0. CONTENTS

1. The Attention Mechanism

The attention mechanism is a relatively recent type of neural network layer that has attracted significant interest, particularly for sequence-to-sequence tasks.

- *What is attention?* At its core, attention is a dynamically calculated weighted average. It allows the model to focus on different parts of the input sequence when producing an output, effectively capturing relationships between different elements.

> **Definition 1**
> The **attention mechanism** describes a weighted average of (sequence) elements with dynamically computed weights.

1.1 History

The "Attention Is All You Need" paper by Vaswani et al. (2017)[1] popularized attention in deep learning, but the concept itself has older roots.

- *Early inspiration.* The idea of "attention" has its roots in cognitive science, where it refers to selectively focusing on certain aspects of information while ignoring others. This concept influenced pre-deep learning work in machine learning and natural language processing.
- *Introduction of Attention in Deep Learning.* The attention mechanism was first introduced in neural machine translation by Bahdanau et al. (2014) in their "Neural Machine Translation by Jointly Learning to Align and Translate" paper[2] .
  - This work addressed limitations of earlier encoder-decoder architecture by allowing the decoder to "attend" to different parts of the source sentence when generating each word of the target sentence.
  - This allows the neural network to effectively capture long-range dependencies.
- *Concurrent explorations in computer vision.* Around the same time (in 2015), researchers began exploring attention in computer vision tasks like image captioning, where the model needed to focus on different parts of an image when generating a descriptive caption.
- *Growing popularity.* During this period, various forms of attention were developed and applied to different tasks, including:
  - Soft attention: Assigns weights to all parts of the input.
  - Hard attention: Selects only a single part of the input
  - Local attention: Focuses on a small window of the input.
- *"Attention Is All You Need" paper.* Vaswani et al. (2017) introduced the Transformer architecture, which relies entirely on attention mechanisms, replacing recurrence (RNNs) and convolutions (CNNs). Key innovations include:

- Self-attention: Allows the model to attend to different parts of the same sequence.
- Multi-head attention: Use multiple attention mechanisms in parallel

This led to significant improvements in performance and efficiency, particularly in NLP tasks, and sparked widespread adoption of attention

- *Modern day.* Since the introduction of Transformers, attention has become a fundamental building block in many deep learning models. Current research focuses on:
  - Sparse attention: reduce the computational cost by only computing attention scores for a subset of all possible pairs
  - Linear attention aims to reduce the computational complexity from quadratic $\mathcal{O}(n^2)$ to linear $\mathcal{O}(n)$ with respect to the sequence length. It achieves this by changing the way attention scores are calculated.

    Example 1. Change the similarity function such that we use kernels instead of a scaled dot product.

  - Attention with inductive biases: incorporate prior knowledge into the attention mechanism.

Key Concept 1. Vaswani et al.'s "Attention Is All You Need" paper built upon prior work by **introducing self-attention and the Transformer architecture**.

Key Concept 2. The Transformer architecture relies exclusively on attention mechanisms.

1.2 What does self-attention accomplish?

Self-attention resolves language ambiguity by modeling relationships between words in a sentence. Consider Jay Alammar's example from the "The Illustrated Transformer" blog post:[3]

"The animal didn't cross the street because it was too tired."

Does "it" refer to "the animal" or "the street"? While humans intuitively understand the connection, pre-Transformer models struggled with this type of ambiguity. Self-attention allows the model to capture dependencies between words, establishing the link between "the animal" and "it."

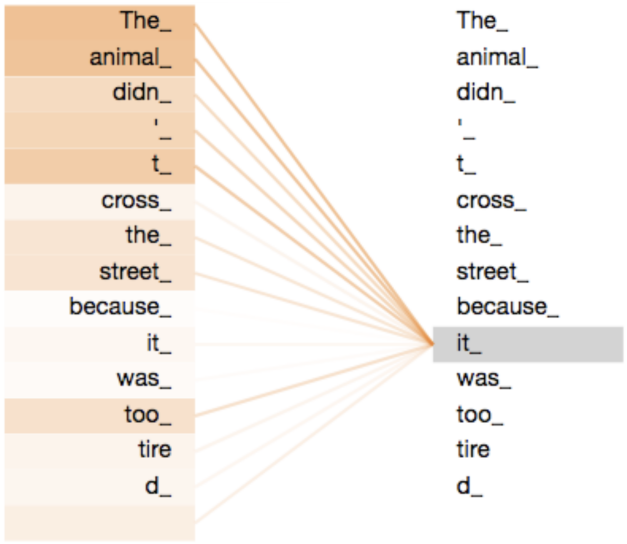


Figure 1: Attention models the dependencies between tokens, resolving language ambiguities (image credit).

Self-attention is a global communication mechanism[4] that update each token's embedding representation based on its surrounding context, which is the embedding representations of other tokens.[5]

- *Example.* In Figure 2-3, self-attention aggregates information about "fluffy", "blue", and "creature" to create a representation of a "fluffy blue creature".

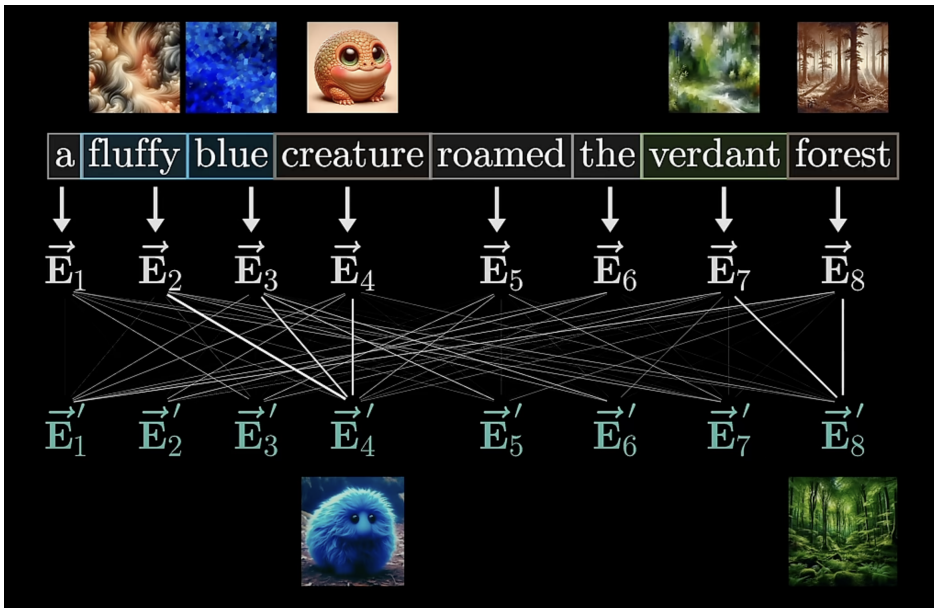


Figure 2: The attention mechanism updates each token's individual embedding representation to include information from its neighboring tokens (image credit).

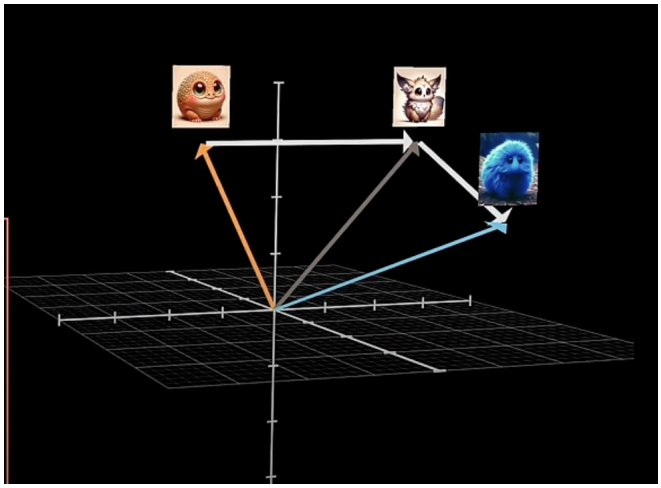


Figure 3: After training with self-attention, our "create" embedding gets updated in the direction of "blue" and "fluffy" in the embedding space. (image credit).

On a more granular level, self-attention calculates how important other tokens are to a given token. Each token's information is scaled against its importance score, and we add every token's scaled information to the given token embedding.[5]

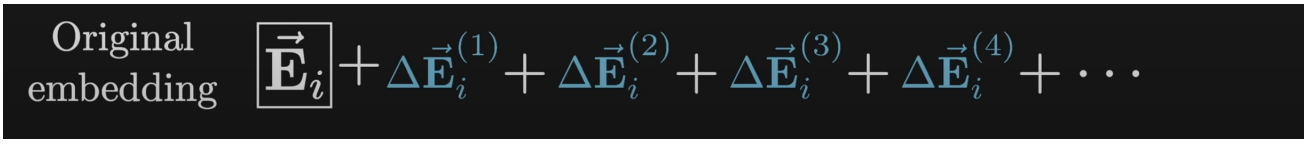


Figure 4: We scale and add all other tokens' information to the original token embedding (image credit).

- *Learning embedding representations.* In practice, we initialize our token embedding representations at random. During training, a Transformer uses self-attention to learn meaningful embedding representations for each token.

> **KEY CONCEPT 3.** Transformers learn the correct token embedding representations on their own. Meaning, they require little manual feature engineering.

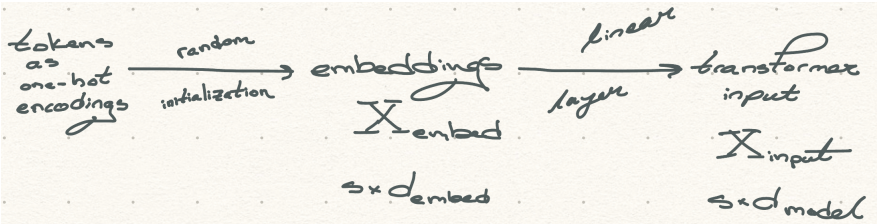


Figure 5: The Transformer learns meaningful token embeddings from a random initialization. Normally, $d_{\text{embed}} = d_{\text{model}}$ and the linear mapping isn't required.

1.3 How does self-attention work?

As a pre-processing step, we convert each input token into a dense embedding vector representation in these two steps:[4,6]

1. Represent the input token as one-hot vectors $X_{\text{raw}} \in \mathbb{R}^{s \times v}$, where $s :=$ sequence length and $v :=$ vocab size.

2. Map these token token representations to dense vector embeddings using a weight matrix to get $X_{\text{embed}} \in \mathbb{R}^{s \times d_{\text{embed}}}$, where $d_{\text{embed}} :=$ embedding hidden dimensionality.
3. The self-attention layer expects an input of $X_{\text{input}} \in \mathbb{R}^{s \times d_{\text{model}}}$, where $d_{\text{model}} :=$ the overall model dimensionality.
    - If $d_{\text{embed}} \neq d_{\text{model}}$, we need to map X_{embed} into the $\mathbb{R}^{s \times d_{\text{model}}}$ space.
    - Normally, $d_{\text{embed}} = d_{\text{model}}$. Thus, no linear mapping is required and $X_{\text{embed}} = X_{\text{input}}$.

This transformation of $X_{\text{raw}} \rightarrow X_{\text{input}}$ only needs to be done once during the forward pass, regardless of the number of self-attention layers.[5] We consider X_{input} to be the input for all self-attention layers, and require the self-attention layer output to also be in the $s \times d_{\text{model}}$ dimensional space.

In the self-attention layer, X_{input} is projected into three feature vectors: Query (Q), Key (K), and Value (V). Meaning, each input element has a Q, K, and V feature vector.[7]

- **Query.** Represents what we're "looking for" in the sequence. In other words, what we want to pay attention to.
- **Keys.** Describes **(a)** what each element "offers", and **(b)** when it might be relevant.
- **Values.** Contains the actual information content of the element, which we will average based on attention weights.

The Q and K matrices are projected into the same-sized dimensional space of $s \times d_k$, where s represents the sequence length (number of input tokens) and d_k the attention head's hidden dimension size. Normally, $d_k \ll d_{\text{model}}$. V can have a different size than the K and Q, but it's usually the same size.

Each feature space has a different learned meaning.

- *Learned meanings.* The meanings of Q, K, and V are learned through backpropagation.
    - Initially, the weight matrices for the Q, K, and V projections are randomly initialized, making them random transformations.
        - ➜ At a high-level, a Transformer is simply predicting the next token in a given sequence. During training, we know what the next token should be. Thus, training becomes a m-class prediction problem, where m represents our (token) vocab size.
        - ➜ We use cross entropy as our objective function, where we compare the predicted next token against its ground truth.
        - ➜ The loss gradients update the weight matrices W^Q, W^K, and W^V (among other model parameters), which are used to generate the q, k, v feature vector projections.
    - The model uses the **score function** f_{attn} to calculate how much a key answers a query.
- *Scoring function.* The score function takes a query and a key as input, and outputs the attention weight of each query-key pair.

- *Implementation.* The scaled dot-product attention (Equation 1) is the most common scoring function used in Transformers. In this context, "attention" normally refers to the "scaled dot-product attention".

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

For a dot product to be possible, the key and query vectors must have the same dimensionality. We refer to this dimensionality as d_k.

➜ *Softmax.* The softmax function transforms a vector of real numbers into a probability distribution. The output is a vector where: **(a)** each number is between 0 and 1, and **(b)** the sum of all numbers equals 1.

➜ *Dot product.* QK^T calculates the similarity between each query-key pair. In particular, this is a matrix of dot products.

➜ *Scaling* $\left(\sqrt{\frac{1}{d_k}}\right)$. We initialize our layers such that Q, K, and V have roughly equal variance. However, the dot product over two vectors with a variance of σ^2 results in a scalar with the variance of $\sigma^4 \cdot d_k$.

If we don't scale the variance back to $\sim \sigma^2$....

1. *One-hot encoding weighted averages.* The softmax will saturate to 1 for one random element and 0 for all others.
2. *Model doesn't learn.* The gradients through the softmax will be close to zero, making it impossible to learn the parameters.

KEY CONCEPT 4. Attention mechanisms differ in their definitions of queries, keys, values, and the score function.

There are various ways to visualize scaled dot-product attention. Figure 6 shows the original visualization from the "Attention Is All You Need Paper" paper.[1]

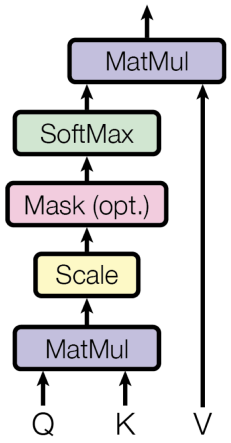


Figure 6: The "Attention Is All You Need" paper visualizes of the scaled dot product calculation (image credit).

