM cell processes the input sequence at each timestep. Figure 30 illustrates a many-to-many architecture with outputs at each timestep. This is an example of a single-layer LSTM network in which all LSTM cells share the same parameters across timesteps, but only one LSTM cell processes the input at each timestep.

However, modern deep learning models often use **multiple stacked recurrent layers**, forming a **multi-layer LSTM** architecture (or **deep LSTM**).

Why go deeper?

Just like in feedforward networks and CNNs, **depth increases representational capacity**.

✗ In a *single-layer LSTM*, th cell sees only the raw sequence input at each timestep. It must learn both low-level and high-level temporal patterns from this input. This makes **learning complex patterns more difficult**.

✓ In a *multi-layer LSTM*, lower layers can learn **low-level temporal features** from the raw input, while higher layers can focus on **high-level temporal abstractions**. So, each layer builds on top of the previous one:

- **Layer 1** learns low-level temporal features from the raw input sequence (e.g., edges in video frames, or phonemes in audio).
- **Layer 2** learns mid-level temporal patterns from Layer 1's features (e.g., shapes in video frames, or syllables in audio).
- **Layer 3** learns high-level temporal abstractions from Layer 2's patterns (e.g., objects in video frames, or words in audio).

Stacking layers **hierarchically organizes information**, just like depth does in CNNs.

How Multi-layer LSTMs work

In a deep LSTM, we have **multiple LSTM networks stacked one above the other**. For each timestep t:

- **Layer 1** receives the raw input $\mathbf{x}^{(t)}$ and processes it through its LSTM cell, producing hidden state $\mathbf{h}_1^{(t)}$ and cell state $\mathbf{c}_1^{(t)}$.
- **Layer 2** receives Layer 1's hidden state $\mathbf{h}_1^{(t)}$ as input, and processes it through its LSTM cell, producing hidden state $\mathbf{h}_2^{(t)}$ and cell state $\mathbf{c}_2^{(t)}$.
- **Layer 3** receives Layer 2's hidden state $\mathbf{h}_2^{(t)}$ as input, and processes it through its LSTM cell, producing hidden state $\mathbf{h}_3^{(t)}$ and cell state $\mathbf{c}_3^{(t)}$.
- This continues for however many layers the network has.

Each layer maintains its own recurrent connections over time, passing its hidden and cell states to the next timestep. The topmost layer produces the final output at each timestep. This architecture allows each layer to learn different levels of temporal features from the input sequence (**different abstraction levels**).

