Generative Models in Finance

Week 1: The Mathematics of Large Language Models

Cristopher Salvi
Imperial College London

Spring 2026

Overview

1. Statistical Language Modelling

Overview

1. Statistical Language Modelling
2. Neural Networks as Function Approximators

Overview

1. Statistical Language Modelling
2. Neural Networks as Function Approximators
3. Recurrent Neural Networks

Overview

1. Statistical Language Modelling
2. Neural Networks as Function Approximators
3. Recurrent Neural Networks
4. The Transformer Architecture

Overview

1. Statistical Language Modelling
2. Neural Networks as Function Approximators
3. Recurrent Neural Networks
4. The Transformer Architecture
5. Scaling Laws

Reference: M. R. Douglas, *Large Language Models*, arXiv:2307.05782, 2023.

Part 1: Statistical Language Modelling

Goal: Build a mathematical framework for the problem of modelling natural language with probability distributions.

- What is language modelling?

Part 1: Statistical Language Modelling

Goal: Build a mathematical framework for the problem of modelling natural language with probability distributions.

- What is language modelling?
- The chain rule decomposition

Part 1: Statistical Language Modelling

Goal: Build a mathematical framework for the problem of modelling natural language with probability distributions.

- What is language modelling?
- The chain rule decomposition
- Tokenisation strategies

Part 1: Statistical Language Modelling

Goal: Build a mathematical framework for the problem of modelling natural language with probability distributions.

- What is language modelling?
- The chain rule decomposition
- Tokenisation strategies
- N-gram models and their limitations

Part 1: Statistical Language Modelling

Goal: Build a mathematical framework for the problem of modelling natural language with probability distributions.

- What is language modelling?
- The chain rule decomposition
- Tokenisation strategies
- N-gram models and their limitations
- Evaluation via perplexity

Part 1: Statistical Language Modelling

Goal: Build a mathematical framework for the problem of modelling natural language with probability distributions.

- What is language modelling?
- The chain rule decomposition
- Tokenisation strategies
- N-gram models and their limitations
- Evaluation via perplexity
- Neural language models

What is Language Modelling?

- A language model is a probability distribution over sequences of tokens (words, characters, or subwords)

What is Language Modelling?

- A language model is a probability distribution over sequences of tokens (words, characters, or subwords)
- Given a vocabulary $\mathcal{V} = \{v_1, v_2, \ldots, v_{|\mathcal{V}|}\}$ (a finite set of tokens), a language model assigns a probability to every finite sequence $(x_1, x_2, \ldots, x_T) \in \mathcal{V}^T$

What is Language Modelling?

- A language model is a probability distribution over sequences of tokens (words, characters, or subwords)
- Given a vocabulary $\mathcal{V} = \{v_1, v_2, \ldots, v_{|\mathcal{V}|}\}$ (a finite set of tokens), a language model assigns a probability to every finite sequence $(x_1, x_2, \ldots, x_T) \in \mathcal{V}^T$
- Core tasks:
  - *Estimation:* Given a sequence $(x_1, \ldots, x_T)$, compute $P(x_1, \ldots, x_T)$
  - *Generation:* Sample new sequences from the learned distribution
  - *Next-token prediction:* Compute $P(x_{t+1} \mid x_1, \ldots, x_t)$

What is Language Modelling?

- A language model is a probability distribution over sequences of tokens (words, characters, or subwords)
- Given a vocabulary $\mathcal{V} = \{v_1, v_2, \ldots, v_{|\mathcal{V}|}\}$ (a finite set of tokens), a language model assigns a probability to every finite sequence $(x_1, x_2, \ldots, x_T) \in \mathcal{V}^T$
- Core tasks:
  - *Estimation:* Given a sequence $(x_1, \ldots, x_T)$, compute $P(x_1, \ldots, x_T)$
  - *Generation:* Sample new sequences from the learned distribution
  - *Next-token prediction:* Compute $P(x_{t+1} \mid x_1, \ldots, x_t)$
- Modern LLMs (GPT, LLaMA, Claude, etc.) are fundamentally autoregressive language models: they are trained to predict the next token given all preceding tokens

The Chain Rule Decomposition

- By the chain rule of probability, any joint distribution over a sequence can be factored as:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, x_2, \ldots, x_{t-1}) \tag{1.1}$$

The Chain Rule Decomposition

- By the chain rule of probability, any joint distribution over a sequence can be factored as:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, x_2, \ldots, x_{t-1}) \tag{1.1}$$

- This is an *exact* identity — no modelling assumptions have been made

The Chain Rule Decomposition

- By the chain rule of probability, any joint distribution over a sequence can be factored as:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, x_2, \ldots, x_{t-1}) \tag{1.1}$$

- This is an *exact* identity — no modelling assumptions have been made
- The entire language modelling problem thus reduces to modelling the conditional distributions

$$P(x_t \mid x_1, \ldots, x_{t-1}) \quad \text{for each } t = 1, \ldots, T$$

The Chain Rule Decomposition

- By the chain rule of probability, any joint distribution over a sequence can be factored as:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, x_2, \ldots, x_{t-1}) \tag{1.1}$$

- This is an *exact* identity — no modelling assumptions have been made
- The entire language modelling problem thus reduces to modelling the conditional distributions

$$P(x_t \mid x_1, \ldots, x_{t-1}) \quad \text{for each } t = 1, \ldots, T$$

- Convention: $P(x_1 \mid x_0) := P(x_1)$ where x_0 is a special start-of-sequence token

The Chain Rule Decomposition

- By the chain rule of probability, any joint distribution over a sequence can be factored as:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, x_2, \ldots, x_{t-1}) \tag{1.1}$$

- This is an *exact* identity — no modelling assumptions have been made
- The entire language modelling problem thus reduces to modelling the conditional distributions

$$P(x_t \mid x_1, \ldots, x_{t-1}) \quad \text{for each } t = 1, \ldots, T$$

- Convention: $P(x_1 \mid x_0) := P(x_1)$ where x_0 is a special start-of-sequence token
- Key challenge: The context $(x_1, \ldots, x_{t-1})$ can be arbitrarily long, making direct estimation intractable for large t

Tokenisation

- Before modelling, raw text must be converted into a sequence of elements from a fixed vocabulary $\mathcal{V}$. This process is called tokenisation

Tokenisation

- Before modelling, raw text must be converted into a sequence of elements from a fixed vocabulary $\mathcal{V}$. This process is called tokenisation
- Word-level tokenisation: Each token is a word
  - Vocabulary can be very large ($|\mathcal{V}| > 100{,}000$)
  - Cannot handle out-of-vocabulary (OOV) words

Tokenisation

- Before modelling, raw text must be converted into a sequence of elements from a fixed vocabulary $\mathcal{V}$. This process is called tokenisation
- Word-level tokenisation: Each token is a word
  - Vocabulary can be very large ($|\mathcal{V}| > 100{,}000$)
  - Cannot handle out-of-vocabulary (OOV) words
- Character-level tokenisation: Each token is a single character
  - Very small vocabulary ($|\mathcal{V}| \approx 256$ for ASCII)
  - Sequences become very long, making modelling harder

Tokenisation

- Before modelling, raw text must be converted into a sequence of elements from a fixed vocabulary $\mathcal{V}$. This process is called tokenisation
- Word-level tokenisation: Each token is a word
  - Vocabulary can be very large ($|\mathcal{V}| > 100{,}000$)
  - Cannot handle out-of-vocabulary (OOV) words
- Character-level tokenisation: Each token is a single character
  - Very small vocabulary ($|\mathcal{V}| \approx 256$ for ASCII)
  - Sequences become very long, making modelling harder
- Byte Pair Encoding (BPE) (Sennrich et al., 2016): A data-driven subword method
  1. Start with character-level vocabulary
  2. Iteratively merge the most frequent pair of adjacent tokens into a new token
  3. Repeat until desired vocabulary size is reached

Tokenisation

- Before modelling, raw text must be converted into a sequence of elements from a fixed vocabulary $\mathcal{V}$. This process is called tokenisation
- Word-level tokenisation: Each token is a word
  - Vocabulary can be very large ($|\mathcal{V}| > 100{,}000$)
  - Cannot handle out-of-vocabulary (OOV) words
- Character-level tokenisation: Each token is a single character
  - Very small vocabulary ($|\mathcal{V}| \approx 256$ for ASCII)
  - Sequences become very long, making modelling harder
- Byte Pair Encoding (BPE) (Sennrich et al., 2016): A data-driven subword method
  1. Start with character-level vocabulary
  2. Iteratively merge the most frequent pair of adjacent tokens into a new token
  3. Repeat until desired vocabulary size is reached
- BPE provides a practical compromise: $|\mathcal{V}| \approx 30{,}000$–$100{,}000$, handles rare words via subword decomposition, and is used by most modern LLMs (GPT uses a variant with $|\mathcal{V}| \approx 50{,}000$)

N-gram Language Models

- Idea: Approximate the conditional distribution by truncating the context to the most recent $n-1$ tokens (a Markov assumption of order $n-1$):

$$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) \tag{1.2}$$

N-gram Language Models

- Idea: Approximate the conditional distribution by truncating the context to the most recent $n-1$ tokens (a Markov assumption of order $n-1$):

$$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) \tag{1.2}$$

- These conditional probabilities are estimated by counting from a training corpus:

$$\hat{P}(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{\text{count}(x_{t-n+1}, \ldots, x_t)}{\text{count}(x_{t-n+1}, \ldots, x_{t-1})} \tag{1.3}$$

N-gram Language Models

- Idea: Approximate the conditional distribution by truncating the context to the most recent $n-1$ tokens (a Markov assumption of order $n-1$):

$$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) \tag{1.2}$$

- These conditional probabilities are estimated by counting from a training corpus:

$$\hat{P}(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{\text{count}(x_{t-n+1}, \ldots, x_t)}{\text{count}(x_{t-n+1}, \ldots, x_{t-1})} \tag{1.3}$$

- Curse of dimensionality: The number of possible n-grams is $|\mathcal{V}|^n$
  - For $|\mathcal{V}| = 50{,}000$ and $n = 5$: there are $50{,}000^5 \approx 3.1 \times 10^{23}$ possible 5-grams
  - Most n-grams will never appear in any finite corpus

N-gram Language Models

- Idea: Approximate the conditional distribution by truncating the context to the most recent $n-1$ tokens (a Markov assumption of order $n-1$):

$$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) \tag{1.2}$$

- These conditional probabilities are estimated by counting from a training corpus:

$$\hat{P}(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{\text{count}(x_{t-n+1}, \ldots, x_t)}{\text{count}(x_{t-n+1}, \ldots, x_{t-1})} \tag{1.3}$$

- Curse of dimensionality: The number of possible n-grams is $|\mathcal{V}|^n$
  - For $|\mathcal{V}| = 50{,}000$ and $n = 5$: there are $50{,}000^5 \approx 3.1 \times 10^{23}$ possible 5-grams
  - Most n-grams will never appear in any finite corpus
- In practice, n-gram models are limited to small n (typically $n \leq 5$) and require smoothing techniques (e.g. Laplace smoothing, Kneser-Ney) to handle unseen n-grams

N-gram Language Models

- Idea: Approximate the conditional distribution by truncating the context to the most recent $n-1$ tokens (a Markov assumption of order $n-1$):

$$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) \tag{1.2}$$

- These conditional probabilities are estimated by counting from a training corpus:

$$\hat{P}(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{\text{count}(x_{t-n+1}, \ldots, x_t)}{\text{count}(x_{t-n+1}, \ldots, x_{t-1})} \tag{1.3}$$

- Curse of dimensionality: The number of possible n-grams is $|\mathcal{V}|^n$
  - For $|\mathcal{V}| = 50{,}000$ and $n = 5$: there are $50{,}000^5 \approx 3.1 \times 10^{23}$ possible 5-grams
  - Most n-grams will never appear in any finite corpus
- In practice, n-gram models are limited to small n (typically $n \leq 5$) and require smoothing techniques (e.g. Laplace smoothing, Kneser-Ney) to handle unseen n-grams
- Fundamental limitation: N-gram models cannot capture long-range dependencies in language

Evaluation: Cross-Entropy and Perplexity

- The evaluation problem: Given a trained model Q and held-out text $(x_1, \ldots, x_T)$, how do we measure how well Q predicts this text?

Evaluation: Cross-Entropy and Perplexity

- The evaluation problem: Given a trained model Q and held-out text $(x_1, \ldots, x_T)$, how do we measure how well Q predicts this text?
- Natural idea: A good model assigns *high probability* to text that actually occurs. So we want to measure the average log-probability the model assigns to each observed token:

$$\frac{1}{T} \sum_{t=1}^{T} \log_2 Q(x_t \mid x_1, \ldots, x_{t-1}) \tag{1.4}$$

Higher is better (each term is ≤ 0, and equals 0 only if the model is certain)

Evaluation: Cross-Entropy and Perplexity

- The evaluation problem: Given a trained model Q and held-out text $(x_1, \ldots, x_T)$, how do we measure how well Q predicts this text?
- Natural idea: A good model assigns *high probability* to text that actually occurs. So we want to measure the average log-probability the model assigns to each observed token:

$$\frac{1}{T} \sum_{t=1}^{T} \log_2 Q(x_t \mid x_1, \ldots, x_{t-1}) \tag{1.4}$$

  Higher is better (each term is ≤ 0, and equals 0 only if the model is certain)
- By convention, we negate this to obtain a loss (lower is better):

$$\hat{H}(P, Q) = -\frac{1}{T} \sum_{t=1}^{T} \log_2 Q(x_t \mid x_1, \ldots, x_{t-1}) \tag{1.5}$$

  This is the per-token cross-entropy of Q on the test corpus

Evaluation: Cross-Entropy and Perplexity

- The evaluation problem: Given a trained model Q and held-out text $(x_1, \ldots, x_T)$, how do we measure how well Q predicts this text?
- Natural idea: A good model assigns *high probability* to text that actually occurs. So we want to measure the average log-probability the model assigns to each observed token:

$$\frac{1}{T} \sum_{t=1}^{T} \log_2 Q(x_t \mid x_1, \ldots, x_{t-1}) \tag{1.4}$$

  Higher is better (each term is ≤ 0, and equals 0 only if the model is certain)
- By convention, we negate this to obtain a loss (lower is better):

$$\hat{H}(P, Q) = -\frac{1}{T} \sum_{t=1}^{T} \log_2 Q(x_t \mid x_1, \ldots, x_{t-1}) \tag{1.5}$$

  This is the per-token cross-entropy of Q on the test corpus
- Units: measured in *bits per token* (when using $\log_2$) or *nats per token* (when using ln). One bit = the information needed to resolve one fair coin flip

From Cross-Entropy to Perplexity

- The per-token cross-entropy $\hat{H}(P, Q)$ (1.5) is an empirical estimate of the cross-entropy of Q relative to the true distribution P:

$$H(P, Q) = -\mathbb{E}_{X \sim P}[\log_2 Q(X)] = -\sum_x P(x) \log_2 Q(x) \tag{1.6}$$

From Cross-Entropy to Perplexity

- The per-token cross-entropy $\hat{H}(P, Q)$ (1.5) is an empirical estimate of the cross-entropy of Q relative to the true distribution P:

$$H(P, Q) = -\mathbb{E}_{X \sim P}[\log_2 Q(X)] = -\sum_x P(x) \log_2 Q(x) \tag{1.6}$$

- Cross-entropy in bits is not always easy to interpret. The perplexity converts it to a more intuitive scale:

$$\mathrm{PP}(Q) = 2^{\hat{H}(P,Q)} = \left(\prod_{t=1}^{T} Q(x_t \mid x_1, \ldots, x_{t-1}) \right)^{-1/T} \tag{1.7}$$

From Cross-Entropy to Perplexity

- The per-token cross-entropy $\hat{H}(P, Q)$ (1.5) is an empirical estimate of the cross-entropy of Q relative to the true distribution P:

$$H(P, Q) = -\mathbb{E}_{X \sim P}[\log_2 Q(X)] = -\sum_x P(x) \log_2 Q(x) \tag{1.6}$$

- Cross-entropy in bits is not always easy to interpret. The perplexity converts it to a more intuitive scale:

$$\mathrm{PP}(Q) = 2^{\hat{H}(P,Q)} = \left(\prod_{t=1}^{T} Q(x_t \mid x_1, \ldots, x_{t-1}) \right)^{-1/T} \tag{1.7}$$

- Interpretation: Perplexity is the (geometric) average number of equally likely tokens the model considers at each step. Lower perplexity = better model

From Cross-Entropy to Perplexity

- The per-token cross-entropy $\hat{H}(P, Q)$ (1.5) is an empirical estimate of the cross-entropy of Q relative to the true distribution P:

$$H(P, Q) = -\mathbb{E}_{X \sim P}[\log_2 Q(X)] = -\sum_x P(x) \log_2 Q(x) \tag{1.6}$$

- Cross-entropy in bits is not always easy to interpret. The perplexity converts it to a more intuitive scale:

$$\mathrm{PP}(Q) = 2^{\hat{H}(P,Q)} = \left(\prod_{t=1}^{T} Q(x_t \mid x_1, \ldots, x_{t-1}) \right)^{-1/T} \tag{1.7}$$

- Interpretation: Perplexity is the (geometric) average number of equally likely tokens the model considers at each step. Lower perplexity = better model
- Boundary cases:
  - A *perfect* model ($Q = P$, certainty on each token) has $\mathrm{PP} = 1$
  - A *uniform random* model ($Q(\cdot) = 1/|\mathcal{V}|$) has $\mathrm{PP} = |\mathcal{V}|$
  - A model that assigns probability 0 to an observed token has $\mathrm{PP} = \infty$

From Cross-Entropy to Perplexity

- The per-token cross-entropy $\hat{H}(P, Q)$ (1.5) is an empirical estimate of the cross-entropy of Q relative to the true distribution P:

$$H(P, Q) = -\mathbb{E}_{X \sim P}[\log_2 Q(X)] = -\sum_x P(x) \log_2 Q(x) \tag{1.6}$$

- Cross-entropy in bits is not always easy to interpret. The perplexity converts it to a more intuitive scale:

$$\mathrm{PP}(Q) = 2^{\hat{H}(P,Q)} = \left(\prod_{t=1}^{T} Q(x_t \mid x_1, \ldots, x_{t-1}) \right)^{-1/T} \tag{1.7}$$

- Interpretation: Perplexity is the (geometric) average number of equally likely tokens the model considers at each step. Lower perplexity = better model
- Boundary cases:
  - A *perfect* model ($Q = P$, certainty on each token) has $\mathrm{PP} = 1$
  - A *uniform random* model ($Q(\cdot) = 1/|\mathcal{V}|$) has $\mathrm{PP} = |\mathcal{V}|$
  - A model that assigns probability 0 to an observed token has $\mathrm{PP} = \infty$
- Typical values: State-of-the-art LLMs achieve perplexities of ∼5–15 on standard benchmarks (e.g. WikiText-103), depending on tokenisation

Cross-Entropy and KL Divergence

- The cross-entropy (1.6) decomposes as:

$$H(P, Q) = H(P) + D_{\mathrm{KL}}(P \,\|\, Q) \tag{1.8}$$

where $H(P) = -\sum_x P(x) \log_2 P(x)$ is the entropy of the true distribution and

$$D_{\mathrm{KL}}(P \,\|\, Q) = \sum_x P(x) \log_2 \frac{P(x)}{Q(x)} \geq 0 \tag{1.9}$$

is the Kullback–Leibler divergence from Q to P

Cross-Entropy and KL Divergence

- The cross-entropy (1.6) decomposes as:

$$H(P, Q) = H(P) + D_{\mathrm{KL}}(P \,\|\, Q) \tag{1.8}$$

  where $H(P) = -\sum_x P(x) \log_2 P(x)$ is the entropy of the true distribution and

$$D_{\mathrm{KL}}(P \,\|\, Q) = \sum_x P(x) \log_2 \frac{P(x)}{Q(x)} \geq 0 \tag{1.9}$$

  is the Kullback–Leibler divergence from Q to P

- Gibbs' inequality: $D_{\mathrm{KL}}(P \,\|\, Q) \geq 0$ with equality if and only if $P = Q$. Therefore:

$$H(P, Q) \geq H(P)$$

  The cross-entropy of any model Q is lower-bounded by the true entropy $H(P)$

Cross-Entropy and KL Divergence

- The cross-entropy (1.6) decomposes as:

$$H(P, Q) = H(P) + D_{\mathrm{KL}}(P \,\|\, Q) \tag{1.8}$$

  where $H(P) = -\sum_x P(x) \log_2 P(x)$ is the entropy of the true distribution and

$$D_{\mathrm{KL}}(P \,\|\, Q) = \sum_x P(x) \log_2 \frac{P(x)}{Q(x)} \geq 0 \tag{1.9}$$

  is the Kullback–Leibler divergence from Q to P

- Gibbs' inequality: $D_{\mathrm{KL}}(P \,\|\, Q) \geq 0$ with equality if and only if $P = Q$. Therefore:

$$H(P, Q) \geq H(P)$$

  The cross-entropy of any model Q is lower-bounded by the true entropy $H(P)$

- Consequence: Minimising the cross-entropy loss $H(P, Q)$ over Q is equivalent to minimising $D_{\mathrm{KL}}(P \,\|\, Q)$, i.e. finding the model closest to the true distribution in KL sense