Alternatively, Jay Alammar visualizes the projection of the word embedding and subsequent calculations as:[3]

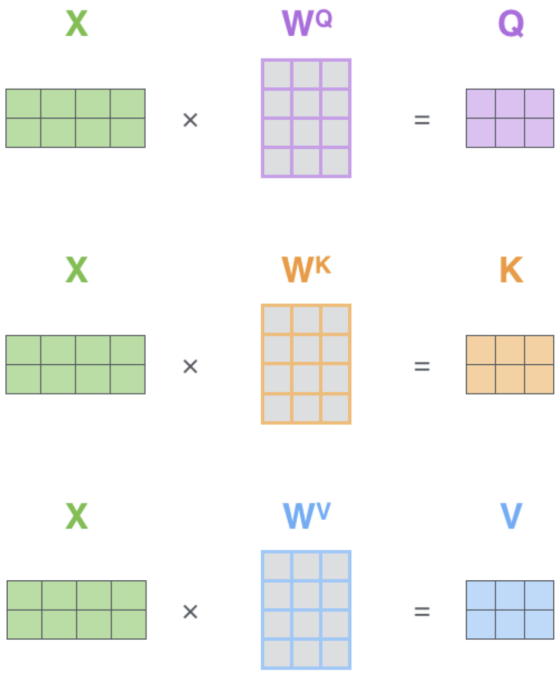


Figure 7: The $X_{\text{input}} \in \mathbb{R}^{s \times d_{\text{model}}}$ represents token embeddings projected into the overall model hidden dimension d_{model}. Using learned weight matrices W^Q, W^K, and $W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$, we project X_{input} into $Q, K, V \in \mathbb{R}^{s \times d_k}$ (image credit).

Once we've computed the Q, K, and V matrices, we calculate the scaled dot-product attention:

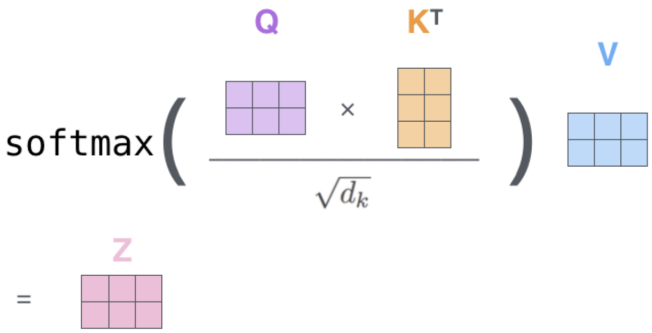


Figure 8: We calculate the outputs of a self-attention layer (matrix Z) by using Equation 1 (image credit).

Finally, we apply a linear layer on the resulting average value features, $\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$, such that our attention layer's output dimensions match those of the original $X_{\text{input}} \in \mathbb{R}^{s \times d_{\text{model}}}$ input.

> ⚠ The self-attention block is not complete without a final linear layer projection into the $\mathbb{R}^{s \times d_{\text{model}}}$ space. Otherwise, we cannot add the self-attention layer inputs back into the residual stream. (See Section 2.1.)

1.4 Multi-Head Attention

Standard scaled dot-product attention produces a single weighted average, focusing on one aspect of the input sequence. However, language (and other sequential inputs) often has multiple aspects

that we should attend to simultaneously.

- *How does each attention head work?* Multi-head attention, also introduced in the "Attention Is All You Need" paper, addresses this by processing multiple Q, K, V triplets in parallel. Specifically, we compute h different linear projections to transform the same input embeddings into h sets of Q, K, and V matrices.[1]

  > **KEY CONCEPT 5.** Each attention head models a different relationship between input elements. Thus, the feature spaces for each set of Q, K, and V represent a different meaning.[4]

  > **KEY CONCEPT 6.** Multi-head attention allows us to capture different contexts across the same sequence **in parallel**, which is essential for properly modeling natural language.[5]

  Each of these sets is then used in a separate scaled dot-product attention calculation ("head").

- *Aggregating results.* The results of the attention heads are then concatenated along a new dimension (Equation 2). A neural linear layer W_O then projects the concatenated output back to the expected input size for subsequent layers.[7]

$$\text{Multihead}(Q, K, V) = \text{Concat}\left(\text{head}_1, ..., \text{head}_h\right) W_O \tag{2}$$

  - *Implementation.* In practice, increasing d_k beyond a certain point yields diminishing returns in terms of model performance. Therefore, a common approach (based on empirical evidence) is to define $d_k = \frac{d_{\text{model}}}{h}$.
    In this case, $\text{Multihead}(Q, K, V) \in \mathbb{R}^{s \times (h \times d_k)}$. We then use W_O to project $\text{Multihead}(Q, K, V)$ back into the final output space $\mathbb{R}^{s \times d_{\text{model}}}$.

- *Visualizations.* Vaswani et al. visualize multi-head attention as a stack of scaled dot-product attention layers followed by this linear projection (Figure 9).[1] Figure 10 visualizes the matrix operations.[3]

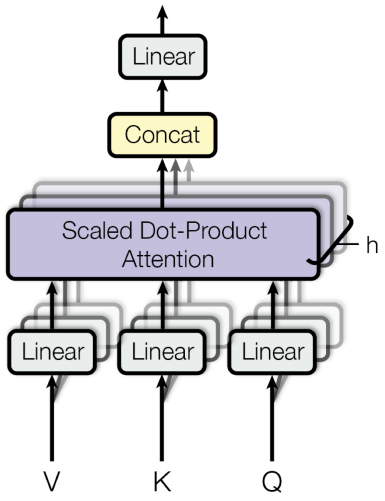


Figure 9: We visualize multi-head attention as a stack of scaled dot product calculations, since we concatenate each head's results together (image credit).

1.5 Properties

The parallel and independent nature of each scaled dot-product attention calculation in multi-head attention results in the following key properties:

1. **Permutation equivariance.** The scaled-dot product attention operates on a global context,[4] where we compute each query-key comparison simultaneously (Equation 1). Hence, the attention score does not consider token position. Meaning, the attention operation is permutation-equivariant with respect to its input.[7]

   Note that the softmax operation is similarly permutation-equivariant.

   > **KEY CONCEPT 7.** A function $f(X_1, X_2, ..., X_n) = \langle Y_1, Y_2, ..., Y_n \rangle$ is permutation-equivariant, if changing the order of the input equally changes the order of the output, but the values of the outputs stay the same, i.e. $f(X_2, X_1, ..., X_i) = \langle Y_2, Y_1, ..., Y_i \rangle$.
   >
   > In contrast, a function $f(X_1, X_2, ..., X_n) = Y$ is permutation-invariant, if changing the order of the input does not impact the output in any way, i.e. it stays invariant: $f(X_2, X_1, ..., X_i) = f(X_1, X_2, ..., X_n) = Y$.

   EXAMPLE 2. Suppose that we have N people sitting in a room and they all change seats.

   - *Permutation invariant.* We count the number of people before and after they've changed seats. Since the total count doesn't change, counting is a permutation invariant operation.
   - *Permutation equivariant.* Persons A and B swap seats, $P_a \leftrightarrow P_b$. They also need to exchange name tags to reflect this change. Hence, name tag placement is permutation equivariant.

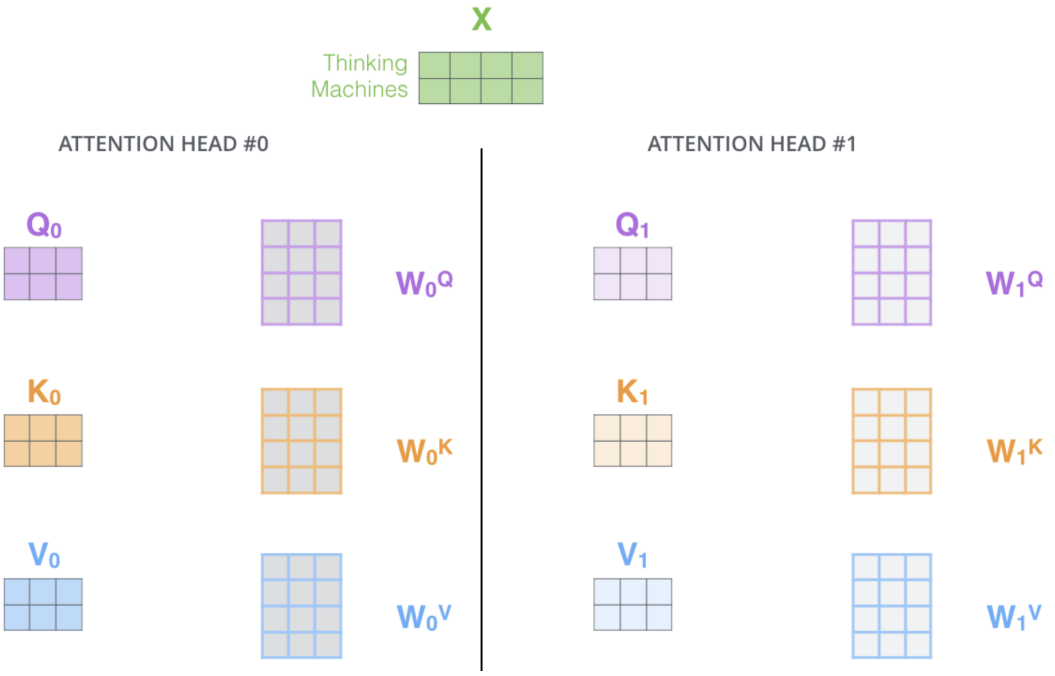
   - *Implications.* This permutation equivariance makes self-attention a global communication mechanism.[4] Thus, Transformers model the global context (including long-range dependencies), increasing their versatility.[5]
   - *Positional information.* If input order is important for a task, we need to encode an element's position as part of its input features.[7] See Section 1.6 for more details.

2. **Partial interpretability.** Self-attention offers some intuitive insight into the model's focus by highlighting relationships between words. However, the attention probabilities alone do not fully explain the model's decisions.[5]

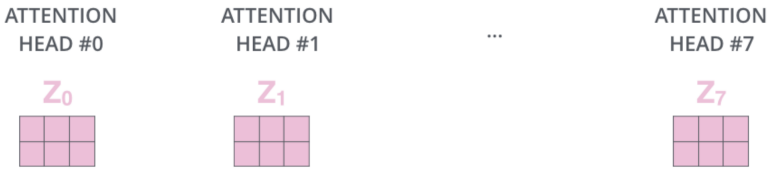
   They interact with other model components to produce the final output. These interacting factors include:

   - *Value vector scaling.* The value vectors, which are weighted by the attention probabilities, can have varying scales (magnitudes).
     ➜ *Implications.* A high attention weight might not translate to a large contribution to the final output if the corresponding value vector has a small magnitude.
   - *Sparsely activated weights.* Some weight dimensions within the attention mechanism (and other parts of the Transformer) can be sparsely activated, meaning they have little to no influence on the model's computations for most inputs.

(a) We maintain separate Q/K/V weight matrices $\in \mathbb{R}^{d_{\text{model}} \times d_k}$ for each head resulting in different Q/K/V matrices $\in \mathbb{R}^{s \times d_k}$.



(b) We have h head calculating different scaled dot product attentions $Z_i \in \mathbb{R}^{s \times d_h}$.



(c) We concatenate our separately calculated attention matrices together and project this matrix into space with the dimensions expected by the next feed forward neural network layer.

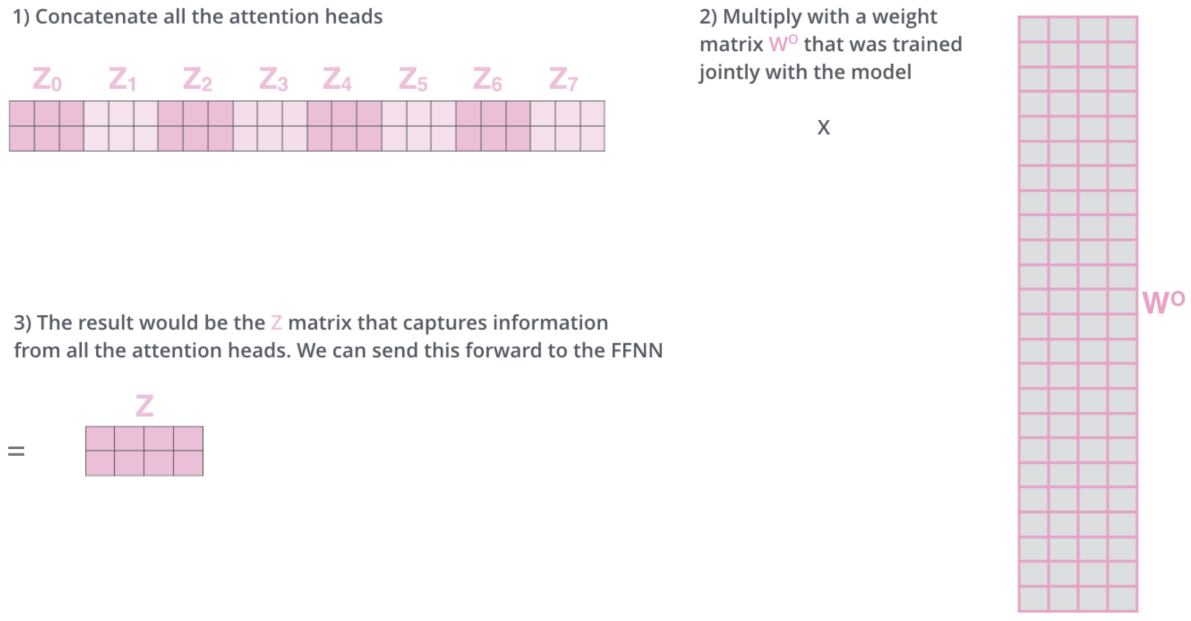


Figure 10: A visualization of multi-head attention matrix operations (image credit).

➜ *Implications.* This obscures the direct relationship between attention probabilities and the final output.

> ⚠ The attention weights, softmax $\left(\frac{QK^1}{\sqrt{d_k}}\right)$, are only partially interpretable. Factors like the magnitudes of V, dead attention heads; etc. also affect what the model attends to.

