Algorithms for Machine Learning Notes

Instructor: Prof. Alina Ene

Student: Wancheng Lin

This document is based on the slides and my notes from Prof. Alina Ene's lectures CS 599 E1: Algorithms for Machine Learning in 2025 fall.

Contents

1 Introduction

1.1 Course Overview

This seminar explores efficient algorithms for training deep learning models at scale, with a strong emphasis on modern **large language models (LLMs)**. We will discuss both classical foundations and cutting-edge techniques from recent research papers.

Topics we will cover:

- **Architectures**: deep neural networks, transformers, state-space models (SSMs), mixture-of-experts (MoE).
- **Algorithms for training, inference, and fine-tuning LLMs**.
- **Attention mechanisms**: efficient GPU implementations, approximations for time/memory reduction.
- **Optimization methods**: stochastic gradient descent (SGD), adaptive algorithms (Adam, AdaGrad), new variants (e.g., Muon).
- **Parallel/distributed training**: ZeRO, DeepSpeed, fully sharded data parallelism (FSDP).
- **Efficiency techniques**: sparsity, low-rank approximations, quantization, graph search, retrieval-augmented generation (RAG).

Not covered: reinforcement learning, alignment, prompting.

1.2 Language Models: The Basics

A **language model (LM)** assigns probabilities to text sequences. For example:

$$\Pr(\text{“the mouse ate the cheese”}) = 0.3, \quad \Pr(\text{“the cheese ate the mouse”}) = 10^{-5}.$$

Definition 1.1 (Language Model)**.** Let $V = \{w_1, w_2, \ldots, w_N\}$ be a vocabulary. A language model defines probabilities

$$\Pr(w_1, w_2, \ldots, w_m) = \prod_{t=1}^{m} \Pr(w_t \mid w_1, \ldots, w_{t-1}),$$

using the chain rule of probability.

Remark. *The above factorization highlights why LMs are difficult: we must model long dependencies. This motivates approximations (e.g., n-grams) or neural networks that learn representations.*

1.3 n-Gram Models

In an n-gram model:

$$\Pr(w_t \mid w_1, \ldots, w_{t-1}) \approx \Pr(w_t \mid w_{t-n+1}, \ldots, w_{t-1}).$$

Counting-based approach:

$$\Pr(w_t \mid w_{t-n+1}, \ldots, w_{t-1}) = \frac{\text{count}(w_{t-n+1}, \ldots, w_{t-1}, w_t)}{\text{count}(w_{t-n+1}, \ldots, w_{t-1})}.$$

Remark. *While intuitive, this approach suffers from:*

- ***Space explosion**: for vocabulary size $|V|$, storing counts for n-grams requires $|V|^n$ space.*
- ***Sparsity**: many sequences never appear in training data.*

Neural-network approach: Instead of counts, embed words into vectors and use a neural network to predict probabilities.

1.4 An Early Neural Language Model (Bengio et al., 2003)

Bengio et al. proposed one of the first neural probabilistic language models:

$$p = \text{softmax}(Wx + U\tanh(Hx + d) + b),$$

where:

- $x = (C(w_{t-n+1}), \ldots, C(w_{t-1}))$ is the concatenation of embeddings.
- C is the embedding matrix.
- $z = \tanh(Hx + d)$ is the hidden representation.
- p is a probability distribution over vocabulary V.

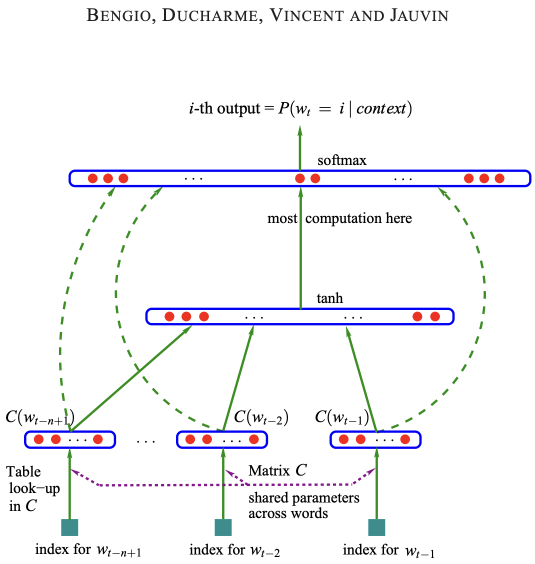


Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

Figure 1: Neural architecture from Bengio et al. (2003): embedding layer → hidden layer → softmax.

Why embeddings matter:

- Capture semantic similarity (*cat* ≈ *dog*, *walk* ≈ *run*).
- Allow generalization to unseen sequences by mapping similar words close in vector space.
- Reduce the curse of dimensionality: number of parameters grows linearly in $|V|$, not exponentially.

1.5 Deep Insights and Illustrations

Example (Embedding Power)**.**
Suppose we want to predict the next word in *"The cat is walking in the ___"*. This exact sequence may not exist in the training data. However, embeddings allow the model to learn:

$$C(\text{cat}) \approx C(\text{dog}),$$
$$C(\text{walking}) \approx C(\text{running}),$$

so the model assigns high probability to *park*, even without explicit training counts.

Remark. *This shift from **memorizing counts** to **learning distributed representations** is the intellectual foundation of modern LLMs like GPT, PaLM, and DeepSeek.*

1.6 Key Takeaways

- Language modeling reduces to conditional probability estimation over sequences.
- n-grams are limited by sparsity and scalability.
- Neural approaches with embeddings overcome these limitations by learning shared representations.
- Bengio et al. (2003) marks the beginning of neural LMs, paving the way for transformers.

