NOTE ON LANGUAGE MODELS

A PREPRINT

Haocheng Dai

December 23, 2024

Contents

1 Transformer deep dive

Definition 1.1. **Transformer** is a deep learning architecture. At each layer, each token is then contextualized within the scope of the context window with other (unmasked) tokens via a parallel multi-head attention mechanism allowing the signal for key tokens to be amplified and less important tokens to be diminished.

1.1 Encoder and decoder of transformer

Definition 1.2. **Encoder in transformer** consists of an embedding layer, followed by multiple encoder layers. Each encoder layer consists of two major components:

1. A **self-attention mechanism** takes an input as a sequence of input vectors, applies the self-attention mechanism, to produce an intermediate sequence of vectors;

2. Then the **feed-forward layer** is applied for each vector individually.

Remark 1.1. *The output from the final encoder layer is then used by the decoder. As the encoder processes the entire input all at once, every token can attend to every other token (all-to-all attention), so there is no need for causal masking.*

Definition 1.3. **Decoder in transformer** contains an embedding layer, followed by multiple decoder layers, followed by an un-embedding layer. Each decoder consists of three major components:

1. **Masked self-attention** for "mixing" information among the input tokens to the decoder (i.e. the tokens generated so far during inference time);allowed to attend to earlier positions in the output sequence;

2. **Cross-attention** for incorporating the output of encoder (contextualized input token representations);

3. **Feed-forward network** for introducing non-linearity and do dimensional transformation, usually using two linear layer to project to a much higher dimension and then project back to the original dimension, with activation function seats in between.

Remark 1.2. *Initially, the shifted target sequence of the decoder is just the start token (like `[BOS]`). As the model generates tokens, the sequence grows, and the model uses the previously generated tokens to predict the next one.*

The original input sequence (from the encoder) is used in the cross-attention layer to provide context to the decoder, helping it generate the appropriate next token based on the input information.

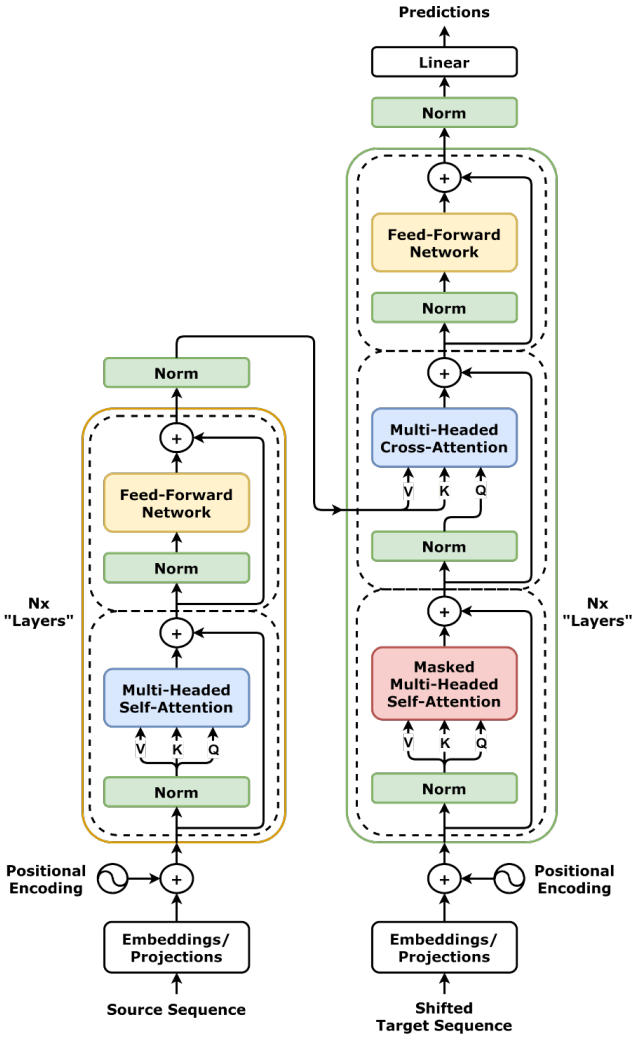

Figure 1: Transformer overview. Left and right branches are the encoder and decoder of the transformer. The $\mathbf{V}$ and $\mathbf{K}$ in the decoder's cross-attention layer are derived by $\mathbf{V} = \mathbf{H}_{\text{enc}} \times \mathbf{W}_{\text{dec}}^{V}, \mathbf{K} = \mathbf{H}_{\text{enc}} \times \mathbf{W}_{\text{dec}}^{K}$, where $\mathbf{H}_{\text{enc}}$ is the hidden state produced by the encoder. The input ("shifted target sequence") of the decoder would be the sequence produced by itself. In the beginning, it will be a `[BOS]` token alone.