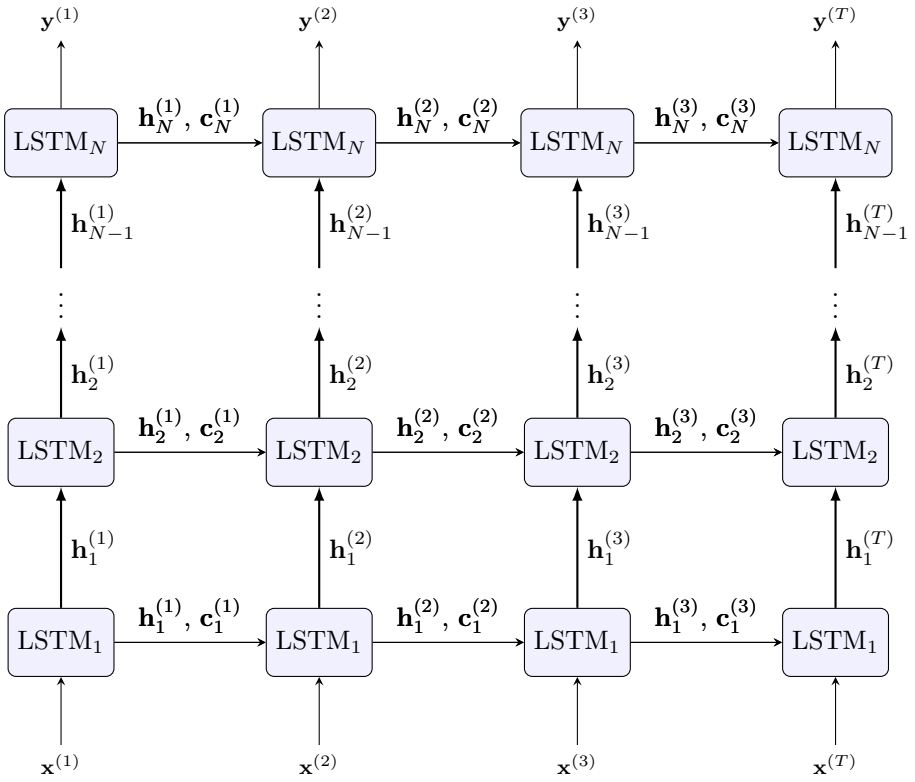


Figure 31: **Multi-layer LSTM network architecture unrolled over time**. Each layer processes the hidden state from the layer below at the same timestep, while maintaining its own recurrent connections over time. The bottom layer receives the raw input sequence, and the top layer produces the output sequence. Each layer has its own LSTM cell with separate parameters.

Formal Equations

We can derive the equations for a multi-layer LSTM by extending the single-layer LSTM equations. For a multi-layer LSTM with N layers, at each timestep t, the computations for layer ℓ ($1 \leq \ell \leq N$) are as follows:

- **Cell update** for layer ℓ:

$$C_t^{(\ell)} = f_t^{(\ell)} \odot C_{t-1}^{(\ell)} + i_t^{(\ell)} \odot \tilde{C}_t^{(\ell)}$$

- **Hidden state update** for layer ℓ:

$$h_t^{(\ell)} = o_t^{(\ell)} \odot \tanh(C_t^{(\ell)})$$

- **Input to layer** ℓ:
  - For layer 1 ($\ell = 1$):

$$h_t^{(0)} = x_t \quad \text{(raw input at timestep } t\text{)}$$

  - For layers $\ell = 2, \ldots, N$:

$$h_t^{(\ell-1)} \quad \text{(hidden state from previous layer at timestep } t\text{)}$$

- **Output from layer** N (topmost layer):

$$y_t = h_t^{(N)} \quad \text{(output at timestep } t\text{)}$$

Weights **are not shared** between layers but are shared across timesteps within each layer. This allows each layer to learn different temporal features at varying levels of abstraction.

⚠ Challenges of Deep LSTMs

While multi-layer LSTMs offer increased representational capacity, they also introduce challenges:

- ✗ **Vanishing/Exploding Gradients across layers**: Although LSTMs mitigate vanishing gradients over time, stacking many layers can still lead to gradient issues during backpropagation through layers.
- ✗ **Heavy Computational Load**: More layers mean more parameters and computations, leading to longer training times and higher resource consumption.
- ✗ **Overfitting Risk**: Deeper models are more prone to overfitting, especially with limited data.
- ✗ **Training Instability**: Deeper LSTMs can be harder to train effectively, requiring careful tuning of hyperparameters and initialization.

To address these challenges, techniques such as **residual connections**, **layer normalization**, and **dropout** are often employed in deep LSTM architectures.

4.6.6 Bidirectional LSTM (BiLSTM) Networks

A standard (unidirectional) LSTM processes a sequence **from left to right**, meaning that at each time step t, the LSTM has access only to the information from the previous time steps $1, 2, \ldots, t-1$:

$$x_1 \to x_2 \to x_3 \to \ldots \to x_{T-1} \to x_T$$

This means the hidden state at time t is based only on: the past inputs $(x_1, \ldots, x_{t-1})$ and the current input (x_t). However, **in many tasks we *also* need to consider future context**, i.e., information from the subsequent time steps $t+1, t+2, \ldots, T$. This is where Bidirectional LSTM (BiLSTM) networks come into play.

Definition 8: Bidirectional LSTM (BiLSTM)

A **Bidirectional LSTM (BiLSTM)** is an **extension** of the traditional LSTM that can **capture information from both past and future contexts by processing the input sequence in both directions**. It consists of two LSTM layers:

- A *forward* LSTM that processes the input sequence from left to right (from x_1 to x_T).
- A *backward* LSTM that processes the input sequence from right to left (from x_T to x_1).