3. **Efficiency.** Unlike recurrent neural networks (RNNs), the self-attention layer runs entirely in parallel.[5] This makes Transformers much faster for shorter sequences, but less efficient for sequences longer than the hidden dimension size.[1]

   - One way to reduce the computational cost for long sequences is to restrict the self-attention to a specific neighborhood of inputs (r) to attend over. This is called **restricted self-attention**.
   - Making the Transformer architecture more efficient is an active area of research.

> ⚠ If the sequence length exceeds the hidden dimension size, then self-attention becomes more expensive than RNNs.

In the original attention paper, Vaswani et. al provided Table 1 to compare the Transformer architecture against other popular neural architectures.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table 1: Complexity of different popular neural architectures. For attention, n represents sequence length, d_k the attention head hidden dimension size, and r restricted neighborhood size. The most efficient architectures are highlighted in pink (table credit).

Here is what each column tells us:

(a) Complexity per Layer - The runtime complexity for each network layer

(b) Sequential Operations - How easy is it to parallelize the operations? $\mathcal{O}(1)$ means that everything can be parallelized (full parallelization).

(c) Maximum Path Length - Measures the number of operations needed to capture the full global context. This represents the communication cost between core operations (e.g., attention vs. convolution).

While Table 1 uses n to represent sequence length, we'll continue to using s for consistency. If we take a closer look at the runtime complexity of each neural architecture layer:

- *Self-attention.* The scaled attention dot product is defined as softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)V$. Meaning, the main operations are:

- Matrix multiplication (QK^T), which scales according **(a)** to the length and width of our matrices and **(b)** requires "sliding" over each row and column. Hence, the runtime complexity is cubic.
  - ➜ Generally, if matrix **A** has the dimensions $m \times n$ and matrix **B** has the dimensions $n \times p$, then $\mathbf{A} \times \mathbf{B} = \mathbf{C}$, where matrix **C** has the dimensions $m \times p$. Thus, the runtime complexity becomes $\mathcal{O}(m \cdot n \cdot p)$.
  - ➜ In this specific case, s is our sequence length and d_k is our key/query dimension. We multiply $Q \in \mathbb{R}^{s \times d_k}$ with $K^T \in \mathbb{R}^{d_k \times s}$ matrix to get a $n \times n$ matrix. Meaning, our complexity "bottleneck" is $\mathcal{O}\left(s^2 \cdot d_k\right)$.
- *Scaling.* Each element in the resulting $s \times s$ matrix is scaled by $\sqrt{\frac{1}{d_k}}$. Since this an element wise operation, its complexity of $\mathcal{O}(s^2)$.
- *Softmax.* The softmax function is then applied row-wise to this scaled matrix. Hence, its complexity is also $\mathcal{O}(s^2)$.

> **KEY CONCEPT 8.** Multiplying $\mathbf{A} \in m \times n$ by $\mathbf{B} \in n \times p$ has a complexity of $\mathcal{O}(m \cdot n \cdot p)$. This information allows us to infer a specific neural architecture's complexity w.r.t. a certain variable.

> **KEY CONCEPT 9.** Self-attention scales quadratically with sequence length ($\mathcal{O}(s^2)$). This is why it becomes less efficient than vanilla RNNs when the sequence length n exceeds the embedding/hidden dimension size d_k.

- *Restricted attention.* If we limit the attention mechanism for each token to a neighborhood of r tokens, the complexity becomes $\mathcal{O}(r \cdot s \cdot d_k)$, where key length is limited to r.
- *Linear Attention.* Linear attention is a simplified version of attention that removes the softmax. Since only matmuls are left, it is essentially a linear function, which one can reformulate in a recurrent form.
  - ➜ This makes it more efficient during inference. In particular, linear attention reduces the quadratic runtime complexity to linear time (over sequence length).
  - ➜ Unfortunately, linear attention has shown lower performance than the original attention mechanism. Thus, it's only used in cases where $s \gg d_k$ and model performance isn't critical.

> **KEY CONCEPT 10.** Linear self-attention is used in applications where **(a)** model accuracy isn't critical **and (b)** input sequence length often exceeds d_k.

- *RNNs.* For a vanilla RNN, the complexity is a function of the input sequence length s and the hidden state size d^2. The runtime complexity is $\mathcal{O}(s \cdot d^2)$.
  - ➜ Note that modern RNNs (e.g. Mamba/xLSTM) use variants with potentially different complexities, but remain linear in sequence length.
- *Convolutional neural networks.* Complexity is a function of the convolution kernel size k, the number of pixels p, the number of input channels d_{in}, and the number of output channels d_{out}. In many cases, $d_{in} = d_{out}$, and we can simplify to $\mathcal{O}(k \cdot n \cdot d^2)$.

The permutation equivariance and efficiency (when $s < d_k$) of Transformers lead to several key paradigm shifts in machine learning (Table 2).

Paradigm	Before Attention	After Attention
Processing	**Sequential processing.** RNNs process data sequentially (one step at a time). This makes them slow and difficult to parallelize, especially for long sequences.	**Parallel processing.** The underlying attention mechanism processes all elements of a sequence simultaneously. This results in faster inference times and better retrieval capabilities in longer sequences.
Context Scope	**Local context.** In Computer Vision, convolutional neural networks captured local patterns in data through convolutional filters. For NLP, RNNs captured some long-range dependencies, but struggled with very long sequences.	**Global context.** The attention mechanism models dependencies across all parts of an input sequence parts, regardless of distance.
Driven by...	**Feature engineering.** Traditional NLP often relies on manual feature engineering to represent words in a way that models could understand.	**Data.** Transformers learn rich representations directly from raw data, which reduces the need for manual feature engineering.
Generalizability	**Task-specific models.** Many AI models were designed for specific tasks, where different architectures were used for different applications.	**Foundation models.** The ability of Transformers to learn from massive datasets has led to the emergence of "foundation models" that are fine-tuned for a wide range of downstream tasks.

Table 2: Paradigm shifts introduced by the attention mechanism.

1.6 Positional Encodings

Because attention is permutation-equivariant, it processes input elements as a set. Meaning, the model has no inherent understanding of word position. Therefore, to effectively process sequential data, we must explicitly provide positional information to the model.[7]

The Transformer architecture achieves this through positional encodings, which are added to the input embeddings. These encodings are designed to provide a unique representation for each position in the sequence, allowing the model to distinguish between different word orderings.[6]

Vaswani et al. proposed using sinusoidal functions of different frequencies for these encodings:[1]

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{\text{model}}}}\right) & \text{if } i \mod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{\text{model}}}}\right) & \text{otherwise} \end{cases} \tag{3}$$

, where pos is the position in the input sequence, i is the dimension of the embedding vector, and d_{model} is the dimensionality of the embeddings.

> **KEY CONCEPT 11.** The positional encoding matrix PE is added element-wise to the embedding matrix X. Thus, each row X_1 (a token's embedding) becomes $X_1 + PE_i$.

The intuition is as follows behind Equation 3:

- *Varying frequencies encode relative position.* Sine and cosine functions are periodic. By using different frequencies (controlled by i and d_{model}), the encodings generate unique patterns for different positions. This allows the model to learn relative positions within the sequence.

- *Even vs. odd dimensions.* Sine is used for even dimensions (i is even), and cosine is used for odd dimensions (i is odd). This provides a distinct representation for each dimension.
- *Geometric progression of wavelengths.* The use of $10000^{i/d_{\text{model}}}$ creates a geometric progression of wavelengths. This means that wavelengths becomes increasingly "stretched out" across at higher levels of d_{model}, allowing the model to easily learn relationships between positions at different distances.

Putting all of this together, we want to have an unique sine/cosine wave for each dimensions where we slowly stretch out the wavelength (Figure 11).[7]

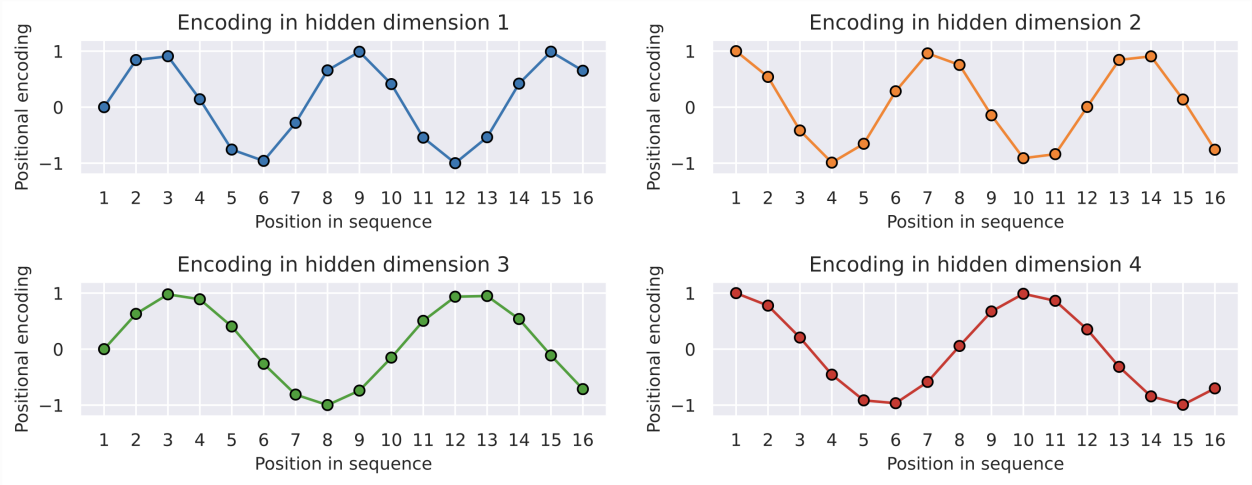


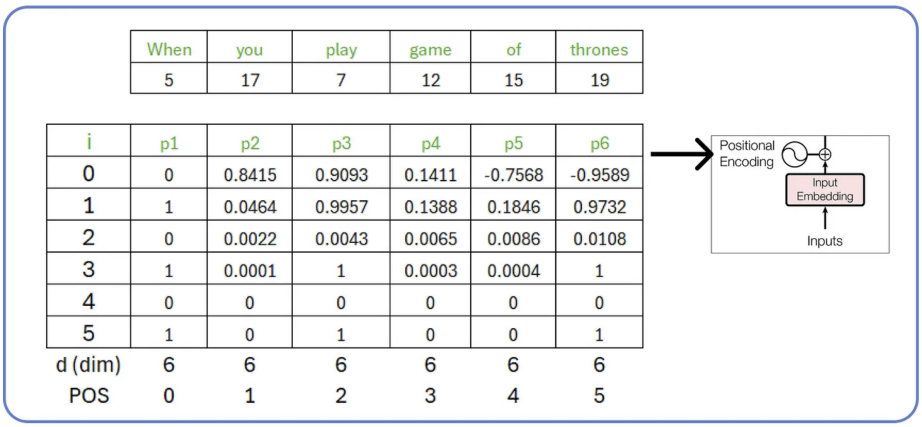
Figure 11: The patterns between the hidden dimension 1 and 2 only differ in the starting angle. The wavelength is 2π, hence the repetition after position 6. The hidden dimensions 2 and 3 have about twice the wavelength (image credit).

EXAMPLE 3. Suppose that our embedding size is 512. Then, we have $\frac{512}{2} = 256$ frequencies, since we use 256 unique sine and 256 unique cosine waves. Hence, we have 512 distinct positional encodings.

KEY CONCEPT 12. Since Transformers are permutation-equivariant, they require explicit positional information to understand word order. Positional encodings provide this crucial information, and are used during both training and inference.

Figure 12 visualizes the matrix math required to add positional encodings into our transformer layer input.[6]

(a) We calculate a PE matrix with the dimensions $s \times d_{\text{model}}$ using Equation 3, where s represents the sequence length and d_{model} the overall model hidden dimension size.



(b) We add matrix PE to our embedding matrix $X_{\text{input}} \in \mathbb{R}^{s \times d_{\text{model}}}$. Then, we feed the resulting matrix into our Transformer blocks.

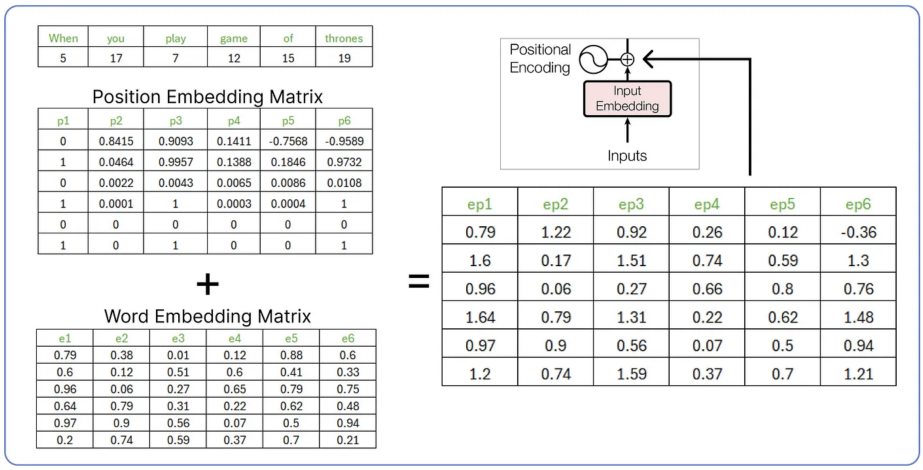


Figure 12: A visualization of computing an input sequence's positional encodings (image credit).

1.7 Training

Transformers are very deep neural networks. Meaning, like any other deep neural network, they are prone to overfitting and have a complex loss space.

1. **Optimization challenges.** Like many very deep layers, Transformers are sensitive to hyperparameters, especially the learning rate, and are susceptible to unstable gradients.

    - *Why?* Transformers lack recurrent memory and have a complex optimization landscape.

        (a) *Lack of recurrent memory.* Unlike RNNs, Transformers process all input tokens simultaneously. Meaning, they lack an inherent "memory" of previous steps to guide the learning process.

        ➜ *Implications.* Each layer receives the full input sequence at once. Therefore, the model's initial state is extremely influential. A poor initialization can easily trap the model in a suboptimal region of the loss landscape, making recovery difficult.

(b) *Complex optimization landscapes.* Transformers are often very deep. Meaning, the non-linearities from the attention mechanism and the numerous cross-layer interactions create a highly complex optimization landscape.

