AI in Mathematics

Elena Bunina, Alexander Shlimovich

1 Lecture 1. Introduction

1.1 Historic Overview: Computers in Mathematics

Computers have changed mathematics in many ways. At first, they were only used for calculations, but over time, they became tools for proofs, collaboration, and sharing ideas.

The Pioneering Era (Mid-20th Century). When Alan Turing worked on the first computers, he was trying to decrypt messages from the German military, which were encoded using the Enigma machine. Deciphering these messages required going through a vast number of possible combinations. Humans struggled with this task, but a machine could solve it relatively quickly. The first computers were used for calculations, codebreaking, and military tasks. They could perform thousands of calculations much faster than humans. However, they were not used for complex mathematical proofs yet. At this time, computers were just tools for number crunching.

The Four Color Theorem Controversy (1976). A big moment in computer-assisted mathematics happened in 1976 when Kenneth Appel and Wolfgang Haken used a computer to help prove the **Four Color Theorem**. This theorem states that any map can be colored using only four colors so that no two neighboring regions have the same color. Their proof relied on a computer checking nearly 2,000 different cases, something no human could do by hand. This caused a debate in the mathematical community:

- Is a proof valid if no human can check all the details?
- Can a computer proof be trusted?

Despite some skepticism, this was the start of **computer-assisted proofs**, which are now becoming an important part of mathematics.

The Digital Revolution (1980s–1990s). As computers became more common, they changed how mathematicians worked. Two key innovations helped:

- **LaTeX** – A system for writing mathematical documents, making them clearer, more structured and more beautiful.

- **arXiv** – A website where researchers can share their work instantly, helping ideas spread quickly.

TeX was created by Donald Knuth in the late 1970s as a response to the poor quality of mathematical typesetting in published papers. At the time, professional typesetting was slow and expensive, making it difficult for mathematicians and scientists to format their work properly. Knuth designed TeX as a high-quality, free, and flexible typesetting system, specifically optimized for mathematical and technical documents. By the 1980s, TeX had gained popularity among academics, and in the 1990s, the development of LaTeX (a structured macro system for TeX) made it even more accessible. Today, TeX and LaTeX are the standard tools for writing research papers in mathematics, physics, and computer science. They are widely used in journals, conferences, and preprint archives like arXiv, ensuring that mathematical writing remains clear, structured, and professional.

arXiv was founded in 1991 by Paul Ginsparg as an online repository for sharing scientific papers before formal peer review. Originally designed for physicists, it quickly expanded to include mathematics, computer science, and other fields. Before arXiv, researchers had to wait months for journal publications or rely on mailing printed preprints to colleagues. With arXiv, scientists could instantly share their work with the global academic community, speeding up research collaboration and discovery. Over time, arXiv became the primary platform for distributing preprints in mathematics and theoretical sciences, hosting millions of papers and serving as a key resource for researchers worldwide. Today, many groundbreaking results appear on arXiv before they are officially published, making it an essential tool for modern academic communication.

These tools made collaboration easier, replacing the slow process of mailing papers to colleagues. Mathematics became faster and more open.

Computers have transformed mathematics. They started as simple calculators but later helped with proofs and made research more accessible. These early developments led to modern AI tools in mathematics, which we will explore next.

1.2 Formal Proof Assistants and Lean

Mathematical proofs have traditionally been written by humans and verified through peer review. However, as they grow increasingly complex and lengthy, the demand for **formal verification**—where every logical step is checked with absolute rigor—has risen. This is where **formal proof assistants** come into play.

Among these tools, **Lean** has recently garnered significant attention. Developed by **Microsoft Research**, Lean offers a user-friendly environment that appeals to mathematicians and programmers. One of its most notable contributions is the **Lean Mathematical Library (mathlib)**, a continuously

expanding repository of formalized mathematics containing rigorous proofs of fundamental theorems.

Various large-scale efforts showcase Lean's potential. For example, a team at **Oxford** is undertaking the challenging project of formalizing **Fermat's Last Theorem** in Lean. The proof of this famous theorem had a lot of discussions of its correctness during the time it was published and original proof contained 100 pages of high-level mathematics which made it difficult to human checking. That's where automatic formal proof assistants may help since the correctness of a proof guaranteed by a code of the assistant.

1.3 Large Language Models and ChatGPT in Mathematics

Recent advances in **Artificial Intelligence (AI)**, particularly **Large Language Models (LLMs)**, have opened new possibilities for using AI in mathematical research. Systems like **ChatGPT**, **Llama**, and other AI models can assist mathematicians in solving problems, verifying proofs, and generating ideas.

1.4 Course Motivation: The Rise of AI in Mathematics

Nowadays Artificial Intelligence (AI) is becoming a powerful tool in a world and in mathematics as well. It helps researchers find proofs, verify results, and even discover new mathematical ideas. We can categorize AI's impact on mathematics into three major areas:

Large Language Models (ChatGPT, DeepSeek, etc.). Large language models (LLMs) have transformed how mathematicians search for information and generate ideas. These models can:

- Act as a **mathematical search engine**, helping users find references, proof ideas, and related concepts.
- Provide **explanations** of complex topics in a more accessible way.
- Even **generate proofs**—though their accuracy varies! Some results are correct, while others may be more "creative" than mathematically sound.

Formal Proof Assistants. Formal proof assistants, such as Lean and Coq, play an essential role in modern mathematics. These tools:

- **Verify** proofs, ensuring correctness, including those generated by AI.
- **Reduce** the need for human reviewers in checking complex proofs.
- **Enable large-scale, collaborative theorem proving**, where multiple researchers contribute to building formalized mathematics.

- **Allows generation structured datasets** for training machine learning models.

Machine Learning for Mathematics – The Focus of This Course!. Machine learning is opening new possibilities in mathematical discovery. This course will focus on how AI can:

- **Generate examples and counterexamples** for mathematical conjectures.
- **Discover new structures and relationships**, revealing insights that may not be obvious through traditional methods.
- **Push the boundaries** of problem-solving where conventional techniques struggle.

AI is rapidly changing the landscape of mathematical research. From automated proof generation to discovering new mathematical concepts, it is becoming an essential part of modern mathematics. In this course, we will explore how machine learning techniques can be applied to mathematical problems and what new insights they can offer.

1.5 Large Language Models (LLMs)

Large Language Models (LLMs) are powerful artificial intelligence systems designed to understand and generate human-like text. They are based on deep learning architectures and have transformed many areas of research, including mathematics. Key characteristics of LLMs include:

- **Massive neural networks** with billions (or even trillions) of parameters.
- **Training on vast datasets**, including books, research papers, websites, programming code, and even mathematical datasets.
- **Token prediction mechanism**: LLMs generate text by predicting the next token based on previously generated tokens and the given context.

Several major LLMs are actively shaping the landscape of AI-assisted mathematics:

- **GPT and o-series** (OpenAI) – Among the most capable models for mathematical tasks, relying on non-linear reasoning to effectively reduce hallucinations.
- **Claude** (Anthropic) – A model that places a strong emphasis on transparency and safety, often praised as one of the best for code generation.

- **LLaMA** (Meta) – A family of open-source LLMs released for research and experimentation, offering the community cutting-edge language modeling. It excels at routine and creative tasks that do not require advanced logical reasoning.
- **Gemini** (Google) – An AI system combining sophisticated language modeling with deep reinforcement learning techniques, capable of generating both text and images using a single model.
- **DeepSeek** – A specialized model focused on robust logical reasoning and mathematical problem-solving, including its noteworthy R1 variant. It performs exceptionally well in both reasoning and coding tasks.

LLMs are evolving rapidly, with ongoing research focusing on:

- **Establishing interpretability** – Understanding how models generate and structure responses. Although not a model-improvement technique, this line of research aims to clarify why the underlying architecture behaves as it does.
- **Enhancing reasoning** – Reducing hallucinations and improving logical consistency. Techniques like non-linear reasoning allow the model to iteratively refine its own steps, significantly boosting performance.
- **Extending long-context memory** – Enabling models to process and recall more extensive sequences of information, crucial for in-depth analysis and multi-step problem-solving.
- **Implementing agents** – Allowing the model to delegate tasks to specialized tools (for example, invoking Wolfram for equation solving or Lean for proof verification), thereby combining different systems to tackle complex tasks more effectively.

Large Language Models have become a significant tool in mathematics, assisting with problem-solving, proof generation, and research exploration. As they continue to improve, they hold great potential for advancing automated reasoning and mathematical discovery.

1.6 Formal Proof Assistants

Formal proof assistants are software tools designed to construct and verify mathematical proofs with **absolute rigor**. Unlike traditional mathematical writing, where mistakes can go unnoticed, proof assistants ensure that every logical step is mechanically checked, eliminating errors. These tools are built on precise logical frameworks, such as **dependent type theory**, which allows formalizing a wide range of mathematical concepts.

Key examples of proof assistants include:

- **Lean** – A modern, user-friendly proof assistant with growing adoption in the mathematical community.
- **Coq** – A powerful system widely used for both mathematics and program verification.
- **Isabelle** – A flexible proof assistant supporting multiple logical formalisms.

Formal proof assistants play an important role in AI-driven mathematical research. They provide structured, reliable data that can be used to train machine learning models for theorem proving. Key reasons why they are valuable for AI include:

- **Generating large datasets**: Proof assistants allow us to create vast collections of theorems and formal proofs, providing structured learning material for AI models. For example Lean community is actively working on a dataset **mathlib4** for proofs in Lean.
- **Enabling automated theorem proving**: AI can use these systems to search for new proofs and assist mathematicians in solving complex problems.
- **Providing noise-free data**: Unlike human-written mathematical text, formal proofs are **100% verifiable**, making them ideal for training AI without inconsistencies or ambiguities.

To learn Lean and explore more on Formal Proof Assistants we encorage you to visit link below:

`https://adam.math.hhu.de`

Formal proof assistants are changing the way mathematics is verified and explored. By combining their precision with machine learning, researchers can push the boundaries of automated reasoning and mathematical discovery.

1.7 LLM + Formal Proof Assistants

Large Language Models (LLMs) have made significant progress in mathematical problem-solving. A major milestone was achieved when an AI system demonstrated **Silver Medal-level** performance in the International Mathematical Olympiad (IMO), proving that LLMs are more than just advanced autocomplete systems.

- The AI wrote proofs in **Lean** and scored **28 out of 42 points** (just one point shy of a gold medal).
- It successfully solved the **hardest problem** of the Olympiad.

- However, it took **three days** to find the solution — far from practical competition timing.
- AI still struggles with **combinatorics problems**, which remain largely unproven. Apparently, AI is not a fan of this field!

Fun Fact: The AI responsible for writing these proofs was built on **AlphaZero** — the same deep reinforcement learning system that mastered Chess and Go. While its performance in mathematical olympiads is impressive, AI models like AlphaZero have already surpassed human abilities in other mathematical tasks.

What's Next? As AI continues to improve, its ability to assist in proof generation and theorem discovery is expanding. In the next section, we will explore areas where AlphaZero-like models are not just competing with humans but actually *beating us* in mathematical problem-solving

1.8 Solutions

1.8.1 Problem 1: Solved

Find all real numbers α so that, for every positive integer n, the integer

$$S(n, \alpha) = \lfloor \alpha \rfloor + \lfloor 2\alpha \rfloor + \lfloor 3\alpha \rfloor + \cdots + \lfloor n\alpha \rfloor$$

is divisible by n.

If α is an integer, then

$$S(n, \alpha) = (\lfloor \alpha \rfloor + \lfloor 2\alpha \rfloor + \cdots + \lfloor n\alpha \rfloor) = (1 + 2 + \cdots + n)\alpha = \frac{n(n+1)}{2} \cdot \alpha.$$

Since $\frac{n(n+1)}{2}$ is always an integer, $S(n, \alpha)$ is divisible by n if and only if $2 \mid \alpha$. If α is an odd integer, then $S(2, \alpha) = 3\alpha$ is not divisible by 2, giving a contradiction.

Now, assume that α is not an integer. We prove that in this case, the divisibility condition fails.

Shifting Argument Since replacing α with $\alpha \pm 2$ changes $S(n, \alpha)$ by

$$S(n \pm 2, \alpha) - S(n, \alpha) = 2(1 + 2 + \cdots + n) = n(n+1) \equiv 0 \pmod{n},$$

we may assume without loss of generality that $-1 < \alpha < 1$ and $\alpha \notin \mathbb{Z}$.

Case 1: $0 < \alpha < 1$ Let $m \geq 2$ be the smallest integer such that $m\alpha \geq 1$. Then,

$$S(m, \alpha) = \lfloor \alpha \rfloor + \lfloor 2\alpha \rfloor + \cdots + \lfloor (m-1)\alpha \rfloor + \lfloor m\alpha \rfloor.$$

Since $\alpha < 1$, it follows that $\lfloor k\alpha \rfloor = 0$ for $1 \leq k \leq m-1$, and $\lfloor m\alpha \rfloor = 1$. Thus,

$$S(m, \alpha) = 0 + \cdots + 0 + 1 = 1.$$

Since 1 is not divisible by m, we get a contradiction.

Case 2: $-1 < \alpha < 0$ Let $m \geq 2$ be the smallest integer such that $m\alpha \leq -1$. Then,

$$S(m, \alpha) = \lfloor \alpha \rfloor + \lfloor 2\alpha \rfloor + \cdots + \lfloor (m-1)\alpha \rfloor + \lfloor m\alpha \rfloor.$$

Since α is negative, $\lfloor k\alpha \rfloor = -1$ for $1 \leq k \leq m-1$, and $\lfloor m\alpha \rfloor = 0$. Thus,

$$S(m, \alpha) = (-1) + (-1) + \cdots + (-1) + 0 = -(m-1).$$

Since $-(m-1)$ is not divisible by m, we again get a contradiction.

1.8.2 Problem 2: Solved

Determine all pairs of positive integers (a, b) such that the sequence

$$x_n = \gcd(a^n + b, b^n + a), \quad n = 1, 2, \ldots$$

is eventually constant.

Key Ideas It is easy to check that $(a, b) = (1, 1)$ works, since

$$x_n = \gcd(1^n + 1, 1^n + 1) = \gcd(2, 2) = 2$$

for all n.

Assume the sequence is eventually constant. The crucial idea is to introduce

$$M := ab + 1.$$

This is inspired by the fact that when we heuristically consider $n = -1$. The reasoning is that the two rational expressions

$$\frac{1}{a} + b = \frac{ab+1}{b}, \quad \frac{1}{b} + a = \frac{ab+1}{a}$$

suggest a large common factor, heuristically corresponding to $x_{-1} = ab + 1$.

Since $\gcd(a, M) = \gcd(b, M) = 1$, we take a sufficiently large multiple of $\varphi(M)$ so that the sequence stabilizes:

$$x_{n-1} = x_n = x_{n+1} = \ldots.$$

Examining the first three stabilized terms:
- **First Term (x_{n-1})**: Using modular reductions, we derive $M \mid x_{n-1}$.
- **Second Term (x_n)**: This forces $a \equiv b \equiv -1 \pmod{M}$.
- **Third Term (x_{n+1})**: We deduce $0 \equiv a + b \pmod{M}$, leading to $M = 2$.

1.8.3 Problem 6: Solved

A function $f : \mathbb{Q} \to \mathbb{Q}$ is called *aquaesulian* if for every $x, y \in \mathbb{Q}$, one of the following holds:

$$f(x + f(y)) = f(x) + y \quad \text{or} \quad f(f(x) + y) = x + f(y).$$

Define $g(x) = f(x) + f(-x)$. Show that there exists an integer c such that there are at most c distinct values of $g(x)$ for rational numbers x, and find the smallest possible c.

Key Ideas A function satisfying the given condition is:

$$f(x) = \{x\} - \lfloor x \rfloor,$$

where $\{x\} = x - \lfloor x \rfloor$ denotes the fractional part of x.

- If x is an integer, then $g(x) = f(x) + f(-x) = 0$.
- If x is not an integer, then $g(x) = -2$.

Thus, this example shows that $g(x)$ can take **at least two distinct values**.

Define a relation:

$$a \sim b \quad \text{if} \quad f(a) = b \quad \text{or} \quad f(b) = a.$$

Using the given functional equation, we deduce:

- The function f must be a **bijection**.
- The property $f(-f(-x)) = x$ holds.
- We derive that $g(x) = f(x) - f^{-1}(x)$, implying that $g(x)$ has at most **two distinct values**. For each two non-zero values they must be equal.

1.8.4 Problem 3: Not solved

Let $(a_n)_{n=1}^{\infty}$ be an infinite sequence of positive integers, and let N be a positive integer. Suppose that for each $n > N$, the term a_n is equal to the number of times a_{n-1} appears in the list $a_1, a_2, \ldots, a_{n-1}$.

Prove that at least one of the sequences $(a_1, a_3, a_5, \ldots)$ or $(a_2, a_4, a_6, \ldots)$ is eventually periodic.

1. If infinitely many distinct values appear in the sequence, then the sequence must contain arbitrarily large numbers. However, each large number must eventually be followed by 1, which contradicts the assumption that new large numbers keep appearing. Thus, the number of distinct values appearing infinitely often must be **finite**.

2. Define $M > \max(a_1, a_2, \ldots, a_N)$. We show that:

   (a) Some integer appears infinitely often.

   (b) Any integer appearing more than M times must be followed by another integer appearing at least M times.

3. Define three categories:

   (a) **Big** numbers: Those appearing at most k times for some k.

   (b) **Small** numbers: Those appearing infinitely often.

   (c) **Medium** numbers: Those appearing more than k but finitely many times.

4. We choose a sufficiently large index N_1 where:

   (a) Every **medium** number has already appeared its final time.

(b) Every **small** number has appeared more than $\max(k, N)$ times.

5. After N_1, the sequence **alternates** between big and small numbers.
6. A key lemma states that if a big number g is followed by a small number h, then h counts how many small numbers have appeared at least g times.
7. The sequence of **small numbers** satisfies a recurrence that ensures it is eventually periodic.

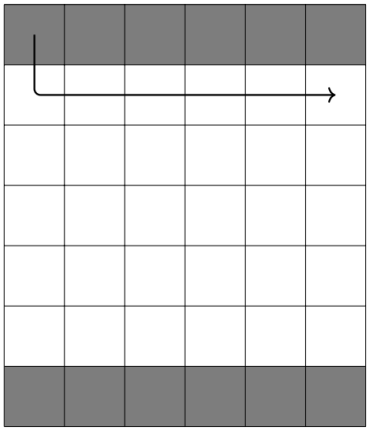
1.8.5 Problem 5: Not solved

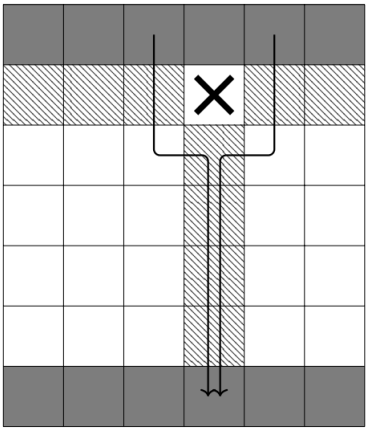
Turbo the snail plays a game on a board with 2024 rows and 2023 columns. There are hidden monsters in 2022 of the cells. Initially, Turbo does not know where any of the monsters are, but he knows that there is exactly one monster in each row except the first row and the last row, and that each column contains at most one monster.

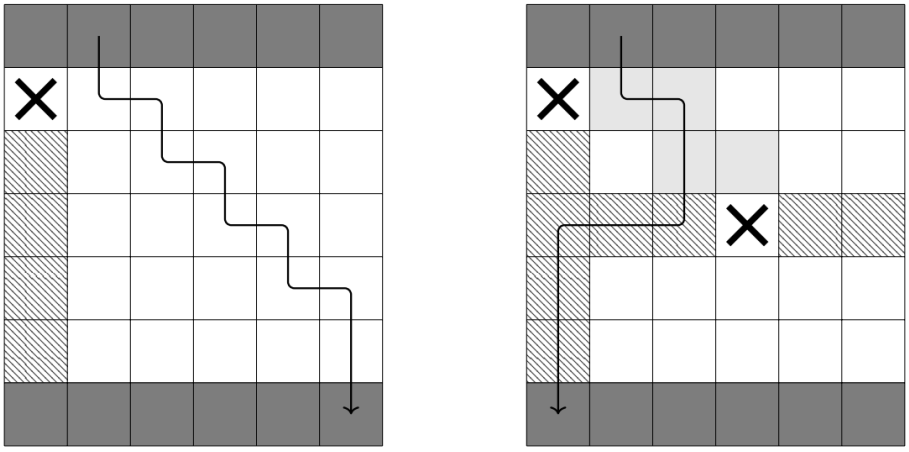
Turbo makes a series of attempts to go from the first row to the last row. On each attempt, he chooses to start on any cell in the first row, then repeatedly moves to an adjacent cell sharing a common side. (He is allowed to return to a previously visited cell.) If he reaches a cell with a monster, his attempt ends and he is transported back to the first row to start a new attempt. The monsters do not move, and Turbo remembers whether or not each cell he has visited contains a monster. If he reaches any cell in the last row, his attempt ends and the game is over.

Determine the minimum value of n for which Turbo has a strategy that guarantees reaching the last row on the n^{th} attempt or earlier, regardless of the locations of the monsters.

Answer: $n = 3$.





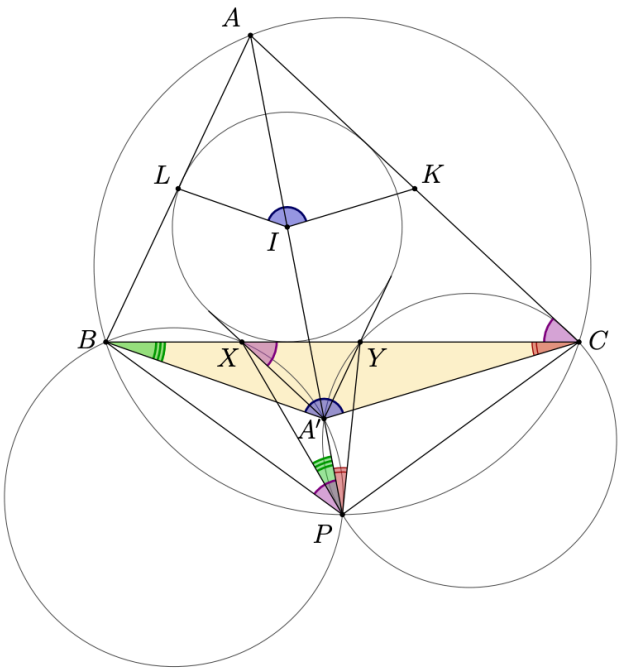


Strategies on a pictures show how to solve the problem in 3 attempts, it is not hard to show that you can not pass even to fourth row generally without the 3rd attempt.

1.9 Problem 5

Let ABC be a triangle with $AB < AC < BC$. Let the incentre and incircle of triangle ABC be I and ω, respectively. Let X be the point on line BC different from C such that the line through X parallel to AC is tangent to ω. Similarly, let Y be the point on line BC different from B such that the line through Y parallel to AB is tangent to ω. Let AI intersect the circumcircle of triangle ABC again at $P \neq A$. Let K and L be the midpoints of AC and AB, respectively.

Prove that $\angle KIL + \angle YPX = 180°$.



1.9.1 Data Generation

For training an ML algorithm, a sufficiently large dataset (potentially on the order of millions) of theorems and their corresponding solutions is required. Constructing this dataset manually is both time-consuming and prone to error, prompting the authors to synthesize theorems programmatically. Their approach begins with a set of pre-generated geometric constructions. From these constructions, they apply a *deduction engine*—an algorithm that uses predefined deduction steps to systematically generate new results.

As this iterative deduction process unfolds, it produces a directed graph whose nodes represent derived facts, and whose edges indicate dependencies. In other words, an edge from Fact A to Fact B signifies that Fact B can be deduced only after Fact A is established. Each fact in this graph is thus associated with a subgraph illustrating the deduction steps it relies upon. This subgraph can be interpreted as a complete proof of the fact, and by choosing any fact in the graph, one obtains the corresponding theorem and its proof structure.

Moreover, this method makes it possible to produce examples involving auxiliary constructions in their proofs. Such constructions may introduce new points or lines that do not appear in the original statement of the selected theorem but are nonetheless essential to the theorem's proof.

1.9.2 AlphaGeometry Model

The AlphaGeometry model reformulates a given theorem into a representation suitable for the system. In the initial prototype, this reformulation was performed manually. More recently, LLMs have been employed to handle the translation, and they have succeeded on 33 out of 44 IMO problems. The authors report that, for simpler problems, the LLM-based reformulation is almost always accurate.

Once the theorem is translated, the model proceeds in iterative cycles of two operations:

1. Deduce all possible facts from the current configuration using a purely computational module that applies the predefined deduction steps.
2. Invoke an ML-based transformer to predict the next auxiliary point or construction needed, based on the problem and the current state.

By separating deduction (a fixed set of inference rules) and construction (an ML-guided search for new elements), the system leverages both logical rigor and predictive flexibility. The process continues until a satisfactory solution is found, at which point the system generates an image highlighting the relevant constructions and facts used in the final proof.

1.9.3 Conclusion

Although AlphaGeometry has demonstrated promising results in automated geometry problem solving, it remains unable to tackle all geometry problems. One key limitation is that the problem statement must be formulated in a manner amenable to the model's language-processing capabilities. An important upgrade from Version 1 to Version 2 was the expansion of the model's accessible dictionary, thereby allowing it to work with a broader range of geometric concepts. Nonetheless, certain areas such as combinatorial geometry and inequality proofs remain beyond the scope of AlphaGeometry in its current form.

And now to the real math problems

1.10 Example: Matrix Multiplication

1.10.1 Classical Approach

Matrix multiplication is a fundamental operation in mathematics and computer science. Consider multiplying two 2×2 matrices:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Question: How many multiplications are required in the classical approach?
Answer: The standard method performs 8 multiplications:

$$\begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}.$$

1.10.2 Strassen's Algorithm

Strassen's Algorithm (1969): A more efficient approach reduces the number of multiplications from 8 to 7, using intermediate matrix products:

$$\begin{aligned} M_1 &= (a_{11} + a_{22}) \cdot (b_{11} + b_{22}), \quad M_2 = (a_{21} + a_{22}) \cdot b_{11}, \\ M_3 &= a_{11} \cdot (b_{12} - b_{22}), \quad M_4 = a_{22} \cdot (b_{21} - b_{11}), \\ M_5 &= (a_{11} + a_{12}) \cdot b_{22}, \quad M_6 = (a_{21} - a_{11}) \cdot (b_{11} + b_{12}), \\ M_7 &= (a_{12} - a_{22}) \cdot (b_{21} + b_{22}). \end{aligned}$$

Using these M_i, the matrix product can be rewritten as:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}.$$

Challenge: Given two matrices of sizes $n \times m$ and $m \times p$, determine the minimum number of scalar multiplications required to compute their product.

1.10.3 Machine Learning Improves Bounds

Recent advancements in deep learning and reinforcement learning have led to the discovery of more efficient matrix multiplication methods. AI has been used to search for new algorithms that outperform classical approaches.

Size (n, m, p)	Best method known	Best rank known	AlphaTensor rank Modular	AlphaTensor rank Standard
(2, 2, 2)	(Strassen, 1969)[2]	7	7	7
(3, 3, 3)	(Laderman, 1976)[15]	23	23	23
(4, 4, 4)	(Strassen, 1969)[2] (2, 2, 2) ⊗ (2, 2, 2)	49	47	49
(5, 5, 5)	(3, 5, 5) + (2, 5, 5)	98	96	98
(2, 2, 3)	(2, 2, 2) + (2, 2, 1)	11	11	11
(2, 2, 4)	(2, 2, 2) + (2, 2, 2)	14	14	14
(2, 2, 5)	(2, 2, 2) + (2, 2, 3)	18	18	18
(2, 3, 3)	(Hopcroft and Kerr, 1971)[16]	15	15	15
(2, 3, 4)	(Hopcroft and Kerr, 1971)[16]	20	20	20
(2, 3, 5)	(Hopcroft and Kerr, 1971)[16]	25	25	25
(2, 4, 4)	(Hopcroft and Kerr, 1971)[16]	26	26	26
(2, 4, 5)	(Hopcroft and Kerr, 1971)[16]	33	33	33
(2, 5, 5)	(Hopcroft and Kerr, 1971)[16]	40	40	40
(3, 3, 4)	(Smirnov, 2013)[18]	29	29	29
(3, 3, 5)	(Smirnov, 2013)[18]	36	36	36
(3, 4, 4)	(Smirnov, 2013)[18]	38	38	38
(3, 4, 5)	(Smirnov, 2013)[18]	48	47	47
(3, 5, 5)	(Sedoglavic and Smirnov, 2021)[19]	58	58	58
(4, 4, 5)	(4, 4, 2) + (4, 4, 3)	64	63	63
(4, 5, 5)	(2, 5, 5) ⊗ (2, 1, 1)	80	76	76

1.10.4 AI Can Discover Faster Matrix Multiplication Algorithms

AlphaTensor has uncovered new multiplication strategies that outperform all previously known results for certain matrix sizes. This AI system is based on **AlphaZero** — the same reinforcement learning model that mastered Go, Chess, and nearly won the IMO. Instead of playing games, it now plays a "game" of discovering efficient matrix multiplication rules.