At each time step t, the outputs from both LSTMs are concatenated (or combined in some other way) to form the final output for that time step. This allows the network to have access to both past and future context when making predictions.

Example 8: BiLSTM Application

For example, in natural language processing tasks: "*I arrived at the bank to **deposit** money*". The word "*bank*" is ambiguous, because we need the *future* ("*to deposit money*") to understand that it refers to a financial institution rather than the side of a river. A BiLSTM can effectively capture this context by considering both preceding and succeeding words in the sentence.

Core Idea

The core idea behind BiLSTMs is to **leverage two separate LSTM networks to process the input sequence in both directions**, allowing the model to capture a more comprehensive understanding of the data by considering both past and future contexts simultaneously. It consists of two LSTM layers:

- **Forward LSTM**: Processes the input sequence from left to right, capturing information from past time steps.

$$\overrightarrow{h}_t = \text{LSTM}_{\text{forward}}(x_t, \overrightarrow{h}_{t-1})$$

  Here, $\overrightarrow{h}_t$ represents the hidden state at time step t from the forward LSTM, and $\overrightarrow{h}_{t-1}$ is the hidden state from the previous time step. This step learns representations from **past context**.

- **Backward LSTM**: Processes the input sequence from right to left, capturing information from future time steps.

$$\overleftarrow{h}_t = \text{LSTM}_{\text{backward}}(x_t, \overleftarrow{h}_{t+1})$$

  Here, $\overleftarrow{h}_t$ represents the hidden state at time step t from the backward LSTM, and $\overleftarrow{h}_{t+1}$ is the hidden state from the next time step. This step learns representations from **future context**.

- **Combining Outputs**: At each time step t, the outputs from both the forward and backward LSTMs are combined (e.g., concatenated) to form the final output representation.

$$h_t = [\overrightarrow{h}_t; \overleftarrow{h}_t]$$

  Here, h_t is the combined hidden state at time step t, which incorporates information from both past and future contexts.

⚠ When BiLSTM cannot be used

Bidirectional models **cannot be used when future inputs are unknown at the time of prediction** (inference time):

- **Real-time applications**: In scenarios where predictions must be made in real-time (e.g., live speech recognition, online translation), future data points are not available, making it impossible to utilize BiLSTMs effectively.
- **Online filtering**: In applications like online filtering or streaming data analysis, where data arrives sequentially and predictions must be made on-the-fly, BiLSTMs cannot be employed since they require access to future inputs.
- **Streaming sensor systems**: In systems that process data from sensors in real-time (e.g., IoT devices, autonomous vehicles), future sensor readings are not accessible at the moment of prediction, rendering BiLSTMs unsuitable.
- **Autoregressive generation**: In tasks such as text generation or time series forecasting, where each output depends on previously generated outputs, future inputs are inherently unknown during the generation process, preventing the use of BiLSTMs.

In these cases, **unidirectional LSTMs** or **GRUs** are typically employed, as they can make predictions based solely on past and current inputs without requiring future context.

4.6.7 Practical Tips: Initialization & Conditioning

This section provides practical tips for initializing and conditioning Long Short-Term Memory (LSTM) networks to enhance their performance and stability during training. We will go through two key recommendations:

1. **Bidirectional conditioning (when and why to use BiLSTM)**
2. **Learnable initial states (why initializing h_0 and C_0 as parameters is better than zeros)**

Both points relate directly to improving performance and stability when training LSTM networks.

Bidirectional Networks: Condition on the Full Sequence

> **Practical Tip #1:** When the entire input sequence is available, a Bidirectional LSTM is usually superior because it conditions each timestep on both past and future information.

This may sound simple on the surface, but it actually encodes one of the most important design principles in sequence modeling: **context matters**.

In a **standard LSTM**, the hidden state at time t depends only on all previous inputs $\{x_1, x_2, \ldots, x_t\}$ and the current input x_t. But it knows *nothing* about future input x_{t+1}, future context or upcoming events. This creates an **information asymmetry** where our representation of x_t (i.e., h_t, the hidden state at time t) is only half-aware.