➜ *Implications.* Small changes in hyperparameters, especially the learning rate, can have a significant impact on the training dynamics and final performance.

- *Solutions.* We use learning rate warm-up when training Transformers alongside the Adam optimizer to better navigate the optimization landscape.[7]
  - *Intuition.* We gradually increase the learning rate from 0 to the target learning rate during the initial training iterations.
    - ➜ This allows the model to explore the loss space slowly at first, preventing large, potentially destabilizing jumps early on.
    - ➜ As training progresses and the model approaches a solution, the learning rate decreases, enabling finer adjustments.
  - *Why isn't Adam enough?* This is an area of active research. Here are the two most common explanations:[7]
    (a) The bias correction factors within Adam can lead to a higher variance in the adaptive learning rate during the first iterations. This results in large gradients during the first iteration, introducing a risk of our model getting stuck in a suboptimal place in the loss function space.
    (b) The iterative application of post-layer normalization across the different transformer blocks also cause very high gradients during the first iterations.[8] Section 2.1 explains how pre-layer normalization solves this.
  - *Alternatives to learning rate warm up.* Improved optimizers like RAdam overcome this issue, but warm-up is such a simple and effective solution that most applications/papers just use the original Adam implementation with learning rate warm-up.
  - *Implementation.* The original "Attention Is All You Need" paper used an exponential decay schedule, but the most popular scheduler is the cosine warm-up scheduler (Figure 13).[7] It combines warm-up with a cosine-shaped learning rate decay.
    - ➜ We often combine learning rate weight warm up with weight decay, which keeps our learned weights closer to zero. This causes each gradient update to have a larger relative impact and speeds up the training process.

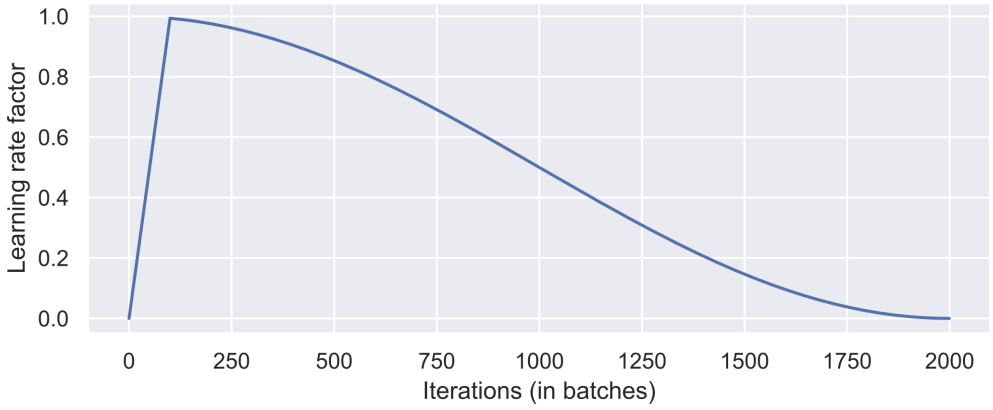


Figure 13: The cosine warm-up learning rate scheduler modulates the learning rate (image credit).

> ⚠ While learning rate scheduling is used in other models, the warm-up phase is particularly crucial for Transformers.

2. **Regularization.** Transformers are especially prone to overfitting.

   - *Why?* Transformers, especially large ones, have a very high capacity due to the numerous parameters in the attention mechanism and feed-forward networks.
   - *Solutions.* There are two possible solutions:
     (a) *Dropout.* Dropout is highly effective within the attention layers and feed-forward networks of Transformers, helping to regularize these high-capacity components.
     (b) *Label smoothing.* In standard classification tasks, we typically use "hard" targets ($p = 1.0$ for one class). Label smoothing replaces these hard targets with "soft" targets.
       - *How?* We use a loss function that supports learning multiple targets.

     EXAMPLE 4. Instead of assigning a probability of 1 to the correct class, it assigns a slightly lower probability (e.g., 0.9) and distributes the remaining probability mass (e.g., 0.1) evenly across all other classes.

     Label smoothing **(a)** helps prevent overconfidence and **(b)** reduces sensitivity to noisy labels.

3. **Memory bottlenecks.** Very deep networks, like transformers, suffer from memory bottlenecks.

   - *Why?* During the forward pass, we need to store each layer's computed activations in memory for the subsequent backward pass. Otherwise, we can't calculate the gradients. The deeper the network, the more layers, incurring higher memory costs.
   - *Solutions.* We use **activation checkpointing**. Rather than storing all activations, we only stores activations from a selected subset of layers (the checkpoints). When the backward pass needs the activations of a non-checkpointed layer, they are recomputed on the fly.
     - We extend computation time to decrease GPU memory consumption.
     - There are various methods for selecting which activation layer computations to save.

In addition to the general challenges of very deep neural networks, Transformers also scale poorly. More specifically, they become computationally expensive and memory-intensive for long sequences. Existing solutions include:

1. *Gradient accumulation.* We simulate larger batch sizes when memory constraints prevent us from using them directly.
    - *How?* We let gradients accumulate over several forward passes before performing back propagation.

      Example 5. Let's say you want to use a batch size of 64, but your GPU memory can only handle a batch size of 16. You would divide your desired batch size (64) by your maximum feasible batch size (16), which gives you 4 accumulation steps.

      ➜ We don't update the weights after each batch of 16. Instead, we perform a forward and backward pass for 4 mini-batches of 16. During each backward pass, we accumulate the gradients across each mini-batch by either summing or averaging them.

      ➜ After accumulating the gradients for all 4 mini-batches, we finally perform a single weight update using the accumulated gradients.

    - *Implications.* This helps if large batch sizes don't fit into your memory, but it doesn't solve the core quadratic complexity problem.
2. *Mixed-precision training.* Mixed-precision training involves using different numerical precisions (usually lower precision like FP16 or BF16) for certain parts of the neural network training process while keeping other parts in higher precision (FP32).
    - *Implications.* Using lower-precision can help with memory, but doesn't address the computational bottleneck.
3. *Attention mechanisms variants.* We choose a slightly different Transformer architecture, where the attention mechanism is less complex. This directly solves the quadratic complexity of standard attention.

Key Concept 13. Transformers suffer from the same challenges as other very deep neural networks, but also from the scaling limitations of the self-attention mechanism.

2. Decoder Only Transformers

The Transformer architecture (introduced in the "All You Need is Attention") paper was an encoder-decoder Transformer.[1] However, OpenAI switched to a decoder-only architecture for GPT-1.[9] This architecture shift allowed OpenAI to train GPT-1 on even more data, leading to significant performance gains.

- *Benefits.* The decoder-only Transformer architecture is easier to train and more versatile.

1. *Simplicity.* In the decoder-only architecture, we remove the entire stack of encoder blocks without modifying the Transformer's core functionality (predicting the next token in a sequence). This simplifies our architecture, and makes it easier to work with.
2. *Easier to train.* The encoder-decoder Transformer architecture requires expected input-output pairs during training. This drastically reduces the data that could be used to train GPT-1. In contrast, the decoder-only architecture can be trained on any sequential data.
3. *Efficiency.* The decoder-only Transformer uses masked attention, where you calculate only half the attention matrix. Thus, we do less calculations to get the same results, making the decoder-only architecture more efficient.
4. *Versatility.* A decoder-only Transformer trained a large dataset (a "chunk of Internet") can perform the same tasks as an encoder-decoder Transformer and other tasks.

2.1 Architecture

The decoder-only architecture consists of an input layer, a stack of decoder blocks (Figure 14), and an output layer. The input layer maps the input tokens to embeddings and adds in positional encodings (see Section 1.6).[10]

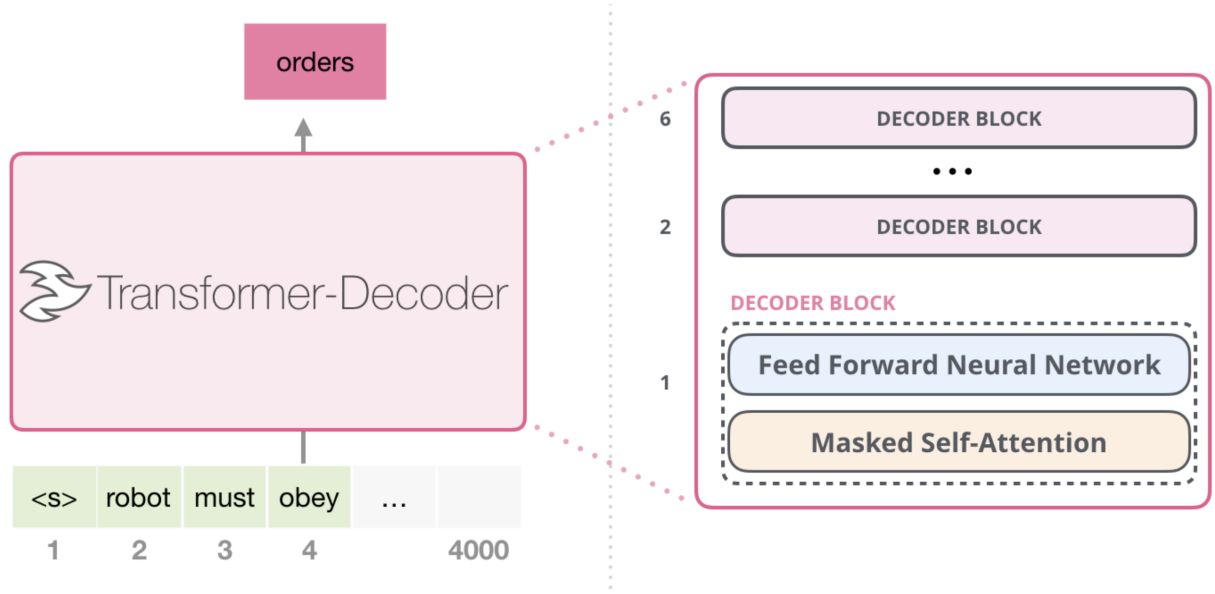


Figure 14: The decoder-only Transformer architecture. All decoder blocks are identical and now use masked self-attention, which increases the number of tokens that a model can handle (image credit).

Each decoder block consists of masked a self-attention layer with residual connection, and a feed forward layer with another residual connection.

- *Masked self-attention.* Masked self-attention ensures "causal" processing, where a token can only look at past tokens but not future ones. This prevents the model from cheating in its next-token prediction task.[5] Figures 15 and 16 visualizes the difference between masked- and self-attention.[10]

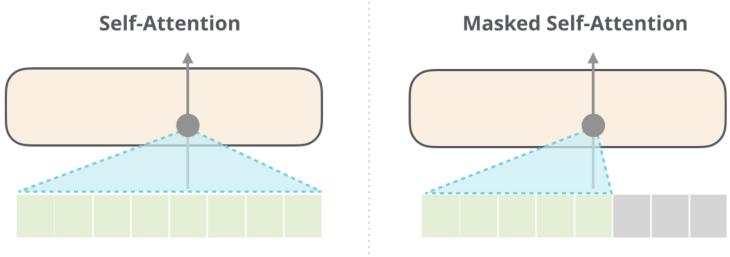


Figure 15: The context captured by self-attention vs. masked self-attention (image credit).

Figure 16: Masked-attention requires less calculations than regular self-attention, which allows the model to handle more tokens (image credit).

- *Feed forward layer.* The decoder's block feed forward layer is a two-layer multi-layer perceptron (MLP) with hidden activation functions (e.g., ReLU), which scales in breadth (hidden size) rather than depth, optimizing computational efficiency.

  It serves two main purposes:

  1. *Introduce non-linearities.* The hidden activation function introduces non-linearities into the model, enhancing its ability to represent complex patterns in data.[11]
  2. *Context aggregation.* The feedforward layer processes and transforms the contextualized token representations produced by the self-attention mechanism.[4]
     - ➜ The MLP usually has a larger hidden dimension than the self-attention layer (d_{model}). This allows the MLP to "inspect" the attention information in a higher-dimensional space.
     - ➜ This "inspection" allows the model to combine and connect learned inter-token dependencies from the self-attention layer into more complex patterns.

- *Layer normalization.* Afterwards, we apply layer normalization. Originally, normalization was done after the residual connections, but this can lead to very high gradients during the first iterations, making training unstable.[7] Pre-layer normalization solves this issue.[8]

  - *Why?* This effect happens as a result of the normalization layer's interaction with the residual stream. (A residual stream is a series of continual residual connections that runs throughout the Transformer.)
  - *Post-layer normalization.* We normalize outside of the decoder blocks by adding normalization layers on top of the residual stream. This distorts the gradient signal by continually scaling it.

➜ We initialize our Transformer weights uniformly. Thus this scaling distortion results in later neural layers have larger gradients than earlier ones during the first iterations (Figure 17).

➜ This results makes training unstable initially, but as the model learns it stabilizes. Thus, this effect diminishes.

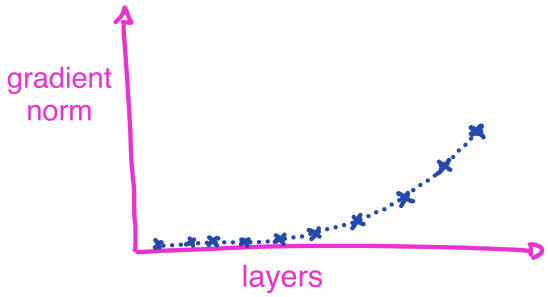


Figure 17: During the first iterations, post-layer normalization and uniform weight initialization result in higher gradient magnitudes for later layers.