1.7 References and Further Reading

- Original slides: CS 599 course website.
- Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). *A Neural Probabilistic Language Model*. JMLR, 3, 1137–1155.
- Stanford CS224N notes: Notes on LM and RNNs.
- Karpathy, A. (2022). Lecture on MLPs.

2 Supervised Learning, Architectures, Transformers

2.1 Supervised Learning Framework

We recall the language modeling problem: given a vocabulary $V = \{w_1, \ldots, w_N\}$ and a large text corpus, the goal is to learn the conditional distribution:

$$\Pr(w_t \mid w_1, \ldots, w_{t-1}).$$

Training data: A dataset $D = \{(x^{(i)}, y^{(i)}) : 1 \le i \le K\}$ consists of pairs where $x^{(i)}$ is a context and $y^{(i)}$ is the next word.

Hypothesis function: A model h_θ : contexts $\to \mathbb{R}^{|V|}$ maps a context to a probability distribution over words. The parameters θ are learned from data.

Loss function: Cross-Entropy. If $y \in \mathbb{R}^{|V|}$ is a one-hot label vector, and $\hat{y} = h_\theta(x)$ is the predicted distribution:

$$\ell_{\text{ce}}(y, \hat{y}) = -\sum_{j=1}^{|V|} y_j \log \hat{y}_j.$$

The empirical risk is

$$L(\theta) = \frac{1}{K} \sum_{i=1}^{K} \ell_{\text{ce}}(y^{(i)}, h_\theta(x^{(i)})).$$

Definition 2.1 (Supervised Learning via ERM)**.** The learning problem is to solve

$$\min_\theta \; L(\theta) = \frac{1}{K} \sum_{i=1}^{K} \ell(y^{(i)}, h_\theta(x^{(i)})).$$

Remark. *For language models, ℓ is almost always cross-entropy, equivalent to maximum likelihood estimation under the categorical distribution.*

2.2 Neural Network Architectures

Feed-forward module:

$$x \mapsto f(Wx + b),$$

where f is applied elementwise.

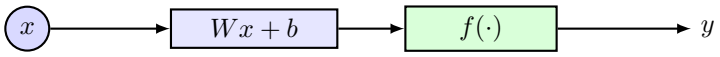


Figure 2: Feed-forward module: linear transformation followed by nonlinearity.

Multi-Layer Perceptron (MLP):

$$x^{(0)} \mapsto x^{(1)} = f(W^{(1)}x^{(0)} + b^{(1)}) \mapsto \cdots \mapsto x^{(L)}.$$

Nonlinearities:

$$\text{Sigmoid: } \sigma(x) = \frac{1}{1 + e^{-x}}, \quad \text{ReLU: } \max\{0, x\}, \quad \text{GELU, SwiGLU, etc.}$$

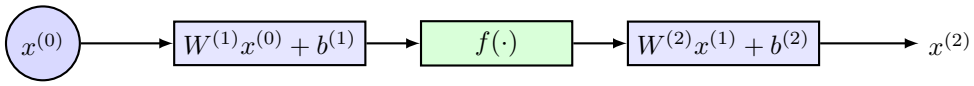


Figure 3: MLP: stacking multiple feed-forward modules.

Remark. *Deep networks are universal function approximators. The nonlinear activation is essential; otherwise the entire network reduces to a linear map.*

Residual connections (He et al., 2015):

$$x^{(i)} = x^{(i-1)} + \mathrm{FFN}(x^{(i-1)}).$$

This eases optimization by letting the model focus on learning residuals rather than full mappings.

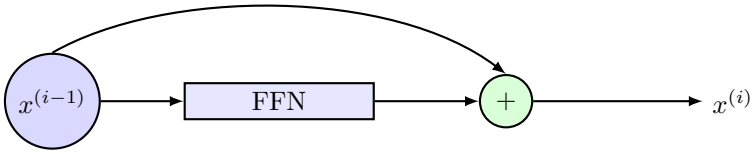


Figure 4: Residual connection: the output is $x^{(i-1)}$ plus the transformation $\mathrm{FFN}(x^{(i-1)})$.

Layer Normalization (Ba et al., 2016): For $x \in \mathbb{R}^d$,

$$\mu(x) = \frac{1}{d}\sum_{i=1}^{d} x_i, \qquad \sigma(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu(x))^2},$$

$$\mathrm{LayerNorm}(x) = \frac{x - \mu(x)\mathbf{1}}{\sigma(x)} \cdot \gamma + \beta.$$

Remark. *Normalization controls covariate shift inside networks, stabilizing gradients and accelerating convergence.*

2.3 From Static to Contextual Embeddings

Static embeddings: Each word $w \in V$ has a fixed vector $C(w) \in \mathbb{R}^m$. This ignores context.

Contextual embeddings: Construct embeddings depending on surrounding words. This motivates the **attention mechanism**.

> "You shall know a word by the company it keeps." — J. R. Firth (1957).