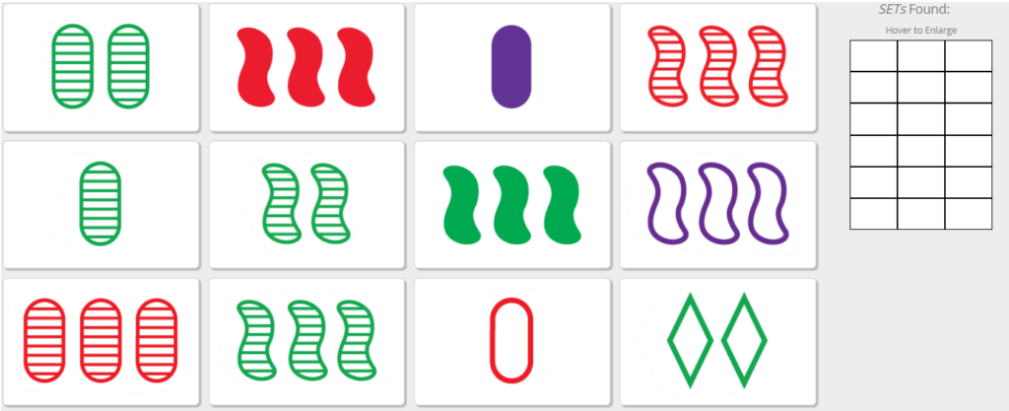
Key Reference: Fawzi, A. *et al.* (2022). *Discovering faster matrix multiplication algorithms with reinforcement learning. Nature 610.*

Why Is This Problem a Good Fit for Machine Learning?

Answer: The key advantage is that **verifying** a solution is easy, even though **finding** the optimal one is extremely difficult. This makes it an ideal problem for AI, where models can explore large search spaces efficiently and propose solutions that can be quickly checked.

1.11 Example: SET Game

SET is a card game based on four attributes: *shape, number, color, and shading.* A **"set"** consists of three cards where, for each attribute, the values are either all the same or all different.



The game has an interesting mathematical structure! Let's explore some key questions.

Question 1: What is the maximum number of SET cards we can place on the table without forming a set?

Answer: The maximum is **20**. Try to find the answer in your free time.

Question 2: What mathematical structure do the cards correspond to?

Each card represents a point in $\mathbb{Z}_3^4$ (a four-dimensional vector space over $\mathbb{Z}_3$). A "set" is a triple of cards whose sum in $\mathbb{Z}_3^4$ is the zero vector.

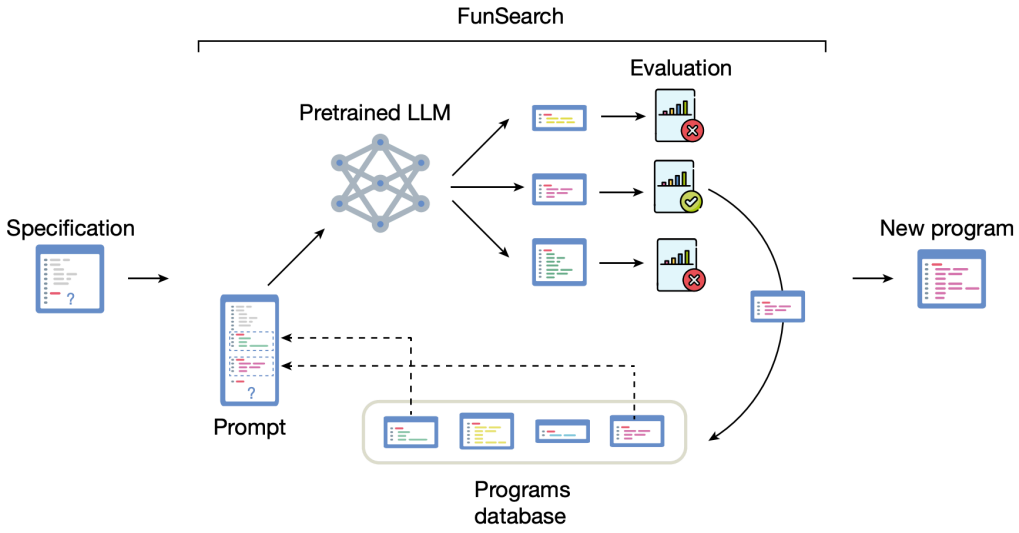
Challenge: Find the largest possible subset of $\mathbb{Z}_3^n$ such that the sum of any triplet does not equal zero.

Observation: Once again, we encounter a **"find an example"** problem — where verifying a solution is easy, but discovering one is difficult.
This problem is known as the **Capset Problem**.

Recently, AI has improved the best-known bound for $n = 8$!!!!

n	3	4	5	6	7	8
Best known	9	20	45	112	236	496
FunSearch	9	20	45	112	236	512

But how did AI achieve this breakthrough? Let's take a closer look.
How did AI improve the capset bound for $n = 8$?



The key idea was to improve a dataset of programs that select large capsets by leveraging a Large Language Model (LLM).
The algorithm followed these iterative steps:

1. Start with a dataset of programs that generate capsets.
2. Select the best programs and ask an LLM to refine them to increase the capset size.
3. Evaluate whether the new program is correct and whether it increases the capset size.

4. Add the best improved programs back into the dataset.

This method is an example of **evolutionary algorithms**, a well-known technique in machine learning that iteratively improves solutions over time. However, the fact that we can **use LLM to evolve a program** is quite surprising.

Reference: Romera-Paredes, B. *et al.* (2024). *Mathematical discoveries from program search with large language models. Nature 625.*

1.12 Example: Lyapunov Functions

A **Lyapunov function** is a mathematical tool used to study the stability of an ordinary differential equation (ODE) of the form:

$$\dot{x} = g(x), \quad x \in \mathbb{R}^n.$$

It is a scalar function that helps us understand whether solutions of the ODE remain near an equilibrium point over time.

Definition 1.1. *Let $V : \mathbb{R}^n \to \mathbb{R}$ be a continuous function. We say that V is a **Lyapunov function** for the system $\dot{x} = g(x)$ if the following conditions hold:*

1. ***Positive definiteness:** $V(x) > 0$ for all $x \neq 0$ and $V(0) = 0$.*
2. ***Non-increasing along trajectories:** The time derivative of V along the trajectories of the system satisfies*

$$\dot{V}(x) = \nabla V(x) \cdot g(x) \leq 0 \quad \text{for all } x \in \mathbb{R}^n.$$

*These conditions ensure that V decreases (or remains constant) along solutions of the ODE, which in turn implies that the equilibrium at $x = 0$ is **stable**.*

The existence of a Lyapunov function for a system is a powerful indicator of stability. In particular:

- If a Lyapunov function exists, then the system is **stable** in the sense of Lyapunov.
- It guarantees that solutions starting close to the equilibrium remain close for all future time.
- In some cases, it can even be used to show that the equilibrium is **asymptotically stable**.

Finding a Lyapunov function for a given ODE is, in general, a difficult task, and there is no universal method for constructing one. Traditional methods rely on deep insight into the system's dynamics, which may not be available for complex or high-dimensional systems.

AI-Assisted Prediction of Lyapunov Functions. Recent advancements in artificial intelligence have shown significant promise in automating the discovery of Lyapunov functions. Previously, available software could only handle a narrow class of functions and offered no general approach for non-polynomial systems. By integrating AI methods, researchers have managed to find Lyapunov functions for a much broader range of problems. While these techniques do not yet solve every conceivable case or discover entirely novel Lyapunov functions unknown to experts, the results so far are nonetheless impressive. In one study, an AI-based tool succeeded in finding Lyapunov functions for 84% of the problems in a test dataset, whereas Master's students in the field solved only 9.3% of them. Such findings underscore the substantial time savings and overall research benefits that AI can provide in this domain.

	Sample	SOSTOOL	Existing AI methods			Forward	Backward models	
Test set	size	`findlyap`	Fossil 2	ANLC	LyzNet	FBarr	PolyM	NonPolyM
Poly3	65,215	1.1	0.9	0.6	4.3	11.7	**11.8**	11.2
Poly5	60,412	0.7	0.3	0.2	2.1	8.0	**10.1**	9.9
NonPoly	19,746	-	1.0	0.6	3.5	-	-	**12.7**

Reference: Alfarano, A., Charton, F., Hayat, A. (2024). *Global Lyapunov functions: a long-standing open problem in mathematics, with symbolic transformers.* arXiv:2410.08304.

Question: How can a function be represented for a machine learning algorithm?

Answer: Functions can be encoded as **tree structures**, where:

- Nodes represent mathematical operations (e.g., $+, -, \times, \div$).
- Leaves represent variables or constants.

This tree can be linearized into a sequence using **Polish notation** (prefix notation), allowing the representation to be processed as a sequence-to-sequence learning task.

Question: How can we create a dataset of ODEs and their corresponding Lyapunov functions?

Answer: Instead of solving ODEs manually, a reverse approach can be taken:

1. First, generate candidate Lyapunov functions.
2. Then, construct or identify ODEs for which these functions satisfy the stability conditions.

Note: This method is not perfect. In practice, the authors improved model performance by including a small number of manually crafted examples to guide the learning process.

Key Takeaway: Verifying a candidate Lyapunov function is relatively straightforward once it is given, but discovering one that works for a given system is highly nontrivial. This characteristic makes the problem well-suited for AI, which can explore vast spaces of candidate functions efficiently.

1.13 Mathematical Data

For machine learning to effectively work with mathematical problems, we must carefully translate abstract mathematical concepts into a format that ML models can process. Unlike natural language or image data, mathematical structures require precise encoding to preserve logical relationships and computational properties.

Throughout this course, we will explore different ways machine learning represents and processes mathematical objects. As we do so, consider these key questions:

- **How should mathematical data be structured for machine learning?** Different mathematical objects (graphs, equations, functions, theorems) require different representations. Should they be encoded as sequences, trees, tensors, or some other format?
- **How can we generate labeled data for problems where manual annotation is impractical?** Many mathematical problems do not have large pre-existing datasets. What strategies can be used to generate high-quality training examples?
- **What does it mean to generate mathematical data "randomly"?** In mathematics, "randomness" is often structured (e.g., uniform sampling, Gaussian distributions, combinatorial choices). How do we define randomness in a way that is meaningful for AI models?

These questions are fundamental to the intersection of mathematics and AI. Understanding how to effectively structure and generate mathematical data can lead to more powerful and insightful machine learning applications.

2 Lecture 2

2.1 Machine Learning

Machine learning (ML) is the study of algorithms that improve automatically through experience. It is particularly useful when we need to solve a problem with software but do not know

how to explicitly define the solution. For example, consider the task of developing a system that classifies mathematical objects based on their properties. While traditional programming may struggle with this, ML can identify hidden patterns in algebraic structures, geometric objects, or number sequences.

In machine learning, it is common practice to structure input data as a matrix, where each row represents an individual data point (such as a mathematical object, a theorem, or a set of equations), and each column corresponds to a specific feature describing that data point.

This format is not inherent to the data itself, but rather reflects the requirements of many traditional machine learning algorithms (though not, for example, large language models). Often, this necessitates a preprocessing step where raw or structured data is transformed into this tabular format. In the context of mathematics, features might include:

- Properties of an algebraic structure (e.g., finiteness, commutativity, order).
- Characteristics of a function (e.g., continuity, differentiability, convexity).
- Attributes of a graph (e.g., number of vertices, connectivity, diameter).

In order to fit such data into a standard machine learning pipeline, these features typically need to be encoded as numerical values. This might involve binary indicators (e.g., whether a group is abelian), categorical encoding(order of a group as a number), or real-valued measurements. Choosing how to represent these features numerically is an important part of the modeling process and often requires mathematical insight.

The output of an ML model depends on the type of problem being solved and can take various forms:

- A numerical value (e.g., predicting the determinant of a matrix in regression tasks).
- A class label (e.g., determining whether a given group is cyclic in classification tasks).
- A sequence (e.g., generating a mathematical proof step-by-step).
- A complex structure (e.g., constructing counterexamples to conjectures).

Supervised and Unsupervised Learning Machine learning algorithms can generally be categorized into two main types: **supervised learning** and **unsupervised learning**.

Supervised learning involves training a model on labeled data, where each input x has a corresponding output y. The goal is to learn a mapping $f(x)$ that generalizes well to unseen data. Examples include **regression** (predicting numerical values) and **classification** (assigning class labels to inputs).

Unsupervised learning, on the other hand, deals with datasets where no explicit labels are provided. The algorithm must uncover hidden structures or patterns within the data. Common approaches include **clustering** (grouping similar data points) and **dimensionality reduction** (simplifying data while preserving key information).

Question: Why do we need unsupervised learning if supervised learning is often more accurate?

Answer: Unsupervised learning can be used when there is no target variable to predict, or it is unclear how to select it. It is valuable for exploratory analysis and feature engineering. For example, clustering can reveal natural groupings in the data, even when no labels are available. Unlike supervised learning, the goal is not prediction, but understanding and organizing the data itself.

Regression is the task of predicting a continuous numerical value based on input features. In a mathematical setting, regression models can be used to estimate the roots of a polynomial function given its coefficients or to approximate the convergence rate of a numerical method.

Classification involves assigning data points to discrete categories. In mathematics, a classification model might determine whether a given number is prime or composite, or categorize geometric shapes based on their symmetry properties.

Both regression and classification rely on learning patterns from data. While regression focuses on numerical predictions, classification deals with categorical decisions.

2.2 Linear Regression

Linear regression is a fundamental supervised learning technique used to model the relationship between input features and a continuous target variable. It assumes that the target variable is a **linear combination** of the input features, plus some noise.

To formalize this, we consider a dataset consisting of m training examples, each with n input features. The **input data matrix X** is structured as:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n}, \tag{1}$$

where each row represents a **training example** $x_i \in \mathbb{R}^n$, and each column corresponds to a **feature**. The **target variable** (or output) is represented as the **response vector**:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \in \mathbb{R}^m. \tag{2}$$

Each entry y_i is a real-valued output corresponding to the input vector x_i.

2.2.1 Why Are X and y Represented as Matrices?

In machine learning, data is commonly represented in **matrix form** to enable efficient computations. Working with matrices allows:

- Fast calculations using **matrix-vector multiplication**.
- The use of powerful **linear algebra techniques** for optimization.

2.2.2 The Linear Regression Model

The goal of linear regression is to find a **weight vector** $\mathbf{w} \in \mathbb{R}^n$ such that the predicted values:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} \tag{3}$$

are as close as possible to the true target values $\mathbf{y}$. The difference between $\hat{\mathbf{y}}$ and $\mathbf{y}$ represents the **error** in the model's predictions.

Question: If $\mathbf{w}$ were a vector of all zeros, what would be the predicted outputs for any input $\mathbf{X}$?

Answer: If $\mathbf{w}$ is the zero vector, then $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} = 0$ for all inputs. This means the model would predict zero regardless of the input features, which is clearly not useful unless all target values in $\mathbf{y}$ are actually zero.

2.2.3 Optimizing the Model: Minimizing the Error

To measure how well the model fits the data, we use a **loss function** that quantifies the difference between the predicted and actual values. One common choice is the **Mean Squared Error (MSE)**:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2. \tag{4}$$

Minimizing this loss ensures that our model makes accurate predictions. This leads to an **optimization problem**, where we seek the optimal $\mathbf{w}$ that minimizes $\mathcal{L}(\mathbf{w})$.

Question: What does it mean if the loss function is exactly zero?

Answer: If the loss function is exactly zero, it means that for every training example, the predicted value $\hat{y}_i$ exactly matches the actual value y_i. This typically happens when the

dependency is perfectly linear, or if the model has too many parameters, potentially leading to **overfitting**.

2.2.4 Finding the Optimal Weights

In linear regression, the goal is to find the optimal weight vector $\mathbf{w}$ that minimizes the loss function. The most common loss function is the **mean squared error (MSE)**:

$$L(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2, \tag{5}$$

where $\mathbf{X} \in \mathbb{R}^{m \times n}$ is the design matrix, $\mathbf{w} \in \mathbb{R}^n$ is the weight vector, and $\mathbf{y} \in \mathbb{R}^m$ is the vector of target values.

To minimize the loss, we take the gradient with respect to $\mathbf{w}$ and set it to zero:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \nabla_{\mathbf{w}} \left(\frac{1}{2}(\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y}) \right) \tag{6}$$

$$= \mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}). \tag{7}$$

Setting the gradient to zero gives the **normal equation**:

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{y}. \tag{8}$$

Solving for $\mathbf{w}$ yields the **closed-form solution**:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{9}$$

This formula provides an exact solution without the need for iterative optimization. However, computing the inverse of $\mathbf{X}^T\mathbf{X}$ is not only **computationally expensive** (roughly $\mathcal{O}(n^3)$), but can also lead to **numerical instability** if the matrix is close to singular.

Question: What problems arise if $\mathbf{X}^T\mathbf{X}$ is not invertible or ill-conditioned?

Answer: If $\mathbf{X}^T\mathbf{X}$ is **singular** (i.e., not invertible), the closed-form solution cannot be computed directly. This happens, for example, when:

- Some features are **linearly dependent** — for instance, one column of $\mathbf{X}$ is an exact linear combination of others.
- The number of features n exceeds the number of training samples m, leading to an underdetermined system with infinitely many solutions.

In such cases, the best remedy is to **remove redundant features**, especially when the dependencies are exact.

Even when $\mathbf{X}^T\mathbf{X}$ is technically invertible, it may be **ill-conditioned** — meaning that its smallest eigenvalues are close to zero. Inverting such matrices can amplify numerical errors and lead to unstable, non-generalizable solutions. We will address this problem with more details a little later.

2.3 Different Loss Functions in Linear Regression

In linear regression, the most common loss function is the **Mean Squared Error (MSE)**:

$$\text{MSE}(y, \hat{y}) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2. \tag{10}$$

Here, y is a true label vector and $\hat{y}$ is a prediction vector. MSE is widely used due to its **smooth differentiability**, which makes it convenient for optimization methods such as **gradient descent**. However, MSE is not the only possible choice. Other loss functions may be preferable in cases where robustness to **outliers, interpretability, or scale sensitivity** is required.

Question: Why is smooth differentiability an important property for a loss function?

Answer: Smooth differentiability ensures that the function has well-defined gradients, allowing optimization algorithms like gradient descent to efficiently update model parameters.

2.3.1 Mean Absolute Error (MAE)

A common alternative is the **Mean Absolute Error (MAE)**:

$$\text{MAE}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|. \tag{11}$$

Unlike MSE, which squares the residuals, MAE **penalizes all errors linearly**. This makes it **less sensitive to large outliers**, since extreme errors contribute less to the total loss. Another key difference is in the statistical interpretation:

1. MSE estimates the **mean** of the conditional distribution $p(y|x)$.
2. MAE estimates the **median** of the conditional distribution.

Thus, if the data contains many **outliers**, MAE may provide more robust predictions.

2.3.2 Mean Absolute Percentage Error (MAPE)

Another useful loss function, particularly in **forecasting tasks**, is the **Mean Absolute Percentage Error (MAPE)**:

$$\text{MAPE}(y, \hat{y}) = \frac{100}{N} \sum_{i=1}^{N} \left| \frac{\hat{y}_i - y_i}{y_i} \right|. \tag{12}$$

MAPE expresses errors **as a percentage** of the true values, which makes it ideal for applications where absolute error magnitudes vary greatly across different examples.

One drawback of MAPE is that it can become problematic when y_i values are close to zero, leading to **very large relative errors**.

Question: Why might MAPE be unsuitable for datasets with values close to zero?

Answer: When y_i is very small, the denominator in MAPE becomes close to zero, making the relative error explode. This can lead to misleadingly high loss values, which distort model performance evaluation.

2.3.3 Huber Loss: A Smooth Hybrid

While MSE is sensitive to **outliers** and MAE lacks differentiability at zero, the **Huber loss** provides a balance between the two:

$$L(f, X, y) = \sum_{i=1}^{N} h_\delta(y_i - \langle w, x_i \rangle), \tag{13}$$

where the function $h_\delta(z)$ is defined as:

$$h_\delta(z) = \begin{cases} \frac{1}{2} z^2, & \text{if } |z| \leq \delta, \\ \delta(|z| - \frac{1}{2}\delta), & \text{if } |z| > \delta. \end{cases} \tag{14}$$

The parameter δ is a **hyperparameter** that determines the threshold at which the loss function transitions between MSE-like behavior (for small errors) and MAE-like behavior (for large errors).

Why Use Huber Loss? - For small errors ($|z| \leq \delta$), it behaves like **MSE**, ensuring smooth optimization. - For large errors ($|z| > \delta$), it **reduces penalization**, preventing outliers from dominating the loss.

Huber loss is widely used in **robust regression tasks**, particularly in cases where **some noise is expected**, but extreme outliers should not dominate training.

Question: What happens if we set δ to be very large in Huber loss?

Answer: If δ is very large, the loss function behaves almost entirely like MSE because all errors will fall into the quadratic region ($|z| \leq \delta$). This reduces the robustness advantage of Huber loss against outliers.

2.3.4 Choosing the Right Loss Function

The choice of loss function depends on the specific **characteristics of the dataset** and the desired behavior of the model:

- **MSE** (Mean Squared Error) penalizes large errors more heavily, making it useful when minimizing large deviations is important — but it is **very sensitive to outliers**.
- **MAE** (Mean Absolute Error) is more **robust to outliers**, but results in a **less smooth optimization landscape** due to its non-differentiability at zero. While MAE can be optimized using subgradients or smoothed approximations, this is rarely done directly with gradient descent. In practice, **proximal methods** or specific solvers are used instead (see, e.g., Parikh & Boyd, 2014).
  - As a side note: even though MAE has desirable robustness properties, it's not widely used in practice — for example, `scikit-learn` does not provide a built-in MAE-based regressor. Most real-world applications default to MSE-based models.
  - This choice of loss function can also influence other parts of the model pipeline — for instance, L1-regularized regression (Lasso) with an MSE loss tends to produce sparse solutions *only if* optimization is done carefully. Naively applying gradient descent may not zero out as many coefficients as desired.
- **MAPE** (Mean Absolute Percentage Error) is useful when **relative errors** matter more than absolute differences — e.g., predicting growth rates or percentages. However, it can behave poorly when true values are close to zero.
- **Huber Loss** combines the benefits of MSE and MAE: it is **quadratic for small errors** (like MSE) and **linear for large errors** (like MAE), making it a solid default in the presence of noisy data and occasional outliers.

2.4 Representing the Intercept as a Weight

In many implementations of **linear regression**, the intercept term w_0 is treated separately. However, we can incorporate it into the weight vector by introducing an additional **dummy feature** that always takes the value 1. This allows us to represent the model in a more **uniform** and **compact** way.

The standard linear regression model is given by:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + \epsilon. \tag{15}$$

Question: Why is w_0 needed in linear regression?

Answer: The intercept w_0 ensures that the model can fit data that does not pass through the origin. Without it, the model would be forced to predict $y = 0$ when all features x_i are zero, which may not be appropriate in many cases.

2.4.1 Augmenting the Feature Matrix

To simplify notation and avoid handling w_0 separately, we define an **augmented feature** $x_0 = 1$, allowing us to rewrite the equation as:

$$y = w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + \epsilon. \tag{16}$$

This transformation allows us to express the model more compactly in **vectorized form**. We define:

- The **augmented weight vector** $\mathbf{w}'$:

$$\mathbf{w}' = (w_0, w_1, \ldots, w_n). \tag{17}$$

- The **augmented feature matrix** $\mathbf{X}'$, which now includes a column of ones:

$$\mathbf{X}' = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1n} \\ 1 & x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix}. \tag{18}$$

Now, the linear regression model simplifies to:

$$\mathbf{y} = \mathbf{X}'\mathbf{w}' + \boldsymbol{\epsilon}, \tag{19}$$

where:

- $\mathbf{y}$ is the vector of target values,
- $\mathbf{X}'$ is the **augmented design matrix** including the column of ones for x_0,
- $\mathbf{w}'$ is the **augmented weight vector**, now including the intercept w_0,
- $\boldsymbol{\epsilon}$ is the vector of error terms.

Question: Why do we add a column of ones to the feature matrix instead of handling w_0 separately?

Answer: By adding a column of ones to the design matrix $\mathbf{X}$, we obtain an augmented matrix $\mathbf{X}'$ that allows the entire linear regression model to be written as a single **matrix multiplication**: $\mathbf{X}'\mathbf{w}'$, where $\mathbf{w}'$ includes both the original weights and the intercept. This formulation simplifies the mathematics in linear regression. However, in practice, all of popular libraries treat the intercept separately and do not modify the input matrix explicitly.

2.5 Polynomial Regression

Polynomial regression extends linear regression by introducing higher-degree terms of input features. Instead of assuming a strictly linear relationship, we model the output as a **polynomial function** of the input:

$$\hat{y} = w_0 + w_1 x + w_2 x^2 + \cdots + w_d x^d, \tag{20}$$

where d is the **polynomial degree**, and ϵ represents the **error term**.

Question: How does polynomial regression differ from standard linear regression?

Answer: Unlike standard linear regression, which assumes a linear relationship between the input and output, polynomial regression allows for **non-linear relationships** by introducing higher-degree terms of the input features.

2.5.1 Transforming Linear Regression into Polynomial Regression

Polynomial regression can be seen as a form of linear regression where we **extend the feature space** by introducing new features derived from existing ones. Given an original feature x, we construct additional features:

$$x_1 = x, \quad x_2 = x^2, \quad x_3 = x^3, \quad \dots, \quad x_d = x^d. \tag{21}$$

We then apply **standard linear regression** on these transformed features:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d. \tag{22}$$

Thus, polynomial regression remains **linear in the parameters** w_i, making it a special case of linear regression after feature transformation.

2.5.2 Polynomial Regression with Multiple Features

Polynomial regression can be extended to multiple features. Suppose we have two input features, x_1 and x_2, we can construct polynomial terms such as:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_1 x_2 + w_5 x_2^2 + \cdots + . \tag{23}$$

This model includes all possible polynomial combinations up to a given degree.

Question: What is the advantage of including interaction terms like $x_1 x_2$ in polynomial regression?

Answer: Interaction terms allow the model to capture relationships between different features, not just their individual effects. This can be useful when the influence of one feature depends on the value of another.

2.6 Underfitting and Overfitting

When training **machine learning models**, a key challenge is balancing **model complexity** to ensure good **generalization** to unseen data. Two common problems that arise are **underfitting** and **overfitting**.

2.6.1 Underfitting

Underfitting occurs when a model is too simple to capture the underlying structure of the data. This happens when the model has **high bias**, meaning it makes overly simplistic assumptions about the data and performs poorly on both training and test sets.

For example, fitting a **linear model** to data that follows a quadratic relationship:

$$y = x^2 + \text{noise} \tag{24}$$

would lead to poor predictions, as a straight-line model does not have the flexibility to represent the actual pattern.

Ways to address underfitting:

- **Increase model complexity**: Use a more expressive model, such as **polynomial regression** instead of linear regression.
- **Add more relevant features**: If important features are missing, the model may fail to learn the true relationships in the data.
- **Reduce regularization**: (**Discussed later**) Strong regularization (such as high values of λ in Ridge or Lasso regression) can overly constrain the model and prevent it from capturing meaningful patterns.
- **Improve optimization techniques**: In many cases, we solve optimization problems approximately using iterative methods. If the optimization is not properly tuned — for example, due to poor choice of hyperparameters or early stopping — the resulting solution may be far from optimal, leading to **underfitting** even when the model itself is sufficiently expressive.

2.6.2 Overfitting

Overfitting occurs when a model is too complex and learns not only the underlying pattern but also the **noise** in the training data. This leads to **high variance**, meaning the model performs well on training data but generalizes poorly to new data.

For example, fitting a **high-degree polynomial** to data might result in a model that perfectly fits the training set but produces unrealistic oscillations in between data points:

$$\hat{y} = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \cdots + w_d x^d \tag{25}$$

where d is too high, capturing unnecessary variations that do not generalize well.

Question: Why does a high-degree polynomial tend to overfit data?

Answer: A high-degree polynomial has enough flexibility to fit the training data almost perfectly, including capturing noise as if it were part of the true pattern. However, this results in a model that lacks generalization and performs poorly on new data.

Ways to address overfitting:

- **Reduce model complexity**: Use a simpler model with fewer parameters, such as a lower-degree polynomial.
- **Use regularization**: Techniques like **Ridge regression** (L2 regularization) and **Lasso regression** (L1 regularization) penalize large coefficients and prevent overfitting.
- **Increase training data**: More training examples help the model generalize better by reducing reliance on specific noisy patterns.
- **Feature selection**: Removing irrelevant or redundant features can reduce overfitting and improve model performance.

Question: Why is it difficult to achieve a perfect balance between underfitting and overfitting?

Answer: There is no universal rule for finding the right complexity of a model. It depends on factors such as the size of the dataset, the amount of noise, and the underlying patterns in the data. Techniques like **cross-validation** and **regularization** help find a suitable balance.

Final Thought: The key to good generalization is to control model complexity and use **appropriate techniques** (like regularization) to ensure the model performs well on both training and unseen data.

2.7 Regularization: Beyond Overfitting Prevention

Regularization is a fundamental technique in regression problems, often introduced to **prevent overfitting**. However, regularization serves an additional crucial role: it helps address **multicollinearity** in the solution.

In a standard linear regression problem, the optimal weight vector $\mathbf{w}$ is found by solving the normal equation:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{26}$$

However, this solution is not always well-defined. If the feature matrix $\mathbf{X}$ contains **linearly dependent columns**, the matrix $\mathbf{X}^T\mathbf{X}$ becomes **singular**, meaning that its inverse does not exist. Handling this case can be relatively easy, since if we have linear dependancy of features it is possible to delete some of them and make the matrix invertable.

But even if the features are only approximately dependent (**multicollinearity**), the matrix is close to singular, and some of its eigenvalues are close to zero. This creates several problems:

- Small numerical errors in $\mathbf{X}$ get **greatly amplified** in $\mathbf{w}$, leading to large, unstable weight values.
- Predictions $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$ can become extremely sensitive to noise in the data.

Question: Why does multicollinearity make the regression solution unstable?

Answer: When two or more features are highly correlated, the matrix $\mathbf{X}^T\mathbf{X}$ becomes **ill-conditioned** — its eigenvalues are very small or nearly zero.

In the closed-form solution for linear regression:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y},$$

the inverse of $\mathbf{X}^T\mathbf{X}$ may amplify noise in the data. Small perturbations in $\mathbf{y}$ or $\mathbf{X}$ can lead to large changes in $\mathbf{w}$, resulting in unstable solutions and poor generalization.