Cross-Entropy and KL Divergence

- The cross-entropy (1.6) decomposes as:

$$H(P, Q) = H(P) + D_{\mathrm{KL}}(P \,\|\, Q) \tag{1.8}$$

  where $H(P) = -\sum_x P(x) \log_2 P(x)$ is the entropy of the true distribution and

$$D_{\mathrm{KL}}(P \,\|\, Q) = \sum_x P(x) \log_2 \frac{P(x)}{Q(x)} \geq 0 \tag{1.9}$$

  is the Kullback–Leibler divergence from Q to P

- Gibbs' inequality: $D_{\mathrm{KL}}(P \,\|\, Q) \geq 0$ with equality if and only if $P = Q$. Therefore:

$$H(P, Q) \geq H(P)$$

  The cross-entropy of any model Q is lower-bounded by the true entropy $H(P)$

- Consequence: Minimising the cross-entropy loss $H(P, Q)$ over Q is equivalent to minimising $D_{\mathrm{KL}}(P \,\|\, Q)$, i.e. finding the model closest to the true distribution in KL sense

- For perplexity: Since $\mathrm{PP}(Q) = 2^{H(P,Q)} \geq 2^{H(P)}$, no model can achieve perplexity below $2^{H(P)}$, the intrinsic perplexity of the language

Perplexity: Worked Example

- Setup: Vocabulary $\mathcal{V} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, test sequence $(\mathtt{a}, \mathtt{b}, \mathtt{a})$, $T = 3$

Perplexity: Worked Example

- Setup: Vocabulary $\mathcal{V} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, test sequence $(\mathtt{a}, \mathtt{b}, \mathtt{a})$, $T = 3$
- Suppose the model assigns the following conditional probabilities:

$$Q(\mathtt{a} \mid \langle \mathsf{start} \rangle) = 0.7, \quad Q(\mathtt{b} \mid \mathtt{a}) = 0.5, \quad Q(\mathtt{a} \mid \mathtt{a}, \mathtt{b}) = 0.6$$

Perplexity: Worked Example

- Setup: Vocabulary $\mathcal{V} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, test sequence $(\mathtt{a}, \mathtt{b}, \mathtt{a})$, $T = 3$
- Suppose the model assigns the following conditional probabilities:

$$Q(\mathtt{a} \mid \langle \mathrm{start} \rangle) = 0.7, \quad Q(\mathtt{b} \mid \mathtt{a}) = 0.5, \quad Q(\mathtt{a} \mid \mathtt{a}, \mathtt{b}) = 0.6$$

- Per-token cross-entropy (using $\log_2$):

$$\begin{aligned} \hat{H} &= -\frac{1}{3}\big[\log_2(0.7) + \log_2(0.5) + \log_2(0.6)\big] \\ &= -\frac{1}{3}\big[-0.515 + (-1.000) + (-0.737)\big] = \frac{2.252}{3} = 0.751 \text{ bits} \end{aligned}$$

Perplexity: Worked Example

- Setup: Vocabulary $\mathcal{V} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, test sequence $(\mathrm{a}, \mathrm{b}, \mathrm{a})$, $T = 3$
- Suppose the model assigns the following conditional probabilities:

$$Q(\mathrm{a} \mid \langle \mathrm{start} \rangle) = 0.7, \quad Q(\mathrm{b} \mid \mathrm{a}) = 0.5, \quad Q(\mathrm{a} \mid \mathrm{a}, \mathrm{b}) = 0.6$$

- Per-token cross-entropy (using $\log_2$):

$$\begin{aligned} \hat{H} &= -\frac{1}{3}\big[\log_2(0.7) + \log_2(0.5) + \log_2(0.6)\big] \\ &= -\frac{1}{3}\big[-0.515 + (-1.000) + (-0.737)\big] = \frac{2.252}{3} = 0.751 \text{ bits} \end{aligned}$$

- Perplexity:

$$\mathrm{PP} = 2^{0.751} \approx 1.68$$

Perplexity: Worked Example

- Setup: Vocabulary $\mathcal{V} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, test sequence $(\mathtt{a}, \mathtt{b}, \mathtt{a})$, $T = 3$
- Suppose the model assigns the following conditional probabilities:

$$Q(\mathtt{a} \mid \langle \mathrm{start} \rangle) = 0.7, \quad Q(\mathtt{b} \mid \mathtt{a}) = 0.5, \quad Q(\mathtt{a} \mid \mathtt{a}, \mathtt{b}) = 0.6$$

- Per-token cross-entropy (using $\log_2$):

$$\begin{aligned} \hat{H} &= -\frac{1}{3}\big[\log_2(0.7) + \log_2(0.5) + \log_2(0.6)\big] \\ &= -\frac{1}{3}\big[-0.515 + (-1.000) + (-0.737)\big] = \frac{2.252}{3} = 0.751 \text{ bits} \end{aligned}$$

- Perplexity:

$$\mathrm{PP} = 2^{0.751} \approx 1.68$$

- Interpretation: On average, the model is as uncertain as choosing uniformly among $\sim$1.68 tokens — close to 1 (the best possible), far below 3 (random guessing over $|\mathcal{V}| = 3$)

Perplexity: Worked Example

- Setup: Vocabulary $\mathcal{V} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, test sequence $(\mathrm{a}, \mathrm{b}, \mathrm{a})$, $T = 3$
- Suppose the model assigns the following conditional probabilities:

$$Q(\mathrm{a} \mid \langle \mathrm{start} \rangle) = 0.7, \quad Q(\mathrm{b} \mid \mathrm{a}) = 0.5, \quad Q(\mathrm{a} \mid \mathrm{a}, \mathrm{b}) = 0.6$$

- Per-token cross-entropy (using $\log_2$):

$$\begin{aligned} \hat{H} &= -\frac{1}{3}\big[\log_2(0.7) + \log_2(0.5) + \log_2(0.6)\big] \\ &= -\frac{1}{3}\big[-0.515 + (-1.000) + (-0.737)\big] = \frac{2.252}{3} = 0.751 \text{ bits} \end{aligned}$$

- Perplexity:

$$\mathrm{PP} = 2^{0.751} \approx 1.68$$

- Interpretation: On average, the model is as uncertain as choosing uniformly among $\sim$1.68 tokens — close to 1 (the best possible), far below 3 (random guessing over $|\mathcal{V}| = 3$)
- Comparison: A uniform model assigns $Q(\cdot) = 1/3$ to every token, giving $\hat{H} = \log_2 3 \approx 1.585$ bits and $\mathrm{PP} = 3$

Neural Language Models: Motivation

- Recall the *n*-gram limitation: the conditional distribution is estimated by counting occurrences of specific token sequences (1.3). Two contexts that are semantically similar but lexically different (e.g. "the cat sat on the" vs. "the dog sat on the") share *no* statistical strength

Neural Language Models: Motivation

- Recall the *n*-gram limitation: the conditional distribution is estimated by counting occurrences of specific token sequences (1.3). Two contexts that are semantically similar but lexically different (e.g. "the cat sat on the" vs. "the dog sat on the") share *no* statistical strength
- Key idea (Bengio et al., 2003): represent each token as a *continuous vector* (an embedding) and use a neural network to predict the next token from the context embeddings

Neural Language Models: Motivation

- Recall the *n*-gram limitation: the conditional distribution is estimated by counting occurrences of specific token sequences (1.3). Two contexts that are semantically similar but lexically different (e.g. "the cat sat on the" vs. "the dog sat on the") share *no* statistical strength
- Key idea (Bengio et al., 2003): represent each token as a *continuous vector* (an embedding) and use a neural network to predict the next token from the context embeddings
- If "cat" and "dog" have similar embedding vectors, the model automatically generalises from one context to the other — this is impossible with discrete counts

Neural Language Models: Motivation

- Recall the *n*-gram limitation: the conditional distribution is estimated by counting occurrences of specific token sequences (1.3). Two contexts that are semantically similar but lexically different (e.g. "the cat sat on the" vs. "the dog sat on the") share *no* statistical strength
- Key idea (Bengio et al., 2003): represent each token as a *continuous vector* (an embedding) and use a neural network to predict the next token from the context embeddings
- If "cat" and "dog" have similar embedding vectors, the model automatically generalises from one context to the other — this is impossible with discrete counts
- Formally, replace the counting-based estimation with a neural network P_θ parameterised by θ:

$$P_\theta(x_t \mid x_1, \ldots, x_{t-1}) = \mathrm{softmax}(f_\theta(x_1, \ldots, x_{t-1}))_{x_t}$$

where $f_\theta : \mathcal{V}^* \to \mathbb{R}^{|\mathcal{V}|}$ maps a context to a vector of logits (unnormalised log-probabilities)

Neural Language Models: Motivation

- Recall the n-gram limitation: the conditional distribution is estimated by counting occurrences of specific token sequences (1.3). Two contexts that are semantically similar but lexically different (e.g. "the cat sat on the" vs. "the dog sat on the") share *no* statistical strength
- Key idea (Bengio et al., 2003): represent each token as a *continuous vector* (an embedding) and use a neural network to predict the next token from the context embeddings
- If "cat" and "dog" have similar embedding vectors, the model automatically generalises from one context to the other — this is impossible with discrete counts
- Formally, replace the counting-based estimation with a neural network P_θ parameterised by θ:

$$P_\theta(x_t \mid x_1, \ldots, x_{t-1}) = \mathrm{softmax}(f_\theta(x_1, \ldots, x_{t-1}))_{x_t}$$

  where $f_\theta : \mathcal{V}^* \to \mathbb{R}^{|\mathcal{V}|}$ maps a context to a vector of logits (unnormalised log-probabilities)
- The softmax function converts logits $z \in \mathbb{R}^{|\mathcal{V}|}$ to a valid probability distribution:

$$\mathrm{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{|\mathcal{V}|} e^{z_j}}, \quad i = 1, \ldots, |\mathcal{V}| \tag{1.10}$$

Neural Language Models: Training and Advantages

- Training objective: Given a training corpus $(x_1, \ldots, x_N)$, find parameters θ that maximise the likelihood of the observed text. Equivalently, minimise the cross-entropy loss:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{t=1}^{N} \log P_\theta(x_t \mid x_1, \ldots, x_{t-1}) \tag{1.11}$$

Neural Language Models: Training and Advantages

- Training objective: Given a training corpus $(x_1, \ldots, x_N)$, find parameters θ that maximise the likelihood of the observed text. Equivalently, minimise the cross-entropy loss:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{t=1}^{N} \log P_\theta(x_t \mid x_1, \ldots, x_{t-1}) \tag{1.11}$$

- Connection to evaluation: The training loss is exactly the per-token cross-entropy (1.5) (using ln instead of $\log_2$). So training directly optimises the quantity we use to evaluate the model

Neural Language Models: Training and Advantages

- Training objective: Given a training corpus $(x_1, \ldots, x_N)$, find parameters θ that maximise the likelihood of the observed text. Equivalently, minimise the cross-entropy loss:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{t=1}^{N} \log P_\theta(x_t \mid x_1, \ldots, x_{t-1}) \tag{1.11}$$

- Connection to evaluation: The training loss is exactly the per-token cross-entropy (1.5) (using ln instead of $\log_2$). So training directly optimises the quantity we use to evaluate the model
- Advantages over *n*-grams:
  - Generalisation: An *n*-gram model that has seen "the *cat* sat on the *mat*" knows nothing about "the *dog* sat on the *mat*" — these are entirely different 6-grams. A neural model maps "cat" and "dog" to nearby vectors in $\mathbb{R}^d$, so a prediction learned from one context automatically transfers to the other
  - Scalability: The number of parameters grows with the network size, *not* exponentially with the context length
  - Flexible context: The context need not be truncated to a fixed n; the architecture determines how much history is used (feedforward, RNN, Transformer, etc.)

Neural Language Models: Training and Advantages

- Training objective: Given a training corpus $(x_1, \ldots, x_N)$, find parameters θ that maximise the likelihood of the observed text. Equivalently, minimise the cross-entropy loss:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{t=1}^{N} \log P_\theta(x_t \mid x_1, \ldots, x_{t-1}) \tag{1.11}$$

- Connection to evaluation: The training loss is exactly the per-token cross-entropy (1.5) (using ln instead of $\log_2$). So training directly optimises the quantity we use to evaluate the model
- Advantages over *n*-grams:
  - Generalisation: An *n*-gram model that has seen "the *cat* sat on the *mat*" knows nothing about "the *dog* sat on the *mat*" — these are entirely different 6-grams. A neural model maps "cat" and "dog" to nearby vectors in $\mathbb{R}^d$, so a prediction learned from one context automatically transfers to the other
  - Scalability: The number of parameters grows with the network size, *not* exponentially with the context length
  - Flexible context: The context need not be truncated to a fixed n; the architecture determines how much history is used (feedforward, RNN, Transformer, etc.)
- The rest of this lecture: What architecture should f_θ be? We will progress from feedforward networks $\rightarrow$ RNNs $\rightarrow$ Transformers

Part 2: Neural Networks as Function Approximators

Goal: Review the mathematical foundations of feedforward neural networks and their approximation-theoretic properties.

- Feedforward architecture

Part 2: Neural Networks as Function Approximators

Goal: Review the mathematical foundations of feedforward neural networks and their approximation-theoretic properties.

- Feedforward architecture
- Activation functions

Part 2: Neural Networks as Function Approximators

Goal: Review the mathematical foundations of feedforward neural networks and their approximation-theoretic properties.

- Feedforward architecture
- Activation functions
- Universal Approximation Theorem

Part 2: Neural Networks as Function Approximators

Goal: Review the mathematical foundations of feedforward neural networks and their approximation-theoretic properties.

- Feedforward architecture
- Activation functions
- Universal Approximation Theorem
- Depth efficiency

Part 2: Neural Networks as Function Approximators

Goal: Review the mathematical foundations of feedforward neural networks and their approximation-theoretic properties.

- Feedforward architecture
- Activation functions
- Universal Approximation Theorem
- Depth efficiency
- SGD and backpropagation

Part 2: Neural Networks as Function Approximators

Goal: Review the mathematical foundations of feedforward neural networks and their approximation-theoretic properties.

- Feedforward architecture
- Activation functions
- Universal Approximation Theorem
- Depth efficiency
- SGD and backpropagation
- Generalisation and double descent

Feedforward Neural Networks

Definition 1.1 (Feedforward Neural Network)

A feedforward neural network (FNN) with L layers is a function $f : \mathbb{R}^{d_0} \to \mathbb{R}^{d_L}$ defined as:

$$f(\mathbf{x}) = W_L \sigma(W_{L-1} \sigma(\cdots \sigma(W_1 \mathbf{x} + \mathbf{b}_1) \cdots) + \mathbf{b}_{L-1}) + \mathbf{b}_L \tag{1.12}$$

where:

- $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ *are weight matrices,* $\mathbf{b}_\ell \in \mathbb{R}^{d_\ell}$ *are bias vectors*
- $\sigma : \mathbb{R} \to \mathbb{R}$ *is a nonlinear activation function applied element-wise*
- d_0 *is the input dimension,* d_L *is the output dimension*
- $d_1, \ldots, d_{L-1}$ *are the hidden layer widths*

Feedforward Neural Networks

Definition 1.1 (Feedforward Neural Network)

A feedforward neural network (FNN) with L layers is a function $f : \mathbb{R}^{d_0} \to \mathbb{R}^{d_L}$ defined as:

$$f(\mathbf{x}) = W_L \sigma(W_{L-1}\sigma(\cdots \sigma(W_1 \mathbf{x} + \mathbf{b}_1) \cdots) + \mathbf{b}_{L-1}) + \mathbf{b}_L \tag{1.12}$$

where:

- $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ *are weight matrices*, $\mathbf{b}_\ell \in \mathbb{R}^{d_\ell}$ *are bias vectors*
- $\sigma : \mathbb{R} \to \mathbb{R}$ *is a nonlinear activation function applied element-wise*
- d_0 *is the input dimension,* d_L *is the output dimension*
- $d_1, \ldots, d_{L-1}$ *are the hidden layer widths*

- The parameters $\theta = \{(W_\ell, \mathbf{b}_\ell)\}_{\ell=1}^{L}$ are learned from data
- Depth = number of layers L; Width = $\max_\ell d_\ell$
- Total number of parameters: $\sum_{\ell=1}^{L}(d_\ell \cdot d_{\ell-1} + d_\ell)$

Activation Functions

- Without nonlinear activations, a composition of affine maps is just another affine map — the network would have no more expressive power than linear regression

Activation Functions

- Without nonlinear activations, a composition of affine maps is just another affine map — the network would have no more expressive power than linear regression
- Common choices of activation function σ:

Activation Functions

- Without nonlinear activations, a composition of affine maps is just another affine map — the network would have no more expressive power than linear regression
- Common choices of activation function σ:
  - Sigmoid: $\sigma(z) = \dfrac{1}{1+e^{-z}} \in (0,1)$
  - Hyperbolic tangent: $\sigma(z) = \tanh(z) = \dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \in (-1,1)$
  - ReLU (Rectified Linear Unit): $\sigma(z) = \max(0,z)$

Activation Functions

- Without nonlinear activations, a composition of affine maps is just another affine map — the network would have no more expressive power than linear regression
- Common choices of activation function σ:
  - Sigmoid: $\sigma(z) = \dfrac{1}{1+e^{-z}} \in (0,1)$
  - Hyperbolic tangent: $\sigma(z) = \tanh(z) = \dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \in (-1,1)$
  - ReLU (Rectified Linear Unit): $\sigma(z) = \max(0, z)$
- ReLU is the most widely used in practice — we explain why on the next slides

The Problem with Sigmoid and Tanh

- Saturation: For large $|z|$, the derivatives of sigmoid and tanh *flatten out*:
  - Sigmoid: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. As $z \to \pm\infty$, $\sigma(z) \to 0$ or 1, so $\sigma'(z) \to 0$. The maximum is $\sigma'(0) = 1/4$
  - Tanh: $\tanh'(z) = 1 - \tanh^2(z)$. As $z \to \pm\infty$, $\tanh(z) \to \pm 1$, so $\tanh'(z) \to 0$. The maximum is $\tanh'(0) = 1$

  In both cases, the derivative is close to zero except near $z = 0$. Any input with large magnitude produces a near-zero gradient — the activation is saturated

The Problem with Sigmoid and Tanh

- Saturation: For large $|z|$, the derivatives of sigmoid and tanh *flatten out*:
  - Sigmoid: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. As $z \to \pm\infty$, $\sigma(z) \to 0$ or 1, so $\sigma'(z) \to 0$. The maximum is $\sigma'(0) = 1/4$
  - Tanh: $\tanh'(z) = 1 - \tanh^2(z)$. As $z \to \pm\infty$, $\tanh(z) \to \pm 1$, so $\tanh'(z) \to 0$. The maximum is $\tanh'(0) = 1$

  In both cases, the derivative is close to zero except near $z = 0$. Any input with large magnitude produces a near-zero gradient — the activation is saturated
- Vanishing gradients: During backpropagation, the gradient of the loss with respect to early layers involves a *product* of activation derivatives $\sigma'(z)$ across layers. For sigmoid:
  - Each layer multiplies the gradient by a factor $\leq 1/4$
  - After L layers: gradient is scaled by $\leq (1/4)^L$
  - For $L = 10$: $(1/4)^{10} \approx 10^{-6}$ — the gradient effectively vanishes

The Problem with Sigmoid and Tanh

- Saturation: For large $|z|$, the derivatives of sigmoid and tanh *flatten out*:
  - Sigmoid: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. As $z \to \pm\infty$, $\sigma(z) \to 0$ or 1, so $\sigma'(z) \to 0$. The maximum is $\sigma'(0) = 1/4$
  - Tanh: $\tanh'(z) = 1 - \tanh^2(z)$. As $z \to \pm\infty$, $\tanh(z) \to \pm 1$, so $\tanh'(z) \to 0$. The maximum is $\tanh'(0) = 1$