2.4 Attention Mechanism

Let $x_1, \ldots, x_n \in \mathbb{R}^d$ be embeddings. Define query, key, and value vectors:

$$q_i = Qx_i, \quad k_i = Kx_i, \quad v_i = Vx_i.$$

Attention score:

$$s_{ij} = \langle q_i, k_j \rangle.$$

Normalized weights:

$$\alpha_{ij} = \frac{\exp(s_{ij})}{\sum_{j'} \exp(s_{ij'})}.$$

Output embedding:

$$o_i = \sum_{j=1}^{n} \alpha_{ij} v_j.$$

Remark. *Each output vector o_i is a contextualized representation of token i, a weighted average of all value vectors, with weights reflecting similarity.*

Remark (Why Three Matrices?)**.** *The decomposition into **query**, **key**, and **value** vectors is not arbitrary—it has an intuitive interpretation.*

- ***Queries (Q):*** *represent the current information need. In analogy to a database, the query is the question we ask: "Given the word* dog*, which other words in the sentence help me predict its meaning?"*

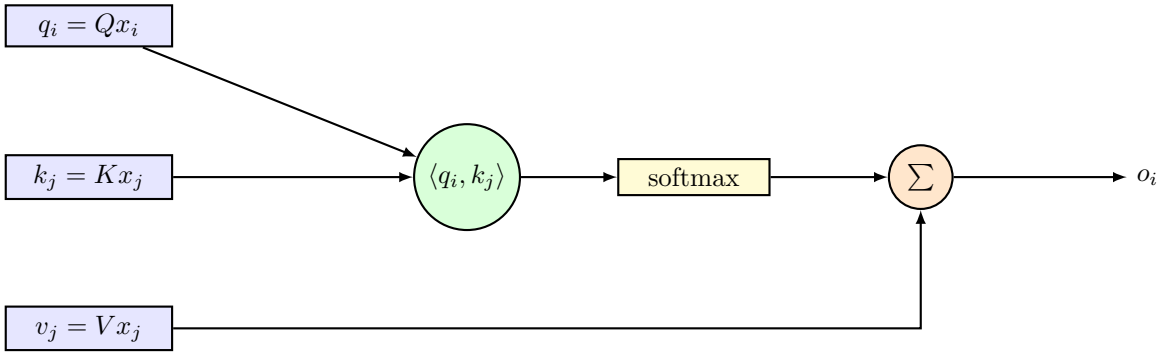


Figure 5: Attention: queries match keys, retrieve values, producing contextual embeddings.

- ***Keys*** *(K): serve as indices for lookup. Each token broadcasts an addressable signature of its content, much like how database records have primary keys used to match queries.*
- ***Values*** *(V): store the actual content to be retrieved. Once a query matches a key (via similarity), the corresponding value is the information passed forward. In a database analogy, values are the data payload stored in the record.*

Thus, the attention mechanism acts as a content-addressable memory: we do not retrieve information by position but by similarity of queries to keys. The weights α_{ij} are learned soft matches, generalizing the notion of exact equality in a database lookup into a graded notion of similarity.

This separation into (Q, K, V) also provides flexibility:

1. *Different subspaces: the same embedding x_i may project differently into query space vs. key space, allowing asymmetric roles (who is asking vs. who is answering).*
2. *Multi-head attention: multiple Q, K, V projections act as different "database queries" running in parallel, retrieving complementary information from the same context.*

Matrix form: If $X \in \mathbb{R}^{n \times d}$ stacks row vectors x_i,

$$O = \text{softmax}\left(\frac{XQK^\top X^\top}{\sqrt{d}}\right) XV.$$

2.5 Extensions of Attention

Scaling (Vaswani et al., 2017): Divide by $\sqrt{d}$ to prevent large dot products, which would cause softmax saturation and small gradients.

Masked attention: For autoregressive LM, enforce causality by setting

$$s_{ij} = \begin{cases} \langle q_i, k_j \rangle, & j \le i, \\ -\infty, & j > i. \end{cases}$$

Position embeddings: Since attention is permutation-invariant, inject position information via sinusoidal or learned embeddings:

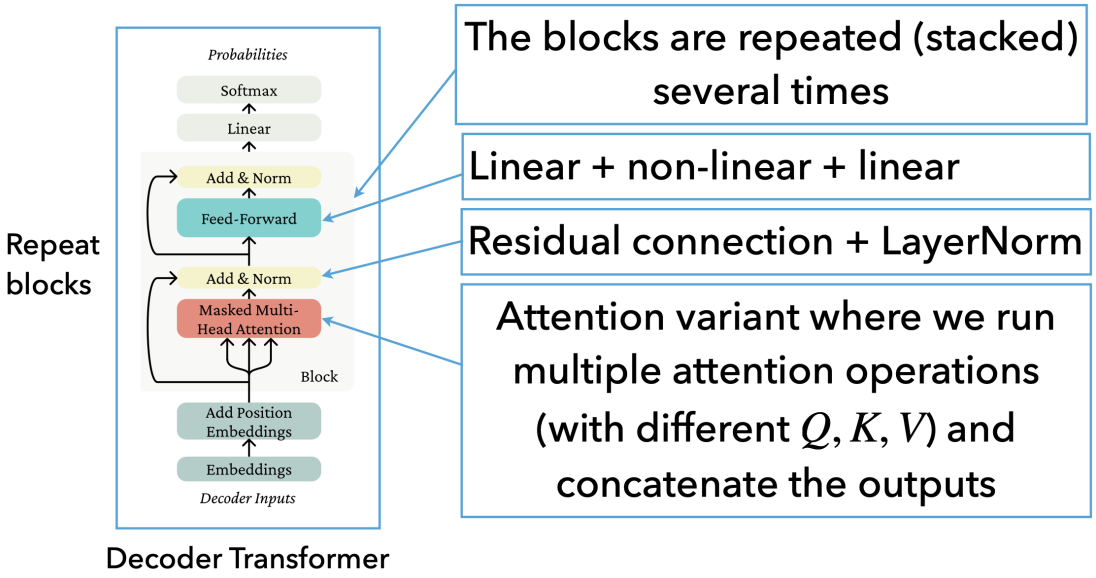
$$\tilde{x}_i = x_i + p_i, \quad p_{i,2t} = \sin\left(\tfrac{i}{10000^{2t/d}}\right), \quad p_{i,2t+1} = \cos\left(\tfrac{i}{10000^{2t/d}}\right).$$

Modern variants: relative position encodings, RoPE (rotary embeddings).