Remark 1.3. *"Target sequence" is the sequence of tokens that the model is expected to generate during the training process. While the "shifted target sequence" means that the decoder's input is the target sequence shifted by one position to the right. The first token of the input sequence is typically a special token like `[BOS]` token.*

Example 1.

1. Given `[BOS]`, predict $\mathbf{y}_1$;
2. Given `[BOS]`, $\mathbf{y}_1$, predict $\mathbf{y}_2$;
3. Given `[BOS]`, $\mathbf{y}_1$, $\mathbf{y}_2$, predict $\mathbf{y}_3$, and so on.

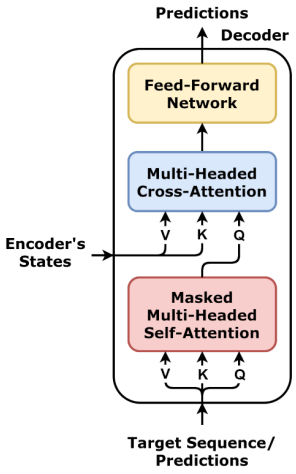

Figure 2: Decoder layer overview. The target sequence/predictions refers to the model has access to the ground truth target sequence during training process and what the model has generated so far during inference process. The encoder's states $\mathbf{V}$ and $\mathbf{K}$ are derived by $\mathbf{V} = \mathbf{H}_{\text{enc}} \times \mathbf{W}_{\text{dec}}^{V}, \mathbf{K} = \mathbf{H}_{\text{enc}} \times \mathbf{W}_{\text{dec}}^{K}$, where $\mathbf{H}_{\text{enc}}$ is the hidden state produced by the encoder.

Remark 1.4. *The role of encoder and decoder in transformer:*

1. *The encoder consists of encoding layers that process all the input tokens together one layer after another;*

2. *The decoder consists of decoding layers that iteratively process the encoder's output and the decoder's output tokens so far, which is called the encoder-decoder attention.*

Definition 1.4. The **linear layer** at the end of the decoder is a simple fully connected neural network that projects the vector corresponding to the last token (the sequence of vectors is the output of the decoder), into a much larger vector called a logits vector.

Remark 1.5. *The encoded output vector $\bar{\mathbf{y}}_i$ should be a good representation of the next target vector $\bar{\mathbf{y}}_{i+1}$ but not of the input vector itself.*

Example 2. Let's assume that our model knows 10,000 unique English words that it is learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the linear layer.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

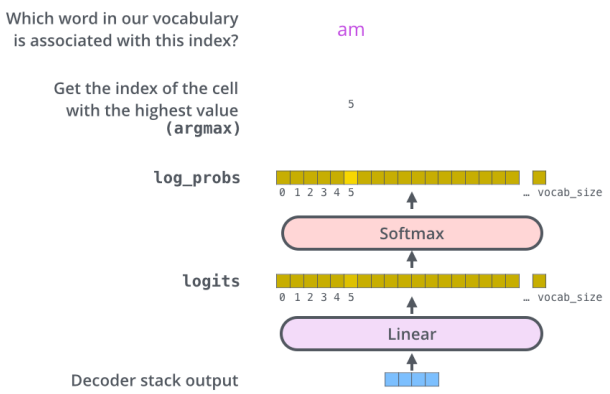

Figure 3: Caption

1.2 Self-attention mechanism

Definition 1.5. **Self-attention function** [Vaswani et al., 2017] is formulated as

$$\text{multiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{concat}(\text{head}_1, ..., \text{head}_h)W^O$$

$$\text{where } \text{head}_i = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V},$$

where d is the dimension of embedding vectors.

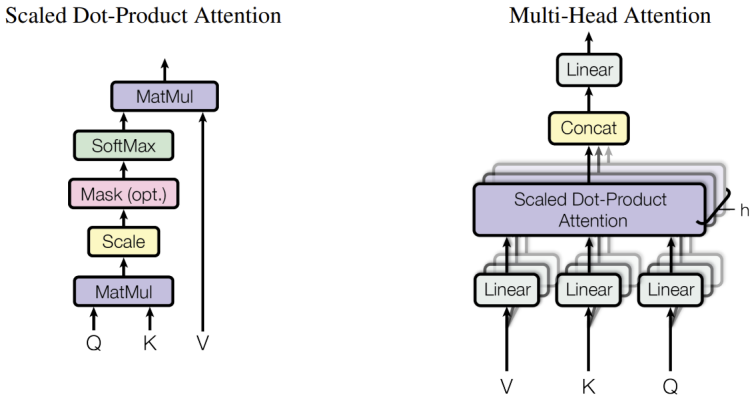

Figure 4: Left: Scaled Dot-Product Attention. Right: Multi-Head attention consists of several attention layers running in parallel.