Why this is a problem: Suppose we want to classify a small window around time t. Patterns like anomalies, spikes, movement patterns often depend on: what *just happened* before and what *will happen* after. A unidirectional model only sees the previous points. Instead, a **Bidirectional LSTM (BiLSTM)** model sees *both sides* of the window. This often gives *significantly* better performance.

When to use BiLSTM: If our application allows us to see the entire sequence before making predictions (e.g., text classification, speech recognition, video analysis), then BiLSTM is usually the better choice. It leverages full context to create richer representations. **When not to use BiLSTM:** However, if we need to make real-time predictions (e.g., online forecasting, live translation), then we must use unidirectional LSTMs since future data is not available.

Learnable Initial States (Initialization as Parameters)

> **Practical Tip #2:** Instead of initializing the initial hidden state h_0 and cell state C_0 with fixed constants (like zeros), treat them as trainable parameters and learn them during training.

Every LSTM sequence starts with initial values:

- Initial **hidden state** h_0
- Initial **cell state** C_0

These two values define the **starting memory** of the LSTM **before it reads the first input** x_1. In most implementations, these are simply initialized to zeros:

$$h_0 = \mathbf{0}, \quad C_0 = \mathbf{0}$$

This is the *default* choice, but not the best.

Why zero initialization is not ideal: Zero initialization has several drawbacks:

- **Not realistic**. Real sequences rarely "start from zero memory". There is almost always some baseline state.
- **Every sequence starts from the same memory**. Even if our dataset contains very different types of sequences, they all begin with identical h_0 and C_0. This limits the model's ability to adapt to different contexts right from the start.
- **Slow convergence**. The model has to learn to build up useful memory from scratch for every sequence, which can slow down training.
- **Weak signal at early timesteps**. The initial outputs may be weak or uninformative since the model starts with no prior knowledge.
- **Doesn't encode dataset priors**. If certain patterns are common at the start of sequences, zero initialization fails to capture this prior knowledge.

The better idea: make h_0 and C_0 learnable parameters. Instead of fixing h_0 and C_0 to zeros, we can treat them as **trainable parameters**:

$$h_0 = \theta_h, \quad C_0 = \theta_C$$

Where θ_h and θ_C are **trainable vectors initialized randomly** (e.g., using Xavier or He initialization, page 172). They are optimized like any other weight via backpropagation. This means that the model learns the best starting hidden state and the best starting cell state for the entire training dataset. These states become part of the model parameters.

Benefits of learnable initial states:

- ✔ **Faster convergence**. The model starts with a better prior memory, leading to quicker learning since it doesn't have to build memory from scratch.
- ✔ **Better performance, especially with short sequences**. The model can adapt its initial memory to the dataset, improving accuracy. If sequences are short or medium length, initial state quality affects the entire trajectory more significantly.
- ✔ **Better early-time gradients**. The model produces more informative outputs from the start, enhancing gradient flow. A strong trained initial state provides clearer gradients and more stable backpropagation.
- ✔ **More expressive temporal priors**. The model can learn common starting patterns in the data. For example, if sequences often start with a certain trend or baseline, the learned initial states can encode this knowledge.
- ✔ **More robust to noise and variability**. The model can adjust its initial memory to handle different sequence types better, especially when early inputs are noisy or unreliable.

⚠ **What about overfitting?** Surprisingly, almost **no risk of overfitting** arises from learning initial states. Because **only 2 vectors** are added to the model (h_0 and C_0), and they are **extremely small compared to network weights**. Also, they capture global dataset-level priors, not sample-level details. Thus, they help generalization rather than harm it.

4.7 Sequential Data Problems

Up to now we mostly considered "classic" deep-learning problems where we have a **fixed-size input** and a **fixed-size output**, for example an image classification problem where the input is an image of fixed size (e.g., 224×224 pixels) and the output is a class label (e.g., *cat*, *dog*, etc.). However, many real-world problems involve **sequential data**, where the input and/or output can vary in length. When dealing with sequential data, we often encounter the following types of problems:

One-to-One Problems. It is the **classic** machine learning setup where we have **one input**, and we want to predict **one output**. There is **no sequence** involved on either side.