  In both cases, the derivative is close to zero except near $z = 0$. Any input with large magnitude produces a near-zero gradient — the activation is saturated
- Vanishing gradients: During backpropagation, the gradient of the loss with respect to early layers involves a *product* of activation derivatives $\sigma'(z)$ across layers. For sigmoid:
  - Each layer multiplies the gradient by a factor $\leq 1/4$
  - After L layers: gradient is scaled by $\leq (1/4)^L$
  - For $L = 10$: $(1/4)^{10} \approx 10^{-6}$ — the gradient effectively vanishes
- Output range also matters:
  - Sigmoid outputs are in $(0, 1)$ — always positive, which can cause systematic bias in downstream layers (all activations shift in one direction)
  - Tanh outputs are in $(-1, 1)$ — centred around zero, which is generally better for optimisation. This is why tanh was preferred over sigmoid in RNNs

ReLU: Advantages and Limitations

- ReLU derivative: $\sigma'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases}$
  - For active neurons ($z > 0$), the gradient passes through *unchanged* (multiplied by 1) — no saturation, no vanishing gradients
  - No attenuation across layers — enables training of much deeper networks

ReLU: Advantages and Limitations

- ReLU derivative: $\sigma'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases}$
  - For active neurons ($z > 0$), the gradient passes through *unchanged* (multiplied by 1) — no saturation, no vanishing gradients
  - No attenuation across layers — enables training of much deeper networks
- Non-differentiability at $z = 0$: ReLU has a kink at the origin. In practice this is not a problem:
  - The set $\{z = 0\}$ has measure zero — it is encountered with probability 0 for continuous-valued pre-activations
  - Implementations simply define $\sigma'(0) = 0$ (or 1); SGD is robust to such choices

ReLU: Advantages and Limitations

- ReLU derivative: $\sigma'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases}$
  - For active neurons ($z > 0$), the gradient passes through *unchanged* (multiplied by 1) — no saturation, no vanishing gradients
  - No attenuation across layers — enables training of much deeper networks
- Non-differentiability at $z = 0$: ReLU has a kink at the origin. In practice this is not a problem:
  - The set $\{z = 0\}$ has measure zero — it is encountered with probability 0 for continuous-valued pre-activations
  - Implementations simply define $\sigma'(0) = 0$ (or 1); SGD is robust to such choices
- Computational efficiency: Only a comparison and a max — no exponentials

ReLU: Advantages and Limitations

- ReLU derivative: $\sigma'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases}$
  - For active neurons ($z > 0$), the gradient passes through *unchanged* (multiplied by 1) — no saturation, no vanishing gradients
  - No attenuation across layers — enables training of much deeper networks
- Non-differentiability at $z = 0$: ReLU has a kink at the origin. In practice this is not a problem:
  - The set $\{z = 0\}$ has measure zero — it is encountered with probability 0 for continuous-valued pre-activations
  - Implementations simply define $\sigma'(0) = 0$ (or 1); SGD is robust to such choices
- Computational efficiency: Only a comparison and a max — no exponentials
- Negative inputs are killed: For $z < 0$, both the output and the gradient are exactly 0. This is a double-edged sword:
  - Advantage (sparsity): Many neurons output 0, leading to sparse representations that can be exploited for computational efficiency
  - Disadvantage ("dying ReLU"): If a neuron's pre-activation z is always negative (e.g. due to a large negative bias or an unlucky weight update), its gradient is permanently 0 and it stops learning entirely

Modern Activation Functions

- The dying ReLU problem and the hard zero for negative inputs motivate smooth variants that maintain a non-zero gradient everywhere:

Modern Activation Functions

- The dying ReLU problem and the hard zero for negative inputs motivate smooth variants that maintain a non-zero gradient everywhere:
- Leaky ReLU (Maas et al., 2013):

$$\sigma(z) = \max(\alpha z,\, z), \quad \text{typically } \alpha = 0.01$$

For $z < 0$, the gradient is α instead of 0 — the neuron never fully dies. Simple but the negative slope α is a fixed hyperparameter

Modern Activation Functions

- The dying ReLU problem and the hard zero for negative inputs motivate smooth variants that maintain a non-zero gradient everywhere:
- Leaky ReLU (Maas et al., 2013):

$$\sigma(z) = \max(\alpha z,\, z), \quad \text{typically } \alpha = 0.01$$

For $z < 0$, the gradient is α instead of 0 — the neuron never fully dies. Simple but the negative slope α is a fixed hyperparameter

- GELU (Hendrycks & Gimpel, 2016):

$$\sigma(z) = z \cdot \Phi(z), \quad \text{where } \Phi \text{ is the standard normal CDF}$$

Smooth approximation of ReLU that "softly" gates the input by its own magnitude. Used in GPT-2, GPT-3, BERT

Modern Activation Functions

- The dying ReLU problem and the hard zero for negative inputs motivate smooth variants that maintain a non-zero gradient everywhere:
- Leaky ReLU (Maas et al., 2013):

$$\sigma(z) = \max(\alpha z,\, z), \quad \text{typically } \alpha = 0.01$$

  For $z < 0$, the gradient is α instead of 0 — the neuron never fully dies. Simple but the negative slope α is a fixed hyperparameter
- GELU (Hendrycks & Gimpel, 2016):

$$\sigma(z) = z \cdot \Phi(z), \quad \text{where } \Phi \text{ is the standard normal CDF}$$

  Smooth approximation of ReLU that "softly" gates the input by its own magnitude. Used in GPT-2, GPT-3, BERT
- SiLU / Swish (Ramachandran et al., 2017):

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

  Similar to GELU but uses the sigmoid instead of Φ. Smooth, non-monotone (slightly negative for $z \lesssim -1$). Used in LLaMA, PaLM, and most modern LLMs (inside the SwiGLU feedforward layer)

Modern Activation Functions

- The dying ReLU problem and the hard zero for negative inputs motivate smooth variants that maintain a non-zero gradient everywhere:
- Leaky ReLU (Maas et al., 2013):

$$\sigma(z) = \max(\alpha z,\, z), \quad \text{typically } \alpha = 0.01$$

  For $z < 0$, the gradient is α instead of 0 — the neuron never fully dies. Simple but the negative slope α is a fixed hyperparameter
- GELU (Hendrycks & Gimpel, 2016):

$$\sigma(z) = z \cdot \Phi(z), \quad \text{where } \Phi \text{ is the standard normal CDF}$$

  Smooth approximation of ReLU that "softly" gates the input by its own magnitude. Used in GPT-2, GPT-3, BERT
- SiLU / Swish (Ramachandran et al., 2017):

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

  Similar to GELU but uses the sigmoid instead of Φ. Smooth, non-monotone (slightly negative for $z \lesssim -1$). Used in LLaMA, PaLM, and most modern LLMs (inside the SwiGLU feedforward layer)
- Empirical finding: GELU and SiLU consistently outperform ReLU in large-scale Transformer training, likely because the smooth gradients improve optimisation dynamics

Universal Approximation Theorem

Theorem 1.2 (Universal Approximation — Cybenko 1989, Hornik 1991)

Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, non-constant, bounded activation function (e.g. sigmoid). Let $K \subset \mathbb{R}^d$ be compact and let $f \in C(K, \mathbb{R})$. Then for every $\varepsilon > 0$, there exist $N \in \mathbb{N}$, weights $W \in \mathbb{R}^{N \times d}$, $\mathbf{v} \in \mathbb{R}^N$, and biases $\mathbf{b} \in \mathbb{R}^N$ such that

$$\sup_{\mathbf{x} \in K} \left| f(\mathbf{x}) - \sum_{i=1}^{N} v_i \, \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i) \right| < \varepsilon \tag{1.13}$$

where $\mathbf{w}_i^\top$ is the i-th row of W.

Universal Approximation Theorem

Theorem 1.2 (Universal Approximation — Cybenko 1989, Hornik 1991)

Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, non-constant, bounded activation function (e.g. sigmoid). Let $K \subset \mathbb{R}^d$ be compact and let $f \in C(K, \mathbb{R})$. Then for every $\varepsilon > 0$, there exist $N \in \mathbb{N}$, weights $W \in \mathbb{R}^{N \times d}$, $\mathbf{v} \in \mathbb{R}^N$, and biases $\mathbf{b} \in \mathbb{R}^N$ such that

$$\sup_{\mathbf{x} \in K} \left| f(\mathbf{x}) - \sum_{i=1}^{N} v_i \, \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i) \right| < \varepsilon \tag{1.13}$$

where $\mathbf{w}_i^\top$ is the i-th row of W.

- A single hidden layer network with N sufficiently large neurons can approximate any continuous function on a compact set to arbitrary accuracy

Universal Approximation Theorem

Theorem 1.2 (Universal Approximation — Cybenko 1989, Hornik 1991)

Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, non-constant, bounded activation function (e.g. sigmoid). Let $K \subset \mathbb{R}^d$ be compact and let $f \in C(K, \mathbb{R})$. Then for every $\varepsilon > 0$, there exist $N \in \mathbb{N}$, weights $W \in \mathbb{R}^{N \times d}$, $\mathbf{v} \in \mathbb{R}^N$, and biases $\mathbf{b} \in \mathbb{R}^N$ such that

$$\sup_{\mathbf{x} \in K} \left| f(\mathbf{x}) - \sum_{i=1}^{N} v_i \, \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i) \right| < \varepsilon \tag{1.13}$$

where $\mathbf{w}_i^\top$ is the i-th row of W.

- A single hidden layer network with N sufficiently large neurons can approximate any continuous function on a compact set to arbitrary accuracy
- Caveat: The theorem is an *existence* result — it says nothing about:
  - How large N needs to be (it may be exponentially large in d)
  - Whether gradient-based training will find the approximating parameters

Depth Efficiency

- The Universal Approximation Theorem concerns *shallow* (single hidden layer) networks. A natural question: does depth help?

Depth Efficiency

- The Universal Approximation Theorem concerns *shallow* (single hidden layer) networks. A natural question: does depth help?
- Answer: Yes. Deep networks can be exponentially more parameter-efficient than shallow ones for certain function classes

Depth Efficiency

- The Universal Approximation Theorem concerns *shallow* (single hidden layer) networks. A natural question: does depth help?
- Answer: Yes. Deep networks can be exponentially more parameter-efficient than shallow ones for certain function classes
- Informally, there exist functions computable by a network of depth L with polynomial width, but any network of depth $L-1$ computing the same function requires exponential width

Depth Efficiency

- The Universal Approximation Theorem concerns *shallow* (single hidden layer) networks. A natural question: does depth help?
- Answer: Yes. Deep networks can be exponentially more parameter-efficient than shallow ones for certain function classes
- Informally, there exist functions computable by a network of depth L with polynomial width, but any network of depth $L-1$ computing the same function requires exponential width
- Intuition: Deep networks compose features hierarchically
  - Layer 1: detect simple patterns (edges, basic motifs)
  - Layer 2: combine simple patterns into intermediate features
  - Layer L: combine all into complex, high-level representations

Depth Efficiency

- The Universal Approximation Theorem concerns *shallow* (single hidden layer) networks. A natural question: does depth help?
- Answer: Yes. Deep networks can be exponentially more parameter-efficient than shallow ones for certain function classes
- Informally, there exist functions computable by a network of depth L with polynomial width, but any network of depth $L-1$ computing the same function requires exponential width
- Intuition: Deep networks compose features hierarchically
  - Layer 1: detect simple patterns (edges, basic motifs)
  - Layer 2: combine simple patterns into intermediate features
  - Layer L: combine all into complex, high-level representations
- This provides a theoretical justification for why modern architectures (including Transformers) use many layers rather than a single wide layer

Depth Efficiency

- The Universal Approximation Theorem concerns *shallow* (single hidden layer) networks. A natural question: does depth help?
- Answer: Yes. Deep networks can be exponentially more parameter-efficient than shallow ones for certain function classes
- Informally, there exist functions computable by a network of depth L with polynomial width, but any network of depth $L-1$ computing the same function requires exponential width
- Intuition: Deep networks compose features hierarchically
  - Layer 1: detect simple patterns (edges, basic motifs)
  - Layer 2: combine simple patterns into intermediate features
  - Layer L: combine all into complex, high-level representations
- This provides a theoretical justification for why modern architectures (including Transformers) use many layers rather than a single wide layer
- Telgarsky (2016): For ReLU networks, there exist functions representable with $O(L)$ parameters at depth L that require $\Omega(2^{L/2})$ parameters at depth $O(1)$

Stochastic Gradient Descent and Backpropagation

- Given a loss function $\mathcal{L}(\theta)$ (e.g. cross-entropy (1.11)), we seek $\theta^* = \arg\min_\theta \mathcal{L}(\theta)$

Stochastic Gradient Descent and Backpropagation

- Given a loss function $\mathcal{L}(\theta)$ (e.g. cross-entropy (1.11)), we seek $\theta^* = \arg\min_\theta \mathcal{L}(\theta)$
- Gradient descent (GD):

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta \mathcal{L}(\theta_k)$$

where $\eta > 0$ is the learning rate

Stochastic Gradient Descent and Backpropagation

- Given a loss function $\mathcal{L}(\theta)$ (e.g. cross-entropy (1.11)), we seek $\theta^* = \arg\min_\theta \mathcal{L}(\theta)$
- Gradient descent (GD):

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta \mathcal{L}(\theta_k)$$

  where $\eta > 0$ is the learning rate
- Stochastic gradient descent (SGD): Replace full gradient with an estimate from a random mini-batch $\mathcal{B} \subset \{1, \ldots, N\}$:

$$\theta_{k+1} = \theta_k - \eta \cdot \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell_i(\theta_k)$$

Stochastic Gradient Descent and Backpropagation

- Given a loss function $\mathcal{L}(\theta)$ (e.g. cross-entropy (1.11)), we seek $\theta^* = \arg\min_\theta \mathcal{L}(\theta)$
- Gradient descent (GD):

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta \mathcal{L}(\theta_k)$$

  where $\eta > 0$ is the learning rate
- Stochastic gradient descent (SGD): Replace full gradient with an estimate from a random mini-batch $\mathcal{B} \subset \{1, \ldots, N\}$:

$$\theta_{k+1} = \theta_k - \eta \cdot \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell_i(\theta_k)$$

- Backpropagation (Rumelhart et al., 1986): An efficient algorithm to compute $\nabla_\theta \mathcal{L}$ by applying the chain rule layer by layer, from output to input

Stochastic Gradient Descent and Backpropagation

- Given a loss function $\mathcal{L}(\theta)$ (e.g. cross-entropy (1.11)), we seek $\theta^* = \arg\min_\theta \mathcal{L}(\theta)$
- Gradient descent (GD):

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta \mathcal{L}(\theta_k)$$

  where $\eta > 0$ is the learning rate
- Stochastic gradient descent (SGD): Replace full gradient with an estimate from a random mini-batch $\mathcal{B} \subset \{1, \ldots, N\}$:

$$\theta_{k+1} = \theta_k - \eta \cdot \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell_i(\theta_k)$$

- Backpropagation (Rumelhart et al., 1986): An efficient algorithm to compute $\nabla_\theta \mathcal{L}$ by applying the chain rule layer by layer, from output to input
- Computational cost of backpropagation: $O(|\theta|)$ per sample — the same order as a single forward pass

Stochastic Gradient Descent and Backpropagation

- Given a loss function $\mathcal{L}(\theta)$ (e.g. cross-entropy (1.11)), we seek $\theta^* = \arg\min_\theta \mathcal{L}(\theta)$
- Gradient descent (GD):

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta \mathcal{L}(\theta_k)$$

  where $\eta > 0$ is the learning rate
- Stochastic gradient descent (SGD): Replace full gradient with an estimate from a random mini-batch $\mathcal{B} \subset \{1, \ldots, N\}$:

$$\theta_{k+1} = \theta_k - \eta \cdot \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell_i(\theta_k)$$

- Backpropagation (Rumelhart et al., 1986): An efficient algorithm to compute $\nabla_\theta \mathcal{L}$ by applying the chain rule layer by layer, from output to input
- Computational cost of backpropagation: $O(|\theta|)$ per sample — the same order as a single forward pass
- Modern optimisers: Adam (Kingma & Ba, 2015) and variants, which adaptively scale learning rates per parameter

Generalisation and the Double Descent Phenomenon

- Classical statistical learning theory: Test error $\approx$ training error $+$ complexity penalty. Overfitting occurs when the model is "too complex" relative to the training data

Generalisation and the Double Descent Phenomenon

- Classical statistical learning theory: Test error $\approx$ training error + complexity penalty. Overfitting occurs when the model is "too complex" relative to the training data
- Puzzle: Modern neural networks are heavily *overparameterised* (more parameters than training examples), yet they generalise well in practice

Generalisation and the Double Descent Phenomenon

- Classical statistical learning theory: Test error $\approx$ training error + complexity penalty. Overfitting occurs when the model is "too complex" relative to the training data
- Puzzle: Modern neural networks are heavily *overparameterised* (more parameters than training examples), yet they generalise well in practice
- Double descent (Belkin et al., 2019): As model complexity grows, test error follows a U-shaped curve (classical regime), but then *decreases again* beyond the interpolation threshold

Generalisation and the Double Descent Phenomenon

- Classical statistical learning theory: Test error $\approx$ training error + complexity penalty. Overfitting occurs when the model is "too complex" relative to the training data
- Puzzle: Modern neural networks are heavily *overparameterised* (more parameters than training examples), yet they generalise well in practice
- Double descent (Belkin et al., 2019): As model complexity grows, test error follows a U-shaped curve (classical regime), but then *decreases again* beyond the interpolation threshold
  1. Underparameterised regime: Classical bias-variance tradeoff applies
  2. Interpolation threshold: Model just barely fits training data — worst generalisation
  3. Overparameterised regime: Among all interpolating solutions, SGD finds one with *low complexity* (implicit regularisation), leading to good generalisation

Generalisation and the Double Descent Phenomenon

- Classical statistical learning theory: Test error $\approx$ training error + complexity penalty. Overfitting occurs when the model is "too complex" relative to the training data
- Puzzle: Modern neural networks are heavily *overparameterised* (more parameters than training examples), yet they generalise well in practice
- Double descent (Belkin et al., 2019): As model complexity grows, test error follows a U-shaped curve (classical regime), but then *decreases again* beyond the interpolation threshold
  1. Underparameterised regime: Classical bias-variance tradeoff applies
  2. Interpolation threshold: Model just barely fits training data — worst generalisation
  3. Overparameterised regime: Among all interpolating solutions, SGD finds one with *low complexity* (implicit regularisation), leading to good generalisation
- Implications for LLMs: Modern LLMs have billions of parameters (far exceeding training examples in many senses), yet they generalise remarkably well — this is consistent with the overparameterised regime of double descent

Part 3: Recurrent Neural Networks

Goal: Introduce sequential architectures for language modelling and analyse their limitations.

- RNN architecture

Part 3: Recurrent Neural Networks

Goal: Introduce sequential architectures for language modelling and analyse their limitations.

- RNN architecture
- Backpropagation through time

Part 3: Recurrent Neural Networks

Goal: Introduce sequential architectures for language modelling and analyse their limitations.

- RNN architecture
- Backpropagation through time
- Vanishing and exploding gradients

Part 3: Recurrent Neural Networks

Goal: Introduce sequential architectures for language modelling and analyse their limitations.

