Generative Models in Finance

Week 1: The Mathematics of Large Language Models

Cristopher Salvi
Imperial College London

Spring 2026

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?
- 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)
- 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:

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

- 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
- 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 $|\mathcal{V}| \approx 50{,}000$)

Sennrich, R., Haddow, B. & Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units. *ACL*.

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?
- 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:

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

- 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

- 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 $(\mathtt{a}, \mathtt{b}, \mathtt{a})$, $T = 3$
- 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 $|\mathcal{V}| = 3$)
- 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
- 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:

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

Bengio, Y., Ducharme, R., Vincent, P. & Jauvin, C. (2003). A Neural Probabilistic Language Model. *JMLR*, 3, 1137–1155.

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
- 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*

- 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
- 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
- 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
- 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:
- 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

Maas, A. L., Hannun, A. Y. & Ng, A. Y. (2013). Rectifier Nonlinearities Improve Neural Network Acoustic Models. *ICML Workshop*.
Hendrycks, D. & Gimpel, K. (2016). Gaussian Error Linear Units (GELUs). arXiv:1606.08415.
Ramachandran, P., Zoph, B. & Le, Q. V. (2017). Searching for Activation Functions. arXiv:1710.05941.

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
- Note: The hypotheses require σ to be *bounded*, which excludes ReLU. Leshno et al. (1993) proved a sharper result: a single-hidden-layer network is a universal approximator if and only if σ is *not a polynomial*. This covers ReLU, Leaky ReLU, GELU, SiLU, and all other activations used in practice
- 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

Cybenko, G. (1989). Approximation by Superpositions of a Sigmoidal Function. *Math. Control Signals Systems.*
Hornik, K. (1991). Approximation Capabilities of Multilayer Feedforward Networks. *Neural Networks.*

Proof Sketch (Cybenko, 1989)

- Let $\mathcal{S} = \operatorname{span}\{\mathbf{x} \mapsto \sigma(\mathbf{w}^\top \mathbf{x} + b) : \mathbf{w} \in \mathbb{R}^d,\ b \in \mathbb{R}\} \subset C(K)$. We want to show $\overline{\mathcal{S}} = C(K)$ (closure in the sup-norm)
- Proof by contradiction. Suppose $\overline{\mathcal{S}} \neq C(K)$. By the Hahn–Banach theorem, there exists a non-zero continuous linear functional Λ on $C(K)$ such that $\Lambda(g) = 0$ for all $g \in \mathcal{S}$
- By the Riesz representation theorem, Λ corresponds to a non-zero signed Borel measure μ on K:

$$\int_K \sigma(\mathbf{w}^\top \mathbf{x} + b)\, d\mu(\mathbf{x}) = 0 \quad \text{for all } \mathbf{w} \in \mathbb{R}^d,\ b \in \mathbb{R}$$

- Key step: Show that this forces $\mu = 0$, contradicting $\Lambda \neq 0$. For sigmoid, as $\lambda \to +\infty$:

$$\sigma(\lambda(\mathbf{w}^\top \mathbf{x} + b)) \longrightarrow \mathbf{1}_{\{\mathbf{w}^\top \mathbf{x} + b > 0\}}$$

  pointwise (except on the hyperplane $\mathbf{w}^\top \mathbf{x} + b = 0$). By dominated convergence, $\int \mathbf{1}_{H^+_{\mathbf{w},b}}\, d\mu = 0$ for every half-space $H^+_{\mathbf{w},b}$
- A signed measure that vanishes on all half-spaces must be zero (this follows from the uniqueness theorem for the Fourier transform of measures). Contradiction $\square$

Depth Efficiency

- The UAT tells us a *single* hidden layer suffices for universal approximation. A natural question: why do modern networks use many layers instead of one wide layer?
- Answer: A single layer may need *exponentially many* neurons. Depth allows the same function to be represented with *far fewer* parameters
- A concrete example (Telgarsky, 2016): Consider a ReLU network of depth L with $O(1)$ neurons per layer (so $O(L)$ parameters total). It computes a "sawtooth" function by composing L tent maps:

$$f_1(x) = \min(2x,\, 2 - 2x), \qquad f_L = f_1 \circ f_1 \circ \cdots \circ f_1 \;\; (L \text{ times})$$

  The function f_L oscillates 2^L times on $[0, 1]$. Any shallow (depth $\leq L/3$) network that approximates f_L needs $\Omega(2^{L/6})$ neurons
- Why does composition help? Each layer can *double* the complexity of the function (e.g. double the number of oscillations). After L layers, the complexity is $\sim 2^L$. A single layer must explicitly represent all 2^L features with individual neurons
- Takeaway: Depth provides exponential compression. This is the theoretical justification for the deep architectures used in Transformers (GPT-3 uses $L = 96$ layers)

Telgarsky, M. (2016). Benefits of Depth in Neural Networks. *COLT*.