It is the standard supervised learning scenario: $x \rightarrow y$. Where x is a fixed-size input (e.g., an image, a feature vector, an audio clip, etc.) and y is a fixed-size output (e.g., a class label, a regression value, etc.). In this setting, the input has **no temporal dimension**, the output has **no temporal dimension**, and the mapping from input to output is **static** (i.e., it does not change over time).

What architecture is used in One-to-One problems? Any model that takes fixed-size input and produces fixed-size output can be used, such as **Feed-Forward Neural Networks (FFNs)**, **Convolutional Neural Networks (CNNs)**, etc.

Architectures that are not used? **Recurrent Neural Networks (RNNs)** and their variants (e.g., LSTMs, GRUs) are generally not used in One-to-One problems, as they are designed to handle sequential data (i.e., data with temporal dependencies).

One-to-Many Problems. In this type of problem, the model receives **one fixed-size input** x and produces a **sequence** as output: $y_1, y_2, \ldots, y_T$. The length of the output sequence may be fixed or variable (stopped by an end-of-sequence token). However, the crucial point is that **the input is static; the output is a sequence generated step-by-step**. This is the first case in which RNNs/LSTMs/GRUs become necessary. Some examples of One-to-Many problems include:

- **Image Captioning**: Given an image, generate a descriptive caption.
- **Music Generation**: Given a seed note or chord, generate a sequence of musical notes.
- **Text Generation**: Given a prompt, generate a sequence of words or sentences.

Why RNN/LSTM is needed here? Because the output must be generated *sequentially*:

- The first output depends on the image.
- The next output depends on:
  - The image representation.
  - The previous output word.

- The internal state of the RNN/LSTM/GRU.

Only RNNs/LSTMs (or later Transformer) can model this dynamic, autoregressive structure. A feedforward network cannot generate a *sequence* of arbitrary length based on a single input.

Example 9: Image Captioning

Image Captioning is a classic example of a One-to-Many problem. In this task, the model takes an image as input and generates a descriptive caption as output. The input is a fixed-size image (e.g., 224×224 pixels), while the output is a sequence of words forming a sentence (e.g., "A dog playing with a ball in the park"). The length of the caption can vary depending on the content of the image.
More specifically, an image is processed once (e.g., by a CNN) to obtain a feature vector. Then an RNN/LSTM/GRU **generates words one at a time**.

1. A CNN encodes the image into a feature vector.
2. The feature vector is fed to an LSTM as the initial input/hidden state.
3. At each timestep, the LSTM outputs **one word**.
4. The sequence continues until an End-Of-Sentence (EOS) token is generated.

Formally, if x is the input image and y_t is the word generated at timestep t, the model learns to predict:

$$P\left(y_1, y_2, \ldots, y_T \mid x\right) = \prod_{t=1}^{T} P\left(y_t \mid y_1, y_2, \ldots, y_{t-1}, x\right)$$

Where T is the length of the generated caption, which can vary for different images.

Many-to-One Problems. In this type of problem, the model receives a **sequence** as input: $x_1, x_2, \ldots, x_T$, and produces a **single fixed-size output** y. The length of the input sequence may be fixed or variable. However, the crucial point is that **the input is a sequence; the output is static**. This means the model must "summarize" the entire sequence into **one decision or prediction**. Some examples of Many-to-One problems include:

How is this solved using RNNs/LSTMs/GRUs? THe RNN/LSTM/GRU reads the input sequence one timestep at a time:

$$h_t = \text{LSTM}\left(x_t, h_{t-1}\right)$$

At the **final timestep** T, the hidden state h_T contains a representation

of the whole sequence:

$$h_T \approx \text{summary of } (x_1, x_2, \ldots, x_T)$$

Then the model passes it to a classifier (e.g., a fully connected layer followed by softmax) to produce the final output y:

$$\hat{y} = g(h_T) \quad \text{linear} \rightarrow \text{softmax or sigmoid}$$

This output is the predicted label or value for the entire input sequence.