- RNN architecture
- Backpropagation through time
- Vanishing and exploding gradients
- Long Short-Term Memory (LSTM)

Recurrent Neural Network (RNN) Architecture

Definition 1.3 (Elman RNN)

A recurrent neural network processes a sequence $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ *with* $\mathbf{x}_t \in \mathbb{R}^d$ *via the recurrence:*

$$\mathbf{h}_t = \sigma(W_h \mathbf{h}_{t-1} + W_x \mathbf{x}_t + \mathbf{b}_h) \tag{1.14}$$

$$\mathbf{y}_t = W_y \mathbf{h}_t + \mathbf{b}_y \tag{1.15}$$

where:

- $\mathbf{h}_t \in \mathbb{R}^{d_h}$ *is the hidden state at time* t*, with* $\mathbf{h}_0 = \mathbf{0}$
- $W_h \in \mathbb{R}^{d_h \times d_h}$, $W_x \in \mathbb{R}^{d_h \times d}$, $W_y \in \mathbb{R}^{d_y \times d_h}$ *are weight matrices*
- σ *is an element-wise activation (typically* $\tanh$*)*

Recurrent Neural Network (RNN) Architecture

Definition 1.3 (Elman RNN)

A recurrent neural network processes a sequence $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ with $\mathbf{x}_t \in \mathbb{R}^d$ via the recurrence:

$$\mathbf{h}_t = \sigma(W_h \mathbf{h}_{t-1} + W_x \mathbf{x}_t + \mathbf{b}_h) \tag{1.14}$$

$$\mathbf{y}_t = W_y \mathbf{h}_t + \mathbf{b}_y \tag{1.15}$$

where:

- $\mathbf{h}_t \in \mathbb{R}^{d_h}$ *is the hidden state at time* t*, with* $\mathbf{h}_0 = \mathbf{0}$
- $W_h \in \mathbb{R}^{d_h \times d_h}$, $W_x \in \mathbb{R}^{d_h \times d}$, $W_y \in \mathbb{R}^{d_y \times d_h}$ *are weight matrices*
- σ *is an element-wise activation (typically* tanh*)*

- The same parameters $(W_h, W_x, W_y, \mathbf{b}_h, \mathbf{b}_y)$ are shared across all time steps

Recurrent Neural Network (RNN) Architecture

Definition 1.3 (Elman RNN)

A recurrent neural network processes a sequence $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ *with* $\mathbf{x}_t \in \mathbb{R}^d$ *via the recurrence:*

$$\mathbf{h}_t = \sigma(W_h \mathbf{h}_{t-1} + W_x \mathbf{x}_t + \mathbf{b}_h) \tag{1.14}$$

$$\mathbf{y}_t = W_y \mathbf{h}_t + \mathbf{b}_y \tag{1.15}$$

where:

- $\mathbf{h}_t \in \mathbb{R}^{d_h}$ *is the hidden state at time* t*, with* $\mathbf{h}_0 = \mathbf{0}$
- $W_h \in \mathbb{R}^{d_h \times d_h}$, $W_x \in \mathbb{R}^{d_h \times d}$, $W_y \in \mathbb{R}^{d_y \times d_h}$ *are weight matrices*
- σ *is an element-wise activation (typically* $\tanh$*)*

- The same parameters $(W_h, W_x, W_y, \mathbf{b}_h, \mathbf{b}_y)$ are shared across all time steps
- The hidden state $\mathbf{h}_t$ serves as a compressed summary of all past inputs $(\mathbf{x}_1, \ldots, \mathbf{x}_t)$ — a form of memory

Recurrent Neural Network (RNN) Architecture

Definition 1.3 (Elman RNN)

A recurrent neural network processes a sequence $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ with $\mathbf{x}_t \in \mathbb{R}^d$ via the recurrence:

$$\mathbf{h}_t = \sigma(W_h \mathbf{h}_{t-1} + W_x \mathbf{x}_t + \mathbf{b}_h) \tag{1.14}$$

$$\mathbf{y}_t = W_y \mathbf{h}_t + \mathbf{b}_y \tag{1.15}$$

where:

- $\mathbf{h}_t \in \mathbb{R}^{d_h}$ *is the hidden state at time* t*, with* $\mathbf{h}_0 = \mathbf{0}$
- $W_h \in \mathbb{R}^{d_h \times d_h}$, $W_x \in \mathbb{R}^{d_h \times d}$, $W_y \in \mathbb{R}^{d_y \times d_h}$ *are weight matrices*
- σ *is an element-wise activation (typically* $\tanh$*)*

- The same parameters $(W_h, W_x, W_y, \mathbf{b}_h, \mathbf{b}_y)$ are shared across all time steps
- The hidden state $\mathbf{h}_t$ serves as a compressed summary of all past inputs $(\mathbf{x}_1, \ldots, \mathbf{x}_t)$ — a form of memory
- For language modelling: $\mathbf{y}_t$ produces logits, and $\mathrm{softmax}(\mathbf{y}_t)$ gives $P_\theta(x_{t+1} \mid x_1, \ldots, x_t)$

Backpropagation Through Time (BPTT)

- To train an RNN, we "unroll" the recurrence over T time steps and apply backpropagation to the resulting computational graph

Backpropagation Through Time (BPTT)

- To train an RNN, we "unroll" the recurrence over T time steps and apply backpropagation to the resulting computational graph
- The loss for a sequence of length T is:

$$\mathcal{L} = \sum_{t=1}^{T} \ell_t(\mathbf{y}_t)$$

Backpropagation Through Time (BPTT)

- To train an RNN, we "unroll" the recurrence over T time steps and apply backpropagation to the resulting computational graph
- The loss for a sequence of length T is:

$$\mathcal{L} = \sum_{t=1}^{T} \ell_t(\mathbf{y}_t)$$

- The gradient with respect to W_h involves a sum over time steps, each requiring a product of Jacobians:

$$\frac{\partial \mathcal{L}}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{y}_t} \frac{\partial \mathbf{y}_t}{\partial \mathbf{h}_t} \left(\sum_{k=1}^{t} \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \cdot \frac{\partial \mathbf{h}_k}{\partial W_h} \right) \tag{1.16}$$

Backpropagation Through Time (BPTT)

- To train an RNN, we "unroll" the recurrence over T time steps and apply backpropagation to the resulting computational graph
- The loss for a sequence of length T is:

$$\mathcal{L} = \sum_{t=1}^{T} \ell_t(\mathbf{y}_t)$$

- The gradient with respect to W_h involves a sum over time steps, each requiring a product of Jacobians:

$$\frac{\partial \mathcal{L}}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{y}_t} \frac{\partial \mathbf{y}_t}{\partial \mathbf{h}_t} \left(\sum_{k=1}^{t} \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \cdot \frac{\partial \mathbf{h}_k}{\partial W_h} \right) \tag{1.16}$$

- The key quantity is the Jacobian product:

$$\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} = \prod_{j=k+1}^{t} \operatorname{diag}(\sigma'(\mathbf{z}_j)) \cdot W_h \tag{1.17}$$

where $\mathbf{z}_j = W_h \mathbf{h}_{j-1} + W_x \mathbf{x}_j + \mathbf{b}_h$

Vanishing and Exploding Gradients

- The Jacobian product (1.17) involves $t - k$ matrix multiplications. As this product length grows:

Vanishing and Exploding Gradients

- The Jacobian product (1.17) involves $t - k$ matrix multiplications. As this product length grows:
  - If $\|W_h\|$ is "small": $\left\| \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \right\| \to 0$ exponentially fast
    $\Rightarrow$ vanishing gradients
  - If $\|W_h\|$ is "large": $\left\| \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \right\| \to \infty$ exponentially fast
    $\Rightarrow$ exploding gradients

Vanishing and Exploding Gradients

- The Jacobian product (1.17) involves $t - k$ matrix multiplications. As this product length grows:
  - If $\|W_h\|$ is "small": $\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \to 0$ exponentially fast
    $\Rightarrow$ vanishing gradients
  - If $\|W_h\|$ is "large": $\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \to \infty$ exponentially fast
    $\Rightarrow$ exploding gradients
- More precisely (Bengio et al., 1994): if λ_1 is the largest singular value of W_h and $\gamma = \max_z |\sigma'(z)|$, then:

$$\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \leq (\gamma \cdot \lambda_1)^{t-k}$$

Vanishing and Exploding Gradients

- The Jacobian product (1.17) involves $t - k$ matrix multiplications. As this product length grows:
  - If $\|W_h\|$ is "small": $\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \to 0$ exponentially fast
    $\Rightarrow$ vanishing gradients
  - If $\|W_h\|$ is "large": $\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \to \infty$ exponentially fast
    $\Rightarrow$ exploding gradients
- More precisely (Bengio et al., 1994): if λ_1 is the largest singular value of W_h and $\gamma = \max_z |\sigma'(z)|$, then:

$$\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \leq (\gamma \cdot \lambda_1)^{t-k}$$

- Consequence: Vanilla RNNs struggle to learn long-range dependencies — the gradient signal from distant time steps is lost

Vanishing and Exploding Gradients

- The Jacobian product (1.17) involves $t - k$ matrix multiplications. As this product length grows:
  - If $\|W_h\|$ is "small": $\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \to 0$ exponentially fast
    $\Rightarrow$ vanishing gradients
  - If $\|W_h\|$ is "large": $\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \to \infty$ exponentially fast
    $\Rightarrow$ exploding gradients
- More precisely (Bengio et al., 1994): if λ_1 is the largest singular value of W_h and $\gamma = \max_z |\sigma'(z)|$, then:

$$\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \leq (\gamma \cdot \lambda_1)^{t-k}$$

- Consequence: Vanilla RNNs struggle to learn long-range dependencies — the gradient signal from distant time steps is lost
- Exploding gradients can be mitigated by *gradient clipping*: $\mathbf{g} \leftarrow \mathbf{g} \cdot \min\left(1, \frac{c}{\|\mathbf{g}\|}\right)$ for some threshold $c > 0$

Vanishing and Exploding Gradients

- The Jacobian product (1.17) involves $t - k$ matrix multiplications. As this product length grows:
  - If $\|W_h\|$ is "small": $\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \to 0$ exponentially fast
    $\Rightarrow$ vanishing gradients
  - If $\|W_h\|$ is "large": $\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \to \infty$ exponentially fast
    $\Rightarrow$ exploding gradients
- More precisely (Bengio et al., 1994): if λ_1 is the largest singular value of W_h and $\gamma = \max_z |\sigma'(z)|$, then:

$$\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \leq (\gamma \cdot \lambda_1)^{t-k}$$

- Consequence: Vanilla RNNs struggle to learn long-range dependencies — the gradient signal from distant time steps is lost
- Exploding gradients can be mitigated by *gradient clipping*: $\mathbf{g} \leftarrow \mathbf{g} \cdot \min\left(1, \frac{c}{\|\mathbf{g}\|}\right)$ for some threshold $c > 0$
- Vanishing gradients require *architectural* solutions $\Rightarrow$ LSTM, GRU

Long Short-Term Memory (LSTM)

Definition 1.4 (LSTM — Hochreiter & Schmidhuber, 1997)

An LSTM cell maintains a cell state $\mathbf{c}_t$ and hidden state $\mathbf{h}_t$ via:

$$\mathbf{f}_t = \sigma_g(W_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \quad \textit{(forget gate)} \tag{1.18}$$

$$\mathbf{i}_t = \sigma_g(W_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \quad \textit{(input gate)} \tag{1.19}$$

$$\tilde{\mathbf{c}}_t = \tanh(W_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c) \quad \textit{(candidate cell state)} \tag{1.20}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \quad \textit{(cell state update)} \tag{1.21}$$

$$\mathbf{o}_t = \sigma_g(W_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \quad \textit{(output gate)} \tag{1.22}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad \textit{(hidden state)} \tag{1.23}$$

where σ_g is the sigmoid function and $\odot$ denotes element-wise (Hadamard) product.

LSTM: Why It Addresses Vanishing Gradients

- The cell state update (1.21) is the key design:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

LSTM: Why It Addresses Vanishing Gradients

- The cell state update (1.21) is the key design:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

- The gradient of $\mathbf{c}_t$ with respect to $\mathbf{c}_{t-1}$ is:

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \mathrm{diag}(\mathbf{f}_t) + (\text{other terms})$$

LSTM: Why It Addresses Vanishing Gradients

- The cell state update (1.21) is the key design:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

- The gradient of $\mathbf{c}_t$ with respect to $\mathbf{c}_{t-1}$ is:

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \operatorname{diag}(\mathbf{f}_t) + (\text{other terms})$$

- When $\mathbf{f}_t \approx \mathbf{1}$ (forget gate open), the gradient flows *unattenuated* through the cell state — this creates a "gradient highway" across time steps

LSTM: Why It Addresses Vanishing Gradients

- The cell state update (1.21) is the key design:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

- The gradient of $\mathbf{c}_t$ with respect to $\mathbf{c}_{t-1}$ is:

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \mathrm{diag}(\mathbf{f}_t) + (\text{other terms})$$

- When $\mathbf{f}_t \approx \mathbf{1}$ (forget gate open), the gradient flows *unattenuated* through the cell state — this creates a "gradient highway" across time steps
- The three gates provide learnable control over information flow:
  - Forget gate $\mathbf{f}_t$: controls what to erase from memory
  - Input gate $\mathbf{i}_t$: controls what new information to store
  - Output gate $\mathbf{o}_t$: controls what to expose as hidden state

LSTM: Why It Addresses Vanishing Gradients

- The cell state update (1.21) is the key design:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

- The gradient of $\mathbf{c}_t$ with respect to $\mathbf{c}_{t-1}$ is:

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \mathrm{diag}(\mathbf{f}_t) + (\text{other terms})$$

- When $\mathbf{f}_t \approx \mathbf{1}$ (forget gate open), the gradient flows *unattenuated* through the cell state — this creates a "gradient highway" across time steps
- The three gates provide learnable control over information flow:
  - Forget gate $\mathbf{f}_t$: controls what to erase from memory
  - Input gate $\mathbf{i}_t$: controls what new information to store
  - Output gate $\mathbf{o}_t$: controls what to expose as hidden state
- Empirically, LSTMs can learn dependencies over sequences of length $\sim$1000, far exceeding vanilla RNNs ($\sim$10–20 steps)

Part 4: The Transformer Architecture

Goal: Develop the complete mathematical description of the Transformer — the architecture underlying modern LLMs.

- Motivation: limitations of recurrence

Part 4: The Transformer Architecture

Goal: Develop the complete mathematical description of the Transformer — the architecture underlying modern LLMs.

- Motivation: limitations of recurrence
- Token embeddings and positional encodings

Part 4: The Transformer Architecture

Goal: Develop the complete mathematical description of the Transformer — the architecture underlying modern LLMs.

- Motivation: limitations of recurrence
- Token embeddings and positional encodings
- Scaled dot-product attention

Part 4: The Transformer Architecture

Goal: Develop the complete mathematical description of the Transformer — the architecture underlying modern LLMs.

- Motivation: limitations of recurrence
- Token embeddings and positional encodings
- Scaled dot-product attention
- Multi-head attention

Part 4: The Transformer Architecture

Goal: Develop the complete mathematical description of the Transformer — the architecture underlying modern LLMs.

- Motivation: limitations of recurrence
- Token embeddings and positional encodings
- Scaled dot-product attention
- Multi-head attention
- Feedforward layers, residual connections, layer normalisation

Part 4: The Transformer Architecture

Goal: Develop the complete mathematical description of the Transformer — the architecture underlying modern LLMs.

- Motivation: limitations of recurrence
- Token embeddings and positional encodings
- Scaled dot-product attention
- Multi-head attention
- Feedforward layers, residual connections, layer normalisation
- Computational complexity

Part 4: The Transformer Architecture

Goal: Develop the complete mathematical description of the Transformer — the architecture underlying modern LLMs.

- Motivation: limitations of recurrence
- Token embeddings and positional encodings
- Scaled dot-product attention
- Multi-head attention
- Feedforward layers, residual connections, layer normalisation
- Computational complexity
- Expressivity and computational limitations

Motivation: From Recurrence to Attention

- Limitations of RNNs/LSTMs for language modelling:

Motivation: From Recurrence to Attention

- Limitations of RNNs/LSTMs for language modelling:
  1. Sequential computation: Hidden states must be computed in order $\mathbf{h}_1 \to \mathbf{h}_2 \to \cdots \to \mathbf{h}_T$. This prevents parallelisation across time steps during training

Motivation: From Recurrence to Attention

- Limitations of RNNs/LSTMs for language modelling:
  1. Sequential computation: Hidden states must be computed in order $\mathbf{h}_1 \to \mathbf{h}_2 \to \cdots \to \mathbf{h}_T$. This prevents parallelisation across time steps during training
  2. Information bottleneck: All past information must be compressed into a fixed-dimensional hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$

Motivation: From Recurrence to Attention

- Limitations of RNNs/LSTMs for language modelling:
  1. Sequential computation: Hidden states must be computed in order $\mathbf{h}_1 \to \mathbf{h}_2 \to \cdots \to \mathbf{h}_T$. This prevents parallelisation across time steps during training
  2. Information bottleneck: All past information must be compressed into a fixed-dimensional hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$
  3. Long-range dependencies: Despite LSTMs, capturing very long-range dependencies remains difficult in practice

Motivation: From Recurrence to Attention

- Limitations of RNNs/LSTMs for language modelling:
  1. Sequential computation: Hidden states must be computed in order $\mathbf{h}_1 \to \mathbf{h}_2 \to \cdots \to \mathbf{h}_T$. This prevents parallelisation across time steps during training
  2. Information bottleneck: All past information must be compressed into a fixed-dimensional hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$
  3. Long-range dependencies: Despite LSTMs, capturing very long-range dependencies remains difficult in practice
- The Transformer (Vaswani et al., 2017) replaces recurrence entirely with an attention mechanism:
  - Every position can directly attend to every other position
  - Computation is fully parallelisable over sequence positions
  - "Attention is All You Need"

Motivation: From Recurrence to Attention

- Limitations of RNNs/LSTMs for language modelling:
  1. Sequential computation: Hidden states must be computed in order $\mathbf{h}_1 \to \mathbf{h}_2 \to \cdots \to \mathbf{h}_T$. This prevents parallelisation across time steps during training
  2. Information bottleneck: All past information must be compressed into a fixed-dimensional hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$
  3. Long-range dependencies: Despite LSTMs, capturing very long-range dependencies remains difficult in practice
- The Transformer (Vaswani et al., 2017) replaces recurrence entirely with an attention mechanism:
  - Every position can directly attend to every other position
  - Computation is fully parallelisable over sequence positions
  - "Attention is All You Need"
- Result: Dramatically better scalability, enabling training on much larger datasets and models (billions of parameters)

Token Embeddings and Positional Encodings

- Each token $x_t \in \mathcal{V}$ is mapped to a continuous vector via a learned embedding matrix $E \in \mathbb{R}^{|\mathcal{V}| \times d}$:

$$\mathbf{e}_t = E_{x_t} \in \mathbb{R}^d$$

Token Embeddings and Positional Encodings

- Each token $x_t \in \mathcal{V}$ is mapped to a continuous vector via a learned embedding matrix $E \in \mathbb{R}^{|\mathcal{V}| \times d}$:

$$\mathbf{e}_t = E_{x_t} \in \mathbb{R}^d$$

- Problem: Unlike RNNs, the Transformer processes all positions simultaneously — there is no inherent notion of sequential order

Token Embeddings and Positional Encodings

- Each token $x_t \in \mathcal{V}$ is mapped to a continuous vector via a learned embedding matrix $E \in \mathbb{R}^{|\mathcal{V}| \times d}$:

$$\mathbf{e}_t = E_{x_t} \in \mathbb{R}^d$$

- Problem: Unlike RNNs, the Transformer processes all positions simultaneously — there is no inherent notion of sequential order
- Solution: Add positional encodings to inject position information:

$$\mathbf{x}_t = \mathbf{e}_t + \mathbf{p}_t$$

Token Embeddings and Positional Encodings

- Each token $x_t \in \mathcal{V}$ is mapped to a continuous vector via a learned embedding matrix $E \in \mathbb{R}^{|\mathcal{V}| \times d}$:

$$\mathbf{e}_t = E_{x_t} \in \mathbb{R}^d$$

- Problem: Unlike RNNs, the Transformer processes all positions simultaneously — there is no inherent notion of sequential order
- Solution: Add positional encodings to inject position information:

$$\mathbf{x}_t = \mathbf{e}_t + \mathbf{p}_t$$

- Vaswani et al. use sinusoidal positional encodings:

$$\mathrm{PE}(\mathrm{pos}, 2i) = \sin\Big(\frac{\mathrm{pos}}{10000^{2i/d}}\Big) \tag{1.24}$$

$$\mathrm{PE}(\mathrm{pos}, 2i+1) = \cos\Big(\frac{\mathrm{pos}}{10000^{2i/d}}\Big) \tag{1.25}$$

for position $\mathrm{pos} = 1, \ldots, T$ and dimension $i = 0, \ldots, d/2 - 1$

Token Embeddings and Positional Encodings

- Each token $x_t \in \mathcal{V}$ is mapped to a continuous vector via a learned embedding matrix $E \in \mathbb{R}^{|\mathcal{V}| \times d}$:

$$\mathbf{e}_t = E_{x_t} \in \mathbb{R}^d$$

- Problem: Unlike RNNs, the Transformer processes all positions simultaneously — there is no inherent notion of sequential order
- Solution: Add positional encodings to inject position information:

$$\mathbf{x}_t = \mathbf{e}_t + \mathbf{p}_t$$

- Vaswani et al. use sinusoidal positional encodings:

$$\mathrm{PE}(\mathrm{pos}, 2i) = \sin\Big(\frac{\mathrm{pos}}{10000^{2i/d}}\Big) \tag{1.24}$$

$$\mathrm{PE}(\mathrm{pos}, 2i+1) = \cos\Big(\frac{\mathrm{pos}}{10000^{2i/d}}\Big) \tag{1.25}$$

  for position $\mathrm{pos} = 1, \ldots, T$ and dimension $i = 0, \ldots, d/2 - 1$
- Property: For any fixed offset k, $\mathrm{PE}(\mathrm{pos} + k)$ is a linear function of $\mathrm{PE}(\mathrm{pos})$, allowing the model to learn relative positional relationships

Why Sinusoidal Positional Encodings?