2.6 The Transformer Architecture

Introduced in *Attention Is All You Need* (Vaswani et al., 2017).

- It is the dominant architecture for modern large language models



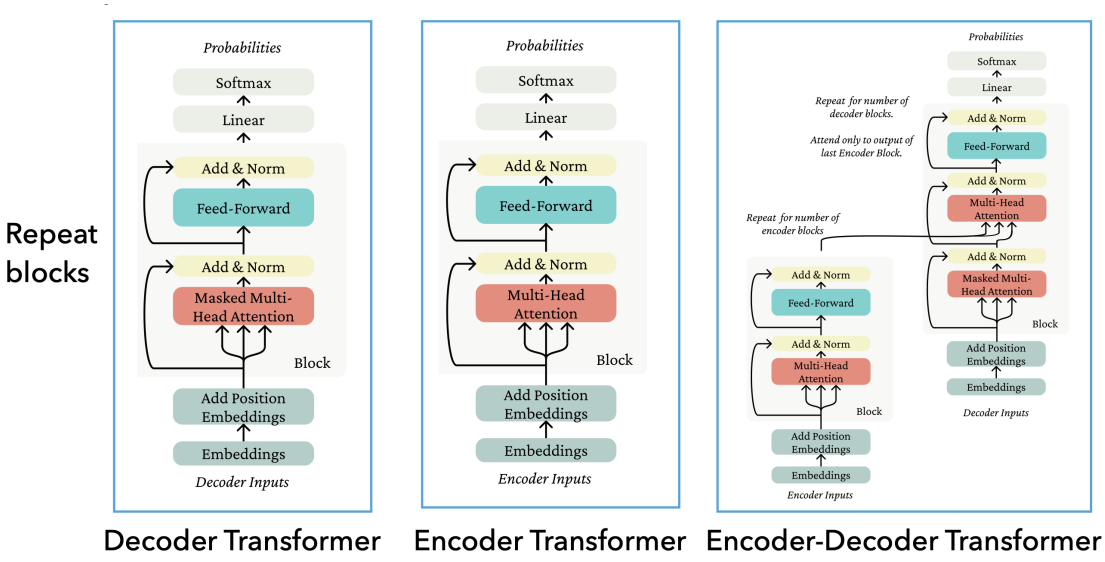
Figures from: https://web.stanford.edu/class/cs224n/readings/cs224n-self-attention-transformers-2023_draft.pdf

Core block:

- Multi-head self-attention.
- Residual connections + LayerNorm.
- Position-wise feed-forward network.

Variants:

- **Encoder:** full self-attention (no masking).
- **Decoder:** masked self-attention (causal).
- **Encoder-Decoder:** combines both for seq2seq tasks.



Figures from: https://web.stanford.edu/class/cs224n/readings/cs224n-self-attention-transformers-2023_draft.pdf

Example (Multi-head Attention).
For k heads, split d into d/k. Each head learns different projections $Q^{(\omega)}, K^{(\omega)}, V^{(\omega)}$ and outputs:

$$\text{head}_\omega = \text{Attention}(XQ^{(\omega)}, XK^{(\omega)}, XV^{(\omega)}).$$

Final output concatenates heads and projects back.

Benefits:

- Parallelizable training on GPUs.
- Strong empirical performance across NLP tasks.

Challenges:

- Quadratic cost of attention ($O(n^2)$).
- Training instabilities, mitigated by normalization and initialization tricks.

2.7 Key References

- Vaswani et al. (2017): *Attention is All You Need* (arXiv:1706.03762).
- Bahdanau et al. (2015): *Neural Machine Translation by Jointly Learning to Align and Translate.*
- He et al. (2015): Residual connections (arXiv:1512.03385).
- Ba et al. (2016): Layer normalization (arXiv:1607.06450).
- Su et al. (2023): RoPE embeddings (arXiv:2104.09864).
- Stanford CS224N draft notes on self-attention and transformers: CS224N Notes.

3 FlashAttention and Efficient Transformers

Reference: Dao et al. (2022), *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*, arXiv:2205.14135.

3.1 Motivation

Self-attention is the core operation in transformers:

$$\mathrm{Attn}(Q,K,V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V,$$

where

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V, \quad X \in \mathbb{R}^{n\times d}.$$

Naive complexity:

- Computing $QK^\top$: $O(n^2 d)$.
- Memory: must materialize $n \times n$ attention score matrix.
- For long sequences ($n = 10^4$ or more), both time and memory become bottlenecks.