When does many-to-one occur in practice? Besides sentiment analysis, common examples include:

- **Sequence classification**: Classify a time-series (e.g., accelerometer data) into categories (e.g., walking, running, sitting). Classify an audio clip into genres (e.g., rock, jazz, classical). Identify speaker emotions from speech (e.g., happy, sad, angry). Human activity recognition from wearable sensor data (e.g., walking, running, sitting). Fault detection in sensor streams (e.g., normal vs. faulty operation).
- **Video-level classification**: Classify an entire video into categories (e.g., action recognition, scene classification).
- **Anomaly detection**: Detect anomalies in sequences (e.g., network traffic, sensor readings).

Many-to-Many Problems. In this type of problem, the model receives a **sequence** as input: $x_1, x_2, \ldots, x_T$, and produces a **sequence** as output: $y_1, y_2, \ldots, y_{T'}$. The lengths of the input and output sequences may be fixed or variable. This is the most general case, where both the input and output are sequences. Some examples of Many-to-Many problems include:

- **Machine Translation**: Given a sentence in one language, translate it into another language.
- **Video Captioning**: Given a video, generate a descriptive caption.
- **Speech Recognition**: Given an audio clip, transcribe it into text.

Why isn't a simple RNN enough? If we tried to output one word for each input word (i.e., $T' = T$), we would have a **Many-to-Many with aligned sequences** problem. However, in many real-world applications, the output sequence length T' may differ from the input sequence length T (e.g., translating a short sentence into a longer one). Therefore, **the model must first understand the entire input sequence, then generate a new sequence**. This motivates the **encoder-decoder architecture**.

How does the Encoder-Decoder architecture work?

The architecture works in two main stages:

1. **Encoder**
   - Reads the entire input sequence $x_1, x_2, \ldots, x_T$.

- Updates its hidden state at each timestep:

$$h_t = \text{LSTM}\,(x_t, h_{t-1})$$

- At the end, produces a **fixed-dimensional representation** (context vector) of the entire input sequence:

$$h_T \approx \text{summary of } (x_1, x_2, \ldots, x_T)$$

2. **Decoder**
   - Takes the context vector from the encoder as input.
   - Generates the output sequence **one element at a time**:

$$y_1, y_2, \ldots, y_{T'}$$

   - At each timestep, the decoder predicts the next output element based on:
     * The context vector (encoder output).
     * The previously generated outputs.
     * Its own internal state.

Many-to-Many Problems (with aligned sequences). In this variant of the Many-to-Many problem, the input and output sequences are of the **same length** ($T' = T$), and there is a direct alignment between each input element and its corresponding output element. This means that for each input x_t, there is a corresponding output y_t.

In this settings:

- **Input** is a sequence: $x_1, x_2, \ldots, x_T$.
- **Output** is also a sequence: $y_1, y_2, \ldots, y_T$.
- There is a **direct correspondence** between each input element and its output element: $x_t \to y_t$ for all $t = 1, 2, \ldots, T$.

So same number of timesteps, same temporal grid, and one output **for each input element**. Some examples of Many-to-Many problems with aligned sequences include:

- **Part-of-Speech Tagging**: Given a sentence, assign a part-of-speech tag to each word.
- **Named Entity Recognition**: Given a sentence, identify and classify named entities (e.g., persons, organizations, locations).
- **Video Frame Labeling**: Given a video, label each frame with an action or object.

How is it modeled? A Recurrent Neural Network (RNN)/LSTM/GRU processes the input sequence one timestep at a time:

$$h_t = \text{LSTM}\,(x_t, h_{t-1})$$

Then, at **every timestep** t, the model produces an output y_t based on the current hidden state h_t:

$$\hat{y}_t = g\,(h_t) \quad \text{linear} \;\to\; \text{softmax or sigmoid}$$

So the model generates outputs:

$$\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_T$$

Often called **Sequence Labeling** or **Sequence Tagging**, since each input element is tagged with a corresponding output label.