Gradient Descent

- 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): Move θ in the direction of steepest decrease of $\mathcal{L}$:

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

  where $\eta > 0$ is the learning rate (step size)
- Why this works: By a first-order Taylor expansion, $\mathcal{L}(\theta_k - \eta \mathbf{g}) \approx \mathcal{L}(\theta_k) - \eta \|\mathbf{g}\|^2$ where $\mathbf{g} = \nabla_\theta \mathcal{L}(\theta_k)$. So the loss decreases at each step (for η small enough)
- Problem: For a training set of N examples, computing the full gradient requires a pass over all N examples:

$$\nabla_\theta \mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \ell_i(\theta)$$

  For GPT-3 with $N \approx 300 \times 10^9$ tokens, a single gradient step would be astronomically expensive
- The learning rate η is critical:
  - Too large: the update overshoots and the loss diverges
  - Too small: convergence is extremely slow
  - Typical practice: start with a "warm-up" schedule, then decay η over training (e.g. cosine annealing)

Stochastic Gradient Descent

- Key idea: Replace the full gradient (over all N examples) with a cheap, *noisy* estimate from a random mini-batch $\mathcal{B} \subset \{1, \ldots, N\}$:

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

- Why this is valid: The mini-batch gradient is an *unbiased* estimator of the full gradient:

$$\mathbb{E}_{\mathcal{B}} \left[\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell_i(\theta) \right] = \nabla_\theta \mathcal{L}(\theta)$$

  So SGD follows the true gradient *on average*, with variance $\propto 1/|\mathcal{B}|$

- Typical mini-batch sizes: $|\mathcal{B}| = 32$–4096, depending on hardware. Each step uses only $|\mathcal{B}|/N$ of the data — for GPT-3, this is a factor of $\sim 10^5$–10^7 cheaper than full GD
- The noise is actually beneficial: SGD noise helps escape sharp local minima and saddle points, biasing the optimiser towards *flatter* regions of the loss landscape that tend to generalise better (Keskar et al., 2017)

Keskar, N. S. et al. (2017). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *ICLR*.

Backpropagation and Modern Optimisers

- Backpropagation (Rumelhart et al., 1986): An efficient algorithm to compute $\nabla_\theta \mathcal{L}$ by applying the chain rule layer by layer, from the output back to the input
- Concrete example (2-layer network, $\hat{y} = W_2\sigma(W_1\mathbf{x})$, loss $\ell = L(\hat{y}, y)$):

$$\frac{\partial \ell}{\partial W_2} = \frac{\partial \ell}{\partial \hat{y}} \cdot \sigma(W_1\mathbf{x})^\top \qquad \text{(outer layer)}$$

$$\frac{\partial \ell}{\partial W_1} = \underbrace{\left(W_2^\top \frac{\partial \ell}{\partial \hat{y}}\right)}_{\text{error propagated back}} \odot \sigma'(W_1\mathbf{x}) \cdot \mathbf{x}^\top \qquad \text{(inner layer)}$$

  Each layer reuses the error signal from the layer above — no redundant computation

- Cost: $O(|\theta|)$ per sample — the same order as a forward pass. This efficiency is what makes training networks with billions of parameters feasible
- Adam (Kingma & Ba, 2015): The standard optimiser for LLMs. Maintains per-parameter running averages of the gradient (momentum m_t) and squared gradient (variance v_t):

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \quad \theta_t = \theta_{t-1} - \eta \frac{m_t}{\sqrt{v_t} + \epsilon}$$

  Parameters with large gradients get smaller effective learning rates, and vice versa

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning Representations by Back-propagating Errors. *Nature*, 323, 533–536.
Kingma, D. P. & Ba, J. (2015). Adam: A Method for Stochastic Optimization. *ICLR*.