Remark 1.6. *Scaling the attention score by $1/\sqrt{d}$ helps to control the magnitude of the value and keep the gradient size reasonable, which leads to easier training.*

Remark 1.7. *Softmax is conducted in each row in the attention filter in Figure 5.*

Remark 1.8. *The role of $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$:*

- ***Q** determines what information is being sought. **Q** is asking the question: "How much attention should I pay to the other tokens?"*
- ***K** provides a means to evaluate how relevant other tokens are in relation to the query. **K** answers the question: "How relevant is this token to the query?"*

- ***V*** *carries the actual information that will be used in the output.*
- ***Self-attention*** *allows the model to focus on different parts of the input sequence, while* ***cross-attention*** *allows the decoder to focus on the relevant parts of the encoder's output.*

Remark 1.9. *In practice,* $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{n \times d}$ *are identical as a concatenated sequence of embedding vectors together, where the dimension of the embedding vector is* d *and the number of vectors is* n*. Consequently,* $\text{softmax}(\mathbf{Q}\mathbf{K}^\top/\sqrt{d}) \in \mathbb{R}^{n \times n}$ *represents the attention score inside of the sequence.*

Remark 1.10. *The reason why the* $\mathbf{Q}, \mathbf{K}$ *use independent linear layer as oppose to use the same is because that if* $\mathbf{Q} = \mathbf{K}$*, then* $\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}$ *would be a symmetric matrix, which can potentially decreases the expressiveness of the model.*

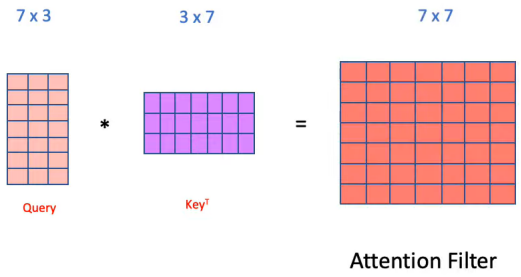

Figure 5: Illustration of how attention score is calculated. 7 is the sequence length and 3 is the dimension of token embedding. If talking about masking in decoder, the upper triangle here would be masked, namely the triangle along with the key dimension.

Definition 1.6. In **masked self-attention**, a triangular (or causal) mask is applied to the attention mechanism, which ensures that when the model is predicting the token at position i, it can only attend to tokens at positions 1 to i and not beyond.

Remark 1.11. *The masked self-attention mechanism allows the decoder to attend to its own previously generated tokens while generating the next token. The key feature here is that it prevents the decoder from "seeing" future tokens in the sequence.*

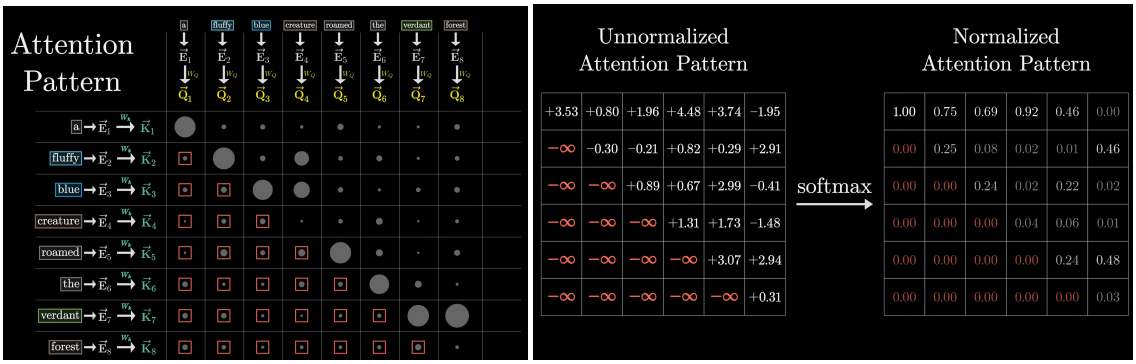

Figure 6: The lower triangle would be masked here, which corresponds to the upper triangle in Figure 5.

Example 3. Below shows the structure of a self attention layer:

```
import torch
```

```
import torch.nn.functional as F

class SelfAttention(nn.Module):
    def __init__(self, dim):
        super(SelfAttention, self).__init__()
        self.heads = heads
        self.dim = d_model

        # Define weight matrices for queries, keys, and values
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)

        # Output linear layer
        self.W_out = nn.Linear(d_model, d_model)

    def forward(self, x):
        batch_size, seq_len, _ = x.size()

        # Linearly transform queries, keys, and values
        queries = self.W_q(x)
        keys = self.W_k(x)
        values = self.W_v(x)

        # Reshape queries, keys, and values to have 'self.heads' as the new dimension
        queries = queries.view(batch_size, seq_len, self.dim)
        keys = keys.view(batch_size, seq_len, self.dim)
        values = values.view(batch_size, seq_len, self.dim)

        # Transpose dimensions for matrix multiplication
        queries = queries.transpose(1, 2)
        keys = keys.transpose(1, 2)
        values = values.transpose(1, 2)

        # Calculate self-attention scores
        scores = torch.matmul(queries, keys.transpose(-2, -1)) / torch.sqrt(torch.tensor(self.dim, dtyp
        attention_weights = F.softmax(scores, dim=-1)
        attended_values = torch.matmul(attention_weights, values)
        output = self.W_out(attended_values)

        return output
```