This problem becomes clearer when we write $\mathbf{X}^T\mathbf{X}$ in terms of its eigen-decomposition:

$$\mathbf{X}^T\mathbf{X} = \mathbf{Q}\,\mathbf{Q}^T,$$

where contains the eigenvalues λ_i. The inverse then becomes:

$$(\mathbf{X}^T\mathbf{X})^{-1} = \mathbf{Q}^{-1}\mathbf{Q}^T.$$

If some λ_i are close to zero, the corresponding entries in $^{-1}$ become very large — amplifying noise in those directions.

To mitigate these issues, we introduce **regularization**, such as in Ridge regression, which adds a multiple of the identity matrix:

$$\mathbf{w}_{\text{ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}.$$

This shifts all eigenvalues by λ, improving the conditioning of the matrix and stabilizing the solution. Regularization effectively controls the magnitude of the weight vector and suppresses the influence of unstable directions.

2.7.1 Ridge Regression and Numerical Stability

Ridge regression modifies the standard least squares loss by adding an **L2 penalty**:

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \lambda\sum_{j=1}^{n} w_j^2. \tag{27}$$

This leads to the following closed-form solution:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}. \tag{28}$$

Even though $\mathbf{X}^T\mathbf{X}$ is typically invertible (if $\mathbf{X}$ has full rank), it can be **ill-conditioned**, meaning that small changes in the data may lead to large fluctuations in the solution. The addition of $\lambda\mathbf{I}$ improves the **conditioning** of the matrix and stabilizes the inversion.

Ridge regression helps in the following ways:

- Improves numerical stability by reducing sensitivity to small eigenvalues of $\mathbf{X}^T\mathbf{X}$.
- Avoids extreme weight values by shrinking coefficients, making predictions more **robust to noise and multicollinearity**.

The **regularization parameter** λ controls the trade-off: - A **small** λ has little effect and behaves like ordinary least squares. - A **large** λ shrinks weights significantly, potentially leading to underfitting.

Question: Why does Ridge regression help when features are highly correlated?

Answer: Ridge regression reduces sensitivity to correlated features by distributing weights more evenly. The penalty term discourages extreme weight values, stabilizing the solution.

2.7.2 Lasso Regression and Feature Selection

Lasso regression introduces an **L1 penalty** instead of L2:

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \lambda\sum_{j=1}^{n} |w_j|. \tag{29}$$

Unlike Ridge, which shrinks all coefficients continuously, Lasso has the unique property of **setting some weights exactly to zero**, effectively performing **automatic feature selection**. This makes it useful when:

- The dataset contains many irrelevant or redundant features.
- We prefer an interpretable model with fewer nonzero weights.

Question: Why does Lasso regression produce sparse models?

Answer: The L1 penalty has a geometric property that encourages sparsity. The constraint region defined by the L1 norm has sharp corners (e.g., in 2D, it forms a diamond shape), which increases the chance that the optimal solution lies exactly on an axis — meaning some coefficients are exactly zero. As a result, Lasso tends to produce **sparse solutions**, especially when the regularization parameter λ is large enough.

2.7.3 Choosing the Regularization Parameter

The value of λ is critical to model performance and must be chosen carefully. Common methods include:

- **Cross-validation**: The most widely used approach, where different values of λ are tested on a validation set, selecting the one that minimizes prediction error.
- **Grid search**: Manually testing a predefined range of λ values to find the best-performing one.

Question: What happens if λ is chosen too large?
Answer: A large λ shrinks all coefficients significantly, leading to an underfitting model that fails to capture important patterns in the data.

2.7.4 Conclusion

Regularization serves **more purposes than just preventing overfitting**. It is essential for **improving numerical stability**. While Ridge regression ensures a stable solution by controlling large weights, Lasso aids in feature selection. The choice of regularization technique and parameter tuning is critical for achieving both **good predictive performance and model reliability**.

2.8 Some Problems with Linear Regression

While **linear regression** is effective for modeling simple relationships, real-world data often involves more complex patterns. A key limitation is that we must **manually decide which interactions or transformations** of the features to include in the model. This can be problematic when the true structure of the data is unknown or highly nonlinear.

Question: What happens when we assume a linear relationship in a complex dataset?
Answer: If the true relationship is nonlinear, a linear model will underfit the data, resulting in poor predictive performance.

To overcome this, more flexible **regression models** can automatically capture complex patterns — and later in the course, we'll see how **neural networks** can be trained to learn highly intricate dependencies directly from data!

3 Lecture 3. Classification tasks. Gradient descent

3.1 Linear classification

Now, let's switch to the *classification problem*. First, let's talk about binary classification, i. e. classification between two classes. It's relatively easy to generalize the problem for K classes. Let our targets encode positive or negative classes, that is, $\{-1, 1\}$. We need to train our linear model so that some plane separates the objects of the classes from one another in the best possible way.

Ideally, we should find a plane that could split the classes: the positive class is on its one side, and the negative class is on the other. The dataset for which this is possible is referred to as a **linearly separable** one. Unfortunately, in reality, this happens pretty rarely.

Later we'll ponder on how to train a linear classification model, but for now it is already clear that to get a final prediction, we should use the formula

$$y = \operatorname{sgn} \langle \mathbf{w}, \mathbf{x}_i \rangle.$$

We want to minimize the number of classifier errors, that is:

$$\sum_i \mathbb{I}[y_i \neq \operatorname{sgn} \langle \mathbf{w}, \mathbf{x}_i \rangle] \to \min.$$

Multiply both parts by y_i and simplify the optimization problem:

$$\sum_i \mathbb{I}[y_i \langle \mathbf{w}, \mathbf{x}_i \rangle < 0] \to \min .$$

The value of $M = y_i \langle \mathbf{w}, \mathbf{x}_i \rangle$ is referred to as the classifier's **margin**. Such a loss function is referred to as a **misclassification loss**. It's easy to see that:

— The margin is positive when $\operatorname{sgn}(y_i) = \operatorname{sgn}(\langle \mathbf{w}, \mathbf{x}_i \rangle)$ (that is, the class has been guessed correctly); note that the higher the margin, the higher the distance between $\mathbf{x}_i$ and the separating hyperplane (that is, the confidence of the classifier).

— The margin is negative when $\operatorname{sgn}(y_i) \neq \operatorname{sgn}(\langle \mathbf{w}, \mathbf{x}_i \rangle)$ (that is, the class is guessed incorrectly); note that the higher the absolute value of the margin, the more devastating the classifier's error.

For each of the margins, we calculate the function

$$F(M) = \mathbb{I}[M < 0] = \begin{cases} 1, & M < 0, \\ 0, & M \geqslant 0. \end{cases}$$

This is a step constant function. That's why we can't use gradient methods to optimize this entire sum: the derivative of this function equals zero at every point where it exists. However, we can use a smoother function for it as an upper estimate and make the problem solvable. We can use different functions as upper estimates, each with its upsides and downsides.

3.2 Maximum likelihood method

Let us have a sample $X_1, \ldots, X_n$ with the same distribution depending of some parameter θ. The **likelihood** (**likelihood function**) of a sample $X_1, \ldots, X_n$ is simply its joint pmf (probability function) or pdf (probability density function). Regardless of the type of distribution, we will denote likelihood as

$$\mathcal{L}(\theta) := p(X_1, \ldots, X_n \mid \theta).$$

If the sample is i.i.d., then the likelihood function decomposes into a product of one-dimensional functions:

$$\mathcal{L}(X_1, \ldots, X_n \mid \theta) = \prod_{i=1}^{n} p(X_i \mid \theta).$$

Maximum likelihood estimation (**MLE**) maximizes likelihood:

$$\hat{\theta}_{ML} = \arg\max_{\theta} \mathcal{L}(\theta).$$

Since it is easier to maximize the sum than the product, we usually switch to the logarithm of likelihood (log-likelihood). This is especially convenient in the case of i.i.d. sampling, then

$$\hat{\theta}_{ML} = \arg\max_{\theta} \log \mathcal{L}(\theta) = \arg\max_{\theta} \sum_{i=1}^{n} \log p(X_i \mid \theta).$$

Example 3.1. *As a result of coin tosses, k "heads" and $n-k$ "tails" fell out. Let's estimate the probability of the "head" falling out by the maximum likelihood method.*

Solution. *Let p be the probability of "head" falling out, then the likelihood is*

$$\mathcal{L}(p) = p^k (1-p)^{n-k}.$$

Differentiating the logarithm of likelihood

$$\log \mathcal{L}(p) = k \log p + (n-k) \log(1-p)$$

and equating the derivative to zero, we find

$$\frac{k}{p} = \frac{n-k}{1-p} \Longleftrightarrow k(1-p) = (n-k)p \Longleftrightarrow p = \frac{k}{n}.$$

It is not difficult to make sure that this is the maximum point. So, the estimate of the maximum likelihood $\hat{p}_{ML} = \frac{k}{n}$ of the probability of "success" in the Bernoulli scheme quite naturally turned out to be equal to the share of "success" in a series of n tests.

Example 3.2. *Let's find the MLE estimates of parameters of the distribution $\mathcal{N}(\mu, \tau)$ for the i.i.d. sample $X_1, \ldots, X_n$.*

Solution. *Let 's write down the likelihood:*

$$\mathcal{L}(\mu, \tau) = \prod_{k=1}^{n} \frac{1}{\sqrt{2\pi\tau}} \exp \frac{-(X_k - \mu)^2}{2\tau}.$$

Let's move on to log-likelihood:

$$\log \mathcal{L}(\mu, \tau) = -\frac{n}{2}(\log \tau + \ln 2\pi) - \frac{1}{2\tau} \sum_{k=1}^{n} (X_k - \mu)^2.$$

Equate the partial derivatives by μ and τ to zero:

$$\frac{\partial \log \mathcal{L}}{\partial \mu} = \frac{1}{\tau} \sum_{k=1}^{n} (X_k - \mu) = 0,$$

$$\frac{\partial \log \mathcal{L}}{\partial \tau} = -\frac{n}{\tau} + \frac{1}{\tau^2} \sum_{k=1}^{n} (X_k - \mu)^2 = 0.$$

Therefore $\hat{\mu}_{ML} = \overline{X}_n$ is a sample mean,

$$\hat{\tau}_{ML} = \frac{1}{n} \sum_{k=1}^{n} X_k^2 - (\overline{X}_n)^2$$

is a sample variance.

Example 3.3. *Let i.i.d. sample $X_1, \ldots, X_n \sim U[a; b]$. Find maximum likelihood estimates for the parameters a and b.*

Solution. *Likelihood here has the form*

$$L(X_1, \ldots, X_n \mid a, b) = \frac{1}{(b-a)^n} \prod_{k=1}^{n} \mathbb{I}(X_k \in [a, b]).$$

If X-s and b are fixed this expression is maximal for $a = X_{(1)}$, because if we take a little bigger number, then the product of indicators becomes zero; if we take a little smaller number, then likelihood becomes smaller because of increasing $(b-a)^n$. By the same arguments $\widehat{b}_{ML} = X_{(n)}$.

Exercise 3.4. *Let the i.i.d. sample $X_1, \ldots, X_n$ is taken from a Poisson distribution with parameter λ. Find the maximum likelihood estimate of λ.*

3.3 Logistic regression

In this section, we'll denote classes as 0 and 1.

Another interesting method emerges from the desire to see classification as predicting probabilities. A good example is click prediction (for example, in advertising and search). If our training log includes a click, it doesn't necessarily mean that if we fully repeat the experiment conditions, the user will click on the item again. More likely, items can be assigned some clickability (that is, a true probability of a click on an item). An event of a click logged for each training object is a realization of this random variable, and we may believe that at each point, the ratio between the positive and negative cases should converge to such a probability.

The challenge is that the probability is, by definition, a value between 0 and 1, and there's no way to train a linear model to meet this constraint. A way out is to train our linear model to correctly predict some object that depends on the probability but has a value range $(-\infty, +\infty)$ and then convert predictions of the model into probabilities. As such an object,

we can use **logit** or **log odds**: a logarithm of the ratio of the probability of the positive and negative event: $\log\left(\frac{p}{1-p}\right)$.

If the prediction of our model is $\log\left(\frac{p}{1-p}\right)$, it's easy to calculate the resulting probability:

$$\langle \mathbf{w}, \mathbf{x}_i \rangle = \log\left(\frac{p}{1-p}\right);$$
$$e^{\langle \mathbf{w}, \mathbf{x}_i \rangle} = \frac{p}{1-p};$$
$$p = \frac{1}{1 + e^{-\langle \mathbf{w}, \mathbf{x}_i \rangle}}.$$

The function on the right side is referred to as the **sigmoid function**, denoted as

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

Therefore, $p = \sigma(\langle \mathbf{w}, \mathbf{x}_i \rangle)$.

How can we optimize $\mathbf{w}$ so that the model can predict logits in the best possible way? Here's where the maximum likelihood method for the Bernoull distribution comes in handy. This is, for example, the distribution of tossing a coin where the tails probability is p. In our case, instead of tails, we have the probability that the user clicked the item.

By evaluating the likelihood, we can see the probability of arriving at given values of the target $\mathbf{y}$ at the given $\mathbf{X}$ and weights of $\mathbf{w}$. The likelihood is calculated by the formula:

$$p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) = \prod_i p(\mathbf{y}_i \mid \mathbf{x}_i, \mathbf{w})$$

and for the Bernoulli distribution, it can be written as:

$$p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) = \prod_i p_i^{y_i} (1-p_i)^{1-y_i},$$

where p_i is the probability calculated from the model predictions. Optimizing products isn't really convenient, it's easier to deal with sums. So, convert the likelihood logarithmically and plug in the probability from above:

$$\ell(\mathbf{w}, \mathbf{X}, \mathbf{y}) = \sum_i (y_i \log(p_i) + (1-y_i)\log(1-p_i)) =$$
$$= \sum_i (y_i \log(\sigma(\langle \mathbf{w}, \mathbf{x}_i \rangle)) + (1-y_i)\log(1-\sigma(\langle \mathbf{w}, \mathbf{x}_i \rangle))).$$

If we notice that

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

We can simplify the expression

$$\ell(\mathbf{w}, \mathbf{X}, \mathbf{y}) = \sum_i (y_i \log(\sigma(\langle \mathbf{w}, \mathbf{x}_i \rangle)) + (1-y_i)\log(\sigma(-\langle \mathbf{w}, \mathbf{x}_i \rangle)))$$

We want to find $\mathbf{w}$ for which the likelihood is maximum. To get the loss to be minimized, multiply it by -1:

$$L(w, X, y) = -\sum_i (y_i \log(\sigma(\langle \mathbf{w}, \mathbf{x}_i \rangle)) + (1-y_i)\log(\sigma(-\langle \mathbf{w}, \mathbf{x}_i \rangle)))$$

In contrast to the linear regression, we have no explicit formula for solving the logistic regression problem. Now, let's use gradient descent. Fortunately, the gradient is really simple:

$$\nabla_{\mathbf{w}} L(\mathbf{w}, \mathbf{X}, \mathbf{y}) = -\sum_i \mathbf{x}_i (y_i - \sigma(\langle \mathbf{w}, \mathbf{x}_i \rangle)).$$

As we have agreed, the model prediction will be calculated as

$$p = \sigma(\langle \mathbf{w}, \mathbf{x}_i \rangle).$$

This is the probability of the positive class, but how can we predict the class itself based on this? In other methods, it was sufficient for us to calculate the prediction sign, but now all our predictions are non-negative numbers between 0 and 1. So, what should we do? An intuitive (but not quite correct) approach is to "take a threshold of 0.5". It would be better to select this threshold individually, for a solved regression, minimizing the metric on a pre-existing test dataset. For example, we can make the ratio between the positive and negative classes near-to-real.

Let's note that the method is referred to as logistic regression rather than logistic classification because we are predicting real numbers (logits) instead of classes.

3.4 Gradient descent

The functional that we are minimizing is smooth and convex. This means that we can efficiently find its minimum using iterative gradient methods. Here, we'll just briefly dwell on the most basic approach.

As we know, the gradient of a function at a point shows its fastest increasing direction, and the anti-gradient (the vector opposing the gradient) shows its fastest decreasing direction. So, if we have some approximation for the optimal value of the parameter $\mathbf{w}$, we can improve it by calculating the gradient of the loss function at the point and then shifting the weights a little in the direction of the anti-gradient:

$$w_j \mapsto w_j - \alpha \frac{\partial}{\partial w_j} L(f_{\mathbf{w}}, \mathbf{X}, \mathbf{y}),$$

where α is the algorithm's parameter (**the learning rate**) that controls the step length in the direction of the anti-gradient. This algorithm is referred to as the **gradient descent**.

Let's now look at the gradient descent for our loss function

$$L(f_{\mathbf{w}}, \mathbf{X}, \mathbf{y}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2.$$

Let us calculate the gradient, i. e. the vector of partial derivatives

$$\frac{\partial L}{\partial w_i} = \frac{\partial}{\partial w_i} \left\| \begin{pmatrix} x_{1,1} & \dots & x_{1,D} \\ \dots & \dots & \dots \\ x_{N,1} & \dots & x_{N,D} \end{pmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_D \end{pmatrix} - \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} \right\|^2 =$$

$$= \frac{\partial}{\partial w_i} \left\| \begin{pmatrix} \sum\limits_{j=1}^{D} x_{1,j} w_j - y_1 \\ \vdots \\ \sum\limits_{j=1}^{D} x_{N,j} w_j - y_N \end{pmatrix} \right\|^2 = \frac{\partial}{\partial w_i} \left(\sum_{j=1}^{D} x_{1,j} w_j - y_1 \right)^2 + \dots + \frac{\partial}{\partial w_i} \left(\sum_{j=1}^{D} x_{N,j} w_j - y_N \right)^2 =$$

$$= \frac{\partial}{\partial w_i} \left(x_{1,i}^2 w_i^2 + \sum_{j \neq i} 2 x_{1,i} x_{1,j} w_i w_j - 2 x_{1,i} w_i y_1 \right) + \dots + \frac{\partial}{\partial w_i} (x_{N,i}^2 w_i^2 + \sum_{j \neq i} 2 x_{N,i} x_{N,j} w_i w_j - 2 x_{N,i} w_i y_N) =$$

$$= \left(2 x_{1,i} \sum_{j=1}^{D} x_{1,j} w_j - y_1 \right) + \dots + \left(2 x_{N,i} \sum_{j=1}^{D} x_{N,j} w_j - y_N \right) =$$

$$= 2 \left\langle (x_{1,i}, \dots, x_{N,i}), \left(\begin{pmatrix} x_{1,1} & \dots & x_{1,D} \\ \dots & \dots & \dots \\ x_{N,1} & \dots & x_{N,D} \end{pmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_D \end{pmatrix} - \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} \right) \right\rangle =$$

$$= 2 \langle (x_{1,i}, \dots, x_{N,i}), \mathbf{X}\mathbf{w} - \mathbf{y} \rangle,$$

therefore we see that

$$\nabla_{\mathbf{w}} L = \begin{pmatrix} \frac{\partial L}{\partial w_1} \\ \dots \\ \frac{\partial L}{\partial w_D} \end{pmatrix} = 2\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}).$$

Hence, by starting from some initial approximation, we can iterate to decrease the value of the function until it converges, at least theoretically, to a minimum. Generally, this should be a local minimum, but in this case (because of convexity), it's a global one.

Gradient descent algorithm

$\mathbf{w}$ = random-normal()
repeat S times:
  $f = \mathbf{X} \cdot \mathbf{w}$ // calculate the prediction
  $err = f - \mathbf{y}$ // calculate the error
  $grad = 2 \cdot \mathbf{X}^T \cdot (err)/n$ // calculate the gradient
  $\mathbf{w} = \mathbf{w} - \alpha \cdot grad$ // update the weights

There are some theoretical results on the convergence rates and guarantees for the gradient descent method, but we will only formulate few general remarks:

— Since the problem is convex, the selection of the initial point affects the convergence rate. However, the effect is not so strong. Hence, in practice, you can always start from zero, or any other point.

— The condition number of $\mathbf{X}$ strongly affects the convergence rate of the gradient descent. The higher the ellipsoid eccentricity of the loss function levels, the worse.

— The learning rate α also strongly affects the gradient descent. Strictly speaking, it's a hyper-parameter of the algorithm, and we may have to select it separately. Other hyper-parameters are the maximum number of iterations S and/or the tolerance threshold.

The computing complexity of the gradient descent method is $O(NDS)$, where, just as above, N is the size of the dataset and D is the number of features per object. Compare it with the estimate of $O(ND^2 + D^3)$ for a "naive" analytical computation.

3.5 Stochastic gradient descent

At each step of the gradient descent, we need to run a potentially costly operation of calculating the gradient across the entire dataset (the complexity is $O(ND)$). We might come up with an idea to replace the gradient by its evaluation on a smaller subset (also known as a **batch** or **minibatch**).

Namely, if the loss function is a sum over individual object-target pairs:

$$L(\mathbf{w}, \mathbf{X}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} L(\mathbf{w}, \mathbf{x}_i, y_i),$$

and the gradient is

$$\nabla_{\mathbf{w}} L(\mathbf{w}, \mathbf{X}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\mathbf{w}} L(\mathbf{w}, \mathbf{x}_i, y_i),$$

then we estimate

$$\nabla_{\mathbf{w}} L(\mathbf{w}, \mathbf{X}, y) \approx \frac{1}{B} \sum_{t=1}^{B} \nabla_{\mathbf{w}} L(\mathbf{w}, \mathbf{x}_{i_t}, y_{i_t})$$

on a certain subset of these pairs $(\mathbf{x}_{i_t}, y_{i_t})_{t=1}^{B}$. Note the multipliers $\frac{1}{N}$ and $\frac{1}{B}$ before the sums. Why do we need them? The full gradient $\nabla_{\mathbf{w}} L(\mathbf{w}, \mathbf{X}, \mathbf{y})$ can be taken as an average gradient across all the objects, that is, the estimate of a mathematical expectation of $\mathbb{E} \nabla_{\mathbf{w}} L(\mathbf{w}, \mathbf{X}, \mathbf{y})$. Then, naturally, the estimated mathematical expectation over the smaller subset will also be represented as an average gradient across the objects in this subset.

How to split the dataset into batches? We could have sampled them from the entire dataset at random. Now, add another parameter to our algorithm: the batch size of B. Iterate through B objects in the batch to calculate the gradient and update the model weights. The number of steps in the algorithm is defined by epochs, E. This is another hyperparameter. One epoch means that we have completely traversed through the dataset. Note that if the dataset is very large and the model is compact, we might not even need to complete the first

traversal. **To improve generalization and avoid overfitting to a specific order of the data, it is common practice to shuffle the dataset at the beginning of each epoch.** This ensures that each epoch presents the model with a different ordering of the data, which can lead to more robust training.

Algorithm:

$\mathbf{w} = \text{normal}(0,\ 1)$
repeat E times:
  for $i = B, i \leqslant n, i+ = B$
    $\mathbf{X}$-batch $= \mathbf{X}[i - B : i]$
    $\mathbf{y}$-batch $= \mathbf{y}[i - B : i]$
    $f = \mathbf{X}\text{-batch} \cdot \mathbf{w}$ // Calculate the prediction
    $err = f - \mathbf{y}\text{-batch}$ // Calculate the error
    $\text{grad} = 2 \cdot \mathbf{X}\text{-batch}^T \cdot (\text{err})$ // Calculate the gradient
    $\mathbf{w} = \mathbf{w} - \alpha \cdot \text{grad}$

The time complexity is $O(NDE)$. At the first glance, this complexity is the same as for the previous gradient descent. But now we have made N/B-times more steps and, consequently, much more model weight updates.

All in all, the difference between the algorithms can be represented like this:

The steps of the stochastic gradient descent are much more noisy, but they are much faster to calculate. Finally, they also converge to the optimal value because the mathematical expectation of the gradient estimated on the batch equals the gradient itself. At least, we can achieve convergence with well-selected learning rates in the case of a convex loss function.

3.6 Multiclass classification

Let each object from your selection belong to one out of K classes: $\mathbb{Y} = 1, \ldots, K$. To predict the classes by a linear model, we need to reduce the multiclass problem to a set of binary problems we know well. We'll discuss the two most popular methods of doing this: one-vs-all and all-vs-all.

3.6.1 One-versus-all

Let's train K-classes linear classifiers $b_1(x), \ldots, b_K(x)$ that evaluate membership in classes $1, \ldots, K$, respectively. For linear models, such classifiers have the format:

$$b_k(x) = \text{sgn}\left(\langle \mathbf{w}_k, \mathbf{x} \rangle + w_{0,k}\right).$$

We are going to train each classifier with the index k by the dataset $(\mathbf{x}_i, 2\mathbb{I}[y_i = k] - 1)_{i=1}^N$. In other words, we'll train the classifier to distinguish the k-th class from the others.

It's logical for the final classifier to output the class that corresponds to the most confident of the binary algorithms. We can measure confidence to a certain extent by using the linear function values.

$$a(x) = \arg\max_k (\langle \mathbf{w}_k, \mathbf{x} \rangle + w_{0,k}\rangle).$$

Let's see what we can gain for our dataset from this approach. Let's train three linear models to distinguish one class from others:

Now, let's compare the values of the linear functions:

and, for each point, select the class corresponding to the highest value (that is, the most "confident" classifier):

The issue with this approach is that each of the classifiers $b_1(x), \ldots, b_K(x)$ trains on its own dataset, and the values of the linear functions $\langle \mathbf{w}_k, \mathbf{x} \rangle + w_{0,k}$ or, to put it simply, the outputs of the classifiers, might have different orders of magnitude. Because of this, it doesn't make sense to compare them. It's also not always reasonable to normalize the weight vectors to the same scale: for example, in the case of the SVM, for normalized weights the norm is different, so they can't be used as the problem's solutions any longer.

3.6.2 All-versus-all

Let's train a set of C_k^2 classifiers $a_{ij}(x)$, $i, j = 1, \ldots, K$, $i \neq j$. For linear models, such classifiers are

$$b_k(x) = \operatorname{sgn}(\langle \mathbf{w}_k, \mathbf{x} \rangle + w_{0,k}\rangle).$$

We are going to set up the classifier $a_{ij}(x)$ by the subset $\mathbf{X}_{ij} \subset \mathbf{X}$ that only includes the objects of the classes i and j. In this manner, the classifier $a_{ij}(x)$ will, for each object, output either class i or j. That's how it will look for our dataset:

To classify some new object, feed it to each of the binary classifiers. Each of them will vote for its own class: as a prediction, take the class that has the highest vote:

$$a(x) = \arg\max_k \sum_{i=1}^{K} \sum_{j \neq i} \mathbb{I}[a_{ij}(\mathbf{x}) = k].$$

3.6.3 Multiclass logistic regression

We can directly generalize some methods of binary classification for multiple classes. Let's look at how to do this using logistic regression.

For a two-class logistic regression, we built a linear model:

$$b(x) = \langle \mathbf{w}, \mathbf{x} \rangle + w_0.$$

Then we convert it to a probabilistic prediction using the sigmoid function

$$\sigma(z) = \frac{1}{1 + \exp(-z)}.$$

Say now we are solving a multiclass problem by building K linear models:

$$b_k(x) = \langle \mathbf{w}_k, \mathbf{x} \rangle + w_{0,k}.$$

Each such model predicts object membership in a certain class. How do we convert the evaluation vector $(b_1(x), \ldots, b_K(x))$ into probabilities? For this, we can use the operator $softmax(z_1, \ldots, z_K)$ that "normalizes" the vector:

$$\operatorname{softmax}(z_1, \ldots, z_K) = \left(\frac{\exp(z_1)}{\sum\limits_{k=1}^{K} \exp(z_k)}, \ldots, \frac{\exp(z_K)}{\sum\limits_{k=1}^{K} \exp(z_k)} \right).$$

The probability of the K-th class is:

$$\mathbb{P}(y = k \mid \mathbf{x}, \mathbf{w}) = \frac{\exp(\langle \mathbf{w}_k, \mathbf{x} \rangle + w_{0,k})}{\sum\limits_{j=1}^{K} \exp(\langle \mathbf{w}_j, \mathbf{x} \rangle + w_{0,j})}.$$

It makes sense to use the maximum likelihood method for training the weights: same as for the two-class logistic regression:

$$\sum_{i=1}^{N} \log \mathbb{P}(y = y_i \mid \mathbf{x}_i, \mathbf{w}) \to \max_{\mathbf{w}_1, \ldots, \mathbf{w}_K}.$$

4 Lecture 4. Fundamental Concepts of Deep Learning: Basic Layers and Activation Functions

4.1 Basic Definitions

An artificial neural network (hereinafter referred to as a "neural network") is a complex differentiable function that maps from the input feature space to the response space, with all its parameters being adjustable simultaneously and interdependently (i.e., the network can be trained end-to-end).

In the specific (and most common) case, it is represented as a sequence of differentiable parametric transformations.

A careful reader may notice that the above definition also covers logistic and linear regression. This is a valid observation: both linear and logistic regression can be regarded as neural networks, mapping respectively to the response space and to the space of logits.

A complex function is conveniently represented as a composition of simple ones, and neural networks are typically presented to the programmer as a constructor consisting of relatively simple blocks (layers). Here are two of the simplest types:

Linear Layer (Dense Layer).

A linear layer (dense layer) is a linear transformation applied to the input data. Its learnable parameters are a matrix W and a vector b:

$$x \mapsto xW^T + b, \quad \text{with } W \in \mathbb{R}^{d \times k},\ x \in \mathbb{R}^d,\ b \in \mathbb{R}^k.$$

This layer transforms d-dimensional vectors into k-dimensional ones.