Remark. *Even though GPUs have high FLOPs, they are often IO-bound. That is, the bottleneck is not arithmetic but reading/writing large intermediate matrices from GPU memory (HBM).*

3.2 Softmax Attention in Detail

For query $q_i \in \mathbb{R}^d$, keys $k_j \in \mathbb{R}^d$, and values $v_j \in \mathbb{R}^d$:

$$o_i = \sum_{j=1}^{n} \alpha_{ij} v_j, \quad \alpha_{ij} = \frac{\exp\left(\frac{q_i \cdot k_j}{\sqrt{d}}\right)}{\sum_{j'=1}^{n} \exp\left(\frac{q_i \cdot k_{j'}}{\sqrt{d}}\right)}.$$

Matrix form:

$$O = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V.$$

Definition 3.1 (Attention Score and Normalization)**.** The numerator $s_{ij} = \exp(\langle q_i, k_j\rangle/\sqrt{d})$ is the unnormalized score. The denominator $Z_i = \sum_j s_{ij}$ ensures row-stochasticity: $\sum_j \alpha_{ij} = 1$.

Remark. *Computing s_{ij} for all (i,j) requires storing n^2 values. This is the key challenge FlashAttention addresses.*

3.3 IO-Aware Algorithm Design

Observation: The naive algorithm materializes the $n \times n$ score matrix $S = QK^\top$, which requires $O(n^2)$ memory traffic to/from HBM. Yet the final output O only requires $O(nd)$ memory.

Goal: Avoid storing the entire score matrix, compute α_{ij} "on the fly" while keeping results exact.

3.4 Numerical Stability and Streaming Softmax

Recall the log-sum-exp trick for stability:

$$\text{softmax}(s_1, \ldots, s_n)_j = \frac{e^{s_j - m}}{\sum_{j'} e^{s_{j'} - m}}, \quad m = \max_j s_j.$$

For each query i, define:

$$m_i = \max_j \frac{q_i \cdot k_j}{\sqrt{d}}, \qquad \ell_i = \sum_{j=1}^{n} \exp\left(\frac{q_i \cdot k_j}{\sqrt{d}} - m_i\right).$$

Then:

$$\alpha_{ij} = \frac{\exp\left(\frac{q_i \cdot k_j}{\sqrt{d}} - m_i\right)}{\ell_i}.$$

Thus,

$$o_i = \frac{1}{\ell_i} \sum_{j=1}^{n} \exp\left(\frac{q_i \cdot k_j}{\sqrt{d}} - m_i\right) v_j.$$

Remark. *We only need to keep track of (m_i, ℓ_i, o_i) while scanning through keys and values sequentially. This enables a streaming algorithm.*

3.5 FlashAttention Algorithm

Idea: Process queries in blocks (tiles) that fit in GPU SRAM (fast on-chip memory). For each tile: 1. Load a block of queries Q_t, keys K_t, values V_t into SRAM. 2. Compute partial scores and update (m_i, ℓ_i, o_i) for queries in the tile. 3. Move to next block, update incrementally. 4. After all blocks, normalize outputs.

Formally: For each query q_i,

$$m_i^{(new)} = \max\{m_i^{(old)}, \max_{j \in \text{block}} s_{ij}\},$$

$$\ell_i^{(new)} = \ell_i^{(old)} e^{m_i^{(old)} - m_i^{(new)}} + \sum_{j \in \text{block}} \exp(s_{ij} - m_i^{(new)}),$$

$$o_i^{(new)} = o_i^{(old)} e^{m_i^{(old)} - m_i^{(new)}} + \sum_{j \in \text{block}} \exp(s_{ij} - m_i^{(new)}) v_j.$$

Remark. *This recurrence ensures numerical stability and exact equivalence to naive attention, while avoiding materialization of the full score matrix.*

3.6 Complexity Analysis

Naive attention:

$$\text{Time: } O(n^2 d), \quad \text{Memory IO: } O(n^2).$$

FlashAttention:

$$\text{Time: } O(n^2 d) \quad \text{(same arithmetic)},$$

$$\text{Memory IO: } O(nd) \quad \text{(linear in sequence length)}.$$

Example (Memory Scaling).
For $n = 16{,}000$, $d = 128$:

- Naive memory: $n^2 = 2.56 \times 10^8$ floats ≈ 1 GB.
- FlashAttention memory: $nd = 2.05 \times 10^6$ floats ≈ 8 MB.

3.7 Variants and Extensions

FlashAttention-2: Improves parallelism by splitting queries/keys into smaller blocks.

FlashAttention-3: Optimized for Hopper GPUs with tensor cores.

Applications: Training GPT-style LLMs with sequence lengths up to 64k tokens.

3.8 Key Insights

- The bottleneck in attention is memory IO, not FLOPs.
- By tiling and streaming, FlashAttention reduces IO from $O(n^2)$ to $O(nd)$.
- Numerical stability is ensured via the log-sum-exp recurrence.
- The algorithm is **exact**, not approximate.

3.9 References and Further Reading

- Dao et al. (2022), *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*, arXiv:2205.14135.
- Tri Dao's blog and talks: https://tridao.me/publications/.
- HuggingFace FlashAttention Docs.

4 Attention Variants for Time and Memory Optimization

Modern transformers rely on *multi-head attention* (MHA). While effective, the quadratic complexity in sequence length n and the memory cost of caching keys and values make inference expensive. Recent work proposes variants to trade off time, memory, and accuracy.

4.1 Dimension Reduction and SVD

We start with the mathematical foundation of low-rank approximation.