Generalisation: The Classical View

- The fundamental question: A model trained on N examples achieves low *training error*. Will it also perform well on *unseen data* (test error)?
- Classical answer (Vapnik, 1998): With high probability,

$$\underbrace{R(f)}_{\text{test error}} \leq \underbrace{\hat{R}_N(f)}_{\text{training error}} + \underbrace{O\left(\sqrt{\frac{d_{\text{VC}}}{N}}\right)}_{\text{complexity penalty}}$$

  where d_{VC} is the VC dimension (a measure of model complexity — roughly, the number of data points the model can shatter)
- Prescription: Choose a model complex enough to fit the data, but not so complex that the complexity penalty dominates. This is the bias-variance tradeoff:
  - Too simple (high bias): cannot fit the training data well
  - Too complex (high variance): fits training data but memorises noise
  - Optimal: somewhere in between
- The puzzle: For a neural network with $|\theta|$ parameters, $d_{\text{VC}} \geq |\theta|$. GPT-3 has $|\theta| = 175 \times 10^9$, so the classical bound predicts *catastrophic* overfitting. Yet GPT-3 generalises extremely well. Something is missing from this picture

Vapnik, V. N. (1998). *Statistical Learning Theory*. Wiley.

The Double Descent Phenomenon

- Belkin et al. (2019) showed that the classical U-shaped test error curve is only *half the story*. As model complexity grows past the interpolation threshold (the point where the model first fits the training data perfectly), test error *decreases again*:
  1. Underparameterised regime ($|\theta| < N$): Classical bias-variance tradeoff. Test error follows a U-shape
  2. Interpolation threshold ($|\theta| \approx N$): The model barely fits the training data, and is forced into a "spiky" function that interpolates every point, including noise. Worst generalisation
  3. Overparameterised regime ($|\theta| \gg N$): Many different functions interpolate the training data. Among these, SGD implicitly selects one with *low complexity* (e.g. small norm), leading to good generalisation
- Why does SGD find "good" interpolators? This is implicit regularisation:
  - For linear models, GD provably converges to the minimum-norm solution
  - For neural networks, SGD + early stopping + small learning rate biases towards "flat" minima in the loss landscape, which correlate with better generalisation (Keskar et al., 2017)
- Implications for LLMs: Modern LLMs operate deep in the overparameterised regime ($|\theta| \gg N$ in many senses). The classical VC bound is vacuous, but double descent explains why they generalise: there are many interpolating solutions, and the training procedure selects a good one

Belkin, M. et al. (2019). Reconciling Modern Machine Learning Practice and the Bias-Variance Trade-off. *PNAS*.
Keskar, N. S. et al. (2017). On Large-Batch Training for Deep Learning. *ICLR*.

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* $\tanh$*)*

- 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
- 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:
  - 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

Bengio, Y., Simard, P. & Frasconi, P. (1994). Learning Long-term Dependencies with Gradient Descent is Difficult. *IEEE Trans. Neural Networks*.

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.

Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735–1780.

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
- 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:
  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)

Vaswani, A. et al. (2017). Attention Is All You Need. *NeurIPS*.

Token Embeddings

- Each token $x_t \in \mathcal{V}$ is an integer (an index into the vocabulary). A neural network needs continuous vectors, not integers
- Solution: Learn an embedding matrix $E \in \mathbb{R}^{|\mathcal{V}| \times d}$ where row x_t is the d-dimensional representation of token x_t:

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

- What does this look like? For $|\mathcal{V}| = 50{,}000$ and $d = 768$ (GPT-2), E is a $50{,}000 \times 768$ matrix. Looking up token 4217 simply extracts row 4217 — this is a differentiable operation, so the embeddings are learned end-to-end via backpropagation
- Key property: Tokens with similar roles in language end up with similar embedding vectors (e.g. "Monday" $\approx$ "Tuesday", "cat" $\approx$ "dog"). This is because the loss function rewards the model for making similar predictions in similar contexts

The Need for Positional Information

- Problem: An RNN processes tokens one at a time ($\mathbf{h}_1 \to \mathbf{h}_2 \to \cdots$), so position is implicit in the order of computation. The Transformer processes all positions *simultaneously* — it sees the set $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_T\}$ but has no way to know which came first
- Consequence: Without position information, the Transformer cannot distinguish:
  - "the cat sat on the mat" from "mat the on sat cat the"
  - "Alice paid Bob" from "Bob paid Alice"

  These have identical token embeddings (just reordered) but completely different meanings
- Solution: Add a positional encoding vector $\mathbf{p}_t \in \mathbb{R}^d$ to each token embedding:

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

  Now $\mathbf{x}_t$ carries both *what* the token is (via $\mathbf{e}_t$) and *where* it is (via $\mathbf{p}_t$)
- Design question: What should $\mathbf{p}_t$ be? We need a different vector for each position, and ideally the encoding should capture the notion that position 5 is "close to" position 6 but "far from" position 500

Sinusoidal Positional Encodings

- Analogy with binary numbers. Consider how we write integers in binary:

Position	bit 3	bit 2	bit 1	bit 0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0

  Bit 0 flips every step, bit 1 every 2 steps, bit 2 every 4 steps. Each bit operates at a different frequency. Together, 4 bits uniquely encode 16 positions

- Sinusoidal PE replaces these discrete bits with *smooth* oscillations. Each dimension pair $(2i, 2i+1)$ uses a sine and cosine at frequency ω_i:

$$\mathrm{PE}(\mathrm{pos}, 2i) = \sin(\omega_i \cdot \mathrm{pos}), \qquad \omega_i = \frac{1}{10000^{2i/d}} \tag{1.24}$$

$$\mathrm{PE}(\mathrm{pos}, 2i+1) = \cos(\omega_i \cdot \mathrm{pos}) \tag{1.25}$$

- Dimension $i = 0$: $\omega_0 = 1$ (fastest oscillation, distinguishes consecutive positions). Dimension $i = d/2 - 1$: $\omega_{d/2-1} \approx 1/10000$ (slowest, distinguishes positions thousands of steps apart). The $d/2$ frequencies together give each position a *unique fingerprint*

Sinusoidal PE: Relative Positions

- A crucial property: The encoding of position $\mathrm{pos} + k$ can be obtained from the encoding of position pos by a *fixed linear transformation* that depends only on the offset k:

$$\begin{pmatrix} \sin(\omega_i(\mathrm{pos}+k)) \\ \cos(\omega_i(\mathrm{pos}+k)) \end{pmatrix} = \underbrace{\begin{pmatrix} \cos(\omega_i k) & \sin(\omega_i k) \\ -\sin(\omega_i k) & \cos(\omega_i k) \end{pmatrix}}_{\text{rotation by angle } \omega_i k} \begin{pmatrix} \sin(\omega_i\,\mathrm{pos}) \\ \cos(\omega_i\,\mathrm{pos}) \end{pmatrix}$$

  This is just the angle addition formula from trigonometry

- Why this matters: The rotation matrix depends only on k (the offset), *not* on pos. So the relationship "3 positions ahead" looks the same whether we are at position 10 or position 1000. The model can learn relative positional patterns (e.g. "the word 2 positions to my left is likely an adjective") via its linear projections W_Q, W_K
- Alternatives in modern LLMs:
  - Learned embeddings (GPT-2): a trainable vector per position. Simple but cannot extrapolate beyond the maximum training length
  - RoPE (Su et al., 2021): applies the rotation directly to Q, K rather than adding to embeddings. Used in LLaMA, Mistral, and most modern LLMs
  - ALiBi (Press et al., 2022): no positional encoding at all; instead adds a linear bias $-m|i-j|$ to attention scores, penalising distant positions

Su, J. et al. (2021). RoFormer: Enhanced Transformer with Rotary Position Embedding. arXiv:2104.09864.
Press, O., Smith, N. & Lewis, M. (2022). Train Short, Test Long: Attention with Linear Biases. *ICLR*.

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.

- 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
- 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)
- 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

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
- 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 token's representation $\mathbf{x} \in \mathbb{R}^d$ is independently processed by a two-layer 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 $W_1 \in \mathbb{R}^{d_{ff} \times d}$ *expands* the dimension from d to d_{ff}, the ReLU introduces nonlinearity, and $W_2 \in \mathbb{R}^{d \times d_{ff}}$ *projects back* to d. Typically $d_{ff} = 4d$ (e.g. $d = 768 \to d_{ff} = 3072$ in GPT-2)

- Why is this needed? Attention computes weighted averages of value vectors — this is a *linear* operation (for fixed attention weights). Without the FFN, stacking attention layers would still only produce linear combinations. The FFN introduces the nonlinearity that gives the Transformer its expressive power
- Division of labour:
  - Attention: moves information *between* positions ("look at what other tokens said")
  - FFN: transforms information *within* each position ("process what I've gathered")
- Interpretation (Geva et al., 2021): The rows of W_1 act as *pattern detectors* (keys), and the columns of W_2 store associated *output updates* (values). The FFN is thus a key-value memory: $W_1\mathbf{x}$ computes a matching score against each pattern, ReLU selects the active patterns, and W_2 retrieves the corresponding information

Geva, M. et al. (2021). Transformer Feed-Forward Layers Are Key-Value Memories. *EMNLP*.