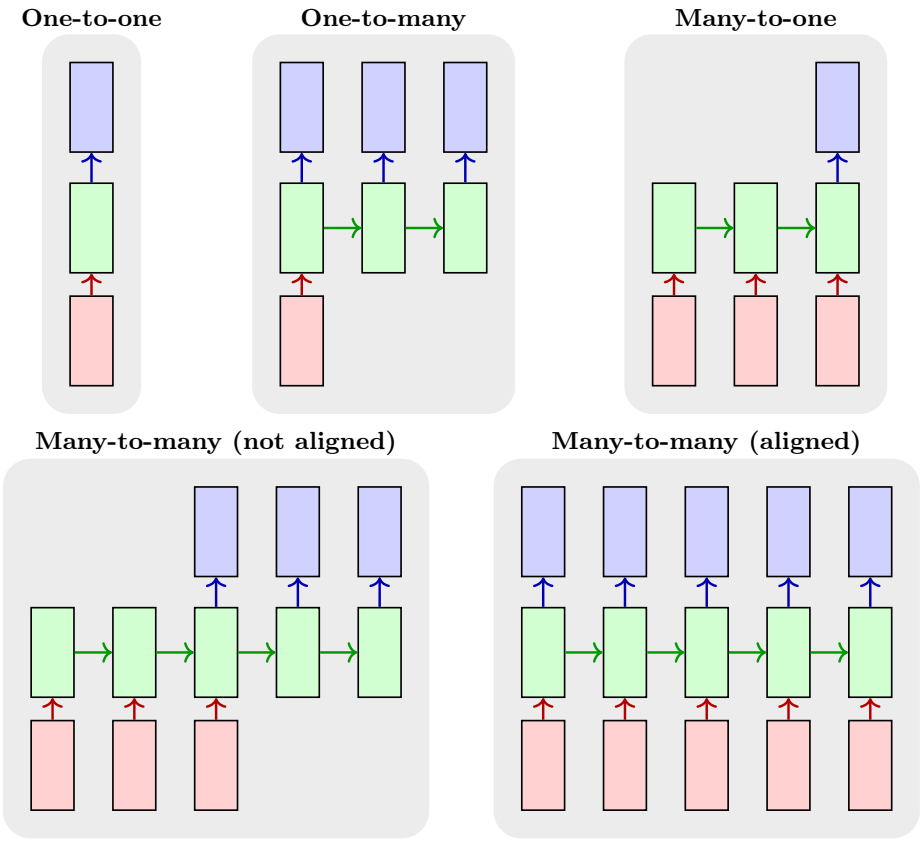


Figure 32: Different types of Sequential Data Problems.

Case	Input	Output	Alignment
One-to-One	Fixed-size	Fixed-size	N/A
One-to-Many	Fixed-size	Sequence	No
Many-to-One	Sequence	Fixed-size	No
Many-to-Many	Sequence	Sequence	No
Many-to-Many (aligned)	Sequence	Sequence	Yes

Table 5: Summary of Sequential Data Problems. "Fixed-size" refers to inputs or outputs that do not vary in length, while "Sequence" indicates variable-length data. The "Alignment" column specifies whether there is a direct correspondence between elements of the input and output sequences.

References

[1] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[4] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[5] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[6] Matteucci Matteo. Artificial neural networks and deep learning. Slides from the HPC-E master's degree course on Politecnico di Milano, 2025-2026.

Index

A

Activation Function 57
Activation Potential 57
ADALINE (Adaptive Linear Neuron) 39
Adam (Adaptive Moment Estimation) 183
Adaptive Learning Rate Methods: Adam (Adaptive Moment Estimation) 183
Adaptive Learning Rate Methods: RMSProp (Root Mean Square Propagation) 183
Artificial Neuron 36
Autoregressive (AR) Model 190

B

Backpropagation 96
Backpropagation Through Time (BPTT) 223
Batch Gradient Descent 115
Batch Normalization (BN) 175
Bayesian Optimization 149
Bias 36
Bias-Variance Trade-off 124
Bidirectional LSTM (BiLSTM) 257
Binary Cross-Entropy (BCE, Log Loss) 73

C

Candidate cell state $\tilde{C}_t$ 245
Categorical Cross-Entropy (CCE) 79
Chain Rule 96
Classification 9
Clustering 12
Constant Error Carousel (CEC) 242
Core Learning Rule 91
Cost Function 86
Covariate Shift 174
Cross-Validation 130