- *Pre-layer normalization.* The normalization layer becomes a part of the decoder block. As a result, each attention/feed forward layer still only see normalized feature inputs. However, the normalization layer no longer directly interacts with the residual stream. Figure 18 visualizes how the normalization layer interacts with the residual stream based on its placement.[8]

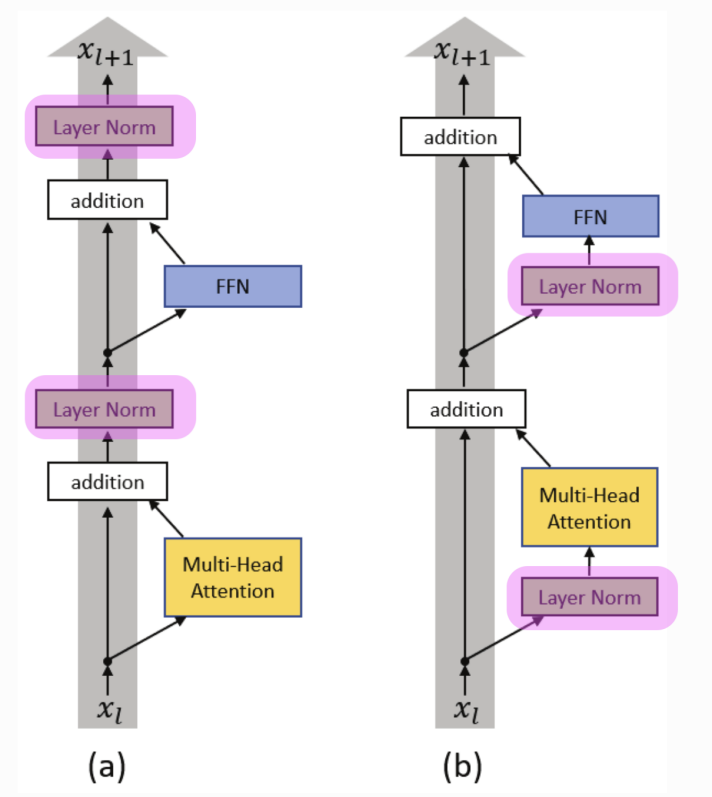


Figure 18: Post-layer normalization **(a)** vs. pre-layer normalization **(b)**. The gray column represents the residual stream (image credit).

We repeat this decoder block N times.[10] Finally, we have our output head. A single linear layer

maps the last decoder block's output to our vocabulary size, which we then apply a softmax over to predict the next token.[4]

> ⚠ In practice, we don't implement the softmax in the output layer during training, since this interferes with the cross entropy loss. Rather, we apply the softmax function over the model output at inference time.

2.2 Training

A commonly raised intuition is to see a Transformer as a lossy compression of a "chunk of internet", where our output is a "zip file" of model parameters. Due to how much information we're tightly compressing, we are not reproducing our training data but rather capturing its high level information.[12]

The decoder-only transformer is trained to predict the next token using cross entropy loss. This technique is called **teacher forcing**, where we always input the "ground truth" tokens and just train transformer to replicate these tokens as outputs.[13]

- *Optimization techniques.* We use the standard Adam optimizer combined with learning rate warm-up and (usually) weight decay.[7]
- *Not many epochs.* In practice, we don't train a Transformers over multiple epochs on the same data (due to the sheer volume of training data).[13]

The gradient stabilization and memory reduction techniques discussed for Transformer in general Section 1.7 still apply.

Definition 2

Teacher forcing is a training technique for sequence models. During training, we use the ground truth output from the previous time step as input to the current step instead of the model's prediction, helping the model learn faster.

2.3 Inference

In contrast to training, the decoder-only transformer now has the task of generating the next token in a sequence. We autoregressively feed the last sampled tokens back into the decoder-only transformer until the model generates a "stop" token (or we reach our maximum sequence length).[14]

- *Sampling the output distribution.* The output head generates a probability distribution of what token is most likely to come next. There are different sampling strategies that we can use to select the next token.
  - Beyond greedy sampling, where we just select the most likely token, we can use temperature T to favor the most likely tokens while allowing for some randomness (Equation 4).
    - ➜ When $T = 1$, the softmax behaves normally. The more we increase T, the more evenly distributed our token probability distribution becomes.
    - ➜ Even if $T = 1$, the Transformer will not reproduce the exact training data. We can think of model training as a lossy compression of the original dataset.

$$P_i = \frac{\exp\left(z_i / T\right)}{\sum_j \exp\left(z_j / T\right)} \tag{4}$$

  - Of course, there are also other more sophisticated sampling techniques. (Ask MD when there's time, but not very important.)

- *Implementation.* In PyTorch, we typically define inference behavior in the `generate` method of our Transformer class definition.

3. Encoder-Decoder Transformers

Thus far, we've focused on decoder-only Transformers, since the are the modern implementation of the Transformer architecture. However, the original Transformer architecture was actually an encoder-decoder Transformer.[1]

3.1 Architecture

The landmark "Attention Is All You Need" paper introduced the architecture shown in Figure 19.[10] The feed-forward layer and self-attention layers are the same those in the decoder-only Transformer. In the encoder-decoder attention layer, we use key-values from the encoder block and do not mask the attention scores $\frac{QK^T}{\sqrt{d_k}}$. Queries originate the decoder block (Figure 20).[10] This process is also called **cross attention**.

3.2 Limitations

As mentioned in Section 2.1, the encoder-decoder architecture requires a set of input and expected output sequences (i.e., "labels"). This greatly limits the amount of data we can train on, and limits the applications for the encoder-decoder architecture.

At the same time, the encoder-decoder Transformer is more computationally expensive than its decoder-only counterpart, because:[9]

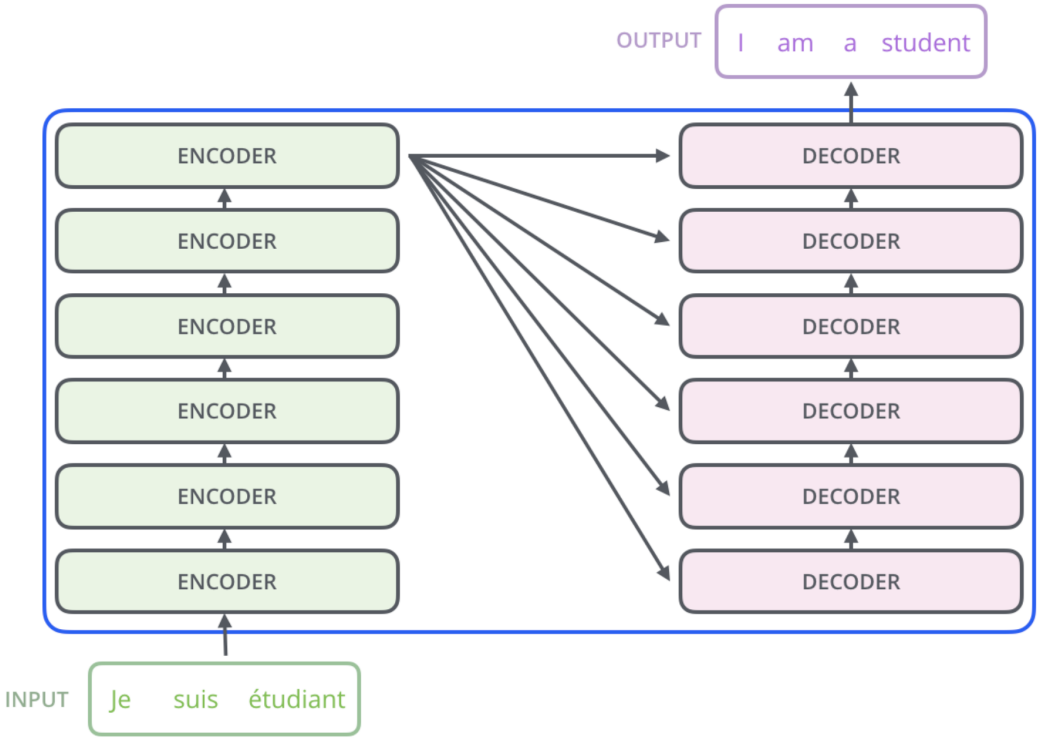
1. The encoder blocks introduce more computations
2. The encoder-decoder attention mechanism does not use causal masking. Meaning, we perform double the calculations for the attention scores $\frac{QK^T}{\sqrt{d_k}}$.

4. Model Compression

4.1 Quantization

Quantization refers to the process of mapping continuous or high-precision values (typically floating-point numbers) to a discrete set of lower-precision values (often integers). In machine learning, we typically quantize a model's weights and activation to reduce its computational footprint.[15]

(a) The original Transformer architecture was a stack of encoder blocks followed by a stack of decoder blocks.



(b) The contents of each encoder and decoder block.

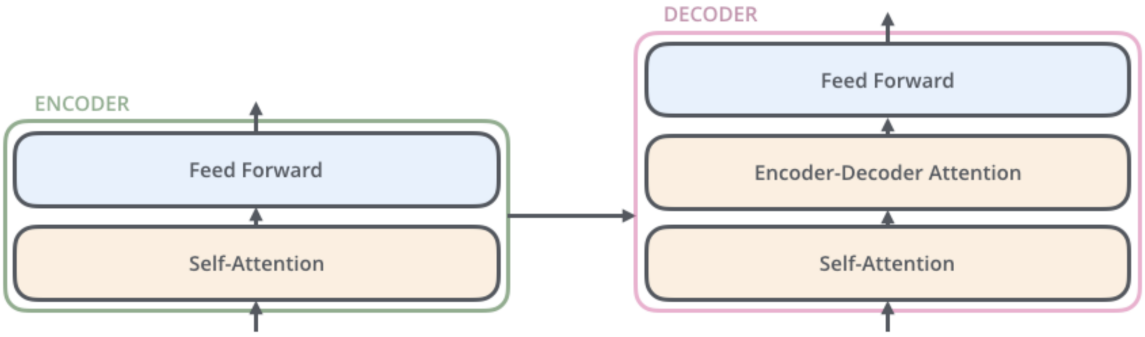


Figure 19: The original encoder-decoder Transformer architecture (image credit).

- *Why?* Large language models consists of billions of parameters. Thus, the effects of compressing a single parameter compound to significant memory costs.[16]
  - *Less memory.* Quantization reduces a model's RAM consumption, making it possible to deploy models on hardware-constrained environments (e.g., edge devices).
  - *Faster computations.* Integer operations are generally faster than floating-point operations, particularly on hardware accelerators. Thus quantizing floating points to integers can lead to significant speedups.
    - ➜ The degree of speedup is hardware-specific.
    - ➜ Some hardware are floating-point optimized. Meaning, the speedup of integer operations may not be that pronounced.
  - *Energy consumption.* Memory-savings and less computationally intensive operations re-

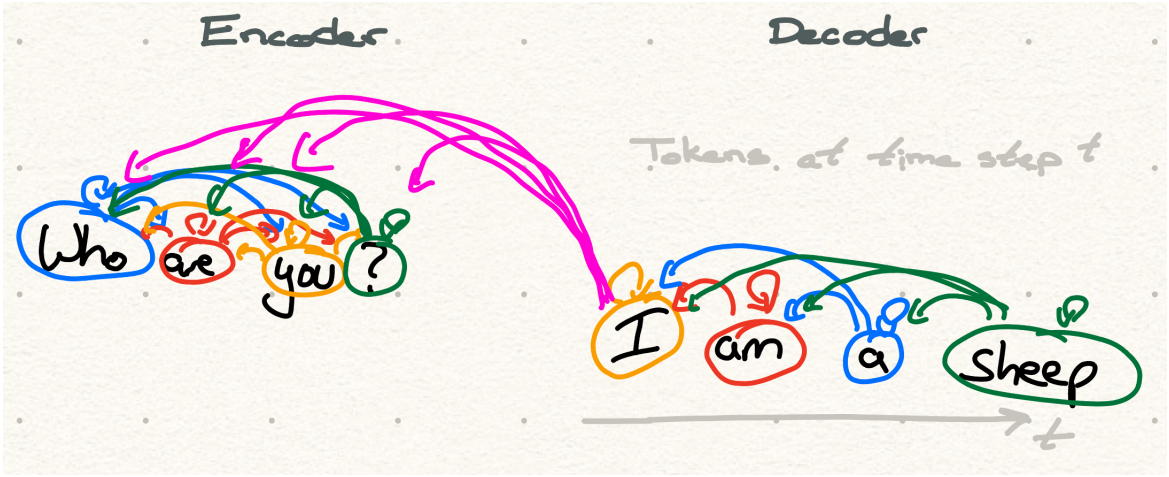


Figure 20: In encoder-decoder attention, we attend to the key-value pairs from the encoder block.

sult in lower power usage. Lower precision also reduces the data movement overhead, which further lowers energy bills.

EXAMPLE 6. Quantizing a 32-bit floating-point model to an 8-bit integer model can significantly reduce memory usage (e.g., 4x less memory). For large language models with billions of parameters, this savings effect is huge.

- *Important terms.* Here are some commonly used terms.
  - *Numerical precision.* The number of bits used to represent numbers (e.g., 32-bit floating point, 8-bit integers).
  - *Dynamic range.* The range of values a number can represent.
- *Trade-offs.* Lower precision models consume less resources, but can become less accurate. Sources of quantization error include precision loss, clipping, and rounding errors.
  - In clipping, quantization maps values outside the representable range to the nearest boundary (e.g., the lowest or highest representable value).
  - We can mitigate these errors source with quantization-aware training, scaling factors to minimize errors, and using mixed-precision techniques.

There are two main types of Quantization methods:[16]

1. **Post-Training Quantization (PTQ).** We apply quantization to after a model has been trained.
   - *Pros.* Simple to implement and doesn't require any training
   - *Cons.* Can degrade model performance, especially for sensitive tasks
2. **Quantization-Aware Training (QAT).** We simulate the effects of quantization during training. This allows the model to adapt to a lower-precision environment and better maintain its accuracy.
   - *Pros.* Retains higher model accuracy when compared to post-training quantization

- *Cons.* Requires retraining the model, which can be computationally expensive

EXAMPLE 7. In the gradient-based quantization technique, we use the gradients of the loss function to infer the importance of weights. We quantize more important weights less aggressively than others, which better preserves model accuracy.

Some popular quantization techniques can be implemented as either as a post-training quantization or quantization-aware training method, while others only belong to specific category.[17]

1. *Uniform quantization.* Values are evenly divided into fixed intervals across a range. This makes it simple and efficient, as each step is of equal size.
   - *Applications.* Situations where the distribution of data is relatively even or predictable.
   - *Category.* Post-training quantization or quantization-aware training

   EXAMPLE 8. We map floating-point values, originally in the range of $[-1, 1]$, to 8-bit integers.

2. *Non-uniform quantization.* Values are divided based on their distribution, with smaller steps where higher precision is needed and larger steps elsewhere
   - *How?* We use the distribution of weight values to select an appropriate non-linear mappings (e.g., logarithmic).
   - *Applications.* Done when the distribution of values is highly skewed or non-linear. Hence, we allocate more precision to the values clustered in a "densely populated" range.
   - *Category.* Post-Training Quantization or Quantization-Aware Training

   KEY CONCEPT 14. Uniform and non-uniform quantization differ in the distribution of the mapped model weights/activations.