- The core problem: the Transformer processes all tokens simultaneously (no recurrence). Without positional information, it cannot distinguish "the cat sat on the mat" from "mat the on sat cat the"

Why Sinusoidal Positional Encodings?

- The core problem: the Transformer processes all tokens simultaneously (no recurrence). Without positional information, it cannot distinguish "the cat sat on the mat" from "mat the on sat cat the"
- Why sines and cosines? Think of how we write integers in binary: position $5 = 101_2$. The least significant bit flips every step, the next every 2, the next every 4. Each bit operates at a different *frequency*

Why Sinusoidal Positional Encodings?

- The core problem: the Transformer processes all tokens simultaneously (no recurrence). Without positional information, it cannot distinguish "the cat sat on the mat" from "mat the on sat cat the"
- Why sines and cosines? Think of how we write integers in binary: position $5 = 101_2$. The least significant bit flips every step, the next every 2, the next every 4. Each bit operates at a different *frequency*
- Sinusoidal PE does the same but with *smooth* oscillations: dimension pair $(2i, 2i+1)$ uses frequency $\omega_i = 1/10000^{2i/d}$
  - Small i: ω_i large $\rightarrow$ fast oscillation $\rightarrow$ distinguishes nearby positions
  - Large i: ω_i small $\rightarrow$ slow oscillation $\rightarrow$ distinguishes distant positions

Why Sinusoidal Positional Encodings?

- The core problem: the Transformer processes all tokens simultaneously (no recurrence). Without positional information, it cannot distinguish "the cat sat on the mat" from "mat the on sat cat the"
- Why sines and cosines? Think of how we write integers in binary: position $5 = 101_2$. The least significant bit flips every step, the next every 2, the next every 4. Each bit operates at a different *frequency*
- Sinusoidal PE does the same but with *smooth* oscillations: dimension pair $(2i, 2i+1)$ uses frequency $\omega_i = 1/10000^{2i/d}$
  - Small i: ω_i large $\rightarrow$ fast oscillation $\rightarrow$ distinguishes nearby positions
  - Large i: ω_i small $\rightarrow$ slow oscillation $\rightarrow$ distinguishes distant positions
- Together, the $d/2$ frequencies give each position a *unique fingerprint*, just as d binary digits uniquely encode 2^d integers — but continuously

Relative Positions and Alternatives

- Key property: for any offset k, $\mathrm{PE}(\mathrm{pos}+k)$ is a linear function of $\mathrm{PE}(\mathrm{pos})$ via a rotation:

$$\begin{pmatrix} \sin(\omega_i(\mathrm{pos}+k)) \\ \cos(\omega_i(\mathrm{pos}+k)) \end{pmatrix} = \begin{pmatrix} \cos(\omega_i k) & \sin(\omega_i k) \\ -\sin(\omega_i k) & \cos(\omega_i k) \end{pmatrix} \begin{pmatrix} \sin(\omega_i\,\mathrm{pos}) \\ \cos(\omega_i\,\mathrm{pos}) \end{pmatrix}$$

  - The rotation matrix depends only on k, not on pos
  - So the model can learn to attend to relative positions via linear projections

Relative Positions and Alternatives

- Key property: for any offset k, $\mathrm{PE}(\mathrm{pos}+k)$ is a linear function of $\mathrm{PE}(\mathrm{pos})$ via a rotation:

$$\begin{pmatrix} \sin(\omega_i(\mathrm{pos}+k)) \\ \cos(\omega_i(\mathrm{pos}+k)) \end{pmatrix} = \begin{pmatrix} \cos(\omega_i k) & \sin(\omega_i k) \\ -\sin(\omega_i k) & \cos(\omega_i k) \end{pmatrix} \begin{pmatrix} \sin(\omega_i\,\mathrm{pos}) \\ \cos(\omega_i\,\mathrm{pos}) \end{pmatrix}$$

  - The rotation matrix depends only on k, not on pos
  - So the model can learn to attend to relative positions via linear projections

- Alternatives used in modern LLMs:
  - Learned embeddings (GPT-2): a trainable vector per position. Simple but limited to the maximum training length
  - RoPE (Su et al., 2021): applies the rotation above directly to Q, K. Used in LLaMA, Mistral, and most modern LLMs
  - ALiBi (Press et al., 2022): no positional encoding; adds a linear bias $-m|i-j|$ to attention logits

Scaled Dot-Product Attention

Definition 1.5 (Scaled Dot-Product Attention)

Given an input matrix $X \in \mathbb{R}^{T \times d}$ (rows are token representations), define:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V \tag{1.26}$$

where $W_Q, W_K \in \mathbb{R}^{d \times d_k}$ and $W_V \in \mathbb{R}^{d \times d_v}$ are learned projection matrices. The scaled dot-product attention is:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V \tag{1.27}$$

where the $\mathrm{softmax}$ is applied row-wise.

Scaled Dot-Product Attention

Definition 1.5 (Scaled Dot-Product Attention)

Given an input matrix $X \in \mathbb{R}^{T \times d}$ (rows are token representations), define:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V \tag{1.26}$$

where $W_Q, W_K \in \mathbb{R}^{d \times d_k}$ and $W_V \in \mathbb{R}^{d \times d_v}$ are learned projection matrices. The scaled dot-product attention is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V \tag{1.27}$$

where the softmax *is applied row-wise.*

- Q (queries): "what am I looking for?"
- K (keys): "what do I contain?"
- V (values): "what information do I provide?"

Scaled Dot-Product Attention

Definition 1.5 (Scaled Dot-Product Attention)

Given an input matrix $X \in \mathbb{R}^{T \times d}$ (rows are token representations), define:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V \tag{1.26}$$

where $W_Q, W_K \in \mathbb{R}^{d \times d_k}$ and $W_V \in \mathbb{R}^{d \times d_v}$ are learned projection matrices. The scaled dot-product attention is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V \tag{1.27}$$

where the softmax *is applied row-wise.*

- Q (queries): "what am I looking for?"
- K (keys): "what do I contain?"
- V (values): "what information do I provide?"
- Row t of the output is a weighted average of all value vectors, where the weights are determined by the similarity of query t to all keys

Why Scale by $\sqrt{d_k}$?

- Consider the dot product $\mathbf{q}^\top \mathbf{k}$ where $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$ with entries drawn independently from a distribution with mean 0 and variance 1

Why Scale by $\sqrt{d_k}$?

- Consider the dot product $\mathbf{q}^\top \mathbf{k}$ where $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$ with entries drawn independently from a distribution with mean 0 and variance 1
- Then:

$$\mathbf{q}^\top \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i \quad \Rightarrow \quad \mathbb{E}[\mathbf{q}^\top \mathbf{k}] = 0, \quad \mathrm{Var}(\mathbf{q}^\top \mathbf{k}) = d_k$$

Why Scale by $\sqrt{d_k}$?

- Consider the dot product $\mathbf{q}^\top \mathbf{k}$ where $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$ with entries drawn independently from a distribution with mean 0 and variance 1
- Then:

$$\mathbf{q}^\top \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i \quad \Rightarrow \quad \mathbb{E}[\mathbf{q}^\top \mathbf{k}] = 0, \quad \mathrm{Var}(\mathbf{q}^\top \mathbf{k}) = d_k$$

- For large d_k, the dot products $\mathbf{q}^\top \mathbf{k}$ have large magnitude, pushing the softmax into saturation regions where gradients are extremely small

Why Scale by $\sqrt{d_k}$?

- Consider the dot product $\mathbf{q}^\top \mathbf{k}$ where $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$ with entries drawn independently from a distribution with mean 0 and variance 1
- Then:

$$\mathbf{q}^\top \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i \quad \Rightarrow \quad \mathbb{E}[\mathbf{q}^\top \mathbf{k}] = 0, \quad \mathrm{Var}(\mathbf{q}^\top \mathbf{k}) = d_k$$

- For large d_k, the dot products $\mathbf{q}^\top \mathbf{k}$ have large magnitude, pushing the softmax into saturation regions where gradients are extremely small
- Dividing by $\sqrt{d_k}$ normalises the variance:

$$\mathrm{Var}\left(\frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{d_k}}\right) = \frac{d_k}{d_k} = 1$$

Why Scale by $\sqrt{d_k}$?

- Consider the dot product $\mathbf{q}^\top \mathbf{k}$ where $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$ with entries drawn independently from a distribution with mean 0 and variance 1
- Then:

$$\mathbf{q}^\top \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i \quad \Rightarrow \quad \mathbb{E}[\mathbf{q}^\top \mathbf{k}] = 0, \quad \mathrm{Var}(\mathbf{q}^\top \mathbf{k}) = d_k$$

- For large d_k, the dot products $\mathbf{q}^\top \mathbf{k}$ have large magnitude, pushing the softmax into saturation regions where gradients are extremely small
- Dividing by $\sqrt{d_k}$ normalises the variance:

$$\mathrm{Var}\left(\frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{d_k}}\right) = \frac{d_k}{d_k} = 1$$

- This keeps the softmax inputs in a regime where the gradients are well-behaved, ensuring stable training

Why Scale by $\sqrt{d_k}$?

- Consider the dot product $\mathbf{q}^\top \mathbf{k}$ where $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$ with entries drawn independently from a distribution with mean 0 and variance 1
- Then:

$$\mathbf{q}^\top \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i \quad \Rightarrow \quad \mathbb{E}[\mathbf{q}^\top \mathbf{k}] = 0, \quad \mathrm{Var}(\mathbf{q}^\top \mathbf{k}) = d_k$$

- For large d_k, the dot products $\mathbf{q}^\top \mathbf{k}$ have large magnitude, pushing the softmax into saturation regions where gradients are extremely small
- Dividing by $\sqrt{d_k}$ normalises the variance:

$$\mathrm{Var}\left(\frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{d_k}}\right) = \frac{d_k}{d_k} = 1$$

- This keeps the softmax inputs in a regime where the gradients are well-behaved, ensuring stable training
- Without scaling: Attention weights tend to concentrate on a single key (near-one-hot), reducing the model's ability to attend to multiple positions simultaneously

Causal Masking for Autoregressive Generation

- For autoregressive language modelling, position t must only attend to positions $1, \ldots, t$ (not future positions)

Causal Masking for Autoregressive Generation

- For autoregressive language modelling, position t must only attend to positions $1, \ldots, t$ (not future positions)
- This is enforced via a causal mask: before applying softmax, set future entries to $-\infty$:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V \tag{1.28}$$

where $M \in \mathbb{R}^{T \times T}$ is the mask matrix:

$$M_{ij} = \begin{cases} 0 & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases}$$

Causal Masking for Autoregressive Generation

- For autoregressive language modelling, position t must only attend to positions $1, \ldots, t$ (not future positions)
- This is enforced via a causal mask: before applying softmax, set future entries to $-\infty$:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V \tag{1.28}$$

where $M \in \mathbb{R}^{T \times T}$ is the mask matrix:

$$M_{ij} = \begin{cases} 0 & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases}$$

- After applying softmax, the $-\infty$ entries become 0 (since $e^{-\infty} = 0$)

Causal Masking for Autoregressive Generation

- For autoregressive language modelling, position t must only attend to positions $1, \ldots, t$ (not future positions)
- This is enforced via a causal mask: before applying softmax, set future entries to $-\infty$:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V \tag{1.28}$$

where $M \in \mathbb{R}^{T \times T}$ is the mask matrix:

$$M_{ij} = \begin{cases} 0 & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases}$$

- After applying softmax, the $-\infty$ entries become 0 (since $e^{-\infty} = 0$)
- Result: The output at position t depends only on tokens at positions $\leq t$, ensuring the autoregressive property:

$$P_\theta(x_t \mid x_1, \ldots, x_{t-1})$$

Causal Masking for Autoregressive Generation

- For autoregressive language modelling, position t must only attend to positions $1, \ldots, t$ (not future positions)
- This is enforced via a causal mask: before applying softmax, set future entries to $-\infty$:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V \tag{1.28}$$

where $M \in \mathbb{R}^{T \times T}$ is the mask matrix:

$$M_{ij} = \begin{cases} 0 & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases}$$

- After applying softmax, the $-\infty$ entries become 0 (since $e^{-\infty} = 0$)
- Result: The output at position t depends only on tokens at positions $\leq t$, ensuring the autoregressive property:

$$P_\theta(x_t \mid x_1, \ldots, x_{t-1})$$

- GPT-style models (decoder-only Transformers) use causal masking throughout. This allows training on all positions simultaneously while respecting the autoregressive structure

Multi-Head Attention

- A single attention head captures one type of relationship between tokens. Multi-head attention allows the model to jointly attend to information from different representation subspaces

Multi-Head Attention

- A single attention head captures one type of relationship between tokens. Multi-head attention allows the model to jointly attend to information from different representation subspaces

Definition 1.6 (Multi-Head Attention)

Given h attention heads, the multi-head attention is:

$$\begin{aligned} \mathrm{MultiHead}(X) &= \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)\, W_O \\ \textit{where } \mathrm{head}_i &= \mathrm{Attention}(XW_Q^{(i)}, XW_K^{(i)}, XW_V^{(i)}) \end{aligned} \tag{1.29}$$

with per-head projections $W_Q^{(i)}, W_K^{(i)} \in \mathbb{R}^{d \times d_k}$, $W_V^{(i)} \in \mathbb{R}^{d \times d_v}$, *and output projection* $W_O \in \mathbb{R}^{hd_v \times d}$.

Multi-Head Attention

- A single attention head captures one type of relationship between tokens. Multi-head attention allows the model to jointly attend to information from different representation subspaces

Definition 1.6 (Multi-Head Attention)

Given h attention heads, the multi-head attention is:

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W_O \tag{1.29}$$
$$\textit{where } \text{head}_i = \text{Attention}(XW_Q^{(i)}, XW_K^{(i)}, XW_V^{(i)})$$

with per-head projections $W_Q^{(i)}, W_K^{(i)} \in \mathbb{R}^{d \times d_k}$, $W_V^{(i)} \in \mathbb{R}^{d \times d_v}$, *and output projection* $W_O \in \mathbb{R}^{hd_v \times d}$.

- Typically $d_k = d_v = d/h$, so the total computational cost is similar to a single head with full dimensionality

Multi-Head Attention

- A single attention head captures one type of relationship between tokens. Multi-head attention allows the model to jointly attend to information from different representation subspaces

Definition 1.6 (Multi-Head Attention)

Given h attention heads, the multi-head attention is:

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W_O \tag{1.29}$$
$$\text{where } \text{head}_i = \text{Attention}(XW_Q^{(i)}, XW_K^{(i)}, XW_V^{(i)})$$

with per-head projections $W_Q^{(i)}, W_K^{(i)} \in \mathbb{R}^{d \times d_k}$, $W_V^{(i)} \in \mathbb{R}^{d \times d_v}$, *and output projection* $W_O \in \mathbb{R}^{hd_v \times d}$.

- Typically $d_k = d_v = d/h$, so the total computational cost is similar to a single head with full dimensionality
- Different heads can specialise: e.g. one head may attend to syntactic structure, another to semantic similarity, another to positional proximity

Position-wise Feedforward Network

- After multi-head attention, each position is independently processed by a position-wise feedforward network (the same network applied to every position):

$$\mathrm{FFN}(\mathbf{x}) = W_2 \,\mathrm{ReLU}(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \tag{1.30}$$

  where $W_1 \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$, and typically $d_{ff} = 4d$

Position-wise Feedforward Network

- After multi-head attention, each position is independently processed by a position-wise feedforward network (the same network applied to every position):

$$\mathrm{FFN}(\mathbf{x}) = W_2 \,\mathrm{ReLU}(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \tag{1.30}$$

  where $W_1 \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$, and typically $d_{ff} = 4d$

- Role: The FFN acts as a per-token nonlinear transformation. It processes each token's representation independently (no interaction between positions)

Position-wise Feedforward Network

- After multi-head attention, each position is independently processed by a position-wise feedforward network (the same network applied to every position):

$$\mathrm{FFN}(\mathbf{x}) = W_2 \,\mathrm{ReLU}(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \tag{1.30}$$

  where $W_1 \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$, and typically $d_{ff} = 4d$
- Role: The FFN acts as a per-token nonlinear transformation. It processes each token's representation independently (no interaction between positions)
- The attention layer handles *cross-position* interactions; the FFN handles *within-position* computation

Position-wise Feedforward Network

- After multi-head attention, each position is independently processed by a position-wise feedforward network (the same network applied to every position):

$$\mathrm{FFN}(\mathbf{x}) = W_2 \,\mathrm{ReLU}(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \tag{1.30}$$

  where $W_1 \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$, and typically $d_{ff} = 4d$
- Role: The FFN acts as a per-token nonlinear transformation. It processes each token's representation independently (no interaction between positions)
- The attention layer handles *cross-position* interactions; the FFN handles *within-position* computation
- Interpretation (Geva et al., 2021): The FFN layers can be viewed as key-value memories, where the first layer computes pattern-matching scores and the second layer retrieves associated information

Position-wise Feedforward Network

- After multi-head attention, each position is independently processed by a position-wise feedforward network (the same network applied to every position):

$$\mathrm{FFN}(\mathbf{x}) = W_2\,\mathrm{ReLU}(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \tag{1.30}$$

  where $W_1 \in \mathbb{R}^{d_{ff}\times d}$, $W_2 \in \mathbb{R}^{d\times d_{ff}}$, and typically $d_{ff} = 4d$
- Role: The FFN acts as a per-token nonlinear transformation. It processes each token's representation independently (no interaction between positions)
- The attention layer handles *cross-position* interactions; the FFN handles *within-position* computation
- Interpretation (Geva et al., 2021): The FFN layers can be viewed as key-value memories, where the first layer computes pattern-matching scores and the second layer retrieves associated information
- Modern variants use SwiGLU (Shazeer, 2020) instead of ReLU:

$$\mathrm{SwiGLU}(\mathbf{x}) = (\mathrm{Swish}(W_1\mathbf{x}) \odot W_3\mathbf{x})W_2$$

Residual Connections

- Problem: stacking L layers means the gradient must pass through L composed nonlinear functions — the same vanishing gradient issue as in RNNs

Residual Connections

- Problem: stacking L layers means the gradient must pass through L composed nonlinear functions — the same vanishing gradient issue as in RNNs
- Residual connection (He et al., 2016): instead of $\mathbf{x}^{(\ell+1)} = F_\ell(\mathbf{x}^{(\ell)})$, compute

$$\mathbf{x}^{(\ell+1)} = \mathbf{x}^{(\ell)} + F_\ell(\mathbf{x}^{(\ell)})$$

where F_ℓ is the sub-layer (attention or FFN). The network only needs to learn a *correction* F_ℓ, not the full mapping

Residual Connections

- Problem: stacking L layers means the gradient must pass through L composed nonlinear functions — the same vanishing gradient issue as in RNNs
- Residual connection (He et al., 2016): instead of $\mathbf{x}^{(\ell+1)} = F_\ell(\mathbf{x}^{(\ell)})$, compute

$$\mathbf{x}^{(\ell+1)} = \mathbf{x}^{(\ell)} + F_\ell(\mathbf{x}^{(\ell)})$$

  where F_ℓ is the sub-layer (attention or FFN). The network only needs to learn a *correction* F_ℓ, not the full mapping

- Why this helps — a two-layer example. Without residuals:

$$\mathbf{x}^{(2)} = F_2(F_1(\mathbf{x}^{(0)})) \quad \Rightarrow \quad \frac{\partial \mathbf{x}^{(2)}}{\partial \mathbf{x}^{(0)}} = \underbrace{J_{F_2}}_{\text{can be small}} \cdot \underbrace{J_{F_1}}_{\text{can be small}}$$

  With residuals:

$$\mathbf{x}^{(2)} = \mathbf{x}^{(0)} + F_1(\mathbf{x}^{(0)}) + F_2(\mathbf{x}^{(0)} + F_1(\mathbf{x}^{(0)}))$$

$$\Rightarrow \quad \frac{\partial \mathbf{x}^{(2)}}{\partial \mathbf{x}^{(0)}} = \underbrace{I}_{\text{always present}} + J_{F_1} + J_{F_2}(I + J_{F_1})$$

Residual Connections

- Problem: stacking L layers means the gradient must pass through L composed nonlinear functions — the same vanishing gradient issue as in RNNs
- Residual connection (He et al., 2016): instead of $\mathbf{x}^{(\ell+1)} = F_\ell(\mathbf{x}^{(\ell)})$, compute

$$\mathbf{x}^{(\ell+1)} = \mathbf{x}^{(\ell)} + F_\ell(\mathbf{x}^{(\ell)})$$

  where F_ℓ is the sub-layer (attention or FFN). The network only needs to learn a *correction* F_ℓ, not the full mapping
- Why this helps — a two-layer example. Without residuals:

$$\mathbf{x}^{(2)} = F_2(F_1(\mathbf{x}^{(0)})) \quad \Rightarrow \quad \frac{\partial \mathbf{x}^{(2)}}{\partial \mathbf{x}^{(0)}} = \underbrace{J_{F_2}}_{\text{can be small}} \cdot \underbrace{J_{F_1}}_{\text{can be small}}$$