FFN: Modern Variants

- The original FFN (1.30) uses ReLU, which zeros out roughly half the hidden units. Modern LLMs replace it with gated variants that give the network more control over information flow
- GLU (Gated Linear Unit): Let the network learn *which* parts of the expanded representation to keep, using a sigmoid gate:

$$\mathrm{GLU}(\mathbf{x}) = (W_1\mathbf{x}) \odot \sigma(W_3\mathbf{x})$$

  Here $W_1\mathbf{x}$ is the "content" and $\sigma(W_3\mathbf{x}) \in (0,1)^{d_{ff}}$ is a learned gate that decides how much of each dimension to pass through ($\odot$ is element-wise multiplication)
- SwiGLU (Shazeer, 2020): Replaces the sigmoid gate with Swish:

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

  Note: SwiGLU uses *three* weight matrices (W_1, W_2, W_3) instead of two, so d_{ff} is typically reduced from $4d$ to $\frac{8}{3}d$ to keep the parameter count comparable
- In practice: SwiGLU consistently outperforms ReLU FFN and is the standard in LLaMA, PaLM, Mistral, and most modern LLMs
- Parameter cost: The FFN is the most parameter-heavy component. In GPT-3 ($d = 12288$, $d_{ff} = 49152$, $L = 96$), the FFN layers alone account for $\sim$2/3 of the total 175B parameters

Shazeer, N. (2020). GLU Variants Improve Transformer. arXiv:2002.05202.

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

He, K. et al. (2016). Deep Residual Learning for Image Recognition. *CVPR*.

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)
  - 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): $\mathbf{x} \leftarrow \mathbf{x} + \mathrm{SubLayer}(\mathrm{LayerNorm}(\mathbf{x}))$
- Modern variants: RMSNorm (Zhang & Sennrich, 2019) drops the mean subtraction and uses only root-mean-square normalisation — slightly faster, used in LLaMA

Zhang, B. & Sennrich, R. (2019). Root Mean Square Layer Normalization. *NeurIPS*.

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, $d_k = d/h$
- 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:
- 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 (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

Katharopoulos, A. et al. (2020). Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. *ICML*.

Alternatives to Attention: Linear Attention and SSMs

- State space models (SSMs): Replace attention with a linear dynamical system $\mathbf{h}_t = \bar{A}\,\mathbf{h}_{t-1} + \bar{B}\,x_t$, $y_t = C\,\mathbf{h}_t$. This is a linear RNN: $O(T)$ at inference, $O(T \log T)$ training via convolution + FFT
- Mamba (Gu & Dao, 2024): makes the SSM parameters *input-dependent* (selective), enabling content-based reasoning like attention while keeping $O(T)$ cost

	Softmax Attention	Linear Attention	SSMs (Mamba)
Training cost	$O(T^2 d)$	$O(Td^2)$	$O(TNd)$
Inference (per step)	$O(td)$ + KV-cache	$O(d^2)$	$O(Nd)$
Memory at inference	$O(T \cdot d)$ KV-cache	$O(d^2)$ fixed	$O(Nd)$ fixed
Content-based	Yes (softmax)	Approximate	Yes (selective)
Proven at scale	Yes (GPT-4, etc.)	Limited	Emerging

- Current landscape: Transformers dominate large-scale LLMs. SSMs are promising for very long contexts. Hybrid architectures (Jamba, Zamba) interleave both

Katharopoulos, A. et al. (2020). Transformers are RNNs. *ICML*.
Gu, A. et al. (2022). Structured State Spaces. *ICLR*.
Gu, A. & Dao, T. (2024). Mamba. *ICLR*.

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

Yun, C. et al. (2020). Are Transformers Universal Approximators of Sequence-to-Sequence Functions? *ICLR*.

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)
- 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

Kaplan, J. et al. (2020). Scaling Laws for Neural Language Models. arXiv:2001.08361.