3. *Dynamic Quantization.* Quantization parameters (e.g., scale and precision type) are determined dynamically during inference, allowing flexibility but introducing runtime overhead.
   - *How?* We convert weights to lower precision, and dynamically computes activation quantization during inference.
   - *Applications.* Suitable for NLP models and use cases where activations vary widely
   - *Category.* Post-Training Quantization

4. *Static Quantization.* Precomputes and fixes model parameters using a calibration dataset. This minimizes inference overhead but requires careful preparation.
   - *How?* We quantize both weights and activations based on a representative dataset.
   - *Applications.* Effective situations where latency and power consumption are critical. e.g., edge device deployment
   - *Category.* Post-Training Quantization

> **KEY CONCEPT 15.** Both dynamic and static quantization are inference-focused methods.

4.2 Model Pruning

In **model pruning**, we remove parts of a trained machine learning model — usually weights, neurons, or layers — to reduce its size. The goal is to make the model more efficient without significantly degrading its accuracy.[18]

- *Why?* Pruned models are smaller. Thus, they **(a)** consume less memory, **(b)** are faster (both to train and at serving time), and **(c)** less prone to over-fitting.
- *What to prune?* We can prune individual weights, parts of neural layers, or entire neural layers.
  - *Weight pruning.* We remove individual network weights by setting them zero
    - ➜ *How?* We prune weights with the smallest absolute values, since they contribute the least to a model's output.
    - ➜ *Implications.* After pruning, the model becomes sparse.
  - *Activation/neuron pruning.* We remove certain dimensions of our hidden space, which reduces the number of actions that we have to make. This is effectively pruning parts of a neural layer.
    - ➜ *How?* We try to prune neurons that show little activation.
    - ➜ *Variants.* In Transformers, we can also prune attention heads which are sub-layers.
  - *Layer pruning.* We remove entire layers from our neural network. The first and last layers are usually critical.
    - ➜ *How?* We identify low-impact layers or redundant layers empirically.
- *How?* Pruning is done in these steps.
  1. *Identify what to prune.* In addition to using weight or activation values, we can also use a trained model's gradients to infer weight/neuron importance.
  2. *Evaluate pruned model.* We test the pruned model's performance on a validation/test set. If performance drops drastically, important model parts were pruned.
  3. *Fine-tuning (optional).* After pruning, we retrained or fine-tune the model to better maintain its accuracy.

4.3 Knowledge Distillation

Knowledge distillation is a model compression technique where a smaller, simpler model (called **the student**) is trained to mimic the behavior of a larger, more complex model (called **the teacher**).

We want the student model to learn to replicate the teacher's predictions, while maintaining comparable performance.[19]

- *Why?* Knowledge distillation decreases a model's overall size while preserving its performance.

- *Efficiency.* Smaller models are faster, consume less memory, and are easier to deploy.
- *Maintains accuracy.* In some cases, the resulting student model performs better than the same architecture trained independently on the original dataset. This happens, because the student benefits from the teacher model's broader knowledge.

- *How?* Unlike traditional training, where a model is trained using hard labels (e.g., class 0, class 1), the student is trained on soft targets — the probability distribution (i.e., logits) produced by the teacher model.
  - Soft targets provide more information about the relative confidence of predictions, which helps the student learn better.
  - The distillation loss is often based on the KL Divergence between the teacher's soft outputs and the student's outputs.
  - After initial training, we sometimes fine-tune the student model to further increase its performance.
- *Variants.* Sometimes, the student is also trained to match the intermediate representations of the teacher model, not just the output. This is called **feature-based distillation**.[20]

5. Inference Optimizations

5.1 Combined QKV Projection

In Section 1.3, we apply three separate linear projections (matrix multiplications) to the input embeddings X_{input} to generate the Q, K, V feature matrices. Alternatively, we can combine the QKV projection into a single operation.[7]

- *How does it work?* We multiply $X_{\text{input}} \in \mathbb{R}^{s \times d_{\text{model}}}$ by a weight matrix $W_{QKV} \in \mathbb{R}^{d_{\text{model}} \times 3 \cdot d_k}$, and then split the result into three components. i.e.,

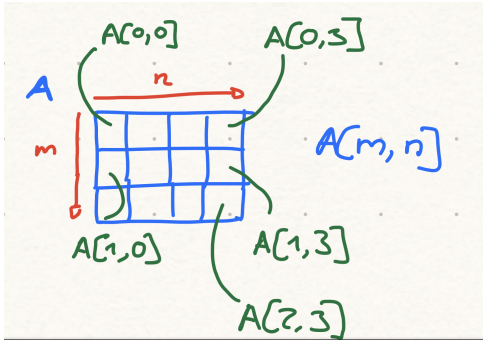
$$QKV = X_{input} W_{QKV} \tag{5}$$

  , where we index $QKV \in \mathbb{R}^{s \times 3 \cdot d_k}$ to get the separate Q, K, and V matrices. We access each specific matrix by selecting its associated columns. i.e.,

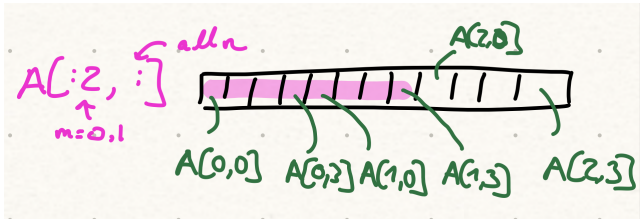
$$Q = QKV[:, : d_k]$$
$$K = QKV[:, d_k : 2d_k]$$
$$V = QKV[:, 2d_k :]$$

- *Understanding low-level memory access.* The combined QKV projection doesn't change the computational complexity (number of operations), but it reduces the number of memory lookups required.

- At the lowest levels in memory, all matrices are stored as one-dimensional arrays.



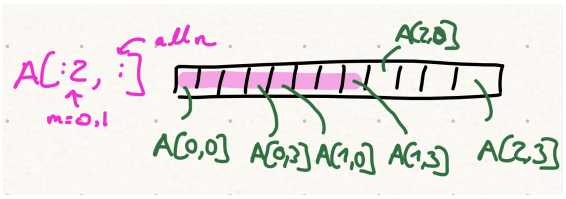
(a) A is a two-dimensional matrix of the size $m \times n$



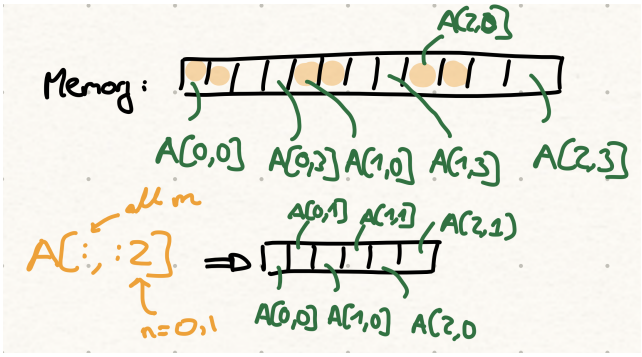
(b) In low-level memory, matrix A is represented as a 1D array

Figure 21: Matrices are always represented as 1D arrays in low-level memory.

- Accessing consecutive elements in a 1D memory array is efficient because they are stored in contiguous memory blocks.
- Accessing non-consecutive elements, such as when selecting specific columns from a matrix (e.g., splitting QKV into Q, K, V), incurs a higher memory lookup cost. This is due to increased cache misses and the overhead of retrieving scattered indices.
- When accessing non-consecutive indices in a large array, the lookup process becomes so inefficient that it's faster to copy the relevant elements into a new contiguous 1D array before performing reoccurring matrix operations.



(a) Accessing consecutive array indices is computationally cheap



(b) Accessing nonconsecutive indices in large arrays is computationally expensive

Figure 22: The memory lookup cost depends on the matrix size and whether we're accessing consecutive/nonconsecutive indices.

- *QKV memory costs.* Performing one large matrix multiplication tends to be more efficient for hardware accelerators (like GPUs or TPUs) than performing multiple separate smaller matrix multiplications so long as QKV doesn't exceed a certain size.

  1. *When is it efficient?* In the combined QKV approach, we only load the input tensors X_{input} and W_{QKV} once and perform a single matmul to generate Q, K, V.
     - In contrast, the traditional self-attention mechanism requires three separate memory accesses to load the W_Q, W_K, and W_V matrices, and X_{input} three times, resulting in additional overhead.

- For smaller sequence lengths (e.g., during inference), the overhead of multiple memory accesses in the classical approach becomes significant. This makes the combined QKV approach more efficient.

2. *When is it inefficient?* Suppose that we have a batch size $b = 1$ and a sequence length $s = 1$. In this case, we have two less dimensions for QKV. This makes the combined QKV 1D array representation contiguous, resulting in no additional overhead.
   - *Non-contiguous arrays.* Normally, $b > 1$ and $s > 1$. Thus, as s (and b) grows, the indexing nonconsecutive QKV elements in its 1D array low-level memory representation becomes increasingly inefficient.
   - *Copying as a solution.* One possible solution is to copy the nonconsecutive elements of QKV into a new 1D array for better access efficiency in following matrix operations. Of course, the copying process incurs its own costs and consumes significant memory when QKV is very large.
   - *Trade-off.* There's a turning point when the cost of indexing nonconsecutive elements in QKV exceeds the benefits of looking up a single memory lookup for both W_{QKV} and X_{input}.
   - *Hardware dependencies* This turning point is hardware-specific, as it depends on the relative cost of memory lookups versus computation (floating-point operations per second) on the hardware being used.

- *When to use QKV?* The weights in W_{QKV} are equivalent to those in $\text{concat}(W_Q, W_K, W_V)$, since W_{QKV} can be constructed by concatenating W_Q, W_K, and W_V.
  - *Training.* Typically, we train a Transformer on very large batches, where the size of QKV becomes substantial. In such cases, indexing nonconsecutive elements from QKV may introduce greater overhead than the separate memory lookups for W_Q, W_K, and W_V.
    ➜ Therefore, during training, we often use separate projections for Q, K, and V to minimize indexing overhead.
  - *Inference.* Input sequences are usually much smaller. In this scenario, the memory costs of indexing nonconsecutive QKV elements are negligible compared to the runtime gains from a single lookup of W_{QKV}.

KEY CONCEPT 16. The decision of whether to combine the Q, K, and V projections is a trade-off between memory and compute efficiency. We have 3 options:

1. *Separate Q, K, V projections.* We have to perform three separate matrix multiplications. For very low batch sizes/sequence lengths, each matmul cannot fully utilize the GPU resources due to the small shapes, leaving idle processors. This implicitly incurs compute costs.

2. *Combined QKV projection, no copying.* We perform a single matrix multiplication, ensuring an efficient utilization of the GPU. However, indexing nonconsecutive QKV elements cannot make use of efficient kernels (e.g., for consecutive matrix multiplications/flash attention).

3. *Combined QKV projection with copying.* We resolve the element access issue from Op-

> tion (2) by copying nonconsecutive QKV elements to new 1D arrays, which then incurs wait time/compute costs that scales with the feature size. For smaller batches/sequence length, this is negligible, but for large training batches with long sequences, this becomes significant.

5.2 KV Caching

The **KV Cache** (Key-Value Cache) optimizes the performance of transformer models at inference time.[21]

	Dynamic KV Cache	Static KV Cache
Main Idea	We dynamically increase the KV Cache tensors as we generate new tokens.	We set a maximum allowed sequence length and set a fixed KV Cache tensor size during its initialization.
Initialization	The KV cache starts with a small initial allocation. Often, the sequence length dimension is 0, meaning no memory is allocated initially for storing keys and values ($\mathbb{R}^{b\times 0\times d_k}$), or sometimes it is initialized with a size of $\mathbb{R}^{b\times 1\times d_k}$.	We allocate memory for the KV cache once, initializing it as a $b \times s_{\text{maximum}} \times d_k$ sized matrix.
Token Generation Loop	We only compute the key and value for the current token. • We concatenate the newly compute keys and values along the s dimension of the KV Cache. Thus, the KV Cache dimension becomes $\mathbb{R}^{b\times s_i\times d_k}$. • We use the entire KV Cache to compute the attention scores, $\frac{QK^T}{\sqrt{d_k}}$, and generate the next token.	We only compute the key and value for the current token. • This key-value pair is stored in the pre-allocated memory block at the next available position. • We use the entire KV Cache to compute the attention scores, $\frac{QK^T}{\sqrt{d_k}}$, but mask the attention scores for keys that we have not filled yet.
Stop Condition	We continue until we reach the desired sequence length or encounter a stop condition (e.g., generating a stop token). The final shape of the KV Cache is $b \times s_{\text{final}} \times d_k$.	We repeat token generation until we run out of pre-allocated memory and then stop, or go over into a sliding window approach (removing first key and adding new one).
Pros	(+) An intuitive implementation (+) Handles memory efficiently when generating varying sequence lengths	(+) Since memory is pre-allocated, we don't have memory allocation overheads during generation (+) The computation graph doesn't change, meaning it can be efficiently compiled
Cons	(-) Resizing the KV Cache means **(a)** allocating new memory and **(b)** copying over existing data. This adds performance overhead, especially for longer sequences. (-) Every time the KV Cache tensor shape changes so does its computation graph. ML compilers can only handle static computation graphs. Thus, the matrix operations are recompiled at every token generation (compute costs).	(-) At every generation step, we calculate the attention scores over the whole KV cache, which is the maximum sequence length. Most of these scores are masked. Thus, we incur unnecessary compute costs during the first steps. (-) If you often generate sequences much shorter than maximum sequence length, you waste a significant portion of the pre-allocated memory (-) There's also overhead in finding the next available position in the KV Cache

Table 3: Paradigm shifts introduced by the attention mechanism.

- *The problem.* Text generation is an autoregressive task, where the model generates text one token at a time. The model uses all previously seen tokens to maintain context.

- Without caching, the model would recompute the keys and values for all preceding tokens, leading to many redundant calculations.

- *The solution.* When the model generates the first word, it calculates the keys and values for this word and stores them in the KV Cache. For each subsequent word:

  1. The model only calculates the new word's key and value.
  2. The model concatenates the newly calculated key and value to their respective cache tensors, where `kv_cache = (key_cache_tensor, value_cache_tensor)`.
  3. The scaled-dot product attention is then calculated using the current token's query feature vector, and the feature vectors from the KV Cache.

- *Scalability.* The KV cache grows in memory with larger and larger context lengths. This memory cost is the main factor limiting transformers in long context applications.