Definition 4.1 (Singular Value Decomposition)**.** Let $A \in \mathbb{R}^{n \times d}$. Then

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^\top = U \Sigma V^\top,$$

where

- $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$ are the singular values,
- $u_i \in \mathbb{R}^n$ are left singular vectors,
- $v_i \in \mathbb{R}^d$ are right singular vectors.

Remark. *The span of the first k right singular vectors $v_1, \ldots, v_k$ is the best-fit k-dimensional subspace, minimizing the squared distance of all rows of A to the subspace.*

Definition 4.2 (Rank-k Approximation)**.** The truncated SVD

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^\top$$

is the best rank-k approximation of A in Frobenius norm.

Remark. *This motivates compressing large parameter matrices (e.g. Q, K, V) into low-rank forms, saving memory and time.*

4.2 Multi-Head Attention (MHA)

Recall self-attention: given input $X \in \mathbb{R}^{n \times d}$, we compute

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V.$$

The attention output is

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V.$$

In MHA with h heads:

$$\text{head}_i = \text{Attn}(Q_i, K_i, V_i), \quad \text{MHA}(X) = [\text{head}_1; \ldots; \text{head}_h] W_O.$$

Remark. *Each head captures different relational patterns (syntax, semantics, position). But memory usage scales with $O(hnd)$, since each head has its own Q, K, V.*

4.3 Multi-Query Attention (MQA)

Definition 4.3 (Multi-Query Attention, Shazeer (2019))**.** Each head uses a distinct Q_i but shares the same K, V across all heads:

$$Q_i = XW_Q^{(i)}, \quad K = XW_K, \quad V = XW_V.$$

Remark. *This reduces memory for key-value caches from $O(hnd)$ to $O(nd)$. However, accuracy degrades since all heads look at the same K, V.*

4.4 Grouped-Query Attention (GQA)

Definition 4.4 (Grouped-Query Attention, Ainslie et al. (2023))**.** Partition h query heads into g groups. Each group shares one K, V pair:

$$Q_i = XW_Q^{(i)}, \quad K^{(j)} = XW_K^{(j)}, \quad V^{(j)} = XW_V^{(j)},$$

where j indexes the group.

Remark. *This interpolates between:*

- $g = h$*: equivalent to MHA (full expressiveness, high memory).*
- $g = 1$*: equivalent to MQA (low memory, lower accuracy).*

Intermediate g balances quality and efficiency.

4.5 Multi-Head Latent Attention (MLA)

Definition 4.5 (Multi-Head Latent Attention, DeepSeek-AI (2024))**.** MLA keeps separate Q, K, V but compresses them into low-rank latent vectors:

$$c_{KV,t} = W_{DKV} x_t \in \mathbb{R}^{d_c}, \quad k_t = W_{UK} c_{KV,t}, \quad v_t = W_{UV} c_{KV,t},$$

$$c_{Q,t} = W_{DQ} x_t \in \mathbb{R}^{d'_c}, \quad q_t = W_{UQ} c_{Q,t}.$$

Remark. *Instead of caching all K, V, we only cache the compressed latent vector $c_{KV,t}$, reducing KV memory from $O(hnd)$ to $O(d_c n)$, where $d_c \ll d$. Empirical results show MLA matches or outperforms MHA while using a fraction of KV cache.*

4.6 Comparisons

Memory Cost per Token. (from DeepSeek, 2024)

Mechanism	KV Cache Size
MHA	$O(hd)$
MQA	$O(d)$
GQA (g groups)	$O(gd)$
MLA	$O(d_c),\ d_c \ll d$

Accuracy Tradeoffs. On benchmarks (MMLU, C-Eval, CMMLU):

- MHA: strongest baseline.
- MQA: large memory savings but accuracy loss.
- GQA: intermediate accuracy, intermediate memory.
- MLA: stronger than MHA, with much smaller KV cache.

Remark. *MLA is enabled by the empirical observation that K, V matrices in MHA are approximately low-rank (Yu et al., 2024).*

4.7 References

- Shazeer (2019): Multi-Query Attention, arXiv:1911.02150.
- Ainslie et al. (2023): Grouped-Query Attention, arXiv:2305.13245.
- DeepSeek-AI (2024): Multi-Head Latent Attention, arXiv:2405.04434.
- Yu et al. (2024): Empirical low-rank structure of K, V, arXiv:2406.07056.

5 Backpropagation Algorithm

5.1 Supervised Learning Framework

We consider a dataset

$$D = \{(x^{(i)}, y^{(i)}) : 1 \le i \le K\},$$

where each $x^{(i)}$ is a context (input) and $y^{(i)}$ is the label (e.g., next-word).

We choose a hypothesis class h_θ (e.g. neural networks) and a loss function $\ell(y, \hat{y})$ (e.g. cross-entropy). The empirical risk is

$$f(\theta) = \frac{1}{K} \sum_{i=1}^{K} \ell(y^{(i)}, h_\theta(x^{(i)})).$$

Definition 5.1 (Supervised Learning via ERM)**.** The learning problem is

$$\min_\theta f(\theta).$$

5.2 Gradient-based Optimization

To minimize $f(\theta)$, we use gradient descent:

$$\theta \leftarrow \theta - \eta \nabla f(\theta).$$

The gradient is

$$\nabla f(\theta) = \left(\frac{\partial f(\theta)}{\partial \theta_1}, \dots, \frac{\partial f(\theta)}{\partial \theta_N} \right)^\top.$$

Key observations:

- $f(\theta)$ is complex but structured.
- We can compute $\nabla_\theta \ell(y, h_\theta(x))$ for a single example and average over dataset.
- $-\nabla f(\theta)$ points in the steepest descent direction.