D

Dead ReLU Problem 166
Decision Boundary 48
Decision Boundary Equation 41
Decoding Problem 208
Deterministic Recurrent Systems 212
Dropout 160, 161
Dying ReLU Problem 166

E

Early Stopping 142
Error Function $E(w)$ 86
Experience (E) 5
Exploding Gradient Problem 165
Exponential Linear Unit (ELU) 168

F
Feature Engineering (Traditional ML) 21
Feed-Forward Neural Network (FNN) 54
Feed-Forward Time-Delay Neural Network (TDNN) 192
Fixed Learning Rate 182
Forget gate f_t 244

G
Gated Recurrent Unit (GRU) 247
Gradient Clipping 234, 239
Gradient Descent 90
Grid Search 148

H
He (Kaiming) Initialization 172
Hebbian Learning Rule 44
Hidden Markov Model (HMM) 203
Hold-Out Method 127, 131
Hold-Out Validation 131
Hyperbolic Tangent (tanh) Activation Function 64
Hyperparameter Tuning 145
Hyperparameter Tuning Algorithm 147
Hyperplane 110

I
Independent and Identically Distributed (i.i.d.) 104
Inductive Bias 123
Inductive Hypothesis 123, 125
Input gate i_t 245
Internal Covariate Shift 174

K
K-Fold Cross-Validation 135
Kalman Filter 198

L
L2 Regularization 151
Leaky ReLU 167
Learned Features (Deep Learning) 21
Learning Rate Scheduling (decay) 182
Learning Rate Scheduling: 1/t Decay 182
Learning Rate Scheduling: Cosine Annealing 182
Learning Rate Scheduling: Exponential Decay 182
Learning Rate Scheduling: Step Decay 182
Leave-One-Out Cross-Validation (LOOCV) 133
Linear Activation Function 59
Linear Dynamical System (LDS) 196
Linearly Separable 51
Logistic Activation Function 61
Long Short-Term Memory Network (LSTM) 241
Loss Function 86

M

MADALINE (Multiple ADALINE network) 39
Many-to-Many Problems 264
Many-to-Many Problems (with aligned sequences) 265
Many-to-One Problems 263
Markov Chain 202
Markov Property 195
Maximum A Posteriori (MAP) 154
Maximum Likelihood Estimation (MLE) 103
Mean Absolute Error 70
Mean Squared Error (MSE) 68
Memory Cell 241
Memoryless Models 188
Mini-Batch Gradient Descent (MBGD) 180
Model Complexity vs. Error Curve 125
Models with Memory 188

N

Nested Cross-Validation 137
Net Input 57
Neural Network 38

O

Ockham's Razor 119
One-Hot Encoding 76
One-to-Many Problems 262
One-to-One Problems 262
Output gate o_t 246
Overfitting 121

P

Parametric ReLU (PReLU) 167
Patience Parameter 143
Patience Window 143
Perceptron 38, 40
Performance measure (P) 5
Probabilistic Dynamical Systems 211

R

Random Search 148
Random Subsampling 127
Rectified Linear Unit (ReLU) 166
Recurrent Neural Network (RNN) 213
Regression 11
Reinforcement Learning (RL) 17
RMSProp (Root Mean Square Propagation) 183

S

Scaled Exponential Linear Unit (SELU) 168
Sequence Labeling 266
Sequence Tagging 266

Sigmoid Activation Function 61
Softmax Activation Function 77
State Estimation Problem 196
State Propagation 187
Stochastic Gradient Descent (SGD) 115
Stratified Sampling 127
Sum of Squared Errors (SSE) 87
Supervised Learning 9, 83

T
Task (T) 5
Task, Experience, Performance 5
Temporal Dimension 187
Test Set 128
Threshold Logic Unit (TLU) 38
Training 84
Training Dataset 128
Training Set 128
Training vs. Validation Error Curve 141

U
Underfitting 122
Universal Approximation Theorem 80, 118
Unsupervised Learning 12

V
Validation Set 128
Vanishing Gradient Problem 163
Viterbi Algorithm 209

W
Weight Decay 151
Weight Scaling Rule 161
Weight Update Rule in Hebbian Learning 45

X
Xavier (Glorot) Initialization 172

Z
Zero-Gradient Problem 163