Activation Function An activation function is a nonlinear transformation applied element-wise to the input. Activation functions enable neural networks to generate more informative feature representations by transforming data nonlinearly. For example, one may use the ReLU (rectified linear unit):

$$\mathrm{ReLU}(x) = \max(0, x),$$

or the sigmoid function:

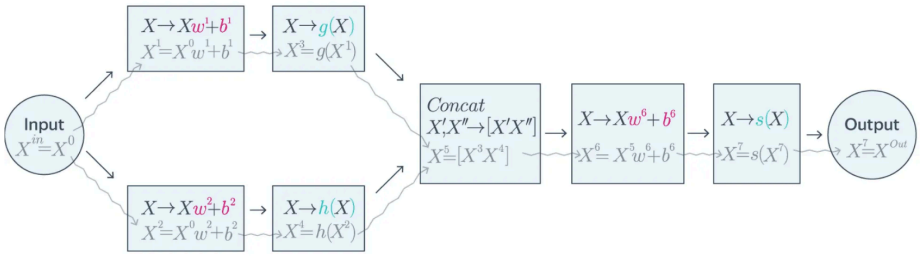
$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

We will discuss various activation functions and their properties in more detail later.

Even the most complex neural networks are usually built from relatively simple blocks like these. Thus, they can be represented as a *computational graph*, where the intermediate nodes correspond to transformations.

4.2 Input and Fully Connected Neural Networks

Input — the entry point of a neural network, where raw data is received. Typically, the input is represented as a matrix (objects × features) or, more generally, as a tensor if the data is multi-dimensional. A network may have multiple inputs when different types of data are processed separately.



For instance, we may provide both an image and associated metadata to a model. Since these data types require different processing pipelines, it is natural to represent them as distinct inputs in the computational graph.

In the diagram above, we see an example of such a network. The input $X^{\text{in}} = X^0$ is split into two branches. Each branch applies a **linear transformation** (also called a **fully connected layer**), which performs an affine transformation of the input:

$$X \to XW + b$$

This results in two intermediate representations: X^1 and X^2. These are also referred to as *hidden states* or *activations* (not to be confused with the nonlinear activation functions applied later).

Each representation then passes through a nonlinear **activation function**, denoted as g and h, producing $X^3 = g(X^1)$ and $X^4 = h(X^2)$. These transformations allow the network to learn complex, non-linear dependencies in the data.

The outputs of the two branches are then concatenated along the feature dimension:

$$X^5 = [X^3, X^4]$$

This combined representation aggregates features learned independently in each branch.

After concatenation, the network applies another fully connected layer, followed by an activation function s, and produces the final output $X^7 = X^{\text{Out}}$.

A neural network that consists only of fully connected (dense) layers and elementwise nonlinear activation functions is called a **fully connected neural network** or a **multilayer perceptron (MLP)**. In such networks, every neuron in a layer is connected to every neuron in the previous layer, and the core learning process happens through compositions of linear transformations and nonlinearities.

4.3 Some words on the power of neural networks

Let us begin by considering a regression problem. It is clear that a linear model (i.e., a single-layer neural network) can approximate only linear functions, whereas a two-layer neural network can approximate almost any function. There are several theorems on this subject; we mention one of them. Note the year: as we have already stated, neural networks began to be studied seriously long before they became state-of-the-art.

Theorem 4.1 (Cybenko's Theorem, 1989)**.** *For any continuous function*

$$f(x) : \mathbb{R}^m \to \mathbb{R},$$

and for any $\varepsilon > 0$, there exists a number N and constants $w_1, \dots, w_N$, $b_1, \dots, b_N$, and $\alpha_1, \dots, \alpha_N$ such that

$$\left| f(x) - \sum_{i=1}^{N} \alpha_i \, \sigma\big(\langle x, w_i \rangle + b_i\big) \right| < \varepsilon,$$

for all x in the unit cube $[0,1]^m \subset \mathbb{R}^m$.

From Cybenko's theorem one can readily recognize the structure of a two-layer neural network with a sigmoid activation function. Indeed, first we transform x to $\langle x, w_i \rangle + b_i$ — this can be represented as a single matrix operation (a linear layer):

$$x \mapsto x^{(1)} = x \cdot \begin{pmatrix} w_1 & \cdots & w_N \end{pmatrix} + \begin{pmatrix} b_1 & \cdots & b_N \end{pmatrix},$$

where each w_i is a column vector and each b_i is added to the corresponding column. Then, we apply the sigmoid function element-wise to obtain

$$x^{(2)} = \sigma\big(x^{(1)}\big),$$

after which we compute

$$\sum_{i=1}^{N} \alpha_i \, x_i^{(2)} = \begin{pmatrix} \alpha_1, \dots, \alpha_N \end{pmatrix} \cdot x.$$

This represents a second linear layer (without a bias term).

Admittedly, the theorem does not help much in actually finding such functions, but that is another matter. In any case, if a neural network is provided with sufficient data, it can indeed learn almost anything.

4.4 Backpropagation Method

Neural networks are trained using various modifications of gradient descent, and to apply it, one must be able to efficiently compute the gradients of the loss function with respect to all training parameters. At first glance, for a complex computational graph this might seem like a very challenging task, but the backpropagation method comes to the rescue.

The discovery of the backpropagation method stands as one of the most significant events in the field of artificial intelligence. In its modern form, it was proposed in 1986 by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, and independently and simultaneously by the Krasnoyarsk mathematicians S. I. Bartsev and V. A. Okhonin.

Since then, the method for computing the gradients of neural network parameters has been based on the chain rule for differentiating composite functions. While calculating these gradients remains a challenging engineering problem, it is no longer considered an art. Despite the simplicity of the underlying mathematical tools, the advent of backpropagation led to a major breakthrough in the development of artificial neural networks.

The essence of the method can be captured by a single formula, which follows trivially from the chain rule. If

$$f(x) = g_m\big(g_{m-1}(\cdots(g_1(x))\cdots)\big),$$

then

$$\frac{\partial f}{\partial x} \;=\; \frac{\partial g_m}{\partial g_{m-1}} \cdot \frac{\partial g_{m-1}}{\partial g_{m-2}} \cdots \frac{\partial g_2}{\partial g_1} \cdot \frac{\partial g_1}{\partial x}.$$

Thus, the gradients can be computed sequentially in a single backward pass, starting with $\frac{\partial g_m}{\partial g_{m-1}}$ and multiplying by the partial derivatives of each preceding layer.

4.4.1 One-dimensional case

In the one-dimensional case, everything becomes especially simple. Let w_0 be the variable with respect to which we wish to differentiate, and suppose the composite function is given by

$$f(w_0) = g_m\Big(g_{m-1}\big(\cdots g_1(w_0)\cdots\big)\Big),$$

where all the g_i are scalar functions. Then, by the chain rule,

$$f'(w_0) = g'_m\Big(g_{m-1}\big(\cdots g_1(w_0)\cdots\big)\Big)\cdot g'_{m-1}\Big(g_{m-2}\big(\cdots g_1(w_0)\cdots\big)\Big)\cdots g'_1(w_0).$$

The essence of this formula is as follows. If we have already performed a forward propagation, then we know

$$g_1(w_0), \quad g_2\big(g_1(w_0)\big), \quad \ldots, \quad g_{m-1}\Big(\cdots g_1(w_0)\cdots\Big).$$

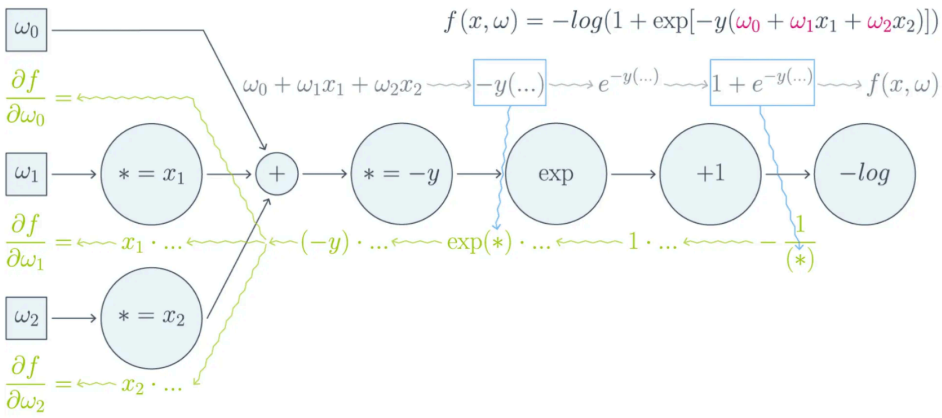
Therefore, we can proceed as follows: take the derivative g'_m at the point

$$g_{m-1}\Big(\cdots g_1(w_0)\cdots\Big),$$

multiply it by the derivative g'_{m-1} at the point

$$g_{m-2}\Big(\cdots g_1(w_0)\cdots\Big),$$

and so on, until we reach $g'_1(w_0)$.



Collecting all the factors for a function on a picture together, we obtain:

$$\frac{\partial f}{\partial w_0} = (-y)\cdot e^{-y(w_0+w_1x_1+w_2x_2)}\cdot\Big[1+e^{-y(w_0+w_1x_1+w_2x_2)}\Big]^{-1},$$

$$\frac{\partial f}{\partial w_1} = x_1\cdot(-y)\cdot e^{-y(w_0+w_1x_1+w_2x_2)}\cdot\Big[1+e^{-y(w_0+w_1x_1+w_2x_2)}\Big]^{-1},$$

$$\frac{\partial f}{\partial w_2} = x_2\cdot(-y)\cdot e^{-y(w_0+w_1x_1+w_2x_2)}\cdot\Big[1+e^{-y(w_0+w_1x_1+w_2x_2)}\Big]^{-1}.$$

Thus, we first perform forward propagation to compute all intermediate values (indeed, all intermediate representations must be stored in memory), and then we run backward propagation, during which all gradients are computed in one pass.

4.4.2 Gradients for Typical Layers

Let's compute gradients for several common types of layers using a simplified style, similar to typical deep learning notation.

Example 4.2 (Element-wise Nonlinearity)**.** *Let $f(x) = u(v(x))$, where x is a vector and $v(x)$ is applied element-wise.*

Then, using the chain rule:

$$\frac{\partial f}{\partial x_i} = \frac{\partial u}{\partial v_i} \cdot v'(x_i).$$

In vector form:

$$\nabla_x f = \nabla_v u \odot v'(x),$$

where $\odot$ denotes element-wise multiplication.

Example 4.3 (Matrix Multiplication – Gradient with Respect to Input)**.** *Let*

$$f(X) = g(XW),$$

where X is an $m \times n$ matrix, W is an $n \times k$ matrix, and $Y = XW$ is an $m \times k$ matrix.

We want to compute the gradient $\frac{\partial f}{\partial x_{ij}}$, using the chain rule:

$$\frac{\partial f}{\partial x_{ij}} = \sum_{l=1}^{k} \frac{\partial f}{\partial y_{il}} \cdot \frac{\partial y_{il}}{\partial x_{ij}}.$$

Since $y_{il} = \sum_{s=1}^{n} x_{is} w_{sl}$, we have:

$$\frac{\partial y_{il}}{\partial x_{ij}} = w_{jl}.$$

Thus,

$$\frac{\partial f}{\partial x_{ij}} = \sum_{l=1}^{k} \frac{\partial f}{\partial y_{il}} \cdot w_{jl}.$$

In matrix notation, this corresponds to:

$$\frac{\partial f}{\partial X} = \frac{\partial f}{\partial Y} \cdot W^T,$$

where the multiplication on the right-hand side is standard matrix multiplication.

Example 4.4 (Matrix Multiplication – Gradient with Respect to Weights)**.** *Let*

$$f(W) = g(XW),$$

where X is an $m \times n$ matrix, W is an $n \times k$ matrix, and $Y = XW$ is an $m \times k$ matrix.

We want to compute the gradient $\frac{\partial f}{\partial w_{ij}}$, using the chain rule:

$$\frac{\partial f}{\partial w_{ij}} = \sum_{r=1}^{m} \frac{\partial f}{\partial y_{rj}} \cdot \frac{\partial y_{rj}}{\partial w_{ij}}.$$

Since $y_{rj} = \sum_{s=1}^{n} x_{rs} w_{sj}$, we have:

$$\frac{\partial y_{rj}}{\partial w_{ij}} = x_{ri}.$$

Thus,

$$\frac{\partial f}{\partial w_{ij}} = \sum_{r=1}^{m} \frac{\partial f}{\partial y_{rj}} \cdot x_{ri}.$$

In matrix notation, this corresponds to:

$$\frac{\partial f}{\partial W} = X^T \cdot \frac{\partial f}{\partial Y},$$

where the multiplication on the right-hand side is standard matrix multiplication.

Example 4.5 (Softmax Layer)**.** *Let*

$$f(X) = g\left(\text{softmax}(X)\right),$$

where softmax is applied row-wise.

Define:

$$s = \text{softmax}(X), \quad \nabla_s g = \frac{\partial g}{\partial s}.$$

Then, for each element:

$$\frac{\partial L}{\partial x_{ij}} = s_{ij}\left(\nabla_{ij} - \sum_t s_{it}\nabla_{it}\right).$$

In vectorized form:

$$\nabla_X f = s \odot \left(\nabla_s g - \text{sum}\left(s \odot \nabla_s g,\ axis = 1\right)\right),$$

where $\odot$ denotes element-wise multiplication and the sum is performed across columns (i.e., per row).

4.5 Pros and Cons of Activation Functions

Activation functions are crucial components of neural networks, as they introduce non-linearity, allowing the model to approximate complex functions. However, not all activation functions behave equally in practice. Below is a qualitative comparison of several widely used activation functions, with a focus on their advantages and limitations during training.

Sigmoid and **tanh** are classical activation functions with smooth and bounded outputs. While they are differentiable and historically important, they suffer from the problem of *vanishing gradients* — especially for inputs with large magnitudes — which can slow down or even stall training in deep networks. This happens because in regions where the activation function saturates (i.e., its output becomes nearly constant), the gradient becomes very small. When multiple layers use such activations, the small gradients are multiplied together during backpropagation, leading to an overall gradient that is nearly zero. As a result, the weights in early layers receive almost no update, hindering learning.

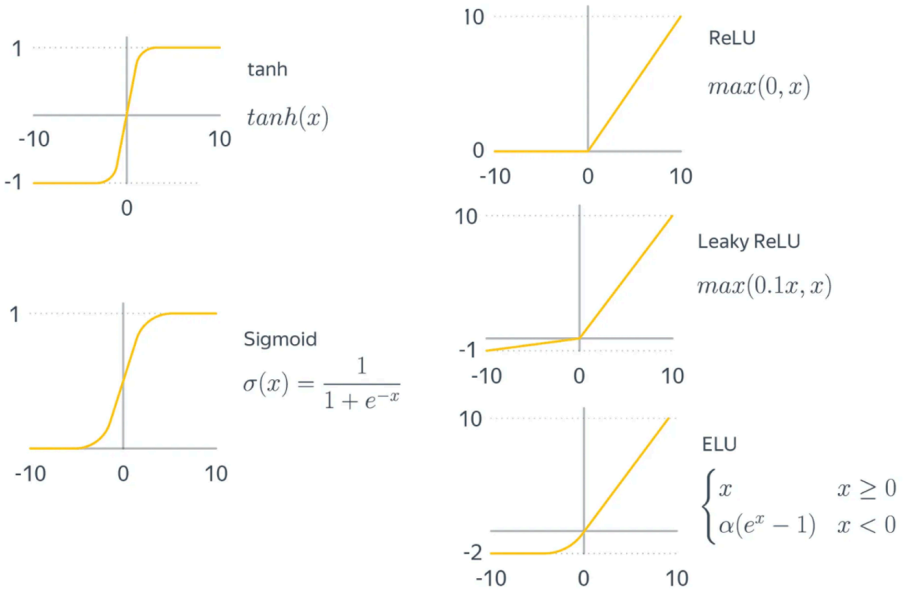
Sigmoid is still used in gating mechanisms, e.g., in recurrent architectures like LSTMs, where we need to control the flow of information through memory cells. In such cases, the sigmoid's output between 0 and 1 is interpreted as a soft on/off switch, allowing the network to decide how much information to let through.

ReLU, by contrast, is simple and efficient. It avoids saturation for positive values and helps deep models converge faster. However, it introduces the problem of *dead neurons* — units that never activate due to zero gradients when the input is negative.

To address this, **Leaky ReLU** allows a small gradient for negative inputs, reducing the risk of dead neurons while retaining simplicity. **ELU** further refines this by using a smooth exponential shape in the negative region, centering the output around zero, which can accelerate learning. However, this comes at the cost of increased computational complexity.

The following lists summarize the key strengths and weaknesses of each function to guide their selection based on network architecture and task requirements.

Activation Functions



Sigmoid:

- **Pros:**
  - Smooth and differentiable
  - Historically popular, useful in probabilistic models
- **Cons:**
  - Saturates for large inputs $\Rightarrow$ vanishing gradients
  - Output not zero-centered
  - Slow convergence in deep networks

Tanh:

- **Pros:**
  - Output is zero-centered
  - Smooth and differentiable
- **Cons:**
  - Still saturates at large $|x|$
  - Suffers from vanishing gradients (though less than sigmoid)

ReLU:

- **Pros:**
  - Simple and computationally efficient

  - Does not saturate in positive region ⇒ faster convergence

- **Cons:**
  - Gradient is zero for $x < 0$ ⇒ dead neurons
  - Not zero-centered

Leaky ReLU:

- **Pros:**
  - Allows small gradient for $x < 0$ ⇒ avoids dead neurons
  - Easy to implement
- **Cons:**
  - Choice of negative slope (α) is arbitrary
  - Still not zero-centered

ELU:

- **Pros:**
  - Smooth transition for negative inputs
  - Output can be negative ⇒ mean closer to zero
- **Cons:**
  - Slower computation due to exponential
  - Requires tuning α

4.6 Gradient-Based Optimization Algorithms

While training neural networks, we typically use **Stochastic Gradient Descent (SGD)** to minimize the loss function. However, vanilla SGD often converges slowly and can get stuck in regions with poor curvature or noisy gradients. To address these issues, several modifications have been proposed to make the optimization more efficient and robust.

In this section, we discuss several popular optimization algorithms based on gradient descent, each incorporating different mechanisms to improve convergence.

4.6.1 Momentum

The **Momentum** method accumulates a moving average of past gradients to smooth out updates and accelerate convergence, especially in the presence of noise or curvature.

$$
\begin{aligned}
g_k &= \nabla_w \mathcal{L}(w_k) \\
\mu_k &= \beta_1 \cdot \mu_{k-1} + (1 - \beta_1) \cdot g_k \\
w_k &= w_{k-1} - \alpha_k \cdot \mu_k
\end{aligned}
$$

Here:

- g_k is the gradient at step k,
- μ_k is the exponentially weighted moving average of gradients,
- $\beta_1 \in [0, 1)$ controls the decay of the momentum,
- α_k is the learning rate.

4.6.2 RMSProp

RMSProp adapts the learning rate per parameter based on a moving average of squared gradients. This helps to normalize updates and mitigate exploding or vanishing gradients.

$$
\begin{aligned}
g_k &= \nabla_w \mathcal{L}(w_k) \\
\nu_k &= \beta_2 \cdot \nu_{k-1} + (1 - \beta_2) \cdot g_k^2 \\
w_k &= w_{k-1} - \alpha_k \cdot \frac{g_k}{\sqrt{\nu_k + \varepsilon}}
\end{aligned}
$$

Here:

- ν_k is the exponentially weighted average of squared gradients,
- β_2 is a decay factor, typically around 0.9,
- ε is a small constant (e.g., 10^{-8}) added for numerical stability.

4.6.3 Adam

Adam (Adaptive Moment Estimation) combines both Momentum and RMSProp. It maintains moving averages of both the gradients (first moment) and the squared gradients (second moment). Since these averages are initialized at zero, they are biased toward zero in the early stages of training. To correct for this, Adam applies **bias correction** to both moment estimates.

$$
\begin{aligned}
g_k &= \nabla_w \mathcal{L}(w_k) \\
\mu_k &= \beta_1 \cdot \mu_{k-1} + (1 - \beta_1) \cdot g_k \\
\nu_k &= \beta_2 \cdot \nu_{k-1} + (1 - \beta_2) \cdot g_k^2 \\
\hat{\mu}_k &= \frac{\mu_k}{1 - \beta_1^k}, \quad \hat{\nu}_k = \frac{\nu_k}{1 - \beta_2^k} \\
w_k &= w_{k-1} - \alpha_k \cdot \frac{\hat{\mu}_k}{\sqrt{\hat{\nu}_k + \varepsilon}}
\end{aligned}
$$

Adam is widely used in practice due to its robust performance and minimal hyperparameter tuning.

5 Lecture 5. Practical applications of neural networks to Math

5.1 Regularization via Model Structure Constraints

Introducing appropriate transformations to the network architecture can be an effective way to achieve desired generalization. Two techniques that significantly impacted the development of neural networks are **dropout** (2014) and **batch normalization** (2015). They helped reduce overfitting and substantially accelerate convergence, respectively.

5.1.1 Dropout

Consider simple fully connected (FC/Dense) networks with multiple layers. Each layer generates a new feature representation x_k of the input object x_{in}:

$$x_k = f_k(x_{k-1}).$$

How can we ensure that the model uses all available parameters efficiently, instead of overfitting to a small subset and dividing the internal representation into "signal" and "noise"?

$$x_k^{\text{overfitted}} = [\text{signal}, \text{noise}]$$

One approach is to randomly "drop" access to some coordinates of internal representations during training. If useful coordinates are dropped, the model's predictions will change drastically, increasing the loss. The resulting gradients will indicate how to adjust and utilize other coordinates. The effect can be visualized by comparing information flow in the original vs. dropout-modified models.

Importantly, dropout can be applied to both input features and intermediate representations. From the perspective of the $(k+1)$-th layer, the data always come from "outside''—from real data when $k = 0$, or from previous layers when $k > 0$.

Technically, this is implemented by multiplying some coordinates of x_k by zero using a binary mask:

$$x_{k+1} = \frac{1}{1-p}\, x_k \odot \text{mask}, \quad \text{mask}_i \sim \text{Bernoulli}(1-p),$$

where p is the dropout rate (the probability of zeroing out a coordinate), often given as a hyperparameter in deep learning frameworks. This mask also affects the gradient calculation:

$$\nabla_{x_k} L = \frac{1}{1-p} \nabla_{x_{k+1}} L \odot \text{mask}.$$

Typically, a new mask is generated at every step of gradient descent. During inference, dropout is **disabled**, i.e.,

$$x_{k+1} = x_k.$$

The factor $\frac{1}{1-p}$ ensures that the expected value and variance of x_{k+1} remain consistent between training and inference. Without this scaling, the variance would decrease due to dropped components.

Note: Dropout can also be applied to input data, especially beneficial when features are noisy or strongly correlated. For instance, in sparse high-dimensional settings (e.g., user preferences over products), input-level dropout can prevent the model from conditioning on a small subset of features and promote more robust learning.

5.1.2 Batch Normalization

Batch normalization significantly accelerated neural network training. While there is still ongoing debate about the exact reasons for its effectiveness (see NeurIPS 2018 discussion), the original intuition remains useful.

Batch normalization ensures that each feature of the internal representation has controlled mean and variance. It operates as follows:

$$X_{k+1} = \frac{X_k - \mu}{\sqrt{\sigma^2 + \varepsilon}},$$

where μ and σ^2 are the mean and variance computed over the current mini-batch, and ε is a small constant added for numerical stability. These statistics are differentiable functions of X_k, and thus participate in backpropagation.

During inference, fixed values μ^* and σ^{*2} are used instead, computed as running averages during training:

$$\mu^* = \lambda\mu^* + (1-\lambda)\mu, \quad \sigma^{*2} = \lambda\sigma^{*2} + (1-\lambda)\sigma^2,$$

where λ is a momentum hyperparameter.

After normalization, a learned rescaling and shifting transformation is applied:

$$X_{k+2} = \beta X_{k+1} + \gamma,$$

where β and γ are trainable parameters allowing the network to recover the original representation scale if needed.

Batch normalization became widely adopted due to its ability to drastically speed up training and improve convergence, often outperforming state-of-the-art architectures from earlier years (e.g., Inception 2014). This efficiency gain is partly due to the ability to use larger learning rates without disrupting the layer-to-layer dependencies.

Despite its success, the theoretical understanding of why batch normalization works so well remains incomplete. Nevertheless, it has become a staple in deep learning.

Note: Other normalization techniques such as instance normalization and layer normalization have also been proposed, each with its own use cases and advantages.

We also recommend reading the original paper by Sergey Ioffe and Christian Szegedy (2015) and the follow-up studies that investigate the backpropagation mechanism through batch normalization layers.

5.2 Bruhat intervals

5.2.1 Bruhat Cells for Symmetric Groups.

In the context of algebraic groups, *Bruhat cells* (also known as *Schubert cells*) are the cells in the Bruhat decomposition of a flag variety. For the full flag variety of type **A** (which corresponds to the group $\mathrm{GL}_n(K)$ or $\mathrm{GL}_n(K)/B$), each Bruhat cell is indexed by a permutation $\mathbf{w} \in \mathbf{S}_n$.

Topologically, the flag variety can be written as a disjoint union of these cells:

$$\mathrm{GL}_n(K) = \bigcup_{\mathbf{w}\in\mathbf{S}_n} \Omega_{\mathbf{w}},$$

where $\Omega_{\mathbf{w}}$ is the Bruhat cell associated to $\mathbf{w}$:

$$\Omega_{\mathbf{w}} = \{[d_1, \dots, d_n](I + a_{12}E_{12}) \dots (I + a_{1n}E_{1n}) \dots (I + a_{n-1,n}E_{n-1,n})) \cdot \mathbf{w} \cdot$$
$$\cdot(I+b_{12}E_{12}) \dots (I+b_{1n}E_{1n}) \dots (I+b_{n-1,n}E_{n-1,n})) \mid a_{i,j}, b_{i,j} \in K, d_i \in K^*1 \leqslant i < j \leqslant n\},$$

where $b_{i,j} = 0$, if the matrix $\mathbf{w}(I + E_{i,j})\mathbf{w}^{-1}$ is lower triangular.

Therefore the dimension of the cell $\Omega_{\mathbf{w}}$ is $n(n+1)/2+\ell(\mathbf{w})$, where $\ell(\mathbf{w})$ is the lengths of $\mathbf{w}$.

The partial order known as the *Bruhat order* captures how these cells fit together in the closure hierarchy:

$$v \leqslant w \iff \Omega_v \subseteq \overline{\Omega_w}.$$

This provides a geometric interpretation: smaller permutations in the order index boundary pieces of the Schubert variety indexed by a larger permutation.

5.2.2 Bruhat Cells for GL_2

In the case of GL_2 we consider $\mathbf{S}_2 = \{e, (1, 2)\}$ and

$$\mathrm{GL}_2(K) = \Omega_e \cup \Omega_{(1,2)},$$

where

$$\Omega_e = B = \left\{ \begin{pmatrix} a & b \\ 0 & c \end{pmatrix} \mid b \in K, a, c \in K^* \right\}$$

and

$$\Omega_{(1,2)} = \left\{ [a, d] \begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix} \mid b, c \in K, a, d \in K^* \right\}$$

i. e.,

$$\Omega_{(1,2)} = \left\{ \begin{pmatrix} ab & a + abc \\ d & cd \end{pmatrix} \mid b, c \in K, a, d \in K^* \right\}$$

So we can analitically (algebraically) see that $\dim \Omega_e = 3$, $\dim \Omega_{(1,2)} = 4$ and $\Omega_e \subset \overline{\Omega}_{(1,2)}$.

5.2.3 Definitions: Bruhat Order, Covers, Length, and Intervals

Definition 5.1. Bruhat order *(sometimes called the* strong Bruhat–Chevalley order*) is a partial order on the symmetric group* $\mathbf{S}_n$*. There are several equivalent definitions of this order:*

1. Matrix (Rank Matrix) Criterion*: If we define the rank matrix* $R_w(i,j)$ *to be the number of entries of* w *that are* $\leqslant i$ *and located in the first* j *positions, then* $u \leqslant v$ *if and only if* $R_u(i,j) \geqslant R_v(i,j)$ *for all* $1 \leqslant i,j \leqslant n$*.*

Intuitively, this means that in the diagram of u*, every upper-left subrectangle contains at least as many inversion points (or 1's in the permutation matrix) as the diagram of* v*. This condition ensures that* u*'s "inversion structure" is a subset of* v*'s in a certain cumulative sense.*

2. Closure of Bruhat Cells*: As mentioned above,* $u \leqslant v$ *if and only if the Schubert cell for* u *is contained in the closure of that for* v*. In particular, the identity permutation* $e = 1\,2\,\ldots\,n$ *is the unique minimum (its cell is a single point, contained in all closures), and the reverse permutation* $w = n\,\ldots\,2\,1$ *is the unique maximum (its cell is the open dense cell in the flag variety).*

A convenient way to visualize the Bruhat order is via its Hasse diagram, where an edge (*cover relation*) $u < v$ indicates that v covers u (i.e. $u < v$ and no element lies strictly between them). In $\mathbf{S}_n$, a cover corresponds to applying a single simple transposition (an adjacent swap) that increases the length of the permutation by 1. Recall that the length $\ell(w)$ of a permutation w is the number of inversions (or equivalently, the minimum number of adjacent swaps needed to transform w into the identity). So, v covers u in the Bruhat order precisely when $v = u \cdot s$ for some adjacent transposition $s = (i, i+1)$ and $\ell(v) = \ell(u) + 1$. In one-line notation, this means v is obtained from u by swapping two adjacent entries that were in increasing order in u, thereby creating exactly one new inversion. For example, in $\mathbf{S}_4$ $u = 1\,3\,2\,4$ (one inversion) is covered by $v = 3\,1\,2\,4$ (two inversions) by swapping the first two entries (1 and 3). Each such cover relation is one "step" upward in the partial order. An **interval** $[x, y]$ in the Bruhat order is defined as usual for posets:

$$[x, y] = \{z \in \mathbf{S}_n \mid x \leqslant z \leqslant y\}.$$

The Bruhat order on $\mathbf{S}_n$ is a graded poset: it has rank function given by the length $\rho(w) = \ell(w)$. Thus, all saturated chains from x up to y have the same length, and the rank difference $\rho(y) - \rho(x)$ equals the number of cover steps in any maximal chain from x to y.

We often refer to $\rho(y) - \rho(x)$ as the *rank* or *length* of the interval $[x, y]$.