5.3 Backpropagation: Computing Gradients

Neural networks are compositions of functions (modules). Backprop computes derivatives module by module, using the chain rule.

Distinction from symbolic calculus:

- Calculus: we compute derivative formulas symbolically (e.g. $f(x) = x^3 \implies f'(x) = 3x^2$).
- Backprop: we only need derivatives at specific parameter values (e.g. $f'(2) = 12$).

This allows efficient numerical evaluation rather than symbolic manipulation.

5.4 Example: Gradient via Chain Rule

Suppose $x \in \mathbb{R}$, define

$$y_1 = x, \qquad y_2 = x^2,$$
$$z = 2y_1 + y_2.$$

Then

$$\frac{\partial z}{\partial y_1} = 2, \quad \frac{\partial z}{\partial y_2} = 1,$$
$$\frac{\partial y_1}{\partial x} = 1, \quad \frac{\partial y_2}{\partial x} = 2x.$$

By chain rule:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x} = 2 \cdot 1 + 1 \cdot (2x) = 2 + 2x.$$

At $x = 3$: $\partial z / \partial x = 8$.

5.5 Forward and Backward Pass

- **Forward pass:** compute values y_1, y_2, z from input x. Example: $x = 3 \implies y_1 = 3, y_2 = 9, z = 15$.
- **Backward pass:** compute gradients $\partial z/\partial y$, $\partial z/\partial x$ using chain rule.

This mimics actual backprop: first evaluate the network, then propagate derivatives backward.

5.6 General Chain Rule (Component Form)

Let $x \in \mathbb{R}^m$, $y = f(x) \in \mathbb{R}^n$, $z = g(y) \in \mathbb{R}^p$. Then

$$\frac{\partial z_i}{\partial x_j} = \sum_{k=1}^{n} \frac{\partial g_i}{\partial y_k} \frac{\partial f_k}{\partial x_j}.$$

This is the multivariate chain rule.

5.7 Matrix Formulation

Define Jacobian of f:

$$J_f(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \in \mathbb{R}^{n \times m}.$$

Chain rule in matrix form:

$$J_{g \circ f}(x) = J_g(f(x)) \cdot J_f(x).$$

Transpose gives backward rule:

$$\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x} \cdot \frac{\partial z}{\partial y}.$$

5.8 Example in Matrix Form

$x = 3$, $y = (x, x^2) = (3, 9)$, $z = 2y_1 + y_2 = 15$. We have

$$\frac{\partial z}{\partial y} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \qquad \frac{\partial y}{\partial x} = \begin{bmatrix} 1 \\ 2x \end{bmatrix}.$$

Thus

$$\frac{\partial z}{\partial x} = \begin{bmatrix} 1 & 2x \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = 2 + 2x = 8.$$

5.9 Backward Function

We define the backward mapping

$$b_{f,x}(v) = \frac{\partial f}{\partial x} \cdot v,$$

where $v = \frac{\partial z}{\partial y}$.

This maps sensitivities w.r.t. outputs to sensitivities w.r.t. inputs. This is the mathematical basis of automatic differentiation and backprop.

5.10 Backpropagation Algorithm

To compute $\nabla_\theta \ell(y, h_\theta(x))$:

1. Do a forward pass: compute intermediate values for each module.
2. Do a backward pass: apply $b_{f,x}$ for each module, propagating derivatives backward from the loss.
3. Accumulate gradients w.r.t. parameters θ.

Formally: for a module $y = f(x)$, given $v = \partial z/\partial y$, backprop computes

$$\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x} \cdot v.$$

This is applied recursively module by module until reaching the input and parameters.

5.11 Remarks

- Backprop is efficient: cost is proportional to a forward pass.
- Each module needs to implement its local backward rule.
- Frameworks like PyTorch implement this via `.backward()`.

5.12 References

- Stanford CS229 Notes: Chapter 7, Deep Learning link
- Dao et al. (2022), FlashAttention: memory and IO efficient attention link

6 Backpropagation: Module Formulas

6.1 Setup

Given a dataset $D = \{(x^{(i)}, y^{(i)})\}$, a neural network with parameters θ makes a prediction

$$\hat{y} = h_\theta(x^{(i)}),$$

and incurs a scalar loss $\ell(\hat{y}) \in \mathbb{R}$. Training requires computing $\nabla_\theta \ell$.

Backpropagation starts by computing

$$\frac{\partial \ell}{\partial \hat{y}},$$

then propagates gradients backwards using the chain rule:

$$\frac{\partial \ell}{\partial x} = \frac{\partial f}{\partial x} \cdot \frac{\partial \ell}{\partial y}.$$

The backward function is

$$b_{f,x}(v) = \frac{\partial f}{\partial x} \cdot v,$$

where $v = \frac{\partial \ell}{\partial y}$ are the sensitivities from later layers.

—

6.2 Linear Module

Forward: $y = Wx$, where $x \in \mathbb{R}^m$, $W \in \mathbb{R}^{n \times m}$, $y \in \mathbb{R}^n$.