  With residuals:

$$\mathbf{x}^{(2)} = \mathbf{x}^{(0)} + F_1(\mathbf{x}^{(0)}) + F_2(\mathbf{x}^{(0)} + F_1(\mathbf{x}^{(0)}))$$

$$\Rightarrow \quad \frac{\partial \mathbf{x}^{(2)}}{\partial \mathbf{x}^{(0)}} = \underbrace{I}_{\text{always present}} + J_{F_1} + J_{F_2}(I + J_{F_1})$$

- The I term guarantees the gradient is *at least* the identity — it cannot vanish to zero, regardless of how small J_{F_1} and J_{F_2} are

Layer Normalisation: The Problem

- Problem: after many layers, the entries of $\mathbf{x}^{(\ell)} \in \mathbb{R}^d$ can grow or shrink in scale. If some dimensions become very large, the softmax in attention saturates; if very small, information is lost

Layer Normalisation: The Problem

- Problem: after many layers, the entries of $\mathbf{x}^{(\ell)} \in \mathbb{R}^d$ can grow or shrink in scale. If some dimensions become very large, the softmax in attention saturates; if very small, information is lost
- Concrete example: suppose $\mathbf{x} = (0.1, 200, -150, 0.3)^\top$. The large entries dominate any dot product or linear transformation, making the network insensitive to the small entries

Layer Normalisation: The Problem

- Problem: after many layers, the entries of $\mathbf{x}^{(\ell)} \in \mathbb{R}^d$ can grow or shrink in scale. If some dimensions become very large, the softmax in attention saturates; if very small, information is lost
- Concrete example: suppose $\mathbf{x} = (0.1, 200, -150, 0.3)^\top$. The large entries dominate any dot product or linear transformation, making the network insensitive to the small entries
- Solution: before each sub-layer, re-centre and re-scale the representation to have zero mean and unit variance across the d dimensions:

$$\mathrm{LayerNorm}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \boldsymbol{\beta} \tag{1.31}$$

  - $\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$ (mean across the d dimensions of this token)
  - $\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$ (variance across the d dimensions)
  - $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^d$ are *learnable* scale and shift parameters
  - $\epsilon > 0$ is a small constant for numerical stability

Layer Normalisation: The Problem

- Problem: after many layers, the entries of $\mathbf{x}^{(\ell)} \in \mathbb{R}^d$ can grow or shrink in scale. If some dimensions become very large, the softmax in attention saturates; if very small, information is lost
- Concrete example: suppose $\mathbf{x} = (0.1, 200, -150, 0.3)^\top$. The large entries dominate any dot product or linear transformation, making the network insensitive to the small entries
- Solution: before each sub-layer, re-centre and re-scale the representation to have zero mean and unit variance across the d dimensions:

$$\mathrm{LayerNorm}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \boldsymbol{\beta} \tag{1.31}$$

  - $\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$ (mean across the d dimensions of this token)
  - $\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$ (variance across the d dimensions)
  - $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^d$ are *learnable* scale and shift parameters
  - $\epsilon > 0$ is a small constant for numerical stability
- After normalisation: $\mathbf{x} = (0.1, 200, -150, 0.3)^\top \;\to\; (-0.56, 1.13, -1.12, -0.55)^\top$ — all dimensions are now on a comparable scale

Layer Normalisation: Where and Why

- Key distinction from BatchNorm: LayerNorm normalises across the *feature dimension* d (independently per token). BatchNorm normalises across the *batch dimension* (across different examples)

Layer Normalisation: Where and Why

- Key distinction from BatchNorm: LayerNorm normalises across the *feature dimension* d (independently per token). BatchNorm normalises across the *batch dimension* (across different examples)
  - BatchNorm: statistics depend on what other examples are in the mini-batch
  - LayerNorm: statistics depend only on the token's own representation
  - For autoregressive generation (one token at a time, batch size 1), BatchNorm is ill-defined. LayerNorm works regardless of batch size

Layer Normalisation: Where and Why

- Key distinction from BatchNorm: LayerNorm normalises across the *feature dimension* d (independently per token). BatchNorm normalises across the *batch dimension* (across different examples)
  - BatchNorm: statistics depend on what other examples are in the mini-batch
  - LayerNorm: statistics depend only on the token's own representation
  - For autoregressive generation (one token at a time, batch size 1), BatchNorm is ill-defined. LayerNorm works regardless of batch size
- Where to place it? Two conventions:
  - Post-norm (Vaswani et al., 2017): $\mathbf{x} \leftarrow \mathrm{LayerNorm}(\mathbf{x} + \mathrm{SubLayer}(\mathbf{x}))$
  - Pre-norm (GPT-2 and later): $\mathbf{x} \leftarrow \mathbf{x} + \mathrm{SubLayer}(\mathrm{LayerNorm}(\mathbf{x}))$

Layer Normalisation: Where and Why

- Key distinction from BatchNorm: LayerNorm normalises across the *feature dimension* d (independently per token). BatchNorm normalises across the *batch dimension* (across different examples)
  - BatchNorm: statistics depend on what other examples are in the mini-batch
  - LayerNorm: statistics depend only on the token's own representation
  - For autoregressive generation (one token at a time, batch size 1), BatchNorm is ill-defined. LayerNorm works regardless of batch size
- Where to place it? Two conventions:
  - Post-norm (Vaswani et al., 2017): $\mathbf{x} \leftarrow \mathrm{LayerNorm}(\mathbf{x} + \mathrm{SubLayer}(\mathbf{x}))$
  - Pre-norm (GPT-2 and later): $\mathbf{x} \leftarrow \mathbf{x} + \mathrm{SubLayer}(\mathrm{LayerNorm}(\mathbf{x}))$
- Why pre-norm is preferred: in post-norm, the LayerNorm sits *on the residual path* — the gradient must pass through its Jacobian. In pre-norm, the residual path $\mathbf{x} \to \mathbf{x}$ is clean — the gradient highway from the residual connection is unobstructed

Layer Normalisation: Where and Why

- Key distinction from BatchNorm: LayerNorm normalises across the *feature dimension* d (independently per token). BatchNorm normalises across the *batch dimension* (across different examples)
  - BatchNorm: statistics depend on what other examples are in the mini-batch
  - LayerNorm: statistics depend only on the token's own representation
  - For autoregressive generation (one token at a time, batch size 1), BatchNorm is ill-defined. LayerNorm works regardless of batch size
- Where to place it? Two conventions:
  - Post-norm (Vaswani et al., 2017): $\mathbf{x} \leftarrow \mathrm{LayerNorm}(\mathbf{x} + \mathrm{SubLayer}(\mathbf{x}))$
  - Pre-norm (GPT-2 and later): $\mathbf{x} \leftarrow \mathbf{x} + \mathrm{SubLayer}(\mathrm{LayerNorm}(\mathbf{x}))$
- Why pre-norm is preferred: in post-norm, the LayerNorm sits *on the residual path* — the gradient must pass through its Jacobian. In pre-norm, the residual path $\mathbf{x} \to \mathbf{x}$ is clean — the gradient highway from the residual connection is unobstructed
- Modern variants: RMSNorm (Zhang & Sennrich, 2019) drops the mean subtraction and uses only root-mean-square normalisation — slightly faster, used in LLaMA

The Transformer Decoder Block

A single Transformer decoder block composes the following operations. Given input $X \in \mathbb{R}^{T \times d}$:

1. Layer Norm + Causal Multi-Head Attention + Residual:

$$X' = X + \mathrm{MultiHead}(\mathrm{LayerNorm}(X))$$

The Transformer Decoder Block

A single Transformer decoder block composes the following operations. Given input $X \in \mathbb{R}^{T \times d}$:

1. Layer Norm + Causal Multi-Head Attention + Residual:

$$X' = X + \mathrm{MultiHead}(\mathrm{LayerNorm}(X))$$

2. Layer Norm + FFN + Residual:

$$X'' = X' + \mathrm{FFN}(\mathrm{LayerNorm}(X'))$$

The Transformer Decoder Block

A single Transformer decoder block composes the following operations. Given input $X \in \mathbb{R}^{T \times d}$:

1. Layer Norm + Causal Multi-Head Attention + Residual:

$$X' = X + \mathrm{MultiHead}(\mathrm{LayerNorm}(X))$$

2. Layer Norm + FFN + Residual:

$$X'' = X' + \mathrm{FFN}(\mathrm{LayerNorm}(X'))$$

A full decoder-only Transformer (e.g. GPT) stacks L such blocks:

$$\begin{aligned} X^{(0)} &= \mathrm{TokenEmbed}(x_1, \ldots, x_T) + \mathrm{PosEncode}(1, \ldots, T) \\ X^{(\ell)} &= \mathrm{TransformerBlock}_\ell(X^{(\ell-1)}), \quad \ell = 1, \ldots, L \\ \mathbf{y}_t &= W_{\text{out}}\, \mathrm{LayerNorm}(X_t^{(L)}) \\ P(x_{t+1} \mid x_{\le t}) &= \mathrm{softmax}(\mathbf{y}_t) \end{aligned}$$

The Transformer Decoder Block

A single Transformer decoder block composes the following operations. Given input $X \in \mathbb{R}^{T \times d}$:

1. Layer Norm + Causal Multi-Head Attention + Residual:

$$X' = X + \mathrm{MultiHead}(\mathrm{LayerNorm}(X))$$

2. Layer Norm + FFN + Residual:

$$X'' = X' + \mathrm{FFN}(\mathrm{LayerNorm}(X'))$$

A full decoder-only Transformer (e.g. GPT) stacks L such blocks:

$$\begin{aligned} X^{(0)} &= \mathrm{TokenEmbed}(x_1, \ldots, x_T) + \mathrm{PosEncode}(1, \ldots, T) \\ X^{(\ell)} &= \mathrm{TransformerBlock}_\ell(X^{(\ell-1)}), \quad \ell = 1, \ldots, L \\ \mathbf{y}_t &= W_{\text{out}}\, \mathrm{LayerNorm}(X_t^{(L)}) \\ P(x_{t+1} \mid x_{\le t}) &= \mathrm{softmax}(\mathbf{y}_t) \end{aligned}$$

Example scales: GPT-3 has $L = 96$ layers, $d = 12288$, $h = 96$ heads, $d_{ff} = 49152$, totalling 175 billion parameters.

Computational Complexity of the Transformer

- Let T = sequence length, d = model dimension, d_{ff} = FFN hidden dimension, h = number of heads, $d_k = d/h$

Computational Complexity of the Transformer

- Let T = sequence length, d = model dimension, d_{ff} = FFN hidden dimension, h = number of heads, $d_k = d/h$
- Self-attention:
  - Computing $QK^\top$: matrix multiplication $\mathbb{R}^{T \times d_k} \times \mathbb{R}^{d_k \times T}$ costs $O(T^2 d_k)$
  - Over h heads and including value projection: $\boxed{O(T^2 d)}$

Computational Complexity of the Transformer

- Let T = sequence length, d = model dimension, d_{ff} = FFN hidden dimension, h = number of heads, $d_k = d/h$
- Self-attention:
  - Computing $QK^\top$: matrix multiplication $\mathbb{R}^{T \times d_k} \times \mathbb{R}^{d_k \times T}$ costs $O(T^2 d_k)$
  - Over h heads and including value projection: $\boxed{O(T^2 d)}$
- Feedforward network:
  - Two matrix multiplications: $\mathbb{R}^{T \times d} \times \mathbb{R}^{d \times d_{ff}}$ and $\mathbb{R}^{T \times d_{ff}} \times \mathbb{R}^{d_{ff} \times d}$
  - Total: $\boxed{O(T d\, d_{ff})}$

Computational Complexity of the Transformer

- Let T = sequence length, d = model dimension, d_{ff} = FFN hidden dimension, h = number of heads, $d_k = d/h$
- Self-attention:
  - Computing $QK^\top$: matrix multiplication $\mathbb{R}^{T \times d_k} \times \mathbb{R}^{d_k \times T}$ costs $O(T^2 d_k)$
  - Over h heads and including value projection: $\boxed{O(T^2 d)}$
- Feedforward network:
  - Two matrix multiplications: $\mathbb{R}^{T \times d} \times \mathbb{R}^{d \times d_{ff}}$ and $\mathbb{R}^{T \times d_{ff}} \times \mathbb{R}^{d_{ff} \times d}$
  - Total: $\boxed{O(Td\, d_{ff})}$
- Memory for attention: Storing the $T \times T$ attention matrix requires $O(T^2)$ memory per head

Computational Complexity of the Transformer

- Let T = sequence length, d = model dimension, d_{ff} = FFN hidden dimension, h = number of heads, $d_k = d/h$
- Self-attention:
  - Computing $QK^\top$: matrix multiplication $\mathbb{R}^{T \times d_k} \times \mathbb{R}^{d_k \times T}$ costs $O(T^2 d_k)$
  - Over h heads and including value projection: $\boxed{O(T^2 d)}$
- Feedforward network:
  - Two matrix multiplications: $\mathbb{R}^{T \times d} \times \mathbb{R}^{d \times d_{ff}}$ and $\mathbb{R}^{T \times d_{ff}} \times \mathbb{R}^{d_{ff} \times d}$
  - Total: $\boxed{O(T d\, d_{ff})}$
- Memory for attention: Storing the $T \times T$ attention matrix requires $O(T^2)$ memory per head
- The $O(T^2)$ scaling with sequence length is the primary bottleneck of the Transformer architecture:
  - GPT-3: $T = 2048$; GPT-4: $T = 8192$ (or 128,000 with extensions)
  - Techniques to mitigate: FlashAttention (hardware-aware), sparse attention, linear attention approximations

The Quadratic Bottleneck: Training vs. Inference

- The $O(T^2)$ cost of attention manifests differently in training vs. inference:

The Quadratic Bottleneck: Training vs. Inference

- The $O(T^2)$ cost of attention manifests differently in training vs. inference:
- Training (parallel): All T tokens are known. The full $T \times T$ attention matrix is computed in one batched matrix multiplication. This is *embarrassingly parallel* on GPUs — the bottleneck is memory, not serial computation

The Quadratic Bottleneck: Training vs. Inference

- The $O(T^2)$ cost of attention manifests differently in training vs. inference:
- Training (parallel): All T tokens are known. The full $T \times T$ attention matrix is computed in one batched matrix multiplication. This is *embarrassingly parallel* on GPUs — the bottleneck is memory, not serial computation
- Inference (autoregressive): Tokens are generated one at a time. At step t, we need $\mathbf{q}_t^\top \mathbf{k}_j$ for all $j \leq t$
  - Naive: recompute all t dot products at each step $\rightarrow$ total cost $O(T^2 d)$
  - KV-cache: store previously computed $\mathbf{k}_j, \mathbf{v}_j$. At step t, only compute $\mathbf{q}_t$ and look up the cached keys/values $\rightarrow$ cost per step is $O(td)$, total $O(T^2 d)$ but with much better constants

The Quadratic Bottleneck: Training vs. Inference

- The $O(T^2)$ cost of attention manifests differently in training vs. inference:
- Training (parallel): All T tokens are known. The full $T \times T$ attention matrix is computed in one batched matrix multiplication. This is *embarrassingly parallel* on GPUs — the bottleneck is memory, not serial computation
- Inference (autoregressive): Tokens are generated one at a time. At step t, we need $\mathbf{q}_t^\top \mathbf{k}_j$ for all $j \leq t$
  - Naive: recompute all t dot products at each step $\rightarrow$ total cost $O(T^2 d)$
  - KV-cache: store previously computed $\mathbf{k}_j, \mathbf{v}_j$. At step t, only compute $\mathbf{q}_t$ and look up the cached keys/values $\rightarrow$ cost per step is $O(td)$, total $O(T^2 d)$ but with much better constants
- For long sequences ($T = 100{,}000+$), even the KV-cache becomes a memory bottleneck: storing T key-value pairs per layer per head

The Quadratic Bottleneck: Training vs. Inference

- The $O(T^2)$ cost of attention manifests differently in training vs. inference:
- Training (parallel): All T tokens are known. The full $T \times T$ attention matrix is computed in one batched matrix multiplication. This is *embarrassingly parallel* on GPUs — the bottleneck is memory, not serial computation
- Inference (autoregressive): Tokens are generated one at a time. At step t, we need $\mathbf{q}_t^\top \mathbf{k}_j$ for all $j \leq t$
  - Naive: recompute all t dot products at each step $\rightarrow$ total cost $O(T^2 d)$
  - KV-cache: store previously computed $\mathbf{k}_j, \mathbf{v}_j$. At step t, only compute $\mathbf{q}_t$ and look up the cached keys/values $\rightarrow$ cost per step is $O(td)$, total $O(T^2 d)$ but with much better constants
- For long sequences ($T = 100{,}000+$), even the KV-cache becomes a memory bottleneck: storing T key-value pairs per layer per head
- This motivates two lines of research:
  1. Efficient attention: reduce the $O(T^2)$ cost while keeping the Transformer architecture
  2. Alternative architectures: replace attention entirely with $O(T)$ mechanisms

Linear Attention

- Key observation: standard attention computes

$$\mathrm{Attn}(Q, K, V)_t = \frac{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)\, \mathbf{v}_j}{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)}$$

The softmax couples all positions, preventing factorisation

Linear Attention

- Key observation: standard attention computes

$$\mathrm{Attn}(Q, K, V)_t = \frac{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)\, \mathbf{v}_j}{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)}$$

  The softmax couples all positions, preventing factorisation

- Linear attention (Katharopoulos et al., 2020): replace $\exp(\mathbf{q}^\top \mathbf{k})$ with a kernel $\phi(\mathbf{q})^\top \phi(\mathbf{k})$ for some feature map ϕ:

$$\mathrm{LinAttn}(Q, K, V)_t = \frac{\phi(\mathbf{q}_t)^\top \sum_{j=1}^{t} \phi(\mathbf{k}_j)\, \mathbf{v}_j^\top}{\phi(\mathbf{q}_t)^\top \sum_{j=1}^{t} \phi(\mathbf{k}_j)}$$

Linear Attention

- Key observation: standard attention computes

$$\mathrm{Attn}(Q,K,V)_t = \frac{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)\,\mathbf{v}_j}{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)}$$

  The softmax couples all positions, preventing factorisation

- Linear attention (Katharopoulos et al., 2020): replace $\exp(\mathbf{q}^\top \mathbf{k})$ with a kernel $\phi(\mathbf{q})^\top \phi(\mathbf{k})$ for some feature map ϕ:

$$\mathrm{LinAttn}(Q,K,V)_t = \frac{\phi(\mathbf{q}_t)^\top \sum_{j=1}^{t} \phi(\mathbf{k}_j)\,\mathbf{v}_j^\top}{\phi(\mathbf{q}_t)^\top \sum_{j=1}^{t} \phi(\mathbf{k}_j)}$$

- The trick: define $\mathbf{S}_t = \sum_{j=1}^{t} \phi(\mathbf{k}_j)\mathbf{v}_j^\top \in \mathbb{R}^{d_k \times d_v}$ and $\mathbf{z}_t = \sum_{j=1}^{t} \phi(\mathbf{k}_j) \in \mathbb{R}^{d_k}$. These can be updated *recurrently*:

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \phi(\mathbf{k}_t)\mathbf{v}_t^\top, \qquad \mathbf{z}_t = \mathbf{z}_{t-1} + \phi(\mathbf{k}_t)$$

Linear Attention

- Key observation: standard attention computes

$$\mathrm{Attn}(Q,K,V)_t = \frac{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)\,\mathbf{v}_j}{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)}$$

  The softmax couples all positions, preventing factorisation

- Linear attention (Katharopoulos et al., 2020): replace $\exp(\mathbf{q}^\top \mathbf{k})$ with a kernel $\phi(\mathbf{q})^\top \phi(\mathbf{k})$ for some feature map ϕ:

$$\mathrm{LinAttn}(Q,K,V)_t = \frac{\phi(\mathbf{q}_t)^\top \sum_{j=1}^{t} \phi(\mathbf{k}_j)\,\mathbf{v}_j^\top}{\phi(\mathbf{q}_t)^\top \sum_{j=1}^{t} \phi(\mathbf{k}_j)}$$