The **Hasse diagram** of the Bruhat order is the graph of these cover relations, drawn with higher rank elements above lower rank ones.

5.2.4 Examples for Small n

To build intuition, it helps to see the Bruhat order for small symmetric groups and specific intervals:

Bruhat order on $\mathbf{S}_3$: Each node is a permutation written in one-line notation. The bottom element is 123 (the identity permutation, length 0) and the top is 321 (the longest permutation, length 3). Edges connect permutations that differ by a single adjacent transposition increasing the length. For instance, 123 is covered by 132 and 213 (each obtained by one adjacent swap). The entire poset forms a diamond shape. Notice it is symmetric about a vertical axis: 123 covers two elements (rank 1), each of those covers two elements (rank 2), and they meet again at 321 (rank 3). This reflects the Eulerian property: there is 1 element of rank 0, 2 of rank 1, 2 of rank 2, and 1 of rank 3, satisfying the equal-even/odd count condition. The interval $[123, 321]$ in $\mathbf{S}_3$ is already the whole poset here.

Bruhat order on $\mathbf{S}_4$: Hasse diagram of the Bruhat order for $\mathbf{S}_4$. The bottom node is 1234 and the top node is 4321. The rank distribution is $1, 3, 5, 6, 5, 3, 1$ (from rank 0 up to 6), again symmetric. To avoid clutter, the diagram does not label every edge orientation, but implicitly edges go upward from lower (shorter) to higher (longer) permutations. We can see, for example, 1234 covers three permutations (those with one inversion: 1243, 1324, 2134).

At the next level (rank 2), there are 5 permutations such as 1342, 1423, 2314, 2143, 3124, etc. The structure is getting more complex, but it is still a lattice-like shape with symmetric layers. Every maximal chain from 1234 to 4321 has length 6.

As an example of an interval, consider $[1234, 3412]$ (where 3412 is a permutation of length 4). This interval consists of all permutations $\leqslant 3412$.

5.2.5 The Weak and Strong Bruhat Orders

Together with this natural (strong) Bruhat order there exists its weak analogue.

For any $i < j$, let $(i\ j)$ denote the transposition exchanging i and j. The *weak order* on $\mathbf{S}_n$ is defined by its covering relations:

$$u \lessdot_{\text{weak}} v \text{ if and only if } v = ut \text{ for some transposition } t \text{ and } \ell(v) = \ell(u) + 1.$$

Both orders have a unique maximal element, the permutation w_0 with one-line notation $n\ (n-1)\ \ldots\ 1$, which has length $\binom{n}{2}$, and a unique minimal element, the identity permutation id, which has length zero.

In the paper under consideration they take a weak order instead of the strong one.

5.3 Bruhat Intervals and KL-Polynomials in S_n

Kazhdan–Lusztig (KL) polynomials $P_{x,y}(q)$ are polynomials associated to pairs (x, y) in a Coxeter group, originally defined via the Hecke algebra. They encode deep geometric and representation-theoretic information. In type A, the coefficients of $P_{u,v}(q)$ correspond to intersection cohomology Betti numbers of Schubert varieties.

The key question: is $P_{u,v}(q)$ determined by the poset structure of $[u, v]$?

5.3.1 The Combinatorial Invariance Conjecture

Conjecture (Lusztig–Dyer). If two Bruhat intervals $[u, v]$ and $[u', v']$ in (possibly different) Coxeter groups are isomorphic as posets, then $P_{u,v}(q) = P_{u',v'}(q)$.

For $\mathbf{S}_n$, this means the KL polynomial $P_{u,v}(q)$ is a poset invariant of the interval $[u, v]$.

5.3.2 Hypercube Decompositions

Many Bruhat intervals contain Boolean subintervals. When k mutually commuting simple reflections are involved, the interval $[u, v]$ may contain a Boolean lattice (a k-cube).

Definition (Hypercube Decomposition). A Bruhat interval $[u, v]$ has a hypercube decomposition if it can be expressed as the union of a Boolean lattice ideal $I \cong B_k$ and a smaller Bruhat interval $[w, v]$.

BBDVW Conjecture (2022). There exist polynomials $N_{u,v,I}(q)$ and $Q_{u,v,I}(q)$, derived from smaller intervals, such that:

$$P_{u,v}(q) = N_{u,v,I}(q) + q^d Q_{u,v,I}(q)$$

This formula uses only the poset structure and would prove combinatorial invariance for $\mathbf{S}_n$ if true for all intervals.

Example 5.2 (Boolean interval)**.** *In $\mathbb{S}_4$, consider the interval $[e; (1,2)(3,4)]$. Here e is the identity permutation and $(1,2)(3,4)$ is the permutation swapping 1, 2 and 3, 4. This top element can be written as the product of two commuting simple transpositions: $s_1 = (1,2)$ and $s_3 = (3,4)$ (using Coxeter generators s_i for adjacent swaps). The Bruhat interval $[e; s_1 s_3]$ has the following elements: $\{e; s_1; s_3; s_1 s_3\}$. Its Hasse diagram is a square (a 4-element diamond): e covers s_1 and s_3, which both cover $s_1 s_3$ at the top. This poset is isomorphic to the Boolean lattice $\mathbb{B}_2$ (rank 2 hypercube).*

All such intervals consisting of two commuting transpositions share this unlabeled structure (often called a 2-crown). The KL polynomial in these cases is trivial: indeed one can show $P_{e;s_1 s_3}(q) = 1$ (since the interval is thin with no lesser elements to force a higher

q-term). Thus any Bruhat interval isomorphic to this 4-element diamond (no matter which specific permutations label the elements) will have $P(q) = 1$. This illustrates the conjecture: the poset type alone dictates the polynomial.

Example 5.3 (Longest Element Interval in $\mathbf{S}_3$)**.** *For $n \leqslant 3$, all Bruhat intervals are fairly simple. In $\mathbf{S}_3$, the largest interval is $[e, w_0]$ where $w_0 = (3, 2, 1)$ is the longest element of S_3. This interval $[e, w_0]$ has 6 elements in a poset of rank 3. Its Hasse diagram can be visualized as two distinct maximal chains from e to w_0 that diverge at the start and reconverge at the top. Specifically, e covers $s_1 = (1, 2)$ and $s_2 = (2, 3)$; then $s_1 < s_1 s_2$ and $s_2 < s_2 s_1$; and finally both $s_1 s_2$ and $s_2 s_1$ are covered by $w_0 = s_1 s_2 s_1 = s_2 s_1 s_2$. This poset is not a Boolean lattice (it has 6 elements rather than $2^3 = 8$), but it does contain a 2-crown as a subposet (for instance, $e < \{s_1, s_2\} < w_0$ in a transitive sense). All intervals of length $\leqslant 2$ in $\mathbf{S}_3$ are either chains or 4-element diamonds, so their KL polynomials are either 1 or 1 (for the diamond) — trivial in either case. The first possibility of a nontrivial polynomial arises in this length 3 interval. Indeed $P_{e,w_0}(q) = 1 + q$ for $\mathbf{S}_3$'s longest interval (the coefficient of q reflects the existence of a "hidden" element in rank 2 forcing a correction term). If any other pair (u, v) in some S_n produced an isomorphic poset (and there are such examples in larger S_n that replicate this shape), they should have the same polynomial $1 + q$ according to the conjecture.*

Evidence for the Conjecture:

- **Lower intervals:** Proven for all $[e, v]$ in $\mathbf{S}_n$.
- **Small ranks:** Verified up to rank 8 (Incitti).
- **Hypercube recurrence:** Proven for lower intervals (Barkley–Gaetz, 2025).
- **No known counterexamples** in any Coxeter group.

5.4 Graph Neural Networks and AI-Guided Discovery in KL Polynomial Computation

5.4.1 Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) are a class of deep learning models designed to operate on graph-structured data. Given a graph $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges, a GNN updates the feature representation of each node by aggregating information from its neighbors.

Let $h_v^{(k)}$ denote the feature vector of node $v \in V$ at layer k. A typical GNN update rule is of the form:

$$h_v^{(k+1)} = \text{UPDATE}^{(k)}\left(h_v^{(k)}, \text{AGGREGATE}^{(k)}\left(\{h_u^{(k)} : u \in \mathcal{N}(v)\}\right)\right),$$

where $\mathcal{N}(v)$ is the set of neighbors of v, and both the aggregation and update steps are usually parameterized by learnable functions (e.g., neural networks). Through repeated iterations, each node's representation becomes increasingly informed by the structure and features of the surrounding subgraph.

GNNs are particularly well-suited for mathematical structures such as Bruhat intervals, which naturally take the form of directed graphs with hierarchical and combinatorial properties.

5.4.2 AI-Guided Discovery of Hypercube Decomposition

In the work, researchers applied supervised learning using graph neural networks to the problem of predicting KL polynomials from Bruhat intervals. The model was trained on a dataset of unlabelled Bruhat intervals represented as directed graphs, with the KL polynomial values as supervised targets.

Through careful experiments, the authors found that certain subgraphs of the Bruhat interval, referred to as *salient subgraphs*, were consistently the most informative for predicting the KL polynomial. These subgraphs were identified using **attribution techniques**,

which measure the contribution of individual components of the input graph to the model's prediction.

A striking empirical observation emerged: extremal reflections—edges corresponding to permutations near the beginning or end of the group (such as transpositions $(0, i)$ or $(i, n-1)$)—were significantly overrepresented in these salient subgraphs, while simple reflections $(i, i+1)$ were underrepresented. Notably, the model was not given any label information about the edges, indicating that it had implicitly learned this structural bias from the data alone.

Motivated by these findings, the authors discovered that every Bruhat interval can be decomposed into two components:

1. A **hypercube subgraph** induced by a subset of extremal reflections;
2. A graph isomorphic to a smaller Bruhat interval in S_{n-1}.

Theorem 5.4 (Canonical Hypercube Decomposition)**.** *Every Bruhat interval admits a canonical decomposition into a hypercube along extremal reflections and a residual Bruhat interval in a smaller symmetric group. The Kazhdan–Lusztig polynomial is directly computable from this decomposition via an explicit formula.*

This decomposition provides an efficient method for computing KL polynomials and offers a new combinatorial structure for studying the combinatorial invariance conjecture.

Conjecture 5.5 (General Hypercube Decomposition)**.** *The KL polynomial of an unlabelled Bruhat interval can be computed using any hypercube decomposition, not necessarily the canonical one.*

This conjecture has been computationally verified for over three million Bruhat intervals in S_n for $n \leq 7$, and for over 10^5 non-isomorphic intervals in S_8 and S_9.

5.5 Connections Between Algebraic and Geometric Invariants of Knots

Low-dimensional topology, particularly the study of knots (embeddings of $S^1 \hookrightarrow \mathbb{R}^3$), is a central area of modern mathematics. A fundamental aim of knot theory is to classify knots and understand their properties through **invariants** — quantities preserved under ambient isotopy.

There are two principal families of knot invariants:

- **Algebraic invariants**, such as the Alexander polynomial, Jones polynomial, and **signature** $\sigma(K)$;
- **Geometric invariants**, especially those arising from the hyperbolic structure on the knot complement, such as **volume** and **injectivity radius**.

These invariants stem from fundamentally different mathematical theories, and it is of great interest to understand relationships between them.

5.5.1 Signature and Hyperbolic Geometry Invariants

Definition 5.6 (Signature)**.** *Let $K \subset \mathbb{S}^3$ be an oriented knot. A* Seifert surface *for K is a connected, oriented, compact surface $S \subset \mathbb{S}^3$ such that $\partial S = K$. A Seifert matrix $V \in \mathbb{Z}^{g \times g}$, associated to a basis of $H_1(S; \mathbb{Z})$, encodes the linking numbers of cycles on S with their push-offs in the ambient space. The **signature** of the knot K is defined as:*

$$\sigma(K) = sign(V + V^\top),$$

where $sign(A)$ denotes the signature of the symmetric matrix A, i.e., the number of positive eigenvalues minus the number of negative eigenvalues. This integer is an ambient isotopy invariant of the knot.

Definition 5.7 (Hyperbolic Volume)**.** *A knot $K \subset \mathbb{S}^3$ is called* hyperbolic *if its complement $\mathbb{S}^3 \setminus K$ admits a complete Riemannian metric of constant sectional curvature -1 and finite volume. The* ***hyperbolic volume*** *of K is then defined as the volume of this hyperbolic 3-manifold:*

$$\mathrm{vol}(K) := \mathrm{vol}(\mathbb{S}^3 \setminus K).$$

By Mostow–Prasad rigidity, the hyperbolic structure on such a 3-manifold (when it exists) is unique up to isometry, and hence the volume is a topological invariant of the knot.

These invariants reflect profoundly different aspects of a knot. The signature is derived from the algebraic topology of surfaces spanning the knot, while the hyperbolic volume arises from the global geometry of the 3-manifold formed by removing the knot from the 3-sphere. Establishing connections between such invariants lies at the heart of modern low-dimensional topology.

Another key geometric quantity is the **injectivity radius** $\mathrm{inj}(K)$, which represents the largest radius r such that the ball of radius r centered at any point in the hyperbolic 3-manifold embeds isometrically into $\mathbb{H}^3$.

The geometry of the **cusp torus** (a toroidal boundary component of the compactified knot complement) plays a central role. Two crucial parameters associated to this torus are:

- **Meridional translation** $\mu \in \mathbb{C}$, representing the complex translation vector of the meridian in the Euclidean structure induced on the cusp torus;
- **Longitudinal translation** $\lambda \in \mathbb{C}$, representing the corresponding translation for the longitude.

These complex numbers describe the holonomy of the cusp and arise naturally in the geometric representation of the knot group.

5.5.2 Discovery of a New Invariant: Natural Slope

A supervised machine learning model was trained on a dataset of knots with known geometric and algebraic invariants. Remarkably, it discovered a strong correlation between the signature $\sigma(K)$ and three real-valued features derived from the cusp geometry:

$$Re(\mu), \quad Re(\lambda), \quad \text{and} \quad Im(\mu).$$

This led to the introduction of a new geometric quantity:

Definition 5.8 (Natural Slope)**.** *The* ***natural slope*** *of a knot K is defined as*

$$\mathrm{slope}(K) = Re\left(\frac{\lambda}{\mu}\right),$$

where Re denotes the real part. Geometrically, this can be interpreted as the signed number of meridional cycles traced when following a geodesic launched orthogonally from the meridian until it returns to intersect it.

5.5.3 Conjecture and Theorem Relating Signature and Slope

Based on empirical observations, the following conjecture was proposed:

Conjecture 5.9. *There exist constants $c_1, c_2 > 0$ such that for every hyperbolic knot K,*

$$|2\sigma(K) - \mathrm{slope}(K) - c_1| < \mathrm{vol}(K) + c_2.$$

Though this relation held for many knots, counterexamples were found (notably using specific braids), prompting the refinement of the conjecture into a provable theorem:

Theorem 5.10 (Refined Signature-Slope Theorem)**.** *There exists a universal constant $c > 0$ such that for every hyperbolic knot K,*

$$|2\sigma(K) - \mathrm{slope}(K) - c| \leq \mathrm{vol}(K) \cdot \mathrm{inj}(K)^{-3}.$$

Since the injectivity radius $\mathrm{inj}(K)$ tends not to be too small in practice, the right-hand side remains moderate even for knots of large volume. This theorem provides one of the first formal bridges between the algebraic and geometric descriptions of knots.

This discovery opens new directions in the interplay between geometry, topology, and machine learning-guided conjecture formulation. Despite the maturity of knot theory, the **natural slope** represents a previously overlooked invariant with deep implications.

6 Deep Learning in Mathematics

6.1 Lyapunov Functions and Function Prediction

In this lecture, we explore a deep learning approach to predicting **Lyapunov functions**, which are fundamental tools for certifying the stability of dynamical systems. A Lyapunov function provides a formal guarantee of stability: if such a function exists for a given system, it typically implies that the system's trajectories remain bounded or converge to equilibrium.

Unlike traditional machine learning tasks—where the output is a finite object such as a class label, real number, or fixed-size vector—our setting involves predicting a *function* as output. This raises a fundamental question: how can a model *generate* or *select* a mathematical function?

To address this, we must confront two central challenges:

- **Representation:** How can functions be encoded as features (either as inputs or outputs)?
- **Structure-aware learning:** How can a model be trained to produce structured mathematical objects such as symbolic functions or expressions?

Earlier approaches were limited to problems with fixed-size inputs and outputs, such as classification or regression. However, symbolic or functional outputs demand a richer and more flexible representation. To overcome this, we turn to the **transformer architecture**, a highly expressive model originally designed for sequences, but well-suited to learning over symbolic mathematical structures.

In today's lecture, we will build up the theoretical tools required to understand this architecture. We will discuss tokenization, embeddings, recurrent networks, sequence-to-sequence models, and attention. In the next lecture, we will turn to the transformer model itself and show how it can be adapted for function prediction.

Transformer Configuration for Lyapunov Learning

Authors of a papers said: We use a transformer model with the following parameters:

- 8 layers
- 10 attention heads
- Embedding dimension of 640
- Mini-batch size: 16 examples
- Optimizer: Adam
- Learning rate: 10^{-4}
- Warm-up phase: 10,000 steps
- Learning rate schedule: inverse square root decay

The **inverse square root schedule** is defined as:

$$\mathrm{lr}(n) = \frac{1}{\sqrt{\max(n, k)}}$$

where n is the current optimization step and k is the warm-up boundary (10,000 steps in our case). This schedule allows for rapid adaptation during early training and gradual decay later, which stabilizes convergence.

6.2 Sequential Data and Modeling

A central motivation for transformer-based architectures is their capacity to model *sequential data*. Many mathematical objects and statements — such as formulas, expressions, or sequences — are inherently ordered, and their meaning depends on the structure of this order.

Examples of sequential mathematical data:

- Integer sequences: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$
- Mathematical expressions: $\sin x + \frac{\cos x}{2x} - 17x + 4$
- Natural language statements of theorems or mathematical descriptions

To apply machine learning to such data, the process typically proceeds in two major stages:

1. Preprocessing: Representation of Data

The first step is to convert raw input into a discrete and numerically meaningful representation:

- **Tokenization:** The input is split into discrete units called *tokens*. These can be words, symbols, numbers, or even subwords, depending on the tokenizer.
- **Embedding:** Each token is mapped to a fixed-size vector in a continuous embedding space, capturing its identity and relationships to other tokens.

Several tokenization strategies exist:

- **Rule-based:** e.g., splitting at spaces, operators, or parentheses.
- **Statistical:** e.g., Byte Pair Encoding (BPE) or WordPiece, which build vocabularies by merging frequent subword units.
- **Grammar-aware:** e.g., parsing mathematical expressions into functional units or syntactic chunks.

Example: Reverse Polish Notation (RPN) Reverse Polish Notation is a syntax for representing mathematical expressions without parentheses, by encoding operator precedence into the order of tokens. For instance:

$$\sin x + \frac{\cos x}{2x} - 17x + 4 \quad \text{becomes} \quad x\ \sin\ x\ \cos\ 2x\ /\ +\ 17x\ -\ 4\ +$$

This representation is useful in symbolic manipulation tasks and can simplify the parsing of expressions by removing the need to learn precedence and grouping rules. In such settings, RPN acts as a **structure-preserving preprocessing step**.

2. Modeling: Learning from Embeddings

After tokenization and embedding, the resulting sequence of vectors is passed to a neural model that learns patterns, dependencies, and structure in the sequence. In this course, we focus primarily on:

- Recurrent neural networks (RNNs)
- Sequence-to-sequence (Seq2Seq) models
- Attention-based architectures such as Transformers

These models are trained to predict outputs (e.g., next token, translated expression, structured summary) from input sequences, capturing both local and global dependencies.

Byte Pair Encoding (BPE): Subword-Based Tokenization

Byte Pair Encoding (BPE) is a widely-used statistical tokenization method that breaks words into subword units based on frequency statistics in the training corpus. It starts with individual characters and iteratively merges the most frequent adjacent pairs of symbols, building up a vocabulary of subword units.

Multilingual Behavior BPE works well for languages like English where many words share frequent prefixes, suffixes, and stems. For example:

`unbelievable` → `un@@ believ@@ able`

However, in languages with rich morphology and lower representation in training corpora (e.g., Hebrew or Russian), BPE tends to split words into subword units that do not align well with actual morphemes. This fragmentation is often arbitrary and inconsistent, especially compared to English, where subword merges more frequently correspond to meaningful linguistic units due to higher frequency and simpler morphology.

- Hebrew: והילדים ("and the children") might be split as ים@@ הילד@@ ו, failing to preserve the prefix-conjunction structure.
- Russian: предсказуемость ("predictability") may be tokenized as пред@@ ска@@ зуем@@ ость, separating parts of the root and suffixes.
- English: `predictability` is more cleanly split into `predict@@ ability`, aligning with its morphological structure.

Because most large-scale corpora (e.g., Wikipedia, Common Crawl) are dominated by English, the learned BPE vocabularies tend to favor English-centric subwords. This creates an **encoding bias** in multilingual models: English words are often encoded in fewer tokens than their morphologically richer counterparts, potentially leading to less efficient learning for underrepresented languages.

Limitations of BPE While BPE captures some morphological regularities, it is ultimately frequency-based and *not linguistically aware*. For example:

- It may correctly segment `predictability` into meaningful morphemes, but this is an emergent property — not a guarantee.
- It often fails to segment low-frequency or domain-specific words meaningfully, especially in mathematical or scientific texts.
- It handles **numbers** and special symbols poorly, splitting long numbers digit by digit or in arbitrary ways:

  `3817925` → `3@@ 8@@ 1@@ 79@@ 25`
- In symbolic domains (e.g., math or programming), BPE may split tokens that should be kept intact — such as variable names, function names, or entire mathematical operators.

Implications for Modeling BPE reduces vocabulary size and enables open-vocabulary modeling, which is crucial for handling rare or unseen words. However, tokenization quality directly impacts model performance, especially in tasks requiring precise structure (e.g., mathematical expressions or code).

Researchers exploring domain-specific or multilingual tasks should be aware of these limitations and may consider:

- Fine-tuning tokenizers on domain-specific data
- Using linguistically informed segmentation (e.g., morphological analyzers)
- Exploring alternative tokenization strategies (e.g., SentencePiece, UnigramLM, character-level models, or tokenization-free approaches)

6.3 Embedding Representations

Each token is embedded into a fixed-dimensional space. For example, the words `I`, `enjoy`, `study`, `math`, `.` might be mapped to vectors:

$$\texttt{I} \to (1.2, 0.3, 5.7), \quad \texttt{enjoy} \to (-2.3, 4.3, 1.9), \quad \texttt{study} \to (9.0, 2.6, -1.3), \quad \texttt{math} \to (0.9, -3.5, 9.1), \quad \texttt{.} \to (0.2, 0.5, -1.0)$$

Traditional one-hot encoding treats each word as a separate category, resulting in sparse and high-dimensional vectors. Embeddings, by contrast, allow for more compact and meaningful representations, capturing semantic similarity and usage context.

For instance, we expect semantically related words like `scientist` and `learned` to have similar embeddings. Such similarity is learned from the **context** in which words appear.

6.4 Word2Vec and Contextual Learning

Word2Vec is a family of shallow, two-layer neural network models that learn vector representations (*embeddings*) of words based on their surrounding context in a corpus. These embeddings capture semantic and syntactic relationships between words, and are useful in downstream NLP tasks.

Word2Vec comes in two primary variants:

- **CBOW (Continuous Bag of Words):** Given a symmetric context window of size c, the model takes the surrounding words $\{w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}\}$ and tries to predict the center word w_t. This can be interpreted as averaging the embeddings of the context words and passing them through a linear layer followed by a softmax over the vocabulary.
- **Skip-Gram:** The reverse of CBOW — given a center word w_t, the model tries to predict each of the surrounding context words. It is especially effective for smaller datasets and infrequent words.

Objective Function

Formally, for a sequence $W = w_1, \ldots, w_T$ and a context window of size c, the Skip-Gram model maximizes the log-likelihood:

$$\sum_{t=1}^{T} \sum_{\substack{-c \leq j \leq c \\ j \neq 0}} \log p(w_{t+j} \mid w_t)$$

The conditional probability $p(w_{t+j} \mid w_t)$ is typically modeled using the softmax:

$$p(w_o \mid w_i) = \frac{\exp(u_{w_o}^\top v_{w_i})}{\sum_{w \in \mathcal{V}} \exp(u_w^\top v_{w_i})}$$

where v_{w_i} is the embedding of the input word, and u_{w_o} is the embedding used to predict output words.

To make this computationally feasible for large vocabularies, techniques like **negative sampling** or **hierarchical softmax** are used to approximate the objective.

Efficient Training: Negative Sampling and Hierarchical Softmax

The softmax function in the Word2Vec objective requires summing over the entire vocabulary $\mathcal{V}$:

$$p(w_o \mid w_i) = \frac{\exp(u_{w_o}^\top v_{w_i})}{\sum_{w \in \mathcal{V}} \exp(u_w^\top v_{w_i})}$$

This is computationally expensive for large vocabularies (e.g., millions of words), as it requires computing a dot product and normalization over all word pairs for every training step.

To address this, two common approximations are used:

1. Negative Sampling Instead of updating all words in the vocabulary, we update only a small number of output words per training example: one **positive** word (the true context word) and a small number of **negative** words, sampled from a noise distribution.

For each training pair (w_i, w_o), we optimize:

$$\log \sigma(u_{w_o}^\top v_{w_i}) + \sum_{j=1}^{k} \mathbb{E}_{w_j \sim P_n(w)} \left[\log \sigma(-u_{w_j}^\top v_{w_i}) \right]$$

where:

- $\sigma(z) = \frac{1}{1+\exp(-z)}$ is the sigmoid function,
- w_o is the true context word (positive example),
- w_j are k negative samples drawn from a noise distribution $P_n(w)$ (often proportional to unigram frequency raised to the 3/4 power).

This makes training highly efficient and empirically effective.

2. Hierarchical Softmax Hierarchical softmax replaces the flat softmax with a binary tree over the vocabulary, where each word corresponds to a leaf. The probability of a word is decomposed as the probability of a path from the root to the word's leaf node:

$$p(w_o \mid w_i) = \prod_{l=1}^{L(w_o)} \sigma(\pm u_l^\top v_{w_i})$$

Here, $L(w_o)$ is the sequence of internal nodes along the path to word w_o, and the $\pm$ sign depends on whether the left or right child is taken at each node.

This reduces the computational cost from $\mathcal{O}(|\mathcal{V}|)$ to $\mathcal{O}(\log |\mathcal{V}|)$, which is especially helpful for very large vocabularies.

Both methods enable Word2Vec to scale to large corpora and vocabularies, making it practical for real-world applications.

Properties of Word2Vec Embeddings

Word2Vec embeddings exhibit several remarkable properties:

- **Vector arithmetic:** Relationships between words can be captured algebraically, e.g.,

$$\texttt{king} - \texttt{man} + \texttt{woman} \approx \texttt{queen}$$

- **Semantic structure:** Words that occur in similar contexts tend to have similar embeddings, leading to clustering of semantically related words.
- **PMI approximation:** With negative sampling, it has been shown that the dot product of learned embeddings approximates a **shifted pointwise mutual information (PMI)** between word pairs:

$$u_c^\top v_w \approx \mathrm{PMI}(w, c) - \log k$$

where k is the number of negative samples.

In the next section, we will move beyond static embeddings and explore how sequence models — such as recurrent neural networks (RNNs) — can model contextual word usage and generate variable-length sequences.

Statistical Embedding Methods

Before the rise of neural embeddings, statistical methods were used to derive token representations from large corpora:

- **Co-occurrence Matrix**: Construct a matrix M where $M_{i,j}$ counts how often word w_i appears in the context of word w_j.
- **Pointwise Mutual Information (PMI)**: Normalize the co-occurrence matrix using PMI:

$$\mathrm{PMI}(w, c) = \log \frac{P(w, c)}{P(w)P(c)} = \log \frac{\#(w, c) \cdot N}{\#(w)\#(c)}$$

These methods capture global statistical relationships in the corpus, unlike Word2Vec which captures local context via predictive training.

However, it has been shown that with appropriate training modifications — such as Skip-Gram with negative sampling — Word2Vec implicitly factorizes a shifted PMI matrix. Specifically, the inner product of learned embeddings approximates:

$$u_c^\top v_w \approx \text{PMI}(w, c) - \log k$$

where k is the number of negative samples used during training.

In the next part of the lecture, we will explore recurrent neural networks (RNNs), sequence-to-sequence architectures, and attention mechanisms.

6.5 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a class of neural architectures tailored for sequential data. At each time step t, the network maintains a hidden state h_t that summarizes information from all previous inputs:

$$h_t = f(W_h h_{t-1} + W_x x_t + b)$$

where x_t is the input at time t, W_h and W_x are learned weight matrices, b is a bias term, and f is typically a non-linear activation function such as tanh or ReLU.

RNNs can, in principle, model long-range dependencies, making them applicable to tasks involving language modeling, speech recognition, and symbolic sequence generation (e.g., in mathematics). However, training RNNs over long sequences is difficult due to the **vanishing** and **exploding gradient** problems. These arise when gradients are propagated through many time steps, leading to unstable or negligible updates.

6.6 Sequence-to-Sequence Models (Seq2Seq)

Sequence-to-sequence (Seq2Seq) models are a foundational architecture for tasks where an input sequence must be transformed into an output sequence, potentially of different length. This includes applications such as machine translation, symbolic equation solving, and program synthesis.

The core idea is to decompose the task into two stages:

- **Encoder:** processes the input sequence $(x_1, x_2, \ldots, x_T)$ and encodes it into a fixed-size context vector c that captures the semantic summary of the sequence.
- **Decoder:** generates the output sequence $(y_1, y_2, \ldots, y_{T'})$ one token at a time, using the context vector c and its own previously generated outputs.