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

- Shape vs. size: Consider two models, both with $N = 100$M parameters:
  - Model A: 12 layers, width 768, 12 heads ("GPT-2 small" shape)
  - Model B: 6 layers, width 1024, 16 heads (wider and shallower)

  Kaplan et al. found that A and B achieve *nearly identical* test loss. The *total* parameter count N is what determines performance — how those parameters are distributed across depth, width, and heads matters surprisingly little. This justifies measuring progress by N alone
- Larger models learn faster per token: A $10\times$ larger model reaches the same loss with $\sim 10\times$ fewer training tokens. Concretely: a 1B-parameter model trained on 10B tokens can match a 100M-parameter model trained on 100B tokens
  - But the 1B model costs $10\times$ more FLOPs *per token* (since each forward/backward pass scales with N). So the total compute $C \approx 6ND$ is similar in both cases — you trade data for model size at roughly constant compute
- Remarkable range of validity: 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 regime. It suggests the power laws reflect something fundamental about the structure of natural language

Joint Scaling and Predictability

- In practice, both N and D are limited. Kaplan et al. proposed a joint scaling law that captures both bottlenecks:

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

- Reading this formula: Think of it as $L \approx (\text{model bottleneck} + \text{data bottleneck})^{\alpha_D}$. Performance is limited by whichever bottleneck is worse. If you have a huge model but little data, the D_c/D term dominates. If you have abundant data but a small model, the N_c/N term dominates. Only when both are large does L become small
- The practical payoff — predicting before training:
  1. Train a family of small models: 10M, 50M, 100M, 500M parameters
  2. Each costs very little. Fit α_N, α_D, N_c, D_c to their test losses
  3. Extrapolate: predict the loss of a 70B or 175B model *before spending a single GPU-hour on it*

  This saves millions of dollars — you know in advance which configurations are worth training. GPT-4's performance was reportedly predicted accurately this way (OpenAI, 2023)

The Compute Budget and the Chinchilla Question

- Training costs $C \approx 6ND$ FLOPs. The factor of 6: forward pass $\approx 2N$ FLOPs/token (one multiply-add per weight), backward pass $\approx 4N$ (gradients w.r.t. activations and weights), total $6N$ per token $\times$ D tokens
- Concrete scale (GPT-3): $6 \times 175\text{B} \times 300\text{B} = 3.15 \times 10^{23}$ FLOPs. On 1024 A100 GPUs: $\sim$10 days, $\sim$\$4–12M
- The central question: Given a fixed budget C, how should we split it between model size N and data $D = C/(6N)$?
  - Kaplan et al. (2020): Scale N faster than D. They found $N_{\text{opt}} \propto C^{0.73}$, $D_{\text{opt}} \propto C^{0.27}$ — spend most of your budget on a bigger model
  - Hoffmann et al. (2022): This was wrong. Using a refined loss model $L(N, D) = E + A/N^{\alpha} + B/D^{\beta}$ (where E is the irreducible entropy of language, A/N^{α} is approximation error, B/D^{β} is estimation error), they showed via Lagrange multipliers that $N_{\text{opt}} \propto C^{0.50}$, $D_{\text{opt}} \propto C^{0.50}$ — model size and data should be scaled equally

Hoffmann, J. et al. (2022). Training Compute-Optimal Large Language Models. *NeurIPS*.

The Chinchilla Result

- The punchline: Most large models before 2022 were *significantly undertrained* — too many parameters, too little data. The Chinchilla-optimal ratio is $\sim$20 tokens per parameter:

Model	N (params)	D (tokens)	D/N	Status
GPT-3 (2020)	175B	300B	1.7	heavily undertrained
Gopher (2021)	280B	300B	1.1	heavily undertrained
Chinchilla (2022)	70B	1.4T	20	compute-optimal
LLaMA (2023)	65B	1.4T	22	$\approx$ optimal
LLaMA-3 (2024)	70B	15T	214	overtrained

- Gopher (280B) and Chinchilla (70B) used the *same compute budget* ($\sim 5 \times 10^{23}$ FLOPs), but Chinchilla outperformed Gopher on nearly every benchmark — a $4\times$ smaller model won by seeing $4.7\times$ more data
- Beyond Chinchilla: LLaMA-3 trains far past the optimal ratio ($D/N = 214$). Why? At deployment, inference cost scales with N (every user query runs through the full model), but D is a one-time training cost. A *smaller, overtrained* model is cheaper to serve

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