- The trick: define $\mathbf{S}_t = \sum_{j=1}^{t} \phi(\mathbf{k}_j)\mathbf{v}_j^\top \in \mathbb{R}^{d_k \times d_v}$ and $\mathbf{z}_t = \sum_{j=1}^{t} \phi(\mathbf{k}_j) \in \mathbb{R}^{d_k}$. These can be updated *recurrently*:

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \phi(\mathbf{k}_t)\mathbf{v}_t^\top, \qquad \mathbf{z}_t = \mathbf{z}_{t-1} + \phi(\mathbf{k}_t)$$

- Each step costs $O(d^2)$ (update $\mathbf{S}_t$) instead of $O(td)$ (attend to all t keys)
- Total cost: $\boxed{O(Td^2)}$ instead of $O(T^2 d)$ — linear in sequence length

Linear Attention

- Key observation: standard attention computes

$$\mathrm{Attn}(Q,K,V)_t = \frac{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)\,\mathbf{v}_j}{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)}$$

  The softmax couples all positions, preventing factorisation

- Linear attention (Katharopoulos et al., 2020): replace $\exp(\mathbf{q}^\top \mathbf{k})$ with a kernel $\phi(\mathbf{q})^\top \phi(\mathbf{k})$ for some feature map ϕ:

$$\mathrm{LinAttn}(Q,K,V)_t = \frac{\phi(\mathbf{q}_t)^\top \sum_{j=1}^{t} \phi(\mathbf{k}_j)\,\mathbf{v}_j^\top}{\phi(\mathbf{q}_t)^\top \sum_{j=1}^{t} \phi(\mathbf{k}_j)}$$

- The trick: define $\mathbf{S}_t = \sum_{j=1}^{t} \phi(\mathbf{k}_j)\mathbf{v}_j^\top \in \mathbb{R}^{d_k \times d_v}$ and $\mathbf{z}_t = \sum_{j=1}^{t} \phi(\mathbf{k}_j) \in \mathbb{R}^{d_k}$. These can be updated *recurrently*:

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \phi(\mathbf{k}_t)\mathbf{v}_t^\top, \qquad \mathbf{z}_t = \mathbf{z}_{t-1} + \phi(\mathbf{k}_t)$$

- Each step costs $O(d^2)$ (update $\mathbf{S}_t$) instead of $O(td)$ (attend to all t keys)
- Total cost: $\boxed{O(Td^2)}$ instead of $O(T^2 d)$ — linear in sequence length
- Trade-off: the feature map ϕ is an approximation; linear attention typically underperforms softmax attention on language modelling benchmarks

State Space Models (SSMs) and Mamba

- An alternative to attention: model the sequence via a continuous-time linear dynamical system:

$$\dot{\mathbf{h}}(t) = A\,\mathbf{h}(t) + B\,x(t)$$
$$y(t) = C\,\mathbf{h}(t)$$

where $A \in \mathbb{R}^{N\times N}$, $B \in \mathbb{R}^{N\times 1}$, $C \in \mathbb{R}^{1\times N}$ are learnable, and $\mathbf{h}(t) \in \mathbb{R}^{N}$ is the hidden state

State Space Models (SSMs) and Mamba

- An alternative to attention: model the sequence via a continuous-time linear dynamical system:

$$\dot{\mathbf{h}}(t) = A\,\mathbf{h}(t) + B\,x(t)$$
$$y(t) = C\,\mathbf{h}(t)$$

  where $A \in \mathbb{R}^{N\times N}$, $B \in \mathbb{R}^{N\times 1}$, $C \in \mathbb{R}^{1\times N}$ are learnable, and $\mathbf{h}(t) \in \mathbb{R}^N$ is the hidden state

- After discretisation (zero-order hold with step Δ): $\mathbf{h}_t = \bar{A}\,\mathbf{h}_{t-1} + \bar{B}\,x_t, \quad y_t = C\,\mathbf{h}_t$
  - This is a *linear RNN* — can be computed recurrently in $O(T)$ at inference
  - But also admits a *convolution* form: $y = \bar{K} * x$ where $\bar{K}_t = C\bar{A}^t\bar{B}$. This enables $O(T \log T)$ parallel training via FFT

State Space Models (SSMs) and Mamba

- An alternative to attention: model the sequence via a continuous-time linear dynamical system:

$$\dot{\mathbf{h}}(t) = A\,\mathbf{h}(t) + B\,x(t)$$
$$y(t) = C\,\mathbf{h}(t)$$

  where $A \in \mathbb{R}^{N\times N}$, $B \in \mathbb{R}^{N\times 1}$, $C \in \mathbb{R}^{1\times N}$ are learnable, and $\mathbf{h}(t) \in \mathbb{R}^N$ is the hidden state

- After discretisation (zero-order hold with step Δ): $\mathbf{h}_t = \bar{A}\,\mathbf{h}_{t-1} + \bar{B}\,x_t, \quad y_t = C\,\mathbf{h}_t$
  - This is a *linear RNN* — can be computed recurrently in $O(T)$ at inference
  - But also admits a *convolution* form: $y = \bar{K} * x$ where $\bar{K}_t = C\bar{A}^t\bar{B}$. This enables $O(T \log T)$ parallel training via FFT

- S4 (Gu et al., 2022): structured state space with HiPPO initialisation of A. First SSM competitive with Transformers on long-range benchmarks

State Space Models (SSMs) and Mamba

- An alternative to attention: model the sequence via a continuous-time linear dynamical system:

$$\dot{\mathbf{h}}(t) = A\,\mathbf{h}(t) + B\,x(t)$$
$$y(t) = C\,\mathbf{h}(t)$$

  where $A \in \mathbb{R}^{N\times N}$, $B \in \mathbb{R}^{N\times 1}$, $C \in \mathbb{R}^{1\times N}$ are learnable, and $\mathbf{h}(t) \in \mathbb{R}^N$ is the hidden state

- After discretisation (zero-order hold with step Δ): $\mathbf{h}_t = \bar{A}\,\mathbf{h}_{t-1} + \bar{B}\,x_t, \quad y_t = C\,\mathbf{h}_t$
  - This is a *linear RNN* — can be computed recurrently in $O(T)$ at inference
  - But also admits a *convolution* form: $y = \bar{K} * x$ where $\bar{K}_t = C\bar{A}^t\bar{B}$. This enables $O(T \log T)$ parallel training via FFT

- S4 (Gu et al., 2022): structured state space with HiPPO initialisation of A. First SSM competitive with Transformers on long-range benchmarks

- Mamba (Gu & Dao, 2024): makes B, C, Δ *input-dependent* (selective SSM). This breaks the convolution form but enables content-based reasoning like attention
  - Inference: $O(T)$ recurrence (like an RNN), no KV-cache needed
  - Training: $O(T)$ via a hardware-aware parallel scan
  - Matches Transformer quality at small-to-medium scale; used in Jamba, Zamba, Codestral Mamba

State Space Models (SSMs) and Mamba

- An alternative to attention: model the sequence via a continuous-time linear dynamical system:

$$\dot{\mathbf{h}}(t) = A\,\mathbf{h}(t) + B\,x(t)$$
$$y(t) = C\,\mathbf{h}(t)$$

  where $A \in \mathbb{R}^{N\times N}$, $B \in \mathbb{R}^{N\times 1}$, $C \in \mathbb{R}^{1\times N}$ are learnable, and $\mathbf{h}(t) \in \mathbb{R}^N$ is the hidden state

- After discretisation (zero-order hold with step Δ): $\mathbf{h}_t = \bar{A}\,\mathbf{h}_{t-1} + \bar{B}\,x_t, \quad y_t = C\,\mathbf{h}_t$
  - This is a *linear RNN* — can be computed recurrently in $O(T)$ at inference
  - But also admits a *convolution* form: $y = \bar{K} * x$ where $\bar{K}_t = C\bar{A}^t\bar{B}$. This enables $O(T \log T)$ parallel training via FFT
- S4 (Gu et al., 2022): structured state space with HiPPO initialisation of A. First SSM competitive with Transformers on long-range benchmarks
- Mamba (Gu & Dao, 2024): makes B, C, Δ *input-dependent* (selective SSM). This breaks the convolution form but enables content-based reasoning like attention
  - Inference: $O(T)$ recurrence (like an RNN), no KV-cache needed
  - Training: $O(T)$ via a hardware-aware parallel scan
  - Matches Transformer quality at small-to-medium scale; used in Jamba, Zamba, Codestral Mamba
- Open question: do SSMs scale as well as Transformers to hundreds of billions of parameters? The scaling law evidence is still limited

Attention, Linear Attention, and SSMs: Summary

- Three approaches to sequence modelling, each with different trade-offs:

	Softmax Attention	Linear Attention	SSMs (Mamba)
Training cost	$O(T^2 d)$	$O(Td^2)$	$O(T)^*$
Inference (per step)	$O(td)$ + KV-cache	$O(d^2)$	$O(d)^*$
Memory at inference	$O(T \cdot d)$ KV-cache	$O(d^2)$ fixed	$O(d)$ fixed
Content-based routing	Yes (softmax)	Approximate	Yes (selective)
Long-range	Direct (any pair)	Via recurrent state	Via recurrent state
Proven at scale	Yes (GPT-4, etc.)	Limited	Emerging

*Ignoring state dimension N for clarity; actual cost is $O(TNd)$.

Attention, Linear Attention, and SSMs: Summary

- Three approaches to sequence modelling, each with different trade-offs:

	Softmax Attention	Linear Attention	SSMs (Mamba)
Training cost	$O(T^2 d)$	$O(Td^2)$	$O(T)^*$
Inference (per step)	$O(td)$ + KV-cache	$O(d^2)$	$O(d)^*$
Memory at inference	$O(T \cdot d)$ KV-cache	$O(d^2)$ fixed	$O(d)$ fixed
Content-based routing	Yes (softmax)	Approximate	Yes (selective)
Long-range	Direct (any pair)	Via recurrent state	Via recurrent state
Proven at scale	Yes (GPT-4, etc.)	Limited	Emerging

*Ignoring state dimension N for clarity; actual cost is $O(TNd)$.

- Current landscape: Transformers with softmax attention dominate large-scale LLMs. SSMs are a promising $O(T)$ alternative, especially for very long contexts

Attention, Linear Attention, and SSMs: Summary

- Three approaches to sequence modelling, each with different trade-offs:

	Softmax Attention	Linear Attention	SSMs (Mamba)
Training cost	$O(T^2 d)$	$O(Td^2)$	$O(T)^*$
Inference (per step)	$O(td)$ + KV-cache	$O(d^2)$	$O(d)^*$
Memory at inference	$O(T \cdot d)$ KV-cache	$O(d^2)$ fixed	$O(d)$ fixed
Content-based routing	Yes (softmax)	Approximate	Yes (selective)
Long-range	Direct (any pair)	Via recurrent state	Via recurrent state
Proven at scale	Yes (GPT-4, etc.)	Limited	Emerging

*Ignoring state dimension N for clarity; actual cost is $O(TNd)$.

- Current landscape: Transformers with softmax attention dominate large-scale LLMs. SSMs are a promising $O(T)$ alternative, especially for very long contexts
- Hybrid architectures (Jamba, Zamba): interleave Transformer layers with Mamba layers, getting the best of both worlds

Expressivity: Universal Approximation for Transformers

Theorem 1.7 (Yun et al., 2020)

Let $1 \leq p < \infty$ and let $\mathcal{F}^p$ denote the class of continuous, permutation-equivariant functions $f : \mathbb{R}^{T \times d} \to \mathbb{R}^{T \times d}$ on compact domains. Then for any $f \in \mathcal{F}^p$ and any $\varepsilon > 0$, there exists a Transformer network g such that:

$$\|f - g\|_p < \varepsilon$$

More precisely, Transformers with $O(1)$ heads, $O(1)$ layers, and sufficient width are universal approximators for sequence-to-sequence functions.

Expressivity: Universal Approximation for Transformers

Theorem 1.7 (Yun et al., 2020)

Let $1 \leq p < \infty$ and let $\mathcal{F}^p$ denote the class of continuous, permutation-equivariant functions $f : \mathbb{R}^{T \times d} \to \mathbb{R}^{T \times d}$ on compact domains. Then for any $f \in \mathcal{F}^p$ and any $\varepsilon > 0$, there exists a Transformer network g such that:

$$\|f - g\|_p < \varepsilon$$

More precisely, Transformers with $O(1)$ heads, $O(1)$ layers, and sufficient width are universal approximators for sequence-to-sequence functions.

- This extends the classical Universal Approximation Theorem (for FNNs) to the sequence-to-sequence setting

Expressivity: Universal Approximation for Transformers

Theorem 1.7 (Yun et al., 2020)

Let $1 \leq p < \infty$ and let $\mathcal{F}^p$ denote the class of continuous, permutation-equivariant functions $f : \mathbb{R}^{T \times d} \to \mathbb{R}^{T \times d}$ on compact domains. Then for any $f \in \mathcal{F}^p$ and any $\varepsilon > 0$, there exists a Transformer network g such that:

$$\|f - g\|_p < \varepsilon$$

More precisely, Transformers with $O(1)$ heads, $O(1)$ layers, and sufficient width are universal approximators for sequence-to-sequence functions.

- This extends the classical Universal Approximation Theorem (for FNNs) to the sequence-to-sequence setting
- The self-attention mechanism is crucial: it allows arbitrary interactions between positions, which pure FFNs applied independently to each position cannot achieve

Expressivity: Universal Approximation for Transformers

Theorem 1.7 (Yun et al., 2020)

Let $1 \leq p < \infty$ and let $\mathcal{F}^p$ denote the class of continuous, permutation-equivariant functions $f : \mathbb{R}^{T \times d} \to \mathbb{R}^{T \times d}$ on compact domains. Then for any $f \in \mathcal{F}^p$ and any $\varepsilon > 0$, there exists a Transformer network g such that:

$$\|f - g\|_p < \varepsilon$$

More precisely, Transformers with $O(1)$ heads, $O(1)$ layers, and sufficient width are universal approximators for sequence-to-sequence functions.

- This extends the classical Universal Approximation Theorem (for FNNs) to the sequence-to-sequence setting
- The self-attention mechanism is crucial: it allows arbitrary interactions between positions, which pure FFNs applied independently to each position cannot achieve
- Key insight: Self-attention can implement *contextual mappings* — the representation of each token can depend on the entire input sequence

Part 5: Scaling Laws

Goal: Understand the empirical relationships between model scale, data, compute, and performance, and their implications for LLM development.

- Neural scaling laws (Kaplan et al., 2020)

Part 5: Scaling Laws

Goal: Understand the empirical relationships between model scale, data, compute, and performance, and their implications for LLM development.

- Neural scaling laws (Kaplan et al., 2020)
- Chinchilla scaling laws (Hoffmann et al., 2022)

Part 5: Scaling Laws

Goal: Understand the empirical relationships between model scale, data, compute, and performance, and their implications for LLM development.

- Neural scaling laws (Kaplan et al., 2020)
- Chinchilla scaling laws (Hoffmann et al., 2022)
- Emergent abilities and their interpretation

Neural Scaling Laws: Setup (Kaplan et al., 2020)

- The experiment: Train a family of Transformer LMs of different sizes on different amounts of data. Measure test loss (cross-entropy on held-out text) as a function of three variables:
  - N = number of non-embedding parameters (model size)
  - D = number of training tokens (dataset size)
  - C = compute budget (in floating-point operations, FLOPs)

Neural Scaling Laws: Setup (Kaplan et al., 2020)

- The experiment: Train a family of Transformer LMs of different sizes on different amounts of data. Measure test loss (cross-entropy on held-out text) as a function of three variables:
  - N = number of non-embedding parameters (model size)
  - D = number of training tokens (dataset size)
  - C = compute budget (in floating-point operations, FLOPs)
- Why these three? They are the three "knobs" you can turn when building an LLM:
  1. How big is the model? (architecture choice $\rightarrow N$)
  2. How much data do you train on? (data collection $\rightarrow D$)
  3. How much compute do you spend? (hardware $\times$ time $\rightarrow C$)

Neural Scaling Laws: Setup (Kaplan et al., 2020)

- The experiment: Train a family of Transformer LMs of different sizes on different amounts of data. Measure test loss (cross-entropy on held-out text) as a function of three variables:
  - N = number of non-embedding parameters (model size)
  - D = number of training tokens (dataset size)
  - C = compute budget (in floating-point operations, FLOPs)
- Why these three? They are the three "knobs" you can turn when building an LLM:
  1. How big is the model? (architecture choice $\rightarrow N$)
  2. How much data do you train on? (data collection $\rightarrow D$)
  3. How much compute do you spend? (hardware $\times$ time $\rightarrow C$)
- The metric: test loss $L = -\frac{1}{T}\sum_{t=1}^{T} \log P_\theta(x_t \mid x_{<t})$ on a fixed held-out corpus
  - This is the same cross-entropy loss from Part 1, but evaluated on unseen text
  - Lower L = better next-token prediction = better language model
  - Recall: perplexity $= e^L$, so lower loss $\Leftrightarrow$ lower perplexity

Neural Scaling Laws: The Power Laws

- Empirical finding: On *log-log axes*, the relationship between loss and each variable is approximately linear. A linear relationship on log-log axes means a power law:

$$\log L \approx -\alpha \log N + \text{const} \quad \Leftrightarrow \quad L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha}$$

Neural Scaling Laws: The Power Laws

- Empirical finding: On *log-log axes*, the relationship between loss and each variable is approximately linear. A linear relationship on log-log axes means a power law:

$$\log L \approx -\alpha \log N + \text{const} \quad \Leftrightarrow \quad L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha}$$

- The three power laws (each holding when the other variables are not bottlenecking):

$$L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha_N}, \quad \alpha_N \approx 0.076 \tag{1.32}$$

$$L(D) \approx \left(\frac{D_c}{D}\right)^{\alpha_D}, \quad \alpha_D \approx 0.095 \tag{1.33}$$

$$L(C) \approx \left(\frac{C_c}{C}\right)^{\alpha_C}, \quad \alpha_C \approx 0.050 \tag{1.34}$$

where N_c, D_c, C_c are fitted constants

Neural Scaling Laws: The Power Laws

- Empirical finding: On *log-log axes*, the relationship between loss and each variable is approximately linear. A linear relationship on log-log axes means a power law:

$$\log L \approx -\alpha \log N + \text{const} \quad \Leftrightarrow \quad L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha}$$

- The three power laws (each holding when the other variables are not bottlenecking):

$$L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha_N}, \quad \alpha_N \approx 0.076 \tag{1.32}$$

$$L(D) \approx \left(\frac{D_c}{D}\right)^{\alpha_D}, \quad \alpha_D \approx 0.095 \tag{1.33}$$

$$L(C) \approx \left(\frac{C_c}{C}\right)^{\alpha_C}, \quad \alpha_C \approx 0.050 \tag{1.34}$$

  where N_c, D_c, C_c are fitted constants

- Reading the exponents: $L(N) \propto N^{-0.076}$. To halve the loss, you need N to increase by $2^{1/0.076} \approx 8{,}000\times$. Progress is real but very expensive

Neural Scaling Laws: The Power Laws

- Empirical finding: On *log-log axes*, the relationship between loss and each variable is approximately linear. A linear relationship on log-log axes means a power law:

$$\log L \approx -\alpha \log N + \text{const} \quad \Leftrightarrow \quad L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha}$$

- The three power laws (each holding when the other variables are not bottlenecking):

$$L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha_N}, \quad \alpha_N \approx 0.076 \tag{1.32}$$

$$L(D) \approx \left(\frac{D_c}{D}\right)^{\alpha_D}, \quad \alpha_D \approx 0.095 \tag{1.33}$$

$$L(C) \approx \left(\frac{C_c}{C}\right)^{\alpha_C}, \quad \alpha_C \approx 0.050 \tag{1.34}$$

  where N_c, D_c, C_c are fitted constants

- Reading the exponents: $L(N) \propto N^{-0.076}$. To halve the loss, you need N to increase by $2^{1/0.076} \approx 8{,}000\times$. Progress is real but very expensive
- Note $\alpha_D > \alpha_N > \alpha_C$: loss improves fastest with more *data*, then more *parameters*, then more *compute* (but $C \approx 6ND$ ties them together)

Scaling Laws: What Matters and What Doesn't

- Surprising finding 1: Model shape matters far less than model size
  - Varying depth vs. width, number of attention heads, FFN dimension — as long as total N is the same, loss is approximately the same
  - This justifies measuring progress by parameter count alone

Scaling Laws: What Matters and What Doesn't

- Surprising finding 1: Model shape matters far less than model size
  - Varying depth vs. width, number of attention heads, FFN dimension — as long as total N is the same, loss is approximately the same
  - This justifies measuring progress by parameter count alone
- Surprising finding 2: Larger models are more sample-efficient
  - A $10\times$ larger model reaches the same loss with $\sim 10\times$ fewer training tokens
  - But: it costs $10\times$ more FLOPs per token, so total compute is similar