The transition from encoder to decoder introduces a crucial **information bottleneck**: the entire input sequence must be compressed into a single vector c. If the input is long or information-dense, this bottleneck can limit performance — the decoder has only access to a fixed summary and may forget fine-grained details. This motivates later enhancements such as attention mechanisms, which allow the decoder to dynamically attend to different parts of the input sequence during generation.

The model is trained to minimize the discrepancy between predicted and true sequences using a cross-entropy loss:

$$\mathcal{L} = -\sum_{t=1}^{T'} \log p_t(y_t)$$

where p_t is the predicted probability distribution over the vocabulary at step t, and y_t is the ground-truth token index.

To improve training efficiency and convergence, we commonly apply **teacher forcing**: instead of feeding the decoder's own prediction at time $t-1$ as input at step t, we feed the ground-truth token y_{t-1}. This prevents error accumulation during training and allows gradients to propagate more smoothly.

While Seq2Seq models with basic RNNs are powerful, their reliance on a fixed context vector can hinder their capacity for long sequences — an issue that attention mechanisms address, which we will study next.

6.7 Attention Mechanism

The **attention mechanism** is a powerful extension to the basic sequence-to-sequence architecture that alleviates the limitations of using a single fixed context vector. Instead of forcing the decoder to rely solely on the final encoder state, attention enables the decoder to dynamically access all encoder hidden states when generating each output token.

At decoding step t, the model computes a context vector as a weighted sum of the encoder hidden states:

$$\text{context}_t = \sum_{i=1}^{T} \alpha_{t,i} h_i$$

Here, h_i is the encoder hidden state at time i, and $\alpha_{t,i}$ is an attention weight that quantifies the relevance of h_i to the decoder's prediction at step t. These weights are learned during training and typically computed using a softmax over a similarity score between the decoder's current state and each encoder state.

Motivation and Intuition

The motivation behind attention is to overcome the **bottleneck** of fixed-length representations. Rather than compressing the entire input into a single vector, the model can "attend" to different parts of the input sequence as needed. This is especially crucial for long sequences or when the input contains multiple interdependent components.

Intuitively, attention acts like a soft lookup: for each output token, the model selectively focuses on the most relevant input tokens, enabling better alignment and richer context.

Attention significantly improves performance in tasks like translation, summarization, and symbolic reasoning. It has also laid the foundation for more advanced architectures like the **Transformer**, where attention is used pervasively — not only from decoder to encoder, but also within sequences themselves via *self-attention*.

In the next lecture, we will study the **Transformer model** in detail, along with its core innovations: **multi-head self-attention**, **positional encoding**, and **feedforward blocks**.

7 Lecture 7. Attention and Transformers.

7.1 Seq2Seq Attention

1. Sequence-to-Sequence Models

A **Sequence-to-Sequence (Seq2Seq)** model is a neural architecture designed to map an input sequence $\mathbf{x} = (x_1, \ldots, x_T)$ to an output sequence $\mathbf{y} = (y_1, \ldots, y_{T'})$. It consists of two parts:

- **Encoder:** Processes the input sequence and compresses it into a fixed-size context vector.
- **Decoder:** Generates the output sequence from this context vector.

Each component is typically implemented using RNNs, LSTMs, or GRUs.

2. Limitations of Vanilla Seq2Seq

The encoder must compress all the information into a single vector (the final hidden state), which creates a bottleneck, especially for long sequences. This can cause the model to forget important information from earlier tokens.

3. Attention Mechanism

To address the bottleneck, the **attention mechanism** allows the decoder to access all encoder hidden states $(\mathbf{h}_1, \ldots, \mathbf{h}_T)$ and assign different weights to them at each decoding step.

Bahdanau Attention (Additive)

For decoder hidden state $\mathbf{s}_t$ and encoder state $\mathbf{h}_i$, define the score function:

$$e_{ti} = \mathbf{v}^\top \tanh(\mathbf{W}_1 \mathbf{h}_i + \mathbf{W}_2 \mathbf{s}_t)$$

The attention weights α_{ti} are obtained via softmax:

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_j \exp(e_{tj})}$$

The context vector is:

$$\mathbf{c}_t = \sum_i \alpha_{ti} \mathbf{h}_i$$

Then, $\mathbf{c}_t$ is concatenated with $\mathbf{s}_t$ to produce the final output.

4. Decoder with Attention

At each decoding step t, the decoder uses the attention-weighted context $\mathbf{c}_t$ and previous hidden state $\mathbf{s}_{t-1}$ to compute:

$$\mathbf{s}_t = \text{RNN}([\mathbf{y}_{t-1}, \mathbf{c}_{t-1}], \mathbf{s}_{t-1})$$

$$\hat{y}_t = \text{Softmax}(W_o[\mathbf{s}_t; \mathbf{c}_t])$$

7.2 Transformers

To gain an intuitive and practical understanding of how transformer architectures work — especially in the context of sequence-to-sequence learning and attention mechanisms — we highly recommend the following interactive and visual resource:

NLP COURSE FOR YOU BY LENA VOITA

Although designed for language tasks, the principles apply directly to mathematical sequence learning tasks like Lyapunov function discovery. The attention mechanism is particularly important for handling variable-length symbolic or functional inputs in a structured way.

7.3 Lyapunov functions

Definition 7.1 (Global Lyapunov Function)**.** *Let the origin be an equilibrium point of the autonomous system*

$$\dot{x} = f(x), \quad f(0) = 0, \quad x \in \mathbb{R}^n.$$

*A continuously differentiable function $V : \mathbb{R}^n \to \mathbb{R}$ is called a **global Lyapunov function** if:*

1. *$V(x) > 0$ for all $x \neq 0$, and $V(0) = 0$ (positive definite),*
2. *$V(x) \to \infty$ as $\|x\| \to \infty$ (radially unbounded),*
3. *$\dot{V}(x) = \nabla V(x)^\top f(x) < 0$ for all $x \neq 0$ (negative definite derivative).*

*Then the equilibrium at the origin is **globally asymptotically stable**.*

Example 7.2 (Globally Asymptotically Stable System)**.** *Consider the nonlinear system:*

$$\dot{x}_1 = -x_1^3, \quad \dot{x}_2 = -x_2^3.$$

Define the Lyapunov function:

$$V(x) = \frac{1}{2}x_1^2 + \frac{1}{2}x_2^2.$$

Clearly, $V(x) > 0$ for all $x \neq 0$, $V(0) = 0$, and $V(x) \to \infty$ as $\|x\| \to \infty$. Compute the derivative:

$$\dot{V}(x) = x_1\dot{x}_1 + x_2\dot{x}_2 = x_1(-x_1^3) + x_2(-x_2^3) = -x_1^4 - x_2^4 < 0 \quad \textit{for all } x \neq 0.$$

Thus, V is a global Lyapunov function and the origin is globally asymptotically stable.

If such a function exists, the system is stable at the origin.

Key Challenge: Discovering Lyapunov functions automatically from system dynamics is a fundamental but difficult problem in control theory.

Transformer Model and Training Setup

Authors used a following architecture:

- Architecture: Transformer encoder-decoder.
- Embedding dimension: 640, Layers: 8, Heads: 10.
- Optimizer: Adam with learning rate 10^{-4}, warm-up of 10,000 steps.
- Schedule: Inverse square root decay.
- Batch size: 16 examples.

Goal: Learn to predict valid Lyapunov functions from tokenized representations of dynamical systems.

Dataset Generation Strategies

In the task of discovering **global Lyapunov functions**, generating appropriate training data is a critical design choice. We consider two main strategies:

Forward Generation

In **forward generation**, we begin with a dynamical system:

$$\dot{x} = f(x),$$

and attempt to compute or verify the existence of a global Lyapunov function $V(x)$. This approach reflects the original problem formulation and produces realistic training pairs (f, V), but it is often computationally intractable due to the difficulty of verifying global stability or finding a valid Lyapunov function.

Backward Generation

In **backward generation**, we start by sampling candidate Lyapunov functions $V(x)$, and then construct systems $f(x)$ for which V is a valid Lyapunov function. For example, we can define:

$$f(x) := -\nabla V(x),$$

so that:

$$\dot{V}(x) = \nabla V(x)^\top \dot{x} = \nabla V(x)^\top(-\nabla V(x)) = -\|\nabla V(x)\|^2 < 0 \quad \text{for all } x \neq 0.$$

This ensures that V is indeed a Lyapunov function.

Problem: While this guarantees validity, it introduces a serious bias: the model may simply learn to **integrate gradients**, which is a different task from learning general Lyapunov reasoning. As a result, the model may fail to generalize to systems not derived via this construction. Thus, this form of backward generation teaches the wrong inductive bias.

Further Reading: *Discovering Lyapunov Functions with Transformers,* NeurIPS 2023. Available at: `https://arxiv.org/abs/2302.12227`

7.4 Learning Linear Algebra with Transformers

Paper: *Linear Algebra with Transformers*, François Charton, Meta AI, 2021

Link: `https://arxiv.org/abs/2112.01898`

Summary: This paper explores how transformers can be trained to perform a wide range of symbolic linear algebra tasks — including computing matrix products, inverses, determinants, eigenvalues, and solving linear systems. Remarkably, the models learn to perform these computations symbolically rather than numerically, generalizing to matrix sizes beyond those seen during training.

Why is this relevant? Even though linear algebra tasks seem far removed from real-world machine learning benchmarks, their successful realization by transformers demonstrates the model's ability to:

- Learn structured symbolic computation purely from data,
- Generalize algorithmic reasoning beyond training examples,
- Capture mathematical patterns and identities in sequence form.

This provides encouraging evidence that transformers can learn similar forms of reasoning required to discover **Lyapunov functions**, where both symbolic structure and global generalization are essential. Such examples support the broader thesis that transformers can assist in tasks of formal mathematical reasoning and automated theorem discovery.

8 Guest Lecture

9 FunSearch

Pre-trained Models

Modern Large Language Models (LLMs) such as **GPT-4o**, **LLaMA**, **Claude**, and **Gemini** are trained on massive and diverse datasets. These models exhibit broad general knowledge across domains and are capable of solving a wide range of problems. However, when it comes to domain-specific or task-specific applications, this general knowledge must be adapted or directed appropriately.

There are two main strategies for leveraging pre-trained LLMs in specific tasks:

- **Prompting**: This is the default and most accessible approach. Prompting involves crafting input queries that guide the model to perform a desired task. For example, to solve a programming task, one might prompt the model with: `"Write a Python function that computes the factorial of a number"`. Prompt engineering — the process of refining prompts — can dramatically influence model performance, especially for tasks requiring structured or multi-step reasoning.
- **Fine-tuning**: When prompting is insufficient to achieve the required performance, we can fine-tune the model on a dataset consisting of task-specific examples. This process adapts the model's internal representations to the nuances of the particular task, often leading to significantly improved accuracy and reliability. Fine-tuning is more resource-intensive and typically requires access to model weights and training infrastructure.

In general, **prompting should be the first line of approach**. It is lightweight, flexible, and often sufficient. Fine-tuning should be considered only when prompt-based interaction fails to yield satisfactory results or when consistent behavior across similar inputs is required.

Temperature in Sampling

In language models and other generative settings, the **temperature parameter** $t > 0$ is used to control the sharpness of the output distribution produced by the softmax function.

Given a vector of logits $(x_1, x_2, \ldots, x_n)$, the temperature-scaled softmax is defined as:

$$\text{Softmax}(x_1, \ldots, x_n; t) = \left(\frac{e^{x_1/t}}{\sum_{i=1}^{n} e^{x_i/t}}, \frac{e^{x_2/t}}{\sum_{i=1}^{n} e^{x_i/t}}, \ldots, \frac{e^{x_n/t}}{\sum_{i=1}^{n} e^{x_i/t}} \right)$$

The temperature t scales the logits before the exponentiation, and its effect on the resulting distribution is as follows:

- **As** $t \to 0$: The distribution becomes increasingly peaked around the largest logit value. In the limit, the softmax approaches a one-hot distribution centered at the $\arg\max_i x_i$. However, this limit is **not used in practice** because very small temperatures cause severe **numerical instability** due to large exponentials and division by very small numbers. Instead, deterministic `argmax` is implemented directly in "greedy decoding" or "no-sampling" modes.
- **As** $t \to \infty$: The scaled logits tend to zero, and the softmax approaches the uniform distribution:
$$\lim_{t \to \infty} \text{Softmax}(x_1, \ldots, x_n; t) = \left(\frac{1}{n}, \ldots, \frac{1}{n} \right)$$
This results in maximum entropy (maximum randomness), which is usually not desirable for coherent text generation.

In practice, temperature values are typically chosen in the range $t \in [0.6, 1.0]$. Values larger than 1 often lead to incoherent or erratic outputs, while smaller values (e.g., $t = 0.6$) encourage more confident and focused predictions without being completely deterministic.

Recommendation: Use a moderate temperature (e.g., $t = 0.6$) for high-quality and diverse outputs.

Top-k and Top-p Sampling

When generating text from language models, it is often useful to avoid sampling from the full softmax distribution. Instead, we can restrict sampling to a more manageable and meaningful subset of candidate tokens. Two popular techniques for this are **top-k sampling** and **top-p (nucleus) sampling**.

Top-k Sampling. In top-k sampling, we restrict the sampling distribution to only the k most likely tokens, and set the probability of all other tokens to zero. The resulting distribution is then renormalized.

- **Example:** Suppose a model outputs the following token probabilities:

  `[hello: 0.35, world: 0.25, cat: 0.15, dog: 0.10, car: 0.08, ...]`

  With $k = 3$, we keep only the top 3 tokens: `hello`, `world`, and `cat`. After renormalizing:
$$P'(\texttt{hello}) = \frac{0.35}{0.75}, \quad P'(\texttt{world}) = \frac{0.25}{0.75}, \quad P'(\texttt{cat}) = \frac{0.15}{0.75}$$
  The rest are discarded.

This method prevents low-probability tokens from being sampled, reducing noise and improving coherence. However, it uses a fixed cutoff that ignores context.

Top-p (Nucleus) Sampling. In top-p sampling, we choose the smallest set of top tokens whose cumulative probability exceeds a threshold $p \in (0, 1)$. This means the number of tokens included varies depending on the output distribution's shape.

- **Example:** Using the same distribution:

  `[hello: 0.35, world: 0.25, cat: 0.15, dog: 0.10, car: 0.08, ...]`

If $p = 0.9$, we accumulate from the top:

$$\texttt{hello (0.35)} + \texttt{world (0.25)} + \texttt{cat (0.15)} + \texttt{dog (0.10)} = 0.85 \quad \text{(not enough)}$$

Adding `car (0.08)` gives $0.93 > 0.9$, so we keep `hello`, `world`, `cat`, `dog`, and `car`, and discard the rest. These are then renormalized before sampling.

This method is adaptive — more flexible than top-k — and better handles flat or steep probability distributions.

Comparison.

- **Top-k** ensures a fixed number of candidate tokens, regardless of their probability mass.
- **Top-p** dynamically adapts to the shape of the distribution and often leads to more stable generation in practice.

In practice: Many modern LLM applications use top-p sampling with p in the range $[0.8, 0.95]$, optionally combined with temperature scaling. Top-k is sometimes used with small values (e.g., $k = 50$) for tighter control.

Evolutionary Algorithms

Evolutionary algorithms are a class of optimization techniques inspired by the principles of natural selection and biological evolution. The core idea is often summarized as **"survival of the fittest"**: in each generation, we preserve and propagate the solutions that perform best according to some fitness criterion.

These algorithms are particularly useful for black-box or non-differentiable optimization problems, where gradient-based methods may not be applicable.

Basic Idea. The algorithm operates on a **population** of candidate solutions. In each iteration (or generation), the population is updated via variation (e.g., random perturbations) and selection (choosing the best-performing candidates).

Example: Minimizing a Function. Suppose we want to minimize a real-valued function $f : \mathbb{R}^d \to \mathbb{R}$. An evolutionary approach might proceed as follows:

1. **Initialize** a population of N candidate points $x_1, x_2, \ldots, x_N \in \mathbb{R}^d$, sampled randomly.
2. For each point x_i, generate a perturbed version $x_i' = x_i + \varepsilon_i$, where $\varepsilon_i \sim \mathcal{N}(0, \sigma^2 I)$.
3. Evaluate $f(x_i')$ and compare to $f(x_i)$. Keep the version with lower function value:
$$x_i \leftarrow \arg\min\{f(x_i), f(x_i')\}$$
4. **Repeat** the perturbation and selection process for multiple generations.

Over time, the population "evolves" toward regions of lower function value. This behavior mimics natural evolution: random mutations introduce variation, and selection favors individuals with higher fitness (i.e., lower function value).

Remarks.

- This class of algorithms does not require derivatives or function continuity.
- They can be parallelized easily, as evaluations are independent.
- They may converge slowly and are sensitive to the choice of mutation size σ.

FunSearch: Program Search with Large Language Models

In this section, we discuss the **FunSearch** framework introduced in the paper *"Mathematical discoveries from program search with large language models"* (2023). This approach demonstrates how Large Language Models (LLMs), when guided properly, can contribute to solving mathematical problems — not merely by completing or verifying formal proofs, but by synthesizing useful programs or constructions.

FunSearch combines the generative power of LLMs with an evolutionary selection loop. Instead of relying on prompting or fine-tuning alone, it employs a **search-and-evaluate** paradigm, in which the LLM proposes candidate programs, and these programs are evaluated according to a domain-specific objective. High-performing programs are preserved and mutated, while low-performing ones are discarded — a principle reminiscent of evolutionary algorithms.

This framework has been successfully applied to several challenging mathematical problems, including:

- Constructing large cap sets in $\mathbb{F}_3^n$
- Solving Online Bin Packing problem
- Finding counterexamples or extremal constructions in combinatorics

The key insight is that LLMs are not just passive repositories of knowledge — they can be used to actively explore large combinatorial search spaces when paired with appropriate selection and mutation mechanisms.

Cap Set Problem and the Game of SET

One of the most well-known applications of the FunSearch framework is in the context of the **cap set problem** in the finite vector space $\mathbb{F}_3^n$, where $\mathbb{F}_3 = \{0, 1, 2\}$ denotes the field with 3 elements.

Definition (Cap Set). A subset $A \subseteq \mathbb{F}_3^n$ is called a **cap set** if it contains no three distinct elements $x, y, z \in A$ such that:

$$x + y + z = 0 \quad (\text{in } \mathbb{F}_3^n)$$

Equivalently, a cap set contains no **three-term arithmetic progression**, since if $x+y+z = 0$, then $z = -x - y = x + (x - y)$, i.e., the three elements lie on a line in the affine space over $\mathbb{F}_3$.

Motivation. The central question in cap set research is:

$$\gamma := \sup_n c_n^{1/n}, \quad \text{where } c_n = \max\{|A| : A \subseteq \mathbb{F}_3^n \text{ is a cap set}\}$$

It is known that $2^n < c_n < 3^n$, but finding precise asymptotics for c_n has been a major challenge in additive combinatorics.

Connection to the Game of SET. The popular card game **SET** provides an intuitive model for this problem. Each SET card can be represented as a vector in $\mathbb{F}_3^4$, where each coordinate corresponds to one of four features (shape, color, shading, number), each taking one of three values.

A **SET** (in the game) is a triple of cards whose corresponding vectors $x, y, z \in \mathbb{F}_3^4$ satisfy:

$$x + y + z = 0 \pmod 3$$

Thus, a collection of SET cards with no valid SET among them corresponds precisely to a **cap set** in $\mathbb{F}_3^4$.

This makes the game of SET a tangible example of cap sets. The largest known cap set in $\mathbb{F}_3^4$ has size 20 (out of the total $3^4 = 81$ cards), and FunSearch has been used to improve lower bounds for larger n, helping to close the gap between known constructions and theoretical upper bounds.

How FunSearch Works: Cap Set Discovery

The central goal of FunSearch in the cap set problem is to automatically generate programs that construct large cap sets in $\mathbb{F}_3^n$. These constructions must not only yield high scores (i.e., large cap sets) but also exhibit **generalizable structure** — they should reflect underlying mathematical patterns rather than overfitting to a particular value of n.

Program Structure. Rather than asking the LLM to directly generate a complete function that outputs a cap set, FunSearch takes a more modular approach. It focuses the model's attention on learning a **scoring function**

$$f : \mathbb{F}_3^n \to \mathbb{R}$$

This function is interpreted as assigning a score to each point in $\mathbb{F}_3^n$, indicating how promising it is to be included in a cap set.

Greedy Selection. Given a learned scoring function f, the actual cap set is constructed using a greedy algorithm:

1. Start with an empty set $A = \emptyset$.
2. Sort all points $x \in \mathbb{F}_3^n$ in decreasing order of $f(x)$.
3. Add points to A one by one, skipping any x that would form a three-term arithmetic progression with two existing elements in A.

This strategy allows the model to influence the construction process indirectly via f, while the combinatorial constraints (avoiding 3-term progressions) are enforced by the greedy selection logic.

Motivation and Design Philosophy. This design is motivated by the desire to evolve only the **core logic** governing how good solutions are formed — rather than forcing the model to memorize large sets or generate rigid constructions. The scoring function acts as an interpretable and flexible interface between the LLM and the problem domain, making it easier to identify and preserve useful patterns across generations.

Key idea: Separate *what to learn* (scoring function f) from *how to build* (greedy combinatorial routine). This modularity helps achieve generalization and enables the use of evolutionary techniques for iterative improvement.

Example: Learned Scoring Functions in Code

To illustrate how FunSearch guides the generation of cap sets, consider the following example from the original paper. The LLM is tasked with generating `priority` functions that assign a score to elements of $\mathbb{F}_3^n$, guiding their inclusion in a greedy cap set construction.

The following is an excerpt from the prompt used to generate new candidate scoring functions. Rather than asking the model to build a solution from zero, the prompt includes previously sampled functions that define the `priority` logic:

```
"""Finds large cap sets."""
import numpy as np
import utils_capset

def priority_v0(element, n):
    """Returns the priority with which we want to add `element` to the cap set."""
    #######
    # Code from lowest-scoring sampled program.
    return ...
    #######

def priority_v1(element, n):
```

```
    """Improved version of `priority_v0`."""
    #######
    # Code from highest-scoring sampled program.
    return ...
    #######

def priority_v2(element, n):
    """Improved version of `priority_v1`."""
    #######
```

Few-shot Structure. Two previous versions, `priority_v0` and `priority_v1`, are included in the prompt as examples. This provides the model with a minimal history of evolving solutions — one low-performing, and one high-performing. By seeing these examples, the model can:

- infer which patterns may be effective (based on performance),
- preserve useful structure (e.g., syntax, variable usage),
- generate plausible improvements in a controlled format.

This setup exploits the model's ability to recognize and generalize patterns from just a few examples, enabling iterative improvement without fine-tuning or large training datasets.

Avoiding Overfitting During Search

One of the challenges in any search-based learning system — particularly those involving neural models or program synthesis — is the risk of **overfitting** to specific examples or narrow regions of the search space. In the context of FunSearch, this means generating programs that perform in the same way.

To mitigate this, the authors introduced a clever strategy inspired by ideas from cross-validation and ensemble learning. The key idea is to periodically reshuffle the search space and allow diversity to emerge across multiple independent search trajectories.

Strategy Overview.

- **Partitioning:** The database of programs is split into k parts. Every 4 hours, the current population of programs is divided into k disjoint subsets.
- **Independent Evolution:** Each subset evolves independently for 4 hours. During this time, programs in each part are mutated, evaluated, and selected according to standard evolutionary procedures — but without cross-interaction between groups.
- **Reshuffling at Epoch Boundaries:** At the beginning of a new epoch (i.e., after each 4-hour cycle), the programs are reshuffled randomly across the k groups. This prevents any one group from over-specializing or getting stuck in a local optimum.
- **Evolutionary Updates:** After reshuffling, new evolutionary steps (mutation, evaluation, selection) are applied to the fresh groupings. This process introduces diversity while still retaining high-quality solutions from earlier stages.

Purpose and Benefits. This structured, periodic diversification serves multiple purposes:

- It prevents early convergence to poor local optima by encouraging broader exploration.
- It reduces the risk of overfitting to a specific input space or scoring idiosyncrasy.
- It increases the probability that useful patterns discovered in one group will influence others after reshuffling.

Specification of the Search Procedure

To guide the program synthesis process in FunSearch, two key components are provided as part of the search specification:

1. Reference Function: `Solve` The reference function `Solve` serves as the **starting point** for the search. It is a function that takes as input the dimension n and returns a candidate cap set in $\mathbb{F}_3^n$. In practical terms, it encodes a baseline or known construction method that generates a valid, if not necessarily optimal, cap set.

Formally:

$$\texttt{Solve}(n) \longrightarrow A \subseteq \mathbb{F}_3^n$$

The returned set A is used to seed the initial population or guide the model in understanding the structure of valid solutions.

2. Evaluation Function: `Evaluate` The evaluation function `Evaluate` plays a critical role in the evolutionary loop by providing a quantitative assessment of each candidate solution. It takes as input a proposed cap set and returns a numerical score indicating its quality.

- If the input set is a valid cap set (i.e., contains no three-term arithmetic progressions), the function returns its size $|A|$.
- If the input is invalid, the function returns 0 — effectively discarding the solution from consideration.

Formally:

$$\texttt{Evaluate}(A) = \begin{cases} |A|, & \text{if } A \text{ is a cap set} \\ 0, & \text{otherwise} \end{cases}$$

Purpose. Together, `Solve` and `Evaluate` define the interface through which LLM-generated programs are assessed. They enable:

- Bootstrapping the search from known constructions,
- Rigorous filtering of incorrect outputs,
- A consistent scoring signal for selection and mutation.

This specification separates the learning objective (build a good scoring function) from the correctness constraint (cap set validity), allowing the model to focus on heuristics while correctness is enforced externally.

A key theoretical quantity in the study of cap sets is the exponential growth rate of the largest cap set size in $\mathbb{F}_3^n$, defined as:

$$\gamma := \sup_n \left(c_n^{1/n}\right) \quad \text{where } c_n = \max\{|A| : A \subseteq \mathbb{F}_3^n \text{ is a cap set}\}$$

This constant γ captures the asymptotic density of the largest cap sets and is a central object of interest in additive combinatorics.

Using FunSearch, one of the authors was able to construct a cap set that led to a new best-known lower bound:

$$2.2202 < \gamma$$

This improves upon the previously best-known bound:

$$2.2180 < \gamma$$

The known upper bound remains unchanged:

$$\gamma < 2.756$$

Significance. Although the numerical improvement in the lower bound is small, it demonstrates the capacity of LLM-guided program synthesis to contribute to real mathematical discovery. In particular, it shows that FunSearch is capable of generating novel constructions that are competitive with or exceed existing human-devised methods.

What Makes a Task Suitable for FunSearch?

Not all problems are equally well-suited for the FunSearch framework. According to the authors, several key conditions must be met in order for the evolutionary search process guided by LLMs to succeed.

Core Requirements (from the authors):

- **Efficient Evaluation Function:** The evaluation function must be fast to compute. Since it is invoked repeatedly across many candidate programs, any overhead can significantly slow the search process.
- **Rich Scoring Signal:** The evaluation must provide more than binary feedback (e.g., valid/invalid). A scalar reward — such as the size of the cap set — helps distinguish between better and worse solutions and guides selection effectively.
- **Modular Program Structure:** Only a small, critical part of the algorithm should be subject to LLM-driven evolution (e.g., a scoring function), while the rest (e.g., the greedy search routine) remains fixed and reliable.

Additional Criterion (from critique): Beyond the explicit design choices, critiques of FunSearch highlight another implicit but essential requirement:

- **Diverse and Rich Search Space:** The space of possible candidate functions must be expressive enough to admit a wide range of solutions — some good, some bad — so that the evolutionary process has room to operate. If all functions are similarly effective (or ineffective), the search becomes uninformative.

Other Problems Tackled by FunSearch

In addition to the cap set problem, the FunSearch framework has been applied to several other challenging problems in mathematics and theoretical computer science. These tasks share key characteristics that make them well-suited for LLM-guided program evolution: they involve combinatorial structures, have well-defined objective functions, and allow for modular, heuristic-based approaches.

1. Online Bin Packing. This is a classic problem in **combinatorial optimization**, where items with varying sizes arrive one at a time, and must be placed into bins of fixed capacity without knowledge of future items. The challenge lies in minimizing the total number of bins used under these **online constraints**.

FunSearch was used to evolve heuristics for making bin-packing decisions on the fly — yielding strategies that outperformed several human-designed baselines.

2. Shannon Capacity of a Graph. This problem arises at the intersection of **combinatorics and information theory**. The Shannon capacity of a graph quantifies the maximum rate at which information can be transmitted through a noisy channel without confusion, where the channel's constraints are modeled by the graph.

3. Corners Problem. The corners problem is a well-known challenge in additive combinatorics and discrete geometry. It asks: how large can a subset of a grid (e.g., $[N]^2$) be, such that it contains no "corner" — a triple of points forming an L-shape:

$$(x, y), \quad (x + d, y), \quad (x, y + d)$$

Online Bin Packing: Classical vs. FunSearch Heuristics

The online bin packing problem involves assigning a sequence of items to bins of fixed capacity as they arrive, without knowledge of future items. The goal is to minimize the number of bins used, under the constraint that each item must be packed as it arrives.

Classical Heuristics. Two well-known greedy heuristics are:

- **First-Fit:** Number the bins in order. Place each item into the first bin that has enough space.
- **Best-Fit:** Place each item into the tightest (most filled) bin where it still fits.

These heuristics are simple, fast, and widely used — but they are not optimal and can be improved.

FunSearch-Discovered Heuristic. FunSearch was used to evolve a more complex scoring function that evaluates bins based on multiple interacting factors, leading to improved performance on benchmark datasets. The evolved function assigns a priority score to each bin and selects the bin with the highest score.

The key logic of the discovered heuristic can be described as follows:

- Start with a high default score (e.g., 1000) for each bin.
- Penalize bins with large remaining capacity: subtract a factor proportional to `bins` $\times$ `(bins - item)`.
- Identify the bin with the best fit (least remaining space after insertion).
- Scale its score by the item size (encouraging larger items to go into tight bins).
- Penalize this bin further if the fit is not tight: subtract $(\texttt{bins[index]} - \texttt{item})^4$.