> **KEY CONCEPT 17.** The KV Cache prevents redundant key and value calculations, which speeds up text generation.

> **KEY CONCEPT 18.** The main factor limiting Transformer applications to longer context lengths is KV Cache memory consumption.

KV Caching can either implemented as a Static KV Cache or Dynamic KV Cache.[22]

1. In a Static KV Cache, we assume a maximum sequence length and set the KV Cache tensor to a fixed size.

2. In Dynamic KV Caching, we continuously grow our KV Cache tensor by appending the current token's K and V values to it along the s dimension.

Table 3 summarizes each KV Caching approach as well its pros and cons. Since the pros and cons of static vs. dynamic KV caching are complimentary, we often implement a hybrid approach where we have a "Static" KV Cache that we resize (e.g., by doubling) once the maximum input sequence length is exceeded.

5.3 Flash Attention

In standard self-attention, the most expensive operation is data movement. Reading and writing to GPU memory is so costly due to the self-attention layer's large intermediary matrices (Figure 23).[23]

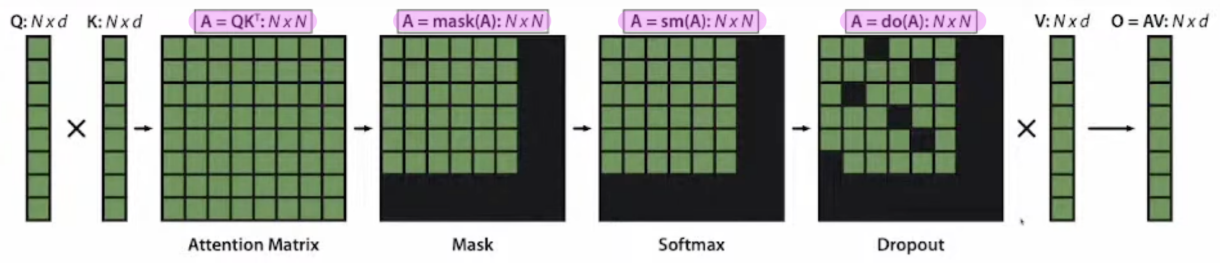


Figure 23: The intermediary self-attention matrices (highlighted in pink) are much larger than the attention layer's inputs or outputs (image credit).

Flash Attention is an optimized implementation of self-attention, where we avoid storing the entire large intermediary QK^T matrix.[24]

- *Note.* Unlike other optimizations in this section, flash attention also optimizes the LLM training process.
- *Intuition.* We rewrite the attention operation to reduce GPU read and write operations. This is a hardware-aware optimization that does not approximate self-attention.
- *Background.* Flash attention is a hardware-aware implementation of attention. Meaning, we need to understand some basic things about GPUs:[23]
  - GPUs have extremely fast compute times relative to their slow memory access times.
  - Maximizing compute and minimizing memory access is the key for efficiency.
  - GPU has a memory hierarchy, where **high bandwidth memory (HPM)** is much slower than static RAM (sRAM). See Figure 24.

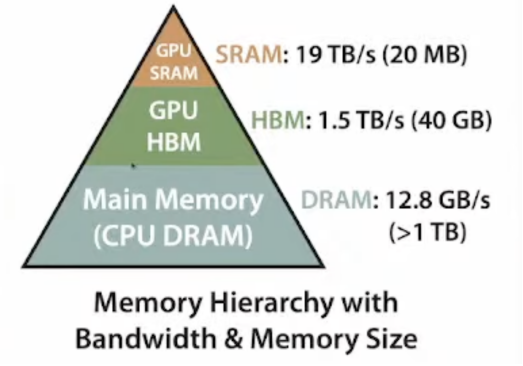


Figure 24: The GPU sRAM is about three times faster than HBM image credit).

- *How?* We break our attention matrix into blocks and recompute intermediary matrices to minimize I/O operations.[23]
  1. *Tiling.* We reformulate the softmax in an iterative way using a mathematical property called "scaling" (Equations 6-7).

$$\text{softmax}([A_1, A_2]) = [\alpha\ \text{softmax}(A_1), \beta\ \text{softmax}(A_2)] \tag{6}$$

$$\text{softmax}([A_1, A_2]) \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \alpha\ \text{softmax}(A_1)V_1 + \beta\ \text{softmax}(A_2)V_2 \tag{7}$$

  This allows us to load data block by block from HBM into sRAM when computing attention. We compute each tile separately and aggregate the results from each tile with Equation 7.

> **KEY CONCEPT 19.** By reformulating the softmax in an iterative way, we no longer need to store the entire attention matrix in HBM.

  2. *Recomputation.* We don't store the attention matrix from the forward pass. We recompute it in the backward pass, since it's cheaper to recompute than to load.

- *Results.* On most hardwares, flash attention offers 2-4x speedup, and 10-20x memory reduction without making any approximations. Exact performance gains depend on the hardware.[23]
    - Flash attention is widely used in LLM development, since it offers significant compute/memory performance gains without sacrificing model performance.
    - This increased memory capacity allows us to increase our context window during training, which ultimately yields better quality model outputs.

5.4 Mixture of Experts

Mixture of Experts (MoE) is an alternative to pruning the feed-forward layer within an attention block.

- *Intuition.* Rather than having one very wide feed-forward network, we have several smaller feed-forward networks running in parallel (Figure 25).[25]
    - Each feed-forward network specializes in a specific type of input (e.g., different modalities).
    - A small model, called a **router**, dynamically selects the correct feed-forward network based on the given input.

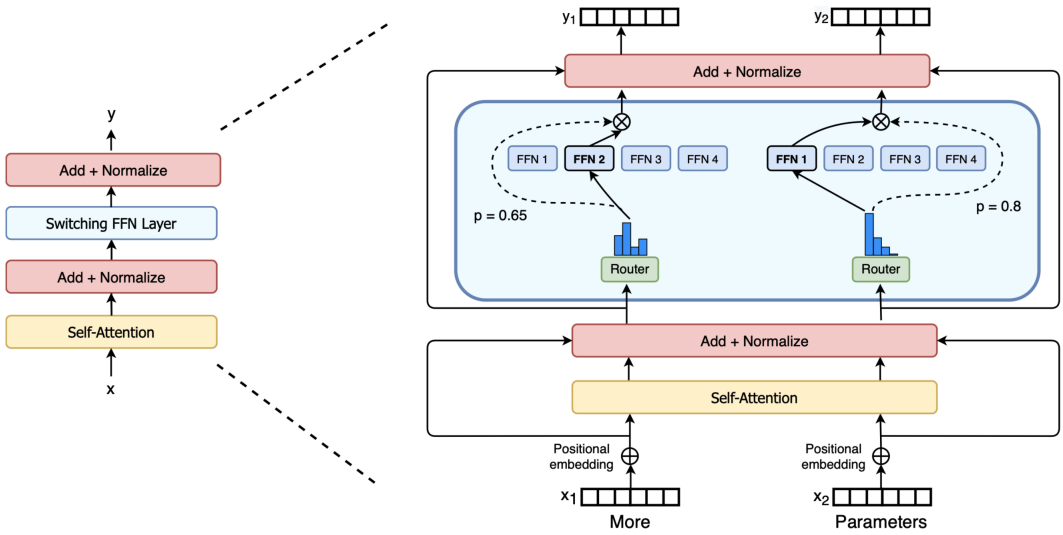
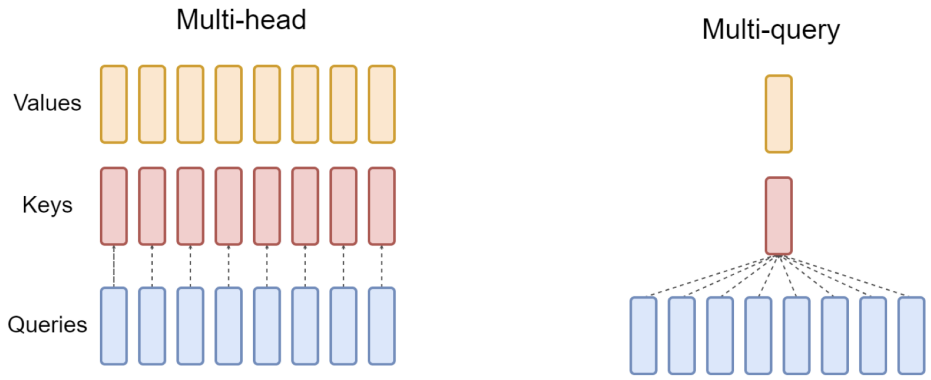


Figure 25: The feed forward layer in classical self-attention vs. mixture of experts implementation (image credit)

- *Implications.* Mixture of Experts speeds up inference times, since we run less computations during each forward pass. However, training becomes more complex since the router needs to learn load balancing.
- *Prevalence.* Almost all larger LLMs (70B+ parameters) use Mixture of Experts at this point.
- *Drawbacks.* Even though a smaller number of parameters are "active" in each forward pass, the inactive parameters still occupy GPU memory. Meaning, MoE is just as memory-expensive as a full-scale dense model.

5.5 Multi-Query Attention

Multi-query attention is an inference optimization technique, where we space key-value pairs across different attention heads in the same self-attention layer.[26]



(a) Each head computes its own key-value pairs **(b)** Key-value pairs are shared across attention heads

Figure 26: Classical multi-head attention vs. multi-query attention (image credit).

> ⚠ If we want to serve the model using multi-query attention, then we need to train it with multi-query attention.

- *Extreme cases.* We can either reuse all key-value pairs across a self-attention layer or only compute separate, unique key-value paris.
  1. In classical self-attention, each attention head computes its own key-value pairs.
  2. In the most extreme application of multi-query attention, all heads share the same key-value pairs and only their queries differ.
- *Why?* Multi-query attention minimizes the number of key-value pairs that we need to compute, which drastically reduces the KV Cache size.
- *Performance.* In the best case, the model's inference predictions are comparable those of a classical self-attention model. In the worst case, multi-query attention performs slightly worse.

KEY CONCEPT 20. Multi-query attention drastically reduces the KV Cache size, which increase our possible context length window.

5.6 Speculative Decoding

Speculative decoding is an LLM inference optimization method, where we use a smaller, faster auxiliary model M_{small} to propose token batches to a larger, more accurate model M_{large}.[27]

- *Why?* LLM inference is an auto-regressive decoding task, where we **(a)** generate the text token-by-token and **(b)** each token depends on the previously generated tokens.[28]
  - *GPU memory consumption.* When we do a forward pass through a model, we need to load all its parameters into memory. The larger the model, the more parameters we have to load into memory.

➜ GPUs have very fast processors, but getting data onto these processors can be slow. This makes GPUs heavily memory bound at smaller batch sizes.

➜ Thus, we need to balance the number of parameters to load into memory against the number of floating point operations per second.

➜ i.e., We need to maximize the number of compute operations we do per byte.

- *Large vs. small models.* Inputting a very small batch size (e.g., 1) into a large model ($> 70B$ parameters) leads to poor GPU utilization. However, larger models also tend to generate higher quality outputs.

- *How?* The smaller model generates multiple tokens, which we call a **speculative batch**. The speculative batch is inputted into the larger model, where we compare the predict last token of the smaller and larger model's.[27]
  - If there's a conflict, the larger model's last generated token is kept. Everything in the speculative batch after the mismatch is discarded (Figure 27).

- *Implementation trade-off.* We want our speculative batch to be large enough to significantly reduces the number of forward passes for M_{large}. If the batch size becomes too large, we are more likely to encounter a mismatch, where we have to redo the computations anyways.

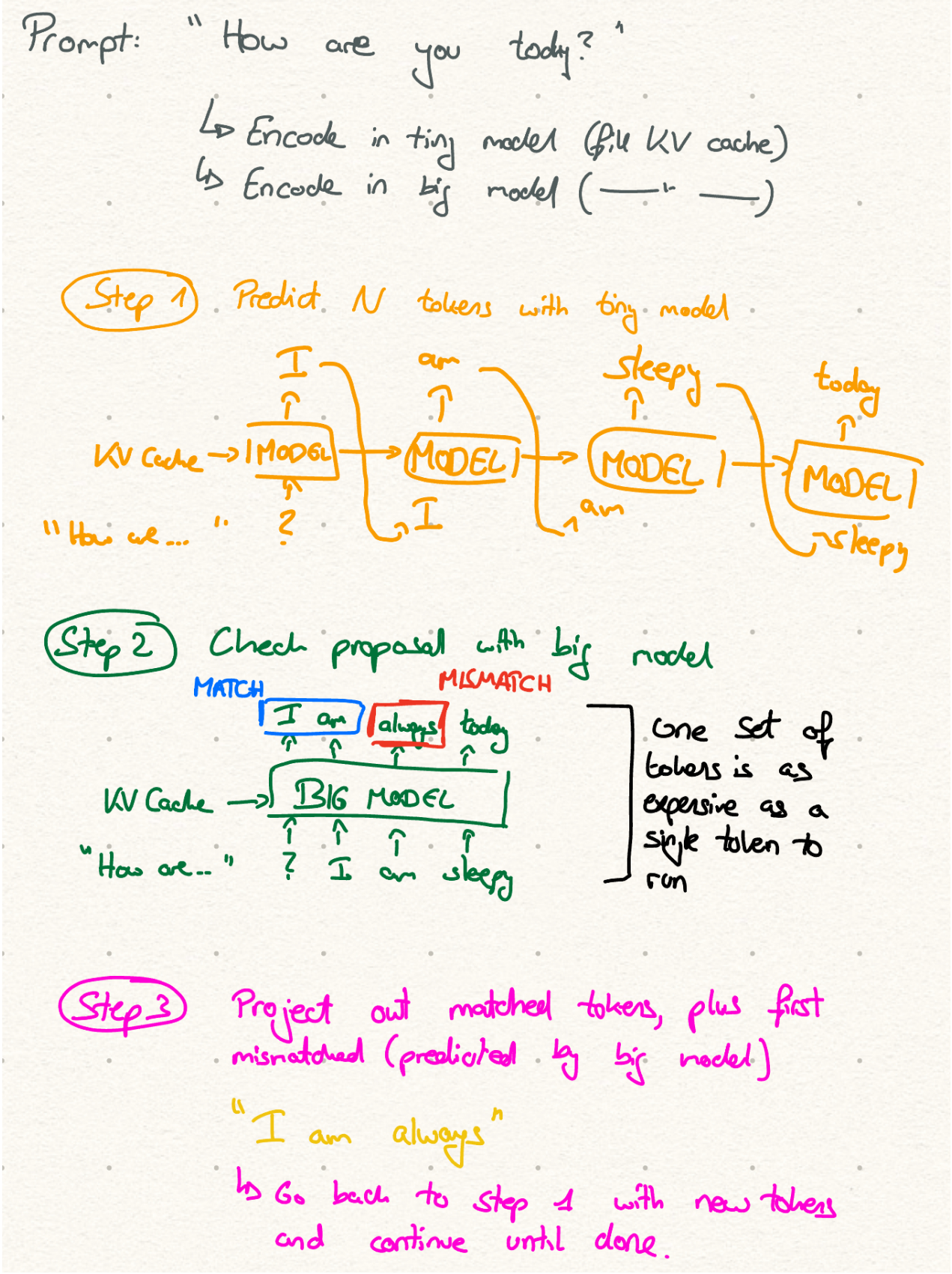


Figure 27: The main steps of speculative decoding

5.7 Continuous Batching

Loading our model parameters into GPU memory is a very expensive operation. Batching allows us to load our model into memory once and then using it to process many input sequences, leading to better GPU utilization.