Backward:

$$b_{Wx,x}(v) = W^\top v, \qquad b_{Wx,W}(v)_{ij} = x_j v_i.$$

Thus:

$$\frac{\partial \ell}{\partial x} = W^\top \frac{\partial \ell}{\partial y}, \qquad \frac{\partial \ell}{\partial W} = \left(\frac{\partial \ell}{\partial y}\right) x^\top.$$

—

6.3 Non-linear Activations

Forward: $y = \sigma(x) = (\sigma(x_1), \ldots, \sigma(x_m))$, elementwise.

Backward:

$$b_{\sigma,x}(v) = \sigma'(x) \odot v,$$

where $\odot$ is elementwise multiplication.

Examples:

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

$$\sigma(x) = \tanh(x) \quad \Rightarrow \quad \sigma'(x) = 1 - \tanh^2(x).$$

$$\sigma(x) = \mathrm{ReLU}(x) \quad \Rightarrow \quad \sigma'(x) = \mathbf{1}_{x>0}.$$

—

6.4 Feed-Forward Network (FFN)

Example: 3-layer FFN

$$z^{(0)} = x,$$

$$y^{(1)} = W^{(1)} z^{(0)}, \quad z^{(1)} = \sigma(y^{(1)}),$$
$$y^{(2)} = W^{(2)} z^{(1)}, \quad z^{(2)} = \sigma(y^{(2)}),$$
$$y^{(3)} = W^{(3)} z^{(2)},$$
$$\ell = \ell(y^{(3)}).$$

Backward pass:

$$\frac{\partial \ell}{\partial y^{(3)}} = b_{\ell, y^{(3)}}(1),$$
$$\frac{\partial \ell}{\partial z^{(2)}} = b_{\text{lin}, z^{(2)}}\left(\frac{\partial \ell}{\partial y^{(3)}}\right), \quad \frac{\partial \ell}{\partial W^{(3)}} = b_{\text{lin}, W^{(3)}}\left(\frac{\partial \ell}{\partial y^{(3)}}\right),$$
$$\frac{\partial \ell}{\partial y^{(2)}} = b_{\sigma, y^{(2)}}\left(\frac{\partial \ell}{\partial z^{(2)}}\right),$$
$$\frac{\partial \ell}{\partial z^{(1)}} = b_{\text{lin}, z^{(1)}}\left(\frac{\partial \ell}{\partial y^{(2)}}\right), \quad \frac{\partial \ell}{\partial W^{(2)}} = b_{\text{lin}, W^{(2)}}\left(\frac{\partial \ell}{\partial y^{(2)}}\right),$$
$$\frac{\partial \ell}{\partial y^{(1)}} = b_{\sigma, y^{(1)}}\left(\frac{\partial \ell}{\partial z^{(1)}}\right),$$
$$\frac{\partial \ell}{\partial z^{(0)}} = b_{\text{lin}, z^{(0)}}\left(\frac{\partial \ell}{\partial y^{(1)}}\right), \quad \frac{\partial \ell}{\partial W^{(1)}} = b_{\text{lin}, W^{(1)}}\left(\frac{\partial \ell}{\partial y^{(1)}}\right).$$

—

6.5 Attention Module

Setup: $Q, K, V \in \mathbb{R}^{n \times d}$.

$$S = QK^\top \in \mathbb{R}^{n \times n}, \quad P = \text{softmax}(S) \in \mathbb{R}^{n \times n}, \quad O = PV \in \mathbb{R}^{n \times d}.$$

We want $\nabla \ell$ w.r.t. Q, K, V.

Step 1. Gradient w.r.t. V:

$$\frac{\partial \ell}{\partial V_{i:}} = \sum_{u=1}^{n} P_{ui} \frac{\partial \ell}{\partial O_{u:}}.$$

Step 2. Gradient w.r.t. P:

$$\frac{\partial \ell}{\partial P_{i:}} = \frac{\partial \ell}{\partial O_{i:}} V.$$

Step 3. Gradient w.r.t. S: Softmax Jacobian: if $p = \text{softmax}(s)$, then

$$\frac{\partial p}{\partial s} = \text{diag}(p) - pp^\top.$$

Thus for row i:

$$\frac{\partial \ell}{\partial S_{i:}} = \left(\text{diag}(P_{i:}) - P_{i:} P_{i:}^\top\right) \frac{\partial \ell}{\partial P_{i:}}.$$

Step 4. Gradient w.r.t. Q, K:

$$\frac{\partial \ell}{\partial Q_{i:}} = \sum_{v=1}^{d} \frac{\partial \ell}{\partial S_{iv}} K_{v:},$$
$$\frac{\partial \ell}{\partial K_{j:}} = \sum_{u=1}^{n} \frac{\partial \ell}{\partial S_{uj}} Q_{u:}.$$

—

6.6 Memory Optimization: Activation Checkpointing

Naive backprop stores all activations $z^{(i)}$ during forward pass. This is memory expensive, especially in attention.

Remark. ***Activation checkpointing:*** *Only store every d-th activation $z^{(id)}$. During backprop, recompute missing activations with mini-forward passes. This trades increased compute (roughly 2x FLOPs) for large memory savings.*

Extensions:

- Store approximate activations (sparse, low-rank, low-precision).
- Combine approximation with checkpointing.
- FlashAttention (Dao et al. 2022) uses memory- and IO-efficient exact formulas.

—

6.7 Takeaways

- Backprop is applying the chain rule module by module.
- Each module needs local forward + backward formulas: linear, activation, FFN, attention.
- Efficient training requires both mathematical understanding and system-level memory optimization.