This leads to more nuanced behavior than either first-fit or best-fit, balancing between efficient space use and tight packing. The heuristic was automatically discovered — not explicitly designed — and yet outperformed classic strategies in controlled experiments.

Online Bin Packing: FunSearch-Discovered Heuristic

The online bin packing problem requires placing a sequence of items into bins of fixed capacity as they arrive, one by one, without knowledge of future items. The goal is to use as few bins as possible.

Classical Heuristics. Two widely used greedy heuristics are:

- **First-fit:** Iterate through the bins in order and place the item in the first bin where it fits.
- **Best-fit:** Place the item in the bin that will be most full after inserting it (i.e., the bin with the tightest fit).

While simple and efficient, these methods may yield suboptimal results for certain sequences. FunSearch was used to explore a space of heuristics.

FunSearch-Discovered Heuristic. Below is an example of a bin packing heuristic discovered via FunSearch:

```
def heuristic(item: float, bins: np.ndarray) -> np.ndarray:
    """Online bin packing heuristic discovered with FunSearch."""
    score = 1000 * np.ones(bins.shape)
    # Penalize bins with large capacities.
    score -= bins * (bins - item)
    # Extract index of bin with best fit.
    index = np.argmin(bins)
```

```
    # Scale score of best fit bin by item size.
    score[index] *= item
    # Penalize best fit bin if fit is not tight.
    score[index] -= (bins[index] - item)**4
    return score
```

Explanation. This heuristic computes a score for each bin and selects the one with the highest score (lowest penalty) to place the current item. Its behavior departs from classical strategies like first-fit or best-fit in important ways.

Key features include:

- A base score of 1000 is initialized for each bin.
- All bins are penalized based on their current remaining capacity: `bins * (bins - item)`. This discourages placing items in bins with large residual space.
- The bin with the smallest remaining capacity is identified, but not necessarily selected. Instead, the score of this **best-fit candidate** is scaled by the item size and then **penalized further** based on how loosely the item fits into it.
- In particular, the penalty term `(bins[index] - item)4` grows rapidly when the item is much smaller than the bin's remaining space — discouraging the use of the best-fit bin unless the fit is extremely tight.

Interpretation. Unlike classical heuristics that either blindly prefer the first bin that fits or the tightest fit, this evolved heuristic introduces a **deliberate hesitation** to use the best-fit bin unless the fit is nearly perfect.

Reproducing the Results: Resources and Cost

Running FunSearch at the scale presented in the original experiments involves significant computational infrastructure and parallel execution. Below is a summary of the configuration and associated cost estimates reported by the authors.

Compute Resources.

- **LLM Inference:** 15 instances of `StarCoder-15B` were used in parallel, each running on an **A100 40GB GPU**.
- **Evaluation Workers:** 5 CPU servers, each running 32 independent evaluator processes (160 evaluators in total), were used to compute the fitness scores of candidate programs.
- **Runtime:** The full run took approximately **2 days**.

Estimated Cost.

- The total monetary cost of running this configuration on **Google Cloud Platform (GCP)** is estimated between **$800 and $1400**, depending on pricing variability and runtime utilization.

Energy Consumption.

- The total energy used across all resources is estimated between **250–500 kWh**. This includes both GPU-based LLM inference and CPU-based evaluation processes.

10 Reinforcement Learning: General Setting

Reinforcement Learning (RL) is a foundational paradigm in machine learning, focused on sequential decision making. Unlike supervised learning, which relies on labeled data, RL is driven by interaction between a learner (the **agent**) and a dynamic **environment**.

10.1 General Definition

Core Components. At the heart of RL lies a feedback loop between the agent and the environment:

- **Agent:** An entity that makes decisions by selecting actions based on observations of the environment.
- **Environment:** The external system with which the agent interacts. It maintains a hidden or observable **state**, which evolves over time in response to the agent's actions.
- **Action:** At each time step t, the agent selects an action a_t from a set of possible actions $\mathcal{A}$.
- **State:** The environment is in a state $s_t \in \mathcal{S}$ at time t, which summarizes everything relevant about the current situation.
- **Reward:** After taking action a_t, the environment returns a scalar feedback signal $r_t \in \mathbb{R}$, indicating the immediate quality of the agent's decision.
- **Transition:** The environment moves to a new state s_{t+1} according to some dynamics (often stochastic), possibly influenced by s_t and a_t.

Interaction Cycle. The RL process unfolds over discrete time steps $t = 0, 1, 2, \ldots$. At each step:

$$s_t \xrightarrow{a_t} (r_t, s_{t+1})$$

That is:

1. The agent observes the state s_t.
2. It chooses an action a_t using its policy.
3. The environment responds with a reward r_t and transitions to the next state s_{t+1}.

Goal. The agent's objective is to learn a policy — a rule for selecting actions — that maximizes the **expected cumulative reward** over time. This involves balancing:

- **Exploration:** Trying new actions to discover their effects.
- **Exploitation:** Using known information to choose actions that yield high rewards.

Mathematical Formalism. This interaction is typically modeled as a **Markov Decision Process (MDP)** — a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ where:

- $\mathcal{S}$: set of states
- $\mathcal{A}$: set of actions
- $P(s' \mid s, a)$: transition probability function
- $R(s, a, s')$: reward function
- $\gamma \in [0, 1]$: discount factor

The agent's task is to learn a policy $\pi(a \mid s)$ that maximizes:

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

This framework is highly general and supports diverse applications — from game playing and robotics to mathematical discovery processes where feedback is sparse and delayed.

Example: Frozen Lake Environment

To ground the abstract reinforcement learning framework in a concrete setting, consider the well-known **Frozen Lake** environment, often used for RL experimentation.

Environment Description. The agent navigates a grid representing a frozen lake. Some tiles are safe, while others are holes (the agent "falls in" if stepped on). The goal is to reach a designated goal tile without falling into a hole.

Formal Components.

- **States:** Each possible position of the agent on the grid. If the grid is $n \times n$, there are n^2 states, typically encoded as discrete indices.
- **Actions:** The agent can move in one of four directions:

$$\mathcal{A} = \{\texttt{left}, \texttt{right}, \texttt{up}, \texttt{down}\}$$

- **Transitions:** The environment is often stochastic — intended movement may not succeed due to slipperiness, and the agent may slide in an unintended direction with some probability.
- **Rewards:**
  - -10 for stepping into a hole (i.e., falling into the lake),
  - $+100$ for reaching the goal (e.g., bottom-left tile),
  - 0 for all other transitions (i.e., safe but non-terminal movements).

Goal. The agent's objective is to learn a policy that maximizes the expected cumulative reward — that is, to reach the goal while avoiding holes, ideally in the fewest steps possible.

This environment is a simple yet rich setting for studying:

- the effect of stochasticity on optimal policies,
- exploration vs. exploitation strategies,
- model-based vs. model-free approaches.

Observability and Challenges. Does the agent see the layout?

In the standard version of the Frozen Lake environment, the agent **does not see the full layout of the lake**. It only observes its current position (state) and must learn through experience which actions are safe and which are dangerous. The locations of holes and the goal are not revealed explicitly.

This makes the task **partially observable and exploration-driven**. The agent must infer the structure of the environment over time by trial and error, receiving feedback in the form of rewards or penalties.

How can it avoid holes if it doesn't see them?

The agent learns to avoid holes through **reinforcement learning**. Specifically:

- When it steps on a hole and receives a reward of -10, this negative feedback is used to adjust its policy or value function.
- Over many episodes, it can estimate the expected return from each state-action pair. This encourages the agent to favor paths that lead to the goal and avoid actions that frequently result in negative outcomes.

Policy: The Agent's Strategy

In reinforcement learning, a **policy** defines the agent's behavior. It specifies how the agent chooses actions based on the current state of the environment.

Definition. A policy is a rule or function π that maps states to actions (or distributions over actions). It guides the agent's decision-making at each time step.

- **Deterministic Policy:**
$$\pi(s) = a$$
This means the agent always chooses action a whenever it is in state s. The policy is a function $\pi : \mathcal{S} \to \mathcal{A}$.
- **Stochastic Policy:**
$$\pi(a \mid s) = \mathbb{P}(a_t = a \mid s_t = s)$$
This gives the probability of selecting action a when the agent is in state s. The policy is a function $\pi : \mathcal{S} \to \Delta(\mathcal{A})$, where $\Delta(\mathcal{A})$ denotes the space of probability distributions over actions.

Usage.

- Deterministic policies are simple to execute and interpret, and often used in environments with deterministic dynamics.
- Stochastic policies are essential in settings where randomness helps with exploration or where the environment is stochastic and partially observable. Many modern RL algorithms (e.g., policy gradient methods) optimize directly over stochastic policies.

Goal of Learning. In both cases, the aim of reinforcement learning is to discover a policy π that maximizes the expected return — the cumulative reward over time.

Types of Reinforcement Learning Methods

Reinforcement Learning methods are typically categorized into two broad families based on how they interact with the environment:

1. Model-Free Methods. In model-free reinforcement learning, the agent interacts with the environment directly and does not attempt to learn a model of the environment's dynamics. Instead, the agent learns either:

- a **value function** that estimates expected future rewards (value-based methods), or
- a **policy** that directly maps states to actions (policy-based methods).

Examples of Model-Free Methods:

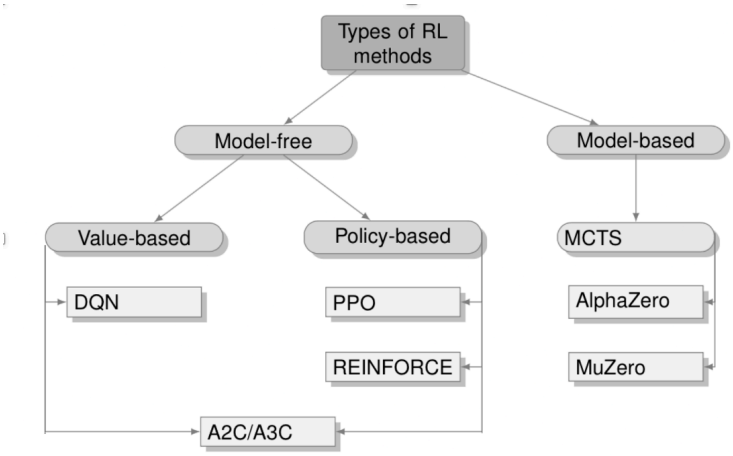
- **Value-based:** Deep Q-Networks (DQN)
- **Policy-based:** REINFORCE, Proximal Policy Optimization (PPO)
- **Hybrid:** Actor-Critic methods like A2C (Advantage Actor Critic) and A3C (Asynchronous A2C)

2. Model-Based Methods. In model-based RL, the agent learns (or is given) a model of the environment's transition dynamics and reward function. This model is then used to plan ahead and simulate outcomes before taking actions.

Examples of Model-Based Methods:

- Monte Carlo Tree Search (MCTS)
- **AlphaZero**: Combines MCTS with neural networks to plan in perfect information games like Chess and Go.
- **MuZero**: Builds a latent-space model of the environment, enabling planning without explicit access to the true transition function.

Visual Taxonomy. The figure below outlines this classification:



Model-Free vs. Model-Based: The Frozen Lake Example

To better understand the distinction between model-free and model-based reinforcement learning, consider how each approach operates in the context of the **Frozen Lake** environment.

Model-Free Approach.

In a model-free setting:

- The agent observes only its current **state index** (i.e., its position on the grid).
- It takes an action (e.g., move left, right, up, or down).
- It receives a reward and observes the next state.
- No attempt is made to model the underlying transition dynamics or map of the lake.

Learning occurs solely from experience. The agent gradually estimates the value of actions in each state through repeated interaction with the environment (e.g., using Q-learning). It does not explicitly reason about which transitions are more or less likely — it simply learns what works.

Model-Based Approach.

In contrast, a model-based agent:

- Either **knows** or **learns** the transition probabilities $P(s' \mid s, a)$.
- For example, it understands how likely it is to *slip* to an unintended tile when trying to move in a given direction.
- It reconstructs or is given access to the full structure of the state space (i.e., the layout of safe tiles and holes).
- It uses this transition model to simulate future outcomes, enabling planning-based strategies.

Summary of the Difference.

- **Model-free:** Learns action values directly from trial and error. No internal model of how the environment works.
- **Model-based:** Builds or uses an internal model of state transitions and outcomes to plan actions ahead of time.

Lecture Focus. In this lecture, we will begin with the **model-free** branch of reinforcement learning, which does not rely on simulating the environment's internal dynamics. Specifically, we will study a simple and elegant method known as the:

Cross-Entropy Method (CEM)

CEM provides a stochastic, optimization-based approach to solving reinforcement learning problems and is a practical starting point for understanding how policies can be discovered without access to gradients or environment models.

The Cross-Entropy Method (CEM)

The **Cross-Entropy Method (CEM)** is a stochastic optimization technique that maintains and iteratively refines a probability distribution over candidate solutions. Rather than using gradients, CEM relies on sampling and evaluation to navigate the solution space.

Algorithm Overview. Each iteration of the CEM algorithm consists of the following steps:

1. **Sampling:** Draw a set of candidate solutions from the current probability distribution.
2. **Evaluation:** Assess the performance (or fitness) of each sample according to a task-specific objective function.
3. **Selection:** Identify a subset of the best-performing samples, often called the **elite set**.
4. **Update:** Adjust the parameters of the sampling distribution to increase the likelihood of generating elite solutions in future iterations.

Over time, this iterative refinement increases the probability of producing high-performing solutions, guiding the search toward optimal or near-optimal regions of the space.

CEM as a General Strategy. CEM is highly flexible and can be applied in domains where gradients are unavailable or difficult to compute — including combinatorial optimization, control, and black-box search problems.

Monte Carlo Estimation: A Motivating Example

To motivate stochastic search, consider the problem of estimating the area of a unit circle — without using the formula $S = \pi r^2$.

Setup.

- Let the circle be centered at the origin with radius $r = 1$.
- Enclose it in a square of side length 2, spanning from $(-1, -1)$ to $(1, 1)$.
- The area of the square is known: 4.

Estimation Strategy. Randomly sample N points uniformly from the square. For each point (x_i, y_i), determine whether it lies inside the circle:

$$x_i^2 + y_i^2 \leq 1$$

Let M be the number of points that fall inside the circle. Then:

$$\text{Estimated Area of Circle} \approx \frac{M}{N} \times \text{Area of Square} = \frac{M}{N} \cdot 4$$

Zooming In: Small Circle and Adaptive Sampling

Now suppose we want to estimate the area of a much smaller circle — for example, one with radius $r = 0.1$, still centered at the origin.

Challenge. If we keep sampling uniformly from the large square $[-1, 1] \times [-1, 1]$, very few of the points will fall inside the small circle. This leads to high variance in the estimation and makes it inefficient.

Solution: Adaptive Sampling via CEM. We can use the Cross-Entropy Method to improve the sampling process:

- Start by sampling uniformly from the full square.
- After the first round, identify which samples fall inside the small circle — call these the **elite samples**.
- Update the sampling distribution (e.g., from a uniform distribution over the square to a denser distribution on a smaller square).
- In the next iteration, generate new samples using this updated distribution — now more focused around the region likely to intersect the circle.

Connection to CEM. While this method is not itself CEM, it shares CEM's core idea: **use random samples and evaluation to estimate or optimize a target quantity**. In CEM, we adapt the sampling distribution toward better regions; in Monte Carlo, we use sampling to estimate properties of a known distribution.

This motivates CEM as a general-purpose tool for solving problems where direct computation or gradient-based optimization is unavailable — a common theme in mathematical applications.

Formal Definition of the Cross-Entropy Method

The Cross-Entropy Method (CEM) is a general-purpose optimization algorithm that iteratively updates a parameterized probability distribution in order to concentrate sampling on high-performing regions of the search space.

Setup. Let $p(x; \theta)$ be a family of probability distributions over the solution space $\mathcal{X}$, parameterized by θ. We wish to find $x \in \mathcal{X}$ that maximizes a given objective function $f(x)$.

Algorithm. CEM proceeds by iteratively updating θ to better sample high-reward regions:

1. **Sampling:** Generate N samples $x_1, x_2, \ldots, x_N \sim p(x; \theta_t)$.
2. **Evaluation:** Compute $f(x_i)$ for each sample.
3. **Elite Selection:** Select the top $\rho \cdot N$ samples (e.g., $\rho = 0.1$) with the highest function values. Denote this elite set as:
$$S_t \subset \{x_1, \ldots, x_N\}, \quad |S_t| = \rho \cdot N$$
4. **Parameter Update:** Update the distribution parameters by maximizing the log-likelihood of the elite set under $p(x; \theta)$:
$$\theta_{t+1} = \arg\max_{\theta} \sum_{x \in S_t} \log p(x; \theta)$$

Intuition. This update step minimizes the Kullback–Leibler divergence between the new distribution and the empirical distribution of elite samples. Over iterations, the sampling distribution concentrates around regions of high function value, guiding the search toward optimal solutions.

CEM is especially useful when gradients are unavailable or the optimization landscape is noisy or combinatorial — common in mathematical and reinforcement learning problems.

Applying CEM to Reinforcement Learning

In reinforcement learning, we often aim to learn a good policy $\pi(s;\theta)$ — a parameterized function that selects actions based on the current state. The Cross-Entropy Method (CEM) provides a simple and effective way to optimize such policies via sampling.

Algorithm Outline. CEM for RL proceeds by iteratively generating full episodes, evaluating them based on total return, and updating the policy to favor high-reward behaviors.

1. **Sampling:** Generate N episodes by interacting with the environment using the current policy $\pi(s;\theta_t)$. Each episode consists of a sequence of states, actions, and rewards:
$$\tau_i = \left(s_0^i, a_0^i, r_0^i, \ldots, s_T^i\right), \quad i = 1, \ldots, N$$
2. **Evaluation:** Compute the **return** R_i of each episode — the cumulative reward:
$$R_i = \sum_{t=0}^{T} r_t^i$$
3. **Elite Selection:** Select the top $\rho \cdot N$ episodes with the highest returns. Denote this elite set as:
$$S_t \subset \{\tau_1, \ldots, \tau_N\}, \quad |S_t| = \rho \cdot N$$
4. **Policy Update:** Update the policy parameters θ_{t+1} by fitting the new policy to favor the actions taken in the elite episodes:
$$\theta_{t+1} = \arg\max_{\theta} \sum_{\tau \in S_t} \log \mathbb{P}(\tau \mid \pi(\cdot;\theta))$$
That is, maximize the log-likelihood of the elite episodes under the updated policy.

Example: Optimizing Spectral Properties of Graphs

As a concrete example of applying reinforcement learning and the Cross-Entropy Method (CEM) to mathematical discovery, consider the following unusual but well-defined problem in spectral graph theory:

Goal. What is the largest possible ratio between the smallest and largest eigenvalues of a graph's adjacency matrix?

This is a challenging **combinatorial optimization problem** with no obvious closed-form solution. We aim to discover graph structures that maximize this spectral ratio — a task well-suited to learning-based search.

Score Function. Let Adj be the adjacency matrix of a graph. Define the objective function:
$$L(\text{Adj}) = \frac{\lambda_{\min}}{\lambda_{\max}}$$
where:

- $\lambda_{\min}$: smallest eigenvalue of the adjacency matrix (possibly negative),
- $\lambda_{\max}$: largest eigenvalue (positive, due to Perron–Frobenius in connected graphs).

Learning Setup with CEM. We use the Cross-Entropy Method to explore the space of adjacency matrices and discover those with high scores:

1. **Generation Process:** We construct the adjacency matrix **element by element**, treating it as a sequence-generation task. At each step, the agent observes the current partial matrix and predicts the next entry (e.g., whether to place a 0 or 1).
2. **Evaluation:** Once a full adjacency matrix is generated, we compute its score $L(\text{Adj})$. If the matrix is not a valid adjacency matrix (e.g., not symmetric), we assign it a low or zero score.

3. **Elite Selection:** We select the top $\rho \cdot N$ matrices with the highest spectral ratios.
4. **Policy Update:** The generation policy is updated to increase the likelihood of reproducing decisions (i.e., sequences of binary entries) that led to these high-scoring graphs.

Interpretation. This approach treats graph generation as a sequential decision process and learns from examples — not by backpropagating through eigenvalues, but by reinforcing decisions that correlate with good outcomes.

Note. The resulting graphs often exhibit surprising structure, and the learning process offers insight into spectral optimization — a topic with deep mathematical roots.

We will explore the results of this method during the practical session!

11 Lecture 11: REINFORCE and A2C

Transition: From CEM to Policy Gradient Methods

In the previous lecture, we explored the Cross-Entropy Method (CEM) — a simple yet powerful black-box optimization algorithm for learning policies. CEM operates by sampling entire episodes, selecting the best, and updating the policy based on elite behavior. While intuitive and useful for many problems, it does not leverage the structure of trajectories or gradients of expected return.

In this lecture, we shift to a more principled and fine-grained class of methods: **policy gradient algorithms**. These methods directly compute an estimate of the gradient of the expected return with respect to policy parameters, and use it to improve the policy step-by-step. Two central algorithms in this category are **REINFORCE** and **A2C (Advantage Actor-Critic)**.

On-Policy vs. Off-Policy Learning

Before diving into specific algorithms, it's important to understand a fundamental distinction in reinforcement learning: the difference between **on-policy** and **off-policy** methods.

On-Policy Algorithms. In on-policy learning, the agent improves its policy using data collected **while following that same policy**. That is, the actions taken to generate experience are sampled from the current version of the policy π_θ, and learning is based on this same distribution. This creates a tight feedback loop between behavior and learning, but can also limit sample efficiency, since data must be constantly regenerated as the policy changes.

Examples:

- REINFORCE (Monte Carlo policy gradient)
- A2C and A3C (actor-critic variants)
- PPO (Proximal Policy Optimization)

Off-Policy Algorithms. Off-policy algorithms, in contrast, allow the agent to learn from trajectories generated by a *different* policy — often called the behavior policy. This enables the reuse of past experiences and supports learning from demonstrations or replay buffers. Off-policy methods are generally more sample-efficient but require careful handling of distributional differences (e.g., via importance sampling).

Examples:

- Q-learning and Deep Q-Networks (DQN)
- DDPG, TD3 (deterministic actor-critic methods)
- SAC (Soft Actor-Critic)

In This Lecture. We will focus on **on-policy** methods, which are conceptually simpler and closely aligned with the policy gradient framework. Specifically, we will:

- Introduce the REINFORCE algorithm — the most basic Monte Carlo policy gradient method.
- Discuss its limitations, such as high variance.
- Motivate and derive A2C, which reduces variance using a learned baseline (value function).

What Is the Best Policy?

The central goal in reinforcement learning is to find a policy $\pi^\star$ that maximizes expected cumulative reward. Formally, we define the optimal policy as:

$$\pi^\star = \arg\max_\pi \ \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} R(s_t, a_t)\right], \quad \text{where } a_t = \pi(s_t)$$

Here:

- $\tau = (s_0, a_0, s_1, a_1, \dots)$ is a trajectory generated by following the policy π,
- $R(s_t, a_t)$ is the expected immediate reward received at time t,
- The expectation is over the distribution of trajectories induced by π and the environment.

Problems with This Objective. Although this expression captures the objective clearly, it suffers from several theoretical and practical issues:

- **Unbounded Returns:** The infinite sum $\sum_{t=0}^{\infty} R(s_t, a_t)$ can diverge — particularly if rewards are positive and the agent acts forever. This makes the objective ill-defined in many environments.
- **Incentivizing Stalling:** If the agent receives positive rewards just for staying alive, it might learn to delay reaching the goal indefinitely.
- **Non-Terminating Episodes:** Environments without time limits may lead to infinite-horizon returns, complicating learning and evaluation.

Solutions: Regularization and Discounting. To address these problems, RL algorithms often modify the return function in one of the following ways:

- **Discounted Return:** Introduce a discount factor $\gamma \in (0, 1)$ to penalize future rewards:

  $$\pi^\star = \arg\max_\pi \ \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right]$$

  This guarantees convergence of the sum and encourages short-term reward accumulation.
- **Per-Step Penalty:** Assign a small negative reward at each time step to penalize delay or wandering and promote quicker task completion.
- **Time Limit Truncation:** Implicitly regularize by terminating episodes after a fixed number of steps, enforcing finite returns.

Objective Function for Learning. In practice, we define a parameterized policy π_θ and aim to maximize the expected discounted return:

$$\mathcal{J}(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right]$$

This function $\mathcal{J}(\pi_\theta)$ becomes the target for gradient-based optimization methods.

Policy Gradient Theorem

In order to optimize the expected return $\mathcal{J}(\pi_\theta)$, we apply gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta \mathcal{J}(\pi_{\theta_k})$$

To apply gradient ascent, we must compute the gradient of the expected return with respect to the policy parameters:

$$\nabla_\theta \mathcal{J}(\pi_\theta)$$

We begin with the definition of the expected return under policy π_θ:

$$\mathcal{J}(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \right]$$

Here, $\tau = (s_0, a_0, s_1, a_1, \dots)$ denotes a trajectory generated by interacting with the environment under policy π_θ. For notational convenience, we define the *return of a trajectory* τ as:

$$R(\tau) := \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)$$

Thus, the objective can be rewritten more compactly as:

$$\mathcal{J}(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$$

This formulation sets the stage for applying the log-derivative trick to compute the policy gradient.

This expectation can be written as an integral:

$$\mathcal{J}(\pi_\theta) = \int R(\tau)\, p_\theta(\tau)\, d\tau$$

Trajectory Probability. The probability of a full trajectory $\tau = (s_0, a_0, s_1, a_1, \dots)$ under policy π_θ is:

$$p_\theta(\tau) = p(s_0) \prod_{t=0}^{\infty} \pi_\theta(a_t \mid s_t) \cdot p(s_{t+1} \mid s_t, a_t)$$

The trajectory probability $p_\theta(\tau)$ includes both policy-dependent components and environment dynamics. However, computing $\nabla_\theta p_\theta(\tau)$ directly is infeasible for two reasons: first, the transition probabilities $p(s_{t+1} \mid s_t, a_t)$ are typically unknown; second, even if all terms were known, differentiating a long product of terms (as appears in $p_\theta(\tau)$) is computationally unstable and inefficient due to the multiplicative chain rule over many steps.

Log-Derivative Trick. We apply the log-derivative identity:

$$\nabla_\theta \mathcal{J}(\pi_\theta) = \int R(\tau) \nabla_\theta p_\theta(\tau)\, d\tau = \int R(\tau) \nabla_\theta \log p_\theta(\tau) \cdot p_\theta(\tau)\, d\tau = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \nabla_\theta \log p_\theta(\tau) \right]$$

Next, observe that $p_\theta(\tau)$ contains only the policy terms that depend on θ:

$$\log p_\theta(\tau) = \log p(s_0) + \sum_{t=0}^{\infty} \log \pi_\theta(a_t \mid s_t) + \log p(s_{t+1} \mid s_t, a_t)$$

Hence,

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

Substituting back, we obtain:

$$\nabla_\theta \mathcal{J}(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right]$$

Time-Localized Return. To make the policy gradient more interpretable and localized in time, we decompose the total return $R(\tau)$ into per-step contributions.

Recall that:

$$R(\tau) = \sum_{t'=0}^{\infty} \gamma^{t'} R(s_{t'}, a_{t'})$$

We define the **return-to-go** from time t onward as:

$$Q(s_t, a_t) := \sum_{t' \geq t} \gamma^{t'-t} R(s_{t'}, a_{t'})$$

Then the gradient becomes:

$$\nabla_\theta \mathcal{J}(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot Q(s_t, a_t) \right]$$

This step crucially relies on the **Markov property** of the environment: the future rewards $R(s_{t'}, a_{t'})$ for $t' \geq t$ depend only on the current state-action pair (s_t, a_t) and not on the earlier parts of the trajectory. As a result, the gradient with respect to θ can be correctly factored through time, and each term only needs to consider its corresponding return-to-go $Q(s_t, a_t)$.

This gives a time-decomposed version of the policy gradient that forms the foundation for practical policy gradient algorithms. This is the **Policy Gradient Theorem**.

Interpretation. It expresses the gradient of the expected return in terms of:

- The score function $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$,
- Weighted by the expected return-to-go $Q(s_t, a_t)$.

Stationary Distribution Form. Under certain assumptions (e.g., ergodicity), we can express the expectation over state-action pairs sampled from the stationary distribution of the policy:

$$\nabla_\theta \mathcal{J}(\pi_\theta) = \mathbb{E}_{(s,a) \sim d^{\pi_\theta}(s) \pi_\theta(a|s)} \left[\nabla_\theta \log \pi_\theta(a \mid s) \cdot Q(s, a) \right]$$

This expression forms the foundation of algorithms like REINFORCE and Advantage Actor-Critic (A2C), which we will now explore.

Time-Localized Return. To make the policy gradient more interpretable and localized in time, we decompose the total return $R(\tau)$ into per-step contributions.

Recall that:

$$R(\tau) = \sum_{t'=0}^{\infty} \gamma^{t'} R(s_{t'}, a_{t'})$$

We define the **return-to-go** from time t onward as:

$$Q(s_t, a_t) := \sum_{t' \geq t} \gamma^{t'-t} R(s_{t'}, a_{t'})$$

Then the gradient becomes:

$$\nabla_\theta \mathcal{J}(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot Q(s_t, a_t) \right]$$

This transformation relies on the **Markov property** of the environment: future rewards $R(s_{t'}, a_{t'})$ for $t' \geq t$ depend only on the current state-action pair (s_t, a_t) and not on the past trajectory. Therefore, each term in the sum can be localized using the return-to-go from that point onward.

Where Does the Distribution Come From? The expectation above is taken over trajectories $\tau \sim \pi_\theta$, which are generated by:

- Sampling an initial state $s_0 \sim p(s_0)$,
- Repeatedly selecting actions from the policy: $a_t \sim \pi_\theta(a_t \mid s_t)$,
- And transitioning according to the environment's dynamics: $s_{t+1} \sim p(s_{t+1} \mid s_t, a_t)$.