There are two main types of batching:[29]

1. **Static batching.** In more traditional deep learning models, the model outputs are uniform in size. Meaning, we can have good GPU utilization by fixing our inference batch size and only returning the results once the entire batch has been processed.

   In contrast, LLM inference is an iterative process where outputs vary in length. In particular, for every request we must:

   (a) *Process the prompt.* We start with a sequence of tokens called the **prefix** or prompt.

   (b) *Complete the sequence.* The LLM produces a sequence of completion tokens until **(a)** a stop token is generated or **(b)** we reach our maximum sequence length.

   Meaning, if we apply static batching to LLM inference, we drastically reduce our model throughput since we can spend a long time waiting for one request to finish before returning the entire batch's results (Figure 28).[29]

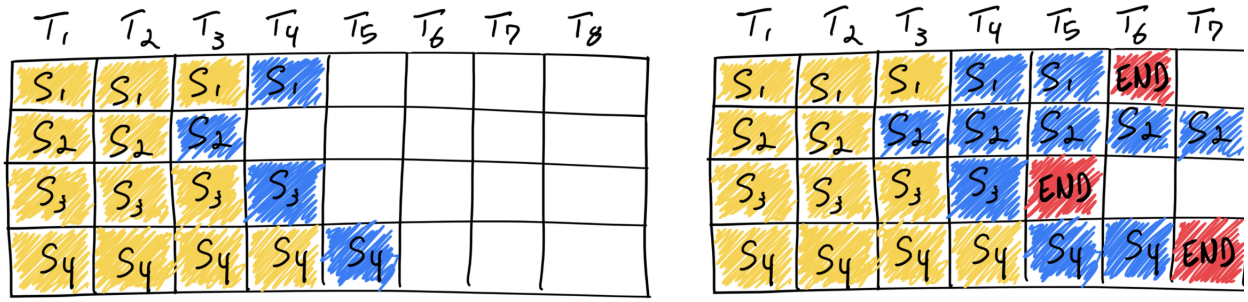


Figure 28: A static LLM batch is not computationally efficient. Yellow represents the prefix token, and blue the end-of-sequence tokens (image credit).

   This reduces the number of FLOPS we can perform while our cost of loading the model parameters into memory remains fixed. i.e., We have poor GPU utilization. The more our generated sequences vary in length, the worse this problem becomes.

2. **Continuous batching** solves this problem by implementing iterative-level scheduling (at each time step).

   - *Intuition.* Once a sequence in our batch is completely generated, we directly send the completed sequence to the client. We use this free memory to process a new incoming request (Figure 29).[29]

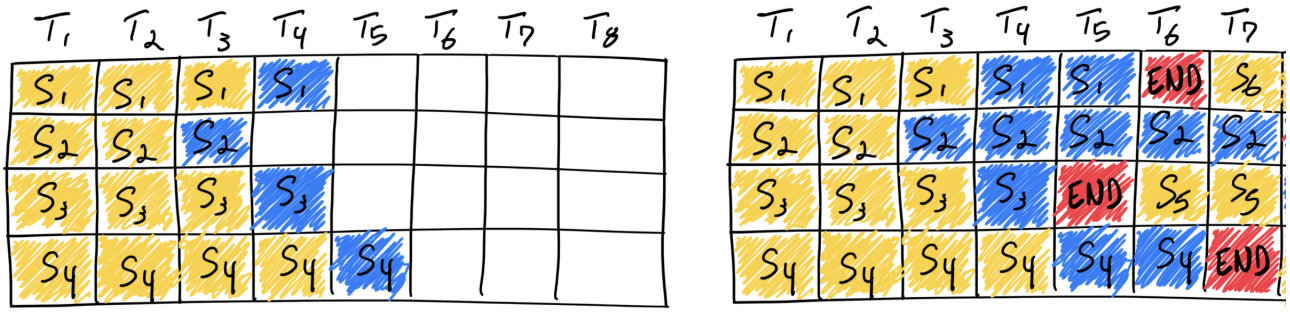


Figure 29: A continuous LLM batch accepts new requests once old requests have been finished, which improves GPU utilization. Prefill tokes are in yellow, end-of-sequence tokens in blue (image credit).

- *Limitations.* Continuous batching only addresses the fact that our end-of-sequence tokens vary in size. However, our prefix token length also varies from request to request. Thus, we just pad our prefix tokens so that they're in an uniform size.

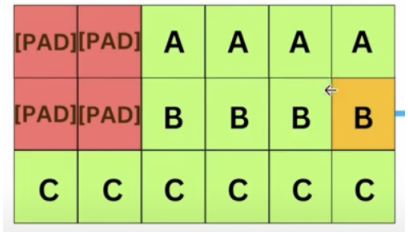


Figure 30: We pad each request's prefix tokens to ensure a uniform sequence length at time step t (image credit).

> ⚠ Continuous batching doesn't resolve the fact that our user prompts also vary in length.

- *Results.* Continuous batching improves model throughput, which in turn improves latency. This can lead up to 20x improvements

6. Fine-Tuning Optimizations

6.1 Low-Rank Adaptation

Low-Rank Adaption (LoRA) is an optimization technique that minimizes the compute costs associated with fine-tuning weight matrices in large LLMs.[30]

- *Intuition.* It's too expensive to update the entire weight matrix W_{original} for very large LLMs. Therefore, we'll learn the changes that we want to make to W_{original} in a lower dimensional space.
  - We save the low-dimensional weight parameter changes needed to fine-tune W into two much smaller matrices A and B such that

$$\delta W_{\text{original}} + W_{\text{original}} = W_{\text{fine-tuned}} \tag{8}$$

$$\delta W_{\text{original}} = BA \tag{9}$$

- Figure 31 visualizes the compute costs saved by using fining a low-rank approximation.[31]

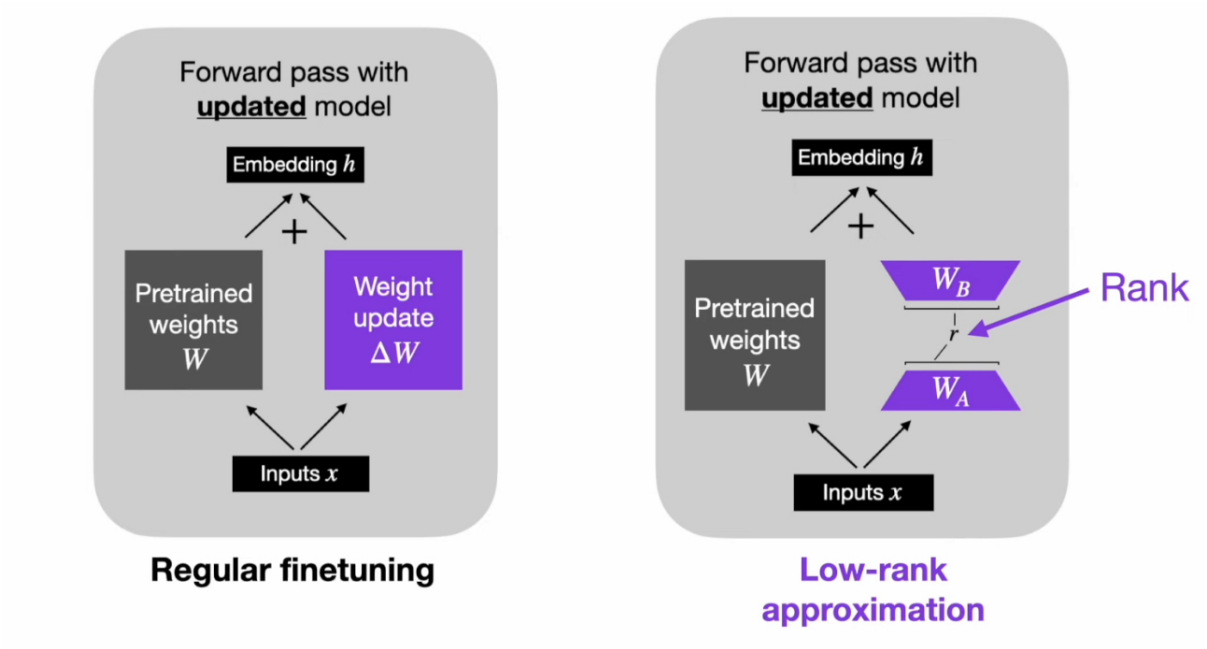


Figure 31: The weight updates in regular fine-tuning vs. low-rank adaptation (image credit)

- *Implementation.* While LoRA can hypothetically be applied to any part of a Transformer, it's been empirically shown to work best when only applied to the self-attention layers.
- *Prevalence.* LoRA is a very popular topic in academia, but it's still important for fine-tuning larger (70B+ parameter) models when:
  1. We have limited compute resources for fine-tuning
  2. We have to fine-tune many different versions of a very large LLM

6. References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, 2017.

[2] D. Bahdanau, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[3] J. Alammar, "The illustrated transformer." Blog post, 2018. https://jalammar.github.io/illustrated-transformer/.

[4] A. Karpathy, "Let's build gpt: from scratch, in code, spelled out." YouTube video, 2023. https://www.youtube.com/watch?v=kCc8FmEb1nY.

[5] 3Blue1Brown, "Attention in transformers, step-by-step | dl6." YouTube video, 2024. https://www.youtube.com/watch?v=eMlx5fFNoYc.

[6] F. Khan, "Solving transformer by hand: A step-by-step math example." Blog post, 2023. https://levelup.gitconnected.com/understanding-transformers-from-start-to-end-a-step-by-step-math-example-16d4e64e6eb1.

[7] P. Lippe, *University of Amsterdam Deep Learning Tutorials*. University of Amsterdam, 2022. https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html.

[8] P. Lippe, *Tutorial 15: Vision Transformers*, ch. Tutorial 15: Vision Transformers. University of Amsterdam, 2022. https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial15/Vision_Transformer.html.

[9] A. Radford, "Improving language understanding by generative pre-training," 2018.

[10] J. Alammar, "The illustrated gpt-2 (visualizing transformer language models)." Blog post, 2019. https://jalammar.github.io/illustrated-gpt2/.

[11] 3Blue1Brown, "How might llms store facts | dl7." YouTube video, 2024. https://www.youtube.com/watch?v=9-Jl0dxWQs8&t=932s.

[12] A. Karpathy, "[1hr talk] intro to large language models." YouTube video, 2024. https://www.youtube.com/watch?v=zjkBMFhNj_g.

[13] Y. Dubois, "Stanford cs229 i machine learning i building large language models (llms)." YouTube lecture, 2024. https://www.youtube.com/watch?v=9vM4p9NN0Ts.

[14] 3Blue1Brown, "Transformers (how llms work) explained visually | dl5." YouTube video, 2024. https://www.youtube.com/watch?v=wjZofJX0v4M&t=1455s/.

[15] H. F. O. Team, "Quantization." Online Documentation, 2024. https://huggingface.co/docs/optimum/en/concept_guides/quantization.

[16] M. Grootendorst, "A visual guide to quantization." Blog post, jul 2024. https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization.

[17] Deepchecks, "Top llm quantization methods and their impact on model quality," *Deepchecks Community Blog*, nov 2024. https://www.deepchecks.com/top-llm-quantization-methods-impact-on-model-quality/.

[18] S. Paul, "Diving into model pruning in deep learning." Blog post, jun 2021. https://wandb.ai/authors/pruning/reports/Diving-Into-Model-Pruning-in-Deep-Learning--VmlldzoxMzcyMDg.

[19] D. Bergmann, "What is knowledge distillation?." Blog post, apr 2024. https://www.ibm.com/think/topics/knowledge-distillation#:~:text=Knowledge%2020distillation%20is%20a%20machine,for%20massive%20deep%20neural%20networks.

[20] C. Yang, X. Yu, Z. An, and Y. Xu, "Categories of response-based, feature-based, and relation-based knowledge distillation," in *Advancements in Knowledge Distillation: Towards New Horizons of Intelligent Systems*, pp. 1–32, Springer, 2023.

[21] M. Oleszak, "Transformers key-value caching explained." Blog post, dec 2024. https://neptune.ai/blog/transformers-key-value-caching.

[22] H. F. T. Team, "Best practices for generation with cache," jan 2025.

[23] T. Dao, "Flashattention - tri dao | stanford mlsys no. 67." YouTube video, 2023. Stanford MLSys Seminars No. 67.

[24] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16344–16359, 2022.

[25] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

[26] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," *arXiv preprint arXiv:2305.13245*, 2023.

[27] Y. Leviathan, M. Kalman, and Y. Matias, "Fast inference from transformers via speculative decoding," in *International Conference on Machine Learning*, pp. 19274–19286, PMLR, 2023.

[28] Y. Leviathan, M. Kalman, and Y. Matias, "Looking back at speculative decoding." Blog post, dec 2024. https://research.google/blog/looking-back-at-speculative-decoding/.

[29] C. Daniel, C. Shen, E. Liang, and R. Liaw, "How continuous batching enables 23x throughput in llm inference while reducing p50 latency." Blog post, jun 2023. https://www.anyscale.com/blog/continuous-batching-llm-inference.

[30] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[31] S. Raschka, "Insights from finetuning llms with low-rank adaptation." YouTube video, 2024. https://www.youtube.com/watch?v=rgmJep4Sba4.