Scaling Laws: What Matters and What Doesn't

- Surprising finding 1: Model shape matters far less than model size
  - Varying depth vs. width, number of attention heads, FFN dimension — as long as total N is the same, loss is approximately the same
  - This justifies measuring progress by parameter count alone
- Surprising finding 2: Larger models are more sample-efficient
  - A $10\times$ larger model reaches the same loss with $\sim 10\times$ fewer training tokens
  - But: it costs $10\times$ more FLOPs per token, so total compute is similar
- Surprising finding 3: Power laws hold over many orders of magnitude
  - Kaplan et al. tested models from $\sim$1K to $\sim$1B parameters — 6 orders of magnitude
  - The same power-law exponents fit the entire range with no sign of saturating
  - This is unusual — most empirical scaling relationships break down outside a narrow range

Joint Scaling and Predictability

- When both N and D are limited, Kaplan et al. proposed a joint scaling law:

$$L(N, D) \approx \left[\left(\frac{N_c}{N} \right)^{\alpha_N / \alpha_D} + \frac{D_c}{D} \right]^{\alpha_D}$$

Joint Scaling and Predictability

- When both N and D are limited, Kaplan et al. proposed a joint scaling law:

$$L(N, D) \approx \left[\left(\frac{N_c}{N} \right)^{\alpha_N / \alpha_D} + \frac{D_c}{D} \right]^{\alpha_D}$$

- Interpretation: L is dominated by whichever bottleneck is worse (smaller N or smaller D). When $N \gg N_c$, the first term vanishes and $L(D) \sim D^{-\alpha_D}$. Symmetrically for $D \gg D_c$

Joint Scaling and Predictability

- When both N and D are limited, Kaplan et al. proposed a joint scaling law:

$$L(N, D) \approx \left[\left(\frac{N_c}{N} \right)^{\alpha_N / \alpha_D} + \frac{D_c}{D} \right]^{\alpha_D}$$

- Interpretation: L is dominated by whichever bottleneck is worse (smaller N or smaller D). When $N \gg N_c$, the first term vanishes and $L(D) \sim D^{-\alpha_D}$. Symmetrically for $D \gg D_c$
- Why this is practically important: you can *predict* the performance of a large model before training it
  - Train a family of small models (e.g. 10M, 50M, 100M, 500M parameters)
  - Fit the power-law exponents α_N, α_D on these small runs
  - Extrapolate to predict the loss of a 70B or 175B model
  - This saves millions of dollars in wasted compute on bad configurations

Joint Scaling and Predictability

- When both N and D are limited, Kaplan et al. proposed a joint scaling law:

$$L(N, D) \approx \left[\left(\frac{N_c}{N} \right)^{\alpha_N / \alpha_D} + \frac{D_c}{D} \right]^{\alpha_D}$$

- Interpretation: L is dominated by whichever bottleneck is worse (smaller N or smaller D). When $N \gg N_c$, the first term vanishes and $L(D) \sim D^{-\alpha_D}$. Symmetrically for $D \gg D_c$
- Why this is practically important: you can *predict* the performance of a large model before training it
  - Train a family of small models (e.g. 10M, 50M, 100M, 500M parameters)
  - Fit the power-law exponents α_N, α_D on these small runs
  - Extrapolate to predict the loss of a 70B or 175B model
  - This saves millions of dollars in wasted compute on bad configurations
- Real-world example: GPT-4's performance was reportedly predicted accurately from scaling laws fitted on much smaller models (OpenAI, 2023)

The Compute Budget: Where Do the FLOPs Go?

- The total compute cost of training is approximately:

$$C \approx 6ND \quad \text{FLOPs}$$

The Compute Budget: Where Do the FLOPs Go?

- The total compute cost of training is approximately:

$$C \approx 6ND \quad \text{FLOPs}$$

- Where does the factor of 6 come from?
  - Each token in the forward pass requires $\sim 2N$ multiply-adds (one for each weight, applied to each input)
  - The backward pass is approximately $2\times$ the forward pass (computing gradients w.r.t. both activations and weights)
  - Total: $\sim 2N + 4N = 6N$ FLOPs per token, times D tokens

The Compute Budget: Where Do the FLOPs Go?

- The total compute cost of training is approximately:

$$C \approx 6ND \quad \text{FLOPs}$$

- Where does the factor of 6 come from?
  - Each token in the forward pass requires $\sim 2N$ multiply-adds (one for each weight, applied to each input)
  - The backward pass is approximately $2\times$ the forward pass (computing gradients w.r.t. both activations and weights)
  - Total: $\sim 2N + 4N = 6N$ FLOPs per token, times D tokens
- Concrete scale:
  - GPT-3 (175B params, 300B tokens): $C \approx 6 \times 175 \times 10^9 \times 300 \times 10^9 = 3.15 \times 10^{23}$ FLOPs
  - On 1024 A100 GPUs at 312 TFLOPS each: $\sim$ 10 days of training
  - Estimated cost: $\sim$\$4–12M (at 2023 cloud prices)

The Compute Budget: Where Do the FLOPs Go?

- The total compute cost of training is approximately:

$$C \approx 6ND \quad \text{FLOPs}$$

- Where does the factor of 6 come from?
  - Each token in the forward pass requires $\sim 2N$ multiply-adds (one for each weight, applied to each input)
  - The backward pass is approximately $2\times$ the forward pass (computing gradients w.r.t. both activations and weights)
  - Total: $\sim 2N + 4N = 6N$ FLOPs per token, times D tokens
- Concrete scale:
  - GPT-3 (175B params, 300B tokens): $C \approx 6 \times 175 \times 10^9 \times 300 \times 10^9 = 3.15 \times 10^{23}$ FLOPs
  - On 1024 A100 GPUs at 312 TFLOPS each: ~ 10 days of training
  - Estimated cost: $\sim$\$4–12M (at 2023 cloud prices)
- The central question of scaling laws: Given a budget of C FLOPs, what is the best split between N and $D = C/(6N)$?

Compute-Optimal Training (Hoffmann et al., 2022)

- Kaplan et al. (2020) answer: Scale N faster than D
  - Their analysis suggested $N_{\text{opt}} \propto C^{0.73}$, $D_{\text{opt}} \propto C^{0.27}$
  - In words: spend most of your budget on a bigger model, even if it sees less data

Compute-Optimal Training (Hoffmann et al., 2022)

- Kaplan et al. (2020) answer: Scale N faster than D
  - Their analysis suggested $N_{\text{opt}} \propto C^{0.73}$, $D_{\text{opt}} \propto C^{0.27}$
  - In words: spend most of your budget on a bigger model, even if it sees less data
- Hoffmann et al. (2022) answer: This was wrong. They proposed a refined loss model:

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}} \tag{1.35}$$

where $E \approx 1.69$ nats is the irreducible entropy of natural language, $\alpha \approx 0.34$, $\beta \approx 0.28$

Compute-Optimal Training (Hoffmann et al., 2022)

- Kaplan et al. (2020) answer: Scale N faster than D
  - Their analysis suggested $N_{\text{opt}} \propto C^{0.73}$, $D_{\text{opt}} \propto C^{0.27}$
  - In words: spend most of your budget on a bigger model, even if it sees less data
- Hoffmann et al. (2022) answer: This was wrong. They proposed a refined loss model:

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}} \tag{1.35}$$

  where $E \approx 1.69$ nats is the irreducible entropy of natural language, $\alpha \approx 0.34$, $\beta \approx 0.28$
- Key insight: The three terms have clear interpretations:
  - E: the entropy of the data — no model can beat this
  - A/N^{α}: the approximation error — the model is too small to represent the true distribution
  - B/D^{β}: the estimation error — not enough data to learn the parameters well

Compute-Optimal Training (Hoffmann et al., 2022)

- Kaplan et al. (2020) answer: Scale N faster than D
  - Their analysis suggested $N_{\text{opt}} \propto C^{0.73}$, $D_{\text{opt}} \propto C^{0.27}$
  - In words: spend most of your budget on a bigger model, even if it sees less data
- Hoffmann et al. (2022) answer: This was wrong. They proposed a refined loss model:

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}} \tag{1.35}$$

  where $E \approx 1.69$ nats is the irreducible entropy of natural language, $\alpha \approx 0.34$, $\beta \approx 0.28$
- Key insight: The three terms have clear interpretations:
  - E: the entropy of the data — no model can beat this
  - A/N^{α}: the approximation error — the model is too small to represent the true distribution
  - B/D^{β}: the estimation error — not enough data to learn the parameters well
- Minimising $L(N, D)$ subject to $C = 6ND$ via Lagrange multipliers gives:

$$N_{\text{opt}} \propto C^{0.50}, \quad D_{\text{opt}} \propto C^{0.50}$$

  Model size and data should be scaled equally

The Chinchilla Result

- The punchline: Most large models before Chinchilla were *significantly undertrained* — too many parameters, too little data

The Chinchilla Result

- The punchline: Most large models before Chinchilla were *significantly undertrained* — too many parameters, too little data
- Concrete example:
  - Gopher (DeepMind, 2021): 280B parameters, trained on 300B tokens
  - Chinchilla (DeepMind, 2022): 70B parameters, trained on 1.4T tokens
  - Same compute budget ($\sim 5 \times 10^{23}$ FLOPs), but Chinchilla outperformed Gopher on nearly every benchmark

The Chinchilla Result

- The punchline: Most large models before Chinchilla were *significantly undertrained* — too many parameters, too little data
- Concrete example:
  - Gopher (DeepMind, 2021): 280B parameters, trained on 300B tokens
  - Chinchilla (DeepMind, 2022): 70B parameters, trained on 1.4T tokens
  - Same compute budget ($\sim 5 \times 10^{23}$ FLOPs), but Chinchilla outperformed Gopher on nearly every benchmark
- Optimal data-to-parameters ratio: approximately 20 tokens per parameter
  - GPT-3 (175B params, 300B tokens): ratio ≈ 1.7 — heavily undertrained
  - Chinchilla (70B params, 1.4T tokens): ratio ≈ 20 — compute-optimal
  - LLaMA (65B params, 1.4T tokens): ratio ≈ 22 — slightly over Chinchilla-optimal

The Chinchilla Result

- The punchline: Most large models before Chinchilla were *significantly undertrained* — too many parameters, too little data
- Concrete example:
  - Gopher (DeepMind, 2021): 280B parameters, trained on 300B tokens
  - Chinchilla (DeepMind, 2022): 70B parameters, trained on 1.4T tokens
  - Same compute budget ($\sim 5 \times 10^{23}$ FLOPs), but Chinchilla outperformed Gopher on nearly every benchmark
- Optimal data-to-parameters ratio: approximately 20 tokens per parameter
  - GPT-3 (175B params, 300B tokens): ratio ≈ 1.7 — heavily undertrained
  - Chinchilla (70B params, 1.4T tokens): ratio ≈ 20 — compute-optimal
  - LLaMA (65B params, 1.4T tokens): ratio ≈ 22 — slightly over Chinchilla-optimal
- The "Chinchilla tax": Compute-optimal training demands far more data than previously assumed. For 175B parameters, the optimal dataset is $\sim$3.5T tokens

The Chinchilla Result

- The punchline: Most large models before Chinchilla were *significantly undertrained* — too many parameters, too little data
- Concrete example:
  - Gopher (DeepMind, 2021): 280B parameters, trained on 300B tokens
  - Chinchilla (DeepMind, 2022): 70B parameters, trained on 1.4T tokens
  - Same compute budget ($\sim 5 \times 10^{23}$ FLOPs), but Chinchilla outperformed Gopher on nearly every benchmark
- Optimal data-to-parameters ratio: approximately 20 tokens per parameter
  - GPT-3 (175B params, 300B tokens): ratio ≈ 1.7 — heavily undertrained
  - Chinchilla (70B params, 1.4T tokens): ratio ≈ 20 — compute-optimal
  - LLaMA (65B params, 1.4T tokens): ratio ≈ 22 — slightly over Chinchilla-optimal
- The "Chinchilla tax": Compute-optimal training demands far more data than previously assumed. For 175B parameters, the optimal dataset is $\sim$3.5T tokens
- Beyond Chinchilla: Recent models (LLaMA-3, Gemma) train well *past* the Chinchilla-optimal point — more data than predicted optimal, because inference cost (which scales with N, not D) matters in deployment

Emergent Abilities

- Wei et al. (2022) observed that certain capabilities appear to "emerge" suddenly at specific model scales:
  - *Few-shot arithmetic*: Near-zero accuracy below 13B, then jumps to $> 50\%$
  - *Multi-step reasoning*: Appears around 100B parameters
  - *Word unscrambling*: Absent below a threshold, then rapidly improves

Emergent Abilities

- Wei et al. (2022) observed that certain capabilities appear to "emerge" suddenly at specific model scales:
  - *Few-shot arithmetic*: Near-zero accuracy below 13B, then jumps to $> 50\%$
  - *Multi-step reasoning*: Appears around 100B parameters
  - *Word unscrambling*: Absent below a threshold, then rapidly improves
- An ability is called emergent if it is "not present in smaller models but is present in larger models" — a *sharp, unpredictable* transition

Emergent Abilities

- Wei et al. (2022) observed that certain capabilities appear to "emerge" suddenly at specific model scales:
  - *Few-shot arithmetic*: Near-zero accuracy below 13B, then jumps to $> 50\%$
  - *Multi-step reasoning*: Appears around 100B parameters
  - *Word unscrambling*: Absent below a threshold, then rapidly improves
- An ability is called emergent if it is "not present in smaller models but is present in larger models" — a *sharp, unpredictable* transition
- This is surprising because the scaling laws predict *smooth* improvement. If the underlying loss improves smoothly, why do specific capabilities appear to jump?

Emergent Abilities: The Metric Artefact Debate

- Schaeffer et al. (2023) argued that emergence is a measurement artefact:

Emergent Abilities: The Metric Artefact Debate

- Schaeffer et al. (2023) argued that emergence is a measurement artefact:
- The argument: Consider a task like 3-digit addition. The metric is exact-match accuracy (the answer is either exactly right or wrong). Suppose the per-token probability improves smoothly from 0.1 to 0.99 as the model scales:
  - Getting all 4 digits right requires $\sim p^4$
  - At $p = 0.5$: accuracy $\approx 0.5^4 = 6\%$ (looks like failure)
  - At $p = 0.9$: accuracy $\approx 0.9^4 = 66\%$ (looks like sudden success)

Emergent Abilities: The Metric Artefact Debate

- Schaeffer et al. (2023) argued that emergence is a measurement artefact:
- The argument: Consider a task like 3-digit addition. The metric is exact-match accuracy (the answer is either exactly right or wrong). Suppose the per-token probability improves smoothly from 0.1 to 0.99 as the model scales:
  - Getting all 4 digits right requires $\sim p^4$
  - At $p = 0.5$: accuracy $\approx 0.5^4 = 6\%$ (looks like failure)
  - At $p = 0.9$: accuracy $\approx 0.9^4 = 66\%$ (looks like sudden success)
- The *underlying* capability (per-token accuracy) improved smoothly, but the *metric* (exact match) introduced a nonlinear threshold that created the appearance of a phase transition

Emergent Abilities: The Metric Artefact Debate

- Schaeffer et al. (2023) argued that emergence is a measurement artefact:
- The argument: Consider a task like 3-digit addition. The metric is exact-match accuracy (the answer is either exactly right or wrong). Suppose the per-token probability improves smoothly from 0.1 to 0.99 as the model scales:
  - Getting all 4 digits right requires $\sim p^4$
  - At $p = 0.5$: accuracy $\approx 0.5^4 = 6\%$ (looks like failure)
  - At $p = 0.9$: accuracy $\approx 0.9^4 = 66\%$ (looks like sudden success)
- The *underlying* capability (per-token accuracy) improved smoothly, but the *metric* (exact match) introduced a nonlinear threshold that created the appearance of a phase transition
- Evidence: When Schaeffer et al. re-evaluated the same tasks with continuous metrics (e.g. Brier score, token-level log-likelihood), the sharp transitions disappeared — performance improved smoothly and predictably with scale

Emergent Abilities: The Metric Artefact Debate

- Schaeffer et al. (2023) argued that emergence is a measurement artefact:
- The argument: Consider a task like 3-digit addition. The metric is exact-match accuracy (the answer is either exactly right or wrong). Suppose the per-token probability improves smoothly from 0.1 to 0.99 as the model scales:
  - Getting all 4 digits right requires $\sim p^4$
  - At $p = 0.5$: accuracy $\approx 0.5^4 = 6\%$ (looks like failure)
  - At $p = 0.9$: accuracy $\approx 0.9^4 = 66\%$ (looks like sudden success)
- The *underlying* capability (per-token accuracy) improved smoothly, but the *metric* (exact match) introduced a nonlinear threshold that created the appearance of a phase transition
- Evidence: When Schaeffer et al. re-evaluated the same tasks with continuous metrics (e.g. Brier score, token-level log-likelihood), the sharp transitions disappeared — performance improved smoothly and predictably with scale
- Current consensus: The underlying capabilities likely improve continuously with scale (consistent with smooth scaling laws), but certain *task-level metrics* can make this look like sudden emergence

Summary

1. Statistical Language Modelling: Language models assign probabilities to token sequences via the chain rule decomposition. Neural language models replace counting-based estimation with learned parameters

Summary

1. Statistical Language Modelling: Language models assign probabilities to token sequences via the chain rule decomposition. Neural language models replace counting-based estimation with learned parameters
2. Neural Networks: Universal approximators (Cybenko, Hornik). Depth provides exponential efficiency gains. Trained via SGD and backpropagation

Summary

1. Statistical Language Modelling: Language models assign probabilities to token sequences via the chain rule decomposition. Neural language models replace counting-based estimation with learned parameters
2. Neural Networks: Universal approximators (Cybenko, Hornik). Depth provides exponential efficiency gains. Trained via SGD and backpropagation
3. Recurrent Neural Networks: Process sequences via hidden states. Suffer from vanishing/exploding gradients. LSTMs partially address this with gating mechanisms. Theoretically Turing-complete but limited in practice

Summary

1. Statistical Language Modelling: Language models assign probabilities to token sequences via the chain rule decomposition. Neural language models replace counting-based estimation with learned parameters
2. Neural Networks: Universal approximators (Cybenko, Hornik). Depth provides exponential efficiency gains. Trained via SGD and backpropagation
3. Recurrent Neural Networks: Process sequences via hidden states. Suffer from vanishing/exploding gradients. LSTMs partially address this with gating mechanisms. Theoretically Turing-complete but limited in practice
4. Transformers: Replace recurrence with self-attention. Enable full parallelisation and direct long-range interactions. $O(T^2 d)$ complexity. Universal approximators for sequence-to-sequence functions. Alternatives: linear attention, SSMs (Mamba)

Summary

1. Statistical Language Modelling: Language models assign probabilities to token sequences via the chain rule decomposition. Neural language models replace counting-based estimation with learned parameters
2. Neural Networks: Universal approximators (Cybenko, Hornik). Depth provides exponential efficiency gains. Trained via SGD and backpropagation
3. Recurrent Neural Networks: Process sequences via hidden states. Suffer from vanishing/exploding gradients. LSTMs partially address this with gating mechanisms. Theoretically Turing-complete but limited in practice
4. Transformers: Replace recurrence with self-attention. Enable full parallelisation and direct long-range interactions. $O(T^2 d)$ complexity. Universal approximators for sequence-to-sequence functions. Alternatives: linear attention, SSMs (Mamba)
5. Scaling Laws: Test loss follows power laws in model size, data, and compute. Chinchilla-optimal training scales N and D equally. Emergent abilities are debated

Optional Reading

- Douglas, M. R. (2023). *Large Language Models*. arXiv:2307.05782.
  - Primary reference for this lecture
- Vaswani, A. et al. (2017). Attention Is All You Need. *NeurIPS*.
  - The original Transformer paper
- Kaplan, J. et al. (2020). Scaling Laws for Neural Language Models. *arXiv:2001.08361*.
- Hoffmann, J. et al. (2022). Training Compute-Optimal Large Language Models. *NeurIPS* (Chinchilla paper).
- Yun, C. et al. (2020). Are Transformers Universal Approximators of Sequence-to-Sequence Functions? *ICLR*.
- Merrill, W. & Sabharwal, A. (2023). The Parallelism Tradeoff: Limitations of Log-Precision Transformers. *TACL*.
- Siegelmann, H. & Sontag, E. (1995). On the Computational Power of Neural Nets. *JCSS*, 50(1), 132–150.
- Schaeffer, R. et al. (2023). Are Emergent Abilities of Large Language Models a Mirage? *NeurIPS*.