This defines a probability distribution over trajectories $p_\theta(\tau)$, and in turn induces a distribution over state-action pairs (s_t, a_t) visited by the agent. It is this distribution — defined implicitly by the policy and environment — that underlies the expectation in the policy gradient.

Understanding this is critical: even though we cannot differentiate through the environment's dynamics $p(s_{t+1} \mid s_t, a_t)$, we can still estimate the gradient with respect to the policy parameters by leveraging the structure of this trajectory distribution and applying the log-derivative trick.

The REINFORCE Algorithm

The **REINFORCE** algorithm (Williams, 1992) is a classic method for estimating policy gradients using Monte Carlo rollouts. It provides a direct implementation of the policy gradient theorem in its most basic form.

Policy Gradient Expression. Recall the core result from the policy gradient theorem:

$$\nabla_\theta \mathcal{J}(\pi_\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a \mid s) \cdot Q(s, a) \right]$$

This expression tells us how to adjust the parameters θ to increase the expected return: reinforce actions that led to high return $Q(s, a)$, and reduce the probability of actions that led to poor outcomes.

Monte Carlo Estimation. In practice, REINFORCE approximates the expectation above using full trajectories sampled from the current policy π_θ. For each timestep t in a sampled trajectory τ, we compute:

$$\nabla_\theta \mathcal{J}(\pi_\theta) \approx \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot \hat{Q}(s_t, a_t)$$

Here, $\hat{Q}(s_t, a_t)$ is the empirical return-to-go from timestep t, obtained from the sampled trajectory:

$$\hat{Q}(s_t, a_t) = \sum_{t'=t}^{T} \gamma^{t'-t} R(s_{t'}, a_{t'})$$

Thus, to compute $Q(s_t, a_t)$, we must observe the entire remainder of the episode — requiring full trajectories.

Variance of the Estimate. A known drawback of REINFORCE is its **high variance**. Since the return $\hat{Q}(s_t, a_t)$ can vary significantly across episodes (especially in stochastic environments or with sparse rewards), the gradient estimates may fluctuate wildly from one batch to the next.

This instability motivates later improvements, such as:

- Using **baselines** to reduce variance (e.g., subtracting a learned value function),
- Employing actor-critic methods like **A2C**, which we will cover next.

Summary. REINFORCE provides a simple and unbiased estimator for the policy gradient:

- It requires no model of the environment.
- It relies solely on trajectory samples.
- But it suffers from high variance due to full-episode return estimates.

Variance Reduction Using Baselines

A major issue with REINFORCE is the high variance in the gradient estimates. Even though the estimator is unbiased, the variability across episodes can make learning slow or unstable. To address this, we can introduce a **baseline** — a common technique in Monte Carlo estimation.

Baseline-Augmented Gradient. We modify the gradient formula by subtracting a baseline $b(s)$ that depends only on the state:

$$\nabla_\theta \mathcal{J}(\pi_\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a \mid s) \cdot (Q(s,a) - b(s))\right]$$

Choosing the Baseline. A natural and effective choice is to set the baseline to the value function:

$$b(s) = V^{\pi_\theta}(s) := \mathbb{E}_{a \sim \pi_\theta} \left[Q(s,a)\right]$$

This choice minimizes the variance of the estimator among all possible functions of the state. The modified gradient becomes:

$$\nabla_\theta \mathcal{J}(\pi_\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a \mid s) \cdot A(s,a)\right]$$

where we define the **advantage function**:

$$A(s,a) := Q(s,a) - V^{\pi_\theta}(s)$$

Interpretation. The advantage $A(s,a)$ tells us how much better (or worse) the chosen action a is compared to the average behavior of the policy π_θ in state s. If $A(s,a) > 0$, the action is better than expected, and the gradient increases its probability. If $A(s,a) < 0$, the action is discouraged.

This interpretation enables a more stable learning process by focusing updates on deviations from average performance rather than total returns alone.

Next, we will see how this idea forms the foundation of actor-critic methods, particularly Advantage Actor-Critic (A2C).

Advantage Actor-Critic (A2C)

The **Advantage Actor-Critic (A2C)** algorithm is a widely-used on-policy reinforcement learning method that addresses the high variance of REINFORCE by combining ideas from policy gradients and value-based learning.

It introduces two key components:

- The **actor** — the policy $\pi_\theta(a \mid s)$, parameterized by θ,
- The **critic** — a learned estimate $V_\phi(s)$ of the value function, parameterized by ϕ.

Learning Setup. At each training step, the agent:

1. **Collects Trajectories.** The agent interacts with the environment using its current policy π_θ, and stores tuples of the form:

   $$(s_t, a_t, r_t, s_{t+1})$$

   This is done over multiple time steps and often across several parallel environments.

2. **Estimates the Advantage.** The critic is used to approximate the value function $V_\phi(s) \approx V^{\pi_\theta}(s)$. The advantage is estimated using a **1-step temporal difference (TD)** approximation:
$$A_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$
3. **Updates the Actor.** The policy parameters are updated by ascending the gradient:
$$\theta \leftarrow \theta + \alpha \cdot \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot A_t$$
4. **Updates the Critic.** The value function parameters ϕ are updated to minimize the squared TD error:
$$L_{\text{critic}} = \big(r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)\big)^2$$

Critic Update in Detail. The goal of the critic is to learn the value function $V^{\pi_\theta}(s)$, which estimates the expected return starting from state s under policy π_θ:

$$V^{\pi_\theta}(s) = \mathbb{E}_{\pi_\theta}\left[\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} \;\middle|\; s_t = s\right]$$

Since we do not know the true expectation, we instead construct a **bootstrapped target** using a single observed reward and the estimated value of the next state:

$$\text{Target}_t = r_t + \gamma V_\phi(s_{t+1})$$

This is a one-step temporal difference (TD) target, which provides a noisy but informative signal for how good the current state's value estimate is.

We then define the **critic loss function** as the squared error between the TD target and the current prediction:

$$L_{\text{critic}}(\phi) = \big(r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)\big)^2$$

Minimizing this loss pushes the predicted value $V_\phi(s_t)$ closer to a self-consistent estimate: the immediate reward plus the discounted prediction of the next state.

Gradient-Based Optimization. We optimize ϕ using stochastic gradient descent or Adam on the loss:

$$\nabla_\phi L_{\text{critic}} = \nabla_\phi \big(r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)\big)^2$$

Only $V_\phi(s_t)$ and $V_\phi(s_{t+1})$ depend on ϕ, so the gradient propagates through these terms. In practice, this is implemented as backpropagation through the critic network, often using mini-batches of transitions.

Why It Works. A2C benefits from lower variance in gradient estimates compared to REINFORCE, because:

- It uses **bootstrapping** via the learned value function instead of relying on full returns,
- It incorporates the **advantage function** to make the updates more informed and stable.

This algorithm is also efficient in practice, as it can be trained using batches of experiences gathered in parallel environments. In the next sections, we will examine further improvements to actor-critic methods, such as Generalized Advantage Estimation (GAE) and A3C.

12 PPO and Andrews-Curtis

Reducing Variance: Generalized Advantage Estimation (GAE)

While the 1-step TD advantage estimate used in A2C is effective, it still suffers from variance due to stochastic rewards and transitions. To obtain more stable and flexible advantage estimates, we can use the **Generalized Advantage Estimation (GAE)** technique, proposed by Schulman et al. (2015).

TD Residual. At the core of GAE is the temporal-difference (TD) residual:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

This measures the difference between the predicted value of the current state and a bootstrapped estimate from the next state.

The GAE Formula. Instead of relying on a single-step or full-return estimate, GAE defines a weighted sum of TD residuals:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \delta_{t+l}$$

Here, $\lambda \in [0, 1]$ is a smoothing parameter that controls the bias-variance tradeoff:

- When $\lambda = 1$, GAE resembles the Monte Carlo-style advantage used in REINFORCE — low bias, high variance.
- When $\lambda = 0$, GAE reduces to a single-step TD estimate — higher bias, lower variance.

Practical Implementation. In practice, the sum is truncated to a finite horizon (e.g., the length of the episode or rollout window), and the advantages $\hat{A}_t$ are computed efficiently in reverse time using dynamic programming.

Summary. GAE provides a flexible and powerful method to estimate the advantage function, allowing smooth interpolation between high-variance, low-bias Monte Carlo estimates and low-variance, high-bias TD methods. It is a key component in modern policy optimization algorithms like PPO.

Off-Policy Learning and Importance Sampling

In reinforcement learning, we often wish to improve a policy π_θ, but the data available comes from an earlier version of the policy, $\pi_{\theta_{\text{old}}}$. This is the setting of **off-policy learning**, and a key idea that enables learning in this setting is **importance sampling**.

Goal. We want to compute expectations under the new policy π_θ, such as:

$$\mathbb{E}_{a\sim\pi_\theta(a|s)}\left[f(a)\right] = \sum_a \pi_\theta(a \mid s) f(a)$$

However, we only have samples from $\pi_{\theta_{\text{old}}}$. Using importance sampling, we can reweight these samples:

$$\sum_a \pi_\theta(a \mid s) f(a) = \sum_a \pi_{\theta_{\text{old}}}(a \mid s) \cdot \frac{\pi_\theta(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)} f(a) = \mathbb{E}_{a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)} f(a)\right]$$

Importance Weight. We define the importance sampling ratio:

$$r(\theta) := \frac{\pi_\theta(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)}$$

This term rescales the contribution of each sample depending on how likely the new policy π_θ is to select the same action in the same state, relative to the old policy.

Practical Implications. This technique allows us to **reuse trajectories** collected under older policies, reducing the need to interact with the environment frequently — a key benefit in computationally expensive or real-world settings.

However, this "cheating" only works reliably if $r(\theta) \approx 1$. That is, the new policy must be close to the one that generated the data. If $r(\theta)$ deviates too far from 1, the variance of the estimate explodes, and learning becomes unstable.

Next, we will see how PPO incorporates this idea and simultaneously constrains policy updates to prevent $r(\theta)$ from growing too far from 1.

Proximal Policy Optimization (PPO)

PPO is a widely used on-policy algorithm that improves stability and performance of policy gradient methods. It builds on the idea of using importance sampling to reuse experience, but avoids large, destabilizing updates by **clipping the objective function**.

Clipped Surrogate Objective. The key innovation in PPO is the use of a clipped loss to prevent large policy updates:

$$L_t^{\text{clip}}(\theta) = \min\Big(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon)\hat{A}_t\Big)$$

where

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \quad \text{is the importance sampling ratio,}$$

and $\hat{A}_t$ is the advantage estimate, typically computed using **Generalized Advantage Estimation (GAE)**:

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \quad \text{with} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Clipping the importance ratio ensures that policy updates remain close to the behavior policy, reducing the chance of overfitting or instability.

Value Function Loss with Frozen Targets. In PPO, the value network $V_\phi(s_t)$ is trained to predict the expected return from state s_t. The target for this prediction is computed as:

$$R_t = \hat{A}_t + V^{\text{old}}(s_t)$$

Here, V^{old} is the value function **frozen at the time trajectories are sampled**. This means it is evaluated once at the start of a training epoch and kept fixed during all optimization steps that follow.

$$L_t^V = \big(V_\phi(s_t) - R_t\big)^2$$

This frozen target prevents the learning target from shifting during backpropagation and ensures a more stable training signal. Multiple gradient steps (over shuffled mini-batches) are taken using this fixed target before new trajectories are collected and the value function is updated accordingly.

This batching strategy enables PPO to perform multiple updates from a single batch of experience, improving data efficiency without sacrificing stability.

Entropy Bonus. To promote exploration, PPO adds an entropy regularization term:

$$\mathcal{S}[\pi_\theta](s_t) = -\sum_a \pi_\theta(a \mid s_t) \log \pi_\theta(a \mid s_t)$$

A low entropy indicates that the policy is nearly deterministic, while high entropy encourages diversity in action selection.

Total Loss. The complete loss function for PPO combines these three terms:

$$L = -L^{\text{clip}} + c_1 L^V - c_2 \mathcal{S}[\pi_\theta]$$

where:

- L^{clip} is the clipped policy loss (actor update),
- L^V is the value function loss (critic update),
- $\mathcal{S}[\pi_\theta]$ is the policy entropy (exploration bonus),
- c_1, c_2 are weighting coefficients.

This carefully balanced loss allows PPO to perform reliable updates, avoid catastrophic forgetting, and maintain exploration throughout training.

Proximal Policy Optimization (PPO): Full Algorithm

PPO alternates between collecting experience with the current policy and performing multiple gradient updates based on that experience. Here's a complete step-by-step breakdown:

1. Collect Trajectories. Execute the current policy π_θ in the environment to gather rollouts consisting of tuples:

$$(s_t, a_t, r_t, s_{t+1}, \pi_\theta(a_t \mid s_t), V_\phi(s_t))$$

This data is collected across multiple trajectories and parallel environments. Importantly, both the policy π_θ and value function V_ϕ are frozen at this stage — we denote them as $\pi_{\theta_{\text{old}}}$ and $V_{\phi_{\text{old}}}$ and reuse them during the optimization phase.

2. Estimate Advantages. Use Generalized Advantage Estimation (GAE) to compute the advantage values:

$$\delta_t = r_t + \gamma V_{\phi_{\text{old}}}(s_{t+1}) - V_{\phi_{\text{old}}}(s_t)$$

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

This provides a smoother, lower-variance estimate of each action's relative value.

3. Compute Loss Terms. Using the frozen policy and value function:

- **Policy Loss (Clipped Surrogate Objective):**

$$L_t^{\text{clip}}(\theta) = \min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right) \quad \text{where} \quad r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

- **Value Function Loss:**

$$L_t^V = \left(V_\phi(s_t) - (\hat{A}_t + V_{\phi_{\text{old}}}(s_t))\right)^2$$

- **Entropy Bonus (for exploration):**

$$\mathcal{S}[\pi_\theta](s_t) = -\sum_a \pi_\theta(a \mid s_t) \log \pi_\theta(a \mid s_t)$$

4. Update Parameters. The total loss is:

$$L = -L^{\text{clip}} + c_1 L^V - c_2 \mathcal{S}[\pi_\theta]$$

Using this loss, update the actor (policy) and critic (value) networks with backpropagation via gradient descent.

5. Repeat Optimization. For each batch of collected trajectories, perform multiple epochs of gradient updates using shuffled mini-batches. The frozen policy $\pi_{\theta_{\text{old}}}$ and value function $V_{\phi_{\text{old}}}$ are used throughout this phase to compute importance weights and targets. Only after completing all gradient steps do we return to Step 1 and collect new rollouts with the updated policy.

This separation between data collection and optimization — along with clipping and entropy regularization — enables PPO to achieve both stability and efficiency in policy learning.

Reinforcement Learning Meets Group Theory: The Andrews–Curtis Conjecture

We now turn to a recent and insightful paper:

> **Gukov et al. (2025)**, *"What Makes Math Problems Hard for Reinforcement Learning: A Case Study"*.

This work explores how reinforcement learning methods can be applied to deep problems in mathematics, using the **Andrews–Curtis conjecture** as a case study. The paper contributes both conceptual insights into what makes mathematical tasks challenging for RL, and presents concrete results using modern RL techniques.

In the following sections, we will introduce the conjecture, define all necessary terms, and then explain how RL is employed to approach this mathematical question.

The Andrews–Curtis Conjecture

Conjecture (J. Andrews and M. Curtis, 1965). Every **balanced presentation** of the trivial group can be transformed into the standard trivial presentation using a sequence of specific elementary moves (called *Andrews–Curtis moves*).

Group Presentations. A **group presentation** is a way to describe a group in terms of generators and relations. It is written in the form:

$$G = \langle x_1, x_2, \ldots, x_n \mid r_1, r_2, \ldots, r_m \rangle$$

where:

- $x_1, \ldots, x_n$ are formal generators,
- $r_1, \ldots, r_m$ are relators (words in the generators and their inverses),

and the group G is the quotient of the free group $F(x_1, \ldots, x_n)$ by the normal closure of the set of relators:

$$G \cong F(x_1, \ldots, x_n)/\langle\langle r_1, \ldots, r_m \rangle\rangle.$$

This defines a group whose elements are equivalence classes of words in the generators, modulo the given relations.

Balanced Presentations. A presentation of a group is said to be *balanced* if the number of generators equals the number of relators:

$$\langle x_1, \ldots, x_n \mid r_1, \ldots, r_n \rangle$$

where each r_i is a word over the generators and their inverses, and the group defined is the trivial group:

$$G \cong \{e\}$$

The conjecture asserts that any such balanced presentation of the trivial group can be transformed into the **trivial presentation**:

$$\langle x_1, \ldots, x_n \mid x_1, \ldots, x_n \rangle$$

by a finite sequence of the following moves, known as **Andrews–Curtis moves**.

Andrews–Curtis Moves. Let r_i and r_j denote relators, and x_k be a generator. The allowed transformations are:

- **(AC1) Relator multiplication:**
$$r_i \rightsquigarrow r_i r_j \quad (i \neq j)$$
- **(AC2) Inversion:**
$$r_i \rightsquigarrow r_i^{-1}$$
- **(AC3) Conjugation:**
$$r_i \rightsquigarrow x_k^{\pm 1} r_i x_k^{\mp 1}$$

These moves are applied only to the relators, while the generators remain unchanged.

Example.
$$\langle x_1, x_2 \mid x_1 x_2,\ x_2 \rangle \quad \rightsquigarrow \quad \langle x_1, x_2 \mid x_1,\ x_2 \rangle$$
This reduction can be achieved via three AC moves.

Simple Invariant: Total Relator Length. An easily computed (though not complete) invariant of a group presentation is the *total relator length*, i.e., the sum of the lengths (number of letters) in all relators:
$$\text{Length}(\mathcal{P}) = \sum_{i=1}^{n} |r_i|$$
This gives a simple heuristic for judging how "far" a presentation is from being trivial.

Mathematical Results: New Reductions in Known Families

The paper provides rigorous group-theoretic evidence supporting the Andrews–Curtis conjecture by exhibiting concrete reductions of certain well-known families of presentations that were historically viewed as potential counterexamples.

Result #1: The Akbulut–Kirby Family. Define the presentation:
$$\mathsf{AK}(n) = \langle x, y \mid x^n = y^{n+1},\ xyx = yxy \rangle$$
Then for every $n \geq 2$, the authors prove that $\mathsf{AK}(n)$ is Andrews–Curtis equivalent to the presentation:
$$\langle x, y \mid x^{-1} yx = xyx^{-1} y,\ xy^{n-1} x = yxy \rangle$$
This alternate presentation has total relator length $n + 11$, which constitutes a strict reduction in total length for all $n \geq 5$. This suggests meaningful structural simplification is possible across this entire family.

Result #2: The Miller–Schupp Family. Define the family of presentations:
$$\mathsf{MS}(n, w) = \langle x, y \mid x^{-1} y^n x = y^{n+1},\ x = w \rangle$$
The authors establish the following:

(i) $\mathsf{MS}(1, w)$ is AC-trivializable for all w,

(ii) $\mathsf{MS}(n, w^\star)$ is AC-trivializable for all $n > 0$, where $w^\star = y^{-1} xyx^{-1}$,

(iii) $\mathsf{MS}(2, w_k)$ is AC-trivializable for all $k \in \mathbb{Z}$, where $w_k = y^{-k} x^{-1} yxy$.

Moreover, for each fixed $n > 0$, all members of the 1-parameter family $\mathsf{MS}(n, w_k)$, parameterized by $k \in \mathbb{Z}$, are AC-equivalent to each other and to $\mathsf{AK}(n)$. This unifies these families under the same equivalence class and offers further support to the validity of the conjecture.

Limitations of Direct RL: A Complexity-Theoretic Perspective

Applying reinforcement learning to general presentations remains extremely challenging.

An Instructive Example (Bridson, 2023). In his paper *"The Complexity of Balanced Presentations and the Andrews–Curtis Conjecture"*, Martin R. Bridson exhibits a balanced presentation of the trivial group that requires more than:

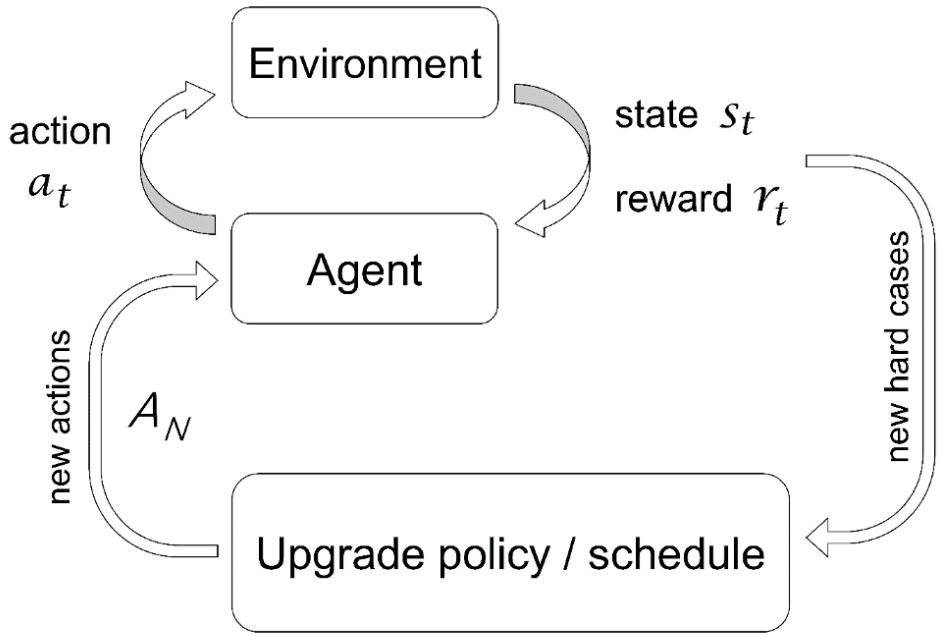
$$10^{10000}$$

Andrews–Curtis moves to trivialize. The presentation involves four generators and nested commutators with high powers. Even though it is AC-trivial, its reduction path is far beyond the reach of brute-force enumeration or shallow exploration strategies.

Implication. This example highlights a core challenge: the space of AC-equivalent presentations can exhibit **astronomically long trivialization sequences**, even for short relators and a small number of generators. This poses a fundamental barrier to naïve applications of reinforcement learning.

Adaptive Action Spaces and the Role of Hard Instances

Motivation. Trivializing complex group presentations like those arising in the Andrews–Curtis conjecture often involves extremely long sequences of moves. Standard reinforcement learning frameworks with fixed primitive actions (e.g., individual AC-moves) are insufficient for navigating such deep search trees efficiently. To address this, the authors propose a dynamic, self-improving system that augments the action space by introducing higher-level actions called **supermoves**.



Supermoves and Learning to Modify the Action Space. A **supermove** is a learned composition of several elementary AC-moves that appears frequently in successful long trivialization sequences. Instead of defining them manually, the algorithm learns both:

- Which sequences should become supermoves,
- When to insert or remove them from the agent's action space.

This is done adaptively, based on the notion of instance **hardness** — how difficult a presentation is to solve using the current policy.

Algorithm 3 Adaptive AI Model Training and Path Discovery

Input:
Family of AI models $\pi(N)$ with common state space S and action space A_0
Initial setting N_0 and ordered range $\{N_1, N_2, \ldots, N_{\max}\}$
Number of epochs for training
Validation set $V \subset S$
Distinguished state $s_0 \in S$
Positive integer n
Output:
For each setting N_i: Set of pairs $\{v, P\}$ where $v \in V$ and P connects v to s_0
Initialize $A(N_1) \leftarrow A_0$
for each N_i in $\{N_1, N_2, \ldots, N_{\max}\}$ **do**
    Train model $\pi(N_i)$ on S for the given number of epochs
    Evaluate $\pi(N_i)$ on V to discover paths connecting V to s_0 using $A(N_i)$
    $V(N_i) \leftarrow \{v \in V \mid v$ can be connected to s_0 using $A(N_i)$, but not by any $\pi(N_j)$ with $j < i\}$
    $W(N_i) \leftarrow \{v \in V(N_i) \mid$ the longest path connecting v to s_0 using $A_0\}$
    if $i \geq n$ **then**
        Compare $W(N_{i-n+1})$ to $W(N_i)$
        Adjust $A(N_{i+1})$ based on the comparison
    else
        $A(N_{i+1}) \leftarrow A(N_i)$
    end if
end for

Adaptive Training Framework. The authors describe a meta-RL framework (see Algorithm 3) that works in rounds indexed by a resource level N, such as the number of training environment interactions. For each N, the system:

1. Trains an agent $\pi(N)$ on the current action space $\mathcal{A}_N$,
2. Identifies **new hard instances** — presentations that are solvable at round N but were not solvable at earlier levels $N' < N$,
3. Examines the **longest solution paths** (AC-trivializations) found for those instances,
4. Selects common or useful sub-sequences of these long paths to promote as supermoves,
5. Updates $\mathcal{A}_{N+1}$ by augmenting or pruning based on performance.

Why Path Length Matters. Longer solution paths typically correspond to harder instances and more meaningful discoveries. By focusing on long successful paths, the system captures structural insights about the problem and generalizes better. This also allows it to compress longer trajectories into shorter effective decision sequences — reducing horizon length and improving credit assignment.

Conclusion. This adaptive strategy, grounded in learning from hardness and reconfiguring the action space over time, offers a promising blueprint for tackling other deep combinatorial problems with reinforcement learning. It combines the strengths of curriculum learning, meta-RL, and representation learning to construct agents that can *learn how to learn*.

Lecture 13: Model-Based Reinforcement Learning

In **model-based reinforcement learning**, the agent explicitly learns or has access to a model of the environment's dynamics. This includes the transition probabilities $P(s' \mid s, a)$ and optionally the reward function $R(s, a)$. The agent uses this model to plan ahead and evaluate potential future trajectories before taking actions.

Key features:

- Explicit reasoning about future states and rewards.
- Requires either a known model or the ability to learn a model from data.
- Can be sample-efficient by reusing the model for planning.

Model-based RL contrasts with **model-free** methods, where the agent directly learns the policy or value function from interactions with the environment without constructing a model.

Monte Carlo Tree Search (MCTS)

MCTS is a powerful planning algorithm used in model-based RL. It incrementally builds a search tree by simulating many possible future sequences of actions and outcomes. It balances **exploration** of new actions with **exploitation** of known rewarding ones.

Key Idea: Instead of evaluating all possible action sequences, MCTS performs **selective exploration** based on statistical estimates from simulated rollouts.

Core Steps of MCTS:

1. **Selection:** Traverse the tree from the root using a selection strategy (e.g., UCB1) to choose child nodes.
2. **Expansion:** Add a new node to the tree if the selected node is not fully expanded.
3. **Simulation:** Perform a random or heuristic rollout from the new node to estimate the outcome.
4. **Backpropagation:** Propagate the simulation result back through the visited nodes, updating statistics.

Application: MCTS is widely used in domains where a model is available, notably in game-playing agents like AlphaGo and MuZero.

MCTS: Four Steps

1. Selection

We use UCB1 to choose the most promising child:

$$\text{score}(s,a) = \frac{Q(s,a)}{N(s,a)} + c \cdot \sqrt{\frac{\ln N(s)}{N(s,a)}}$$

- $Q(s,a)$: cumulative reward from action a in state s
- $N(s,a)$: number of times action a was selected from s
- $N(s)$: total visits to state s
- c: exploration constant (tunable)

2. Expansion

- If all children of a node have been visited, MCTS selects one of them using UCB1.
- If there exists an unvisited action a from a state s, apply it:
  - Simulate the environment to obtain s'
  - Add s' as a new child node

3. Simulation

- From the newly added node s', simulate a full trajectory.
- Use a default policy:
  - A **random policy** (uniform legal action)
  - Or a **simple heuristic policy** (e.g., favoring central moves)
- This policy should be fast and lightweight

4. Backpropagation

- After simulation, obtain a reward R
- Traverse back along the path, and for each (s,a):
  - Update reward: $Q(s,a)$ += R
  - Update visit count: $N(s,a)$ += 1

Final Action Selection

After running MCTS for N simulations, we obtain a partial search tree rooted at the initial state s_0. We must now choose an action a^* to take in the real environment.

There are two common strategies for selecting the final action:

1. **Highest visit count (most common in practice):**

   $$a^* = \arg\max_a N(s_0, a)$$

   This reflects the action that MCTS has explored most frequently, indicating the highest confidence. It is more robust and less sensitive to noise in estimated rewards.

2. **Highest average reward (more greedy):**

   $$a^* = \arg\max_a \frac{Q(s_0, a)}{N(s_0, a)}$$

   This selects the action with the best estimated value, regardless of how often it was visited. It may be less reliable due to lower sample counts.

Interpretation: Once an action a^* is selected, MCTS discards the rest of the tree and continues the search from the child state s' resulting from a^*. Thus, the benefit of the entire subtree rooted at s' is inherited by taking this action.

Neural-Guided MCTS: Predicting Policy and Value

A simple yet powerful modification of MCTS integrates a machine learning model that helps guide the search. The model predicts:

- **Policy** $\pi(a \mid s)$: a probability distribution over actions from state s.
- **Value** $V(s)$: an estimate of the expected return from state s.

Usage in MCTS:

- **Policy Prior:** The predicted policy is used to guide exploration during the **selection** phase, biasing UCB scores toward more promising actions.
- **Value Estimate:** Instead of simulating all the way to a terminal state in the **simulation** phase, we can stop early and use the model's predicted $V(s)$ as a proxy for the final reward. This reduces rollout cost.
- **Training:** After collecting data from completed MCTS searches, the model is trained to match the improved policy (e.g., visit count distribution) and the empirical return from root state:

$$\text{Loss} = \text{CrossEntropy}(\hat{\pi}, \text{MCTS Visits}) + \text{MSE}(\hat{V}(s), R)$$

This idea underpins modern systems like AlphaZero, where neural networks generalize across states and provide powerful heuristics for action selection and value estimation.