Remark 1.12. ***How is multi-head attention connected?*** *Multi-head attention is concatenated along the feature vector dimensions then go through a linear layer.*

Remark 1.13. *The relationship between head number, head size and hidden state size:*

$$\textit{hidden state size} = \textit{\# of head} \times \textit{head size}$$

For example, in BERT base, head size is 64, head number is 12, hence the hidden state size is $64 \times 12 = 768$.

Remark 1.14. *The role of "multi-headed" attention [Chatterjee, 2022]:*

1. *It expands the model's ability to focus on different positions.*
2. *It gives the attention layer multiple "representation subspaces". With multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.*

Definition 1.7. In **cross-attention**, the queries $\mathbf{Q}$ come from the decoder's previous layer (often the output of the masked self-attention layer), while the keys $\mathbf{K}$ and values $\mathbf{V}$ come from the encoder's final hidden states. These are to be used by each decoder layer, which allows the decoder to align its generated tokens with relevant parts of the input sequence.

Remark 1.15. *The cross-attention layer enables the decoder to attend to the entire output of the encoder, allowing it to integrate information from the input sequence into its generation process. This is crucial in tasks like machine translation, where the decoder needs to understand the entire input sentence (processed by the encoder) to generate a coherent output.*

1.3 Positional embedding mechanism

Definition 1.8. **Sinusoidal positional embeddings**: given a token at position i in a sequence, and a dimensional index d in the embedding vector, the positional embedding $Pos_{i,d}$ is defined as:

$$\mathrm{Pos}_{i,d} = \begin{cases} \sin\left(\frac{i}{10000^{\frac{d}{D}}}\right) & \text{, if } d \text{ is even} \\ \cos\left(\frac{i}{10000^{\frac{d}{D}}}\right) & \text{, if } d \text{ is odd} \end{cases}$$

where: i is the position of the token in the sequence, d is the dimension index of the embedding vector, D is the total dimension of the embedding vector, $10000^{\frac{d}{D}}$ scales the frequency of the sine and cosine functions.

Remark 1.16. *Motivation of sinusoidal positional embeddings:*

- ***Frequency Scaling:*** *The use of* $10000^{\frac{d}{D}}$ *ensures that each dimension of the positional embedding corresponds to a different frequency. This allows the model to encode positions at various scales, capturing both short-term and long-term dependencies.*
- ***Sine and Cosine Functions:*** *These functions are used because they create a continuous, non-linear embedding that varies smoothly with the position* i*. This smooth variation allows the model to easily interpolate between positions and capture sequential patterns.*

Remark 1.17. *Property of sinusoidal positional embeddings:*

- ***Deterministic****: The positional embedding is fixed and does not depend on the specific input data.* [1]
- ***Smooth variation****: Positional embeddings for nearby positions are similar, reflecting the sequential nature of the data.*
- ***Interoperability****: The sinusoidal functions allow the model to generalize to sequences longer than those seen during training.*

[1] The sinusoidal positional embeddings belongs to absolute positional embedding family.

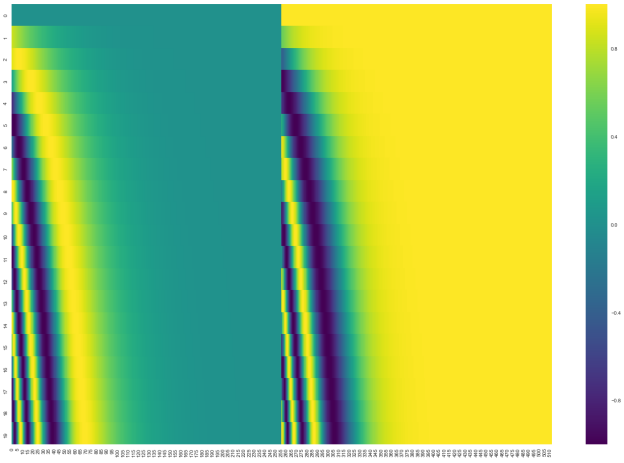

Figure 7: `Tensor2Tensor` implementation of the positional encoding for 20 words (rows) with an embedding size of 512 (columns). The values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). They're then concatenated to form each of the positional encoding vectors. The rows would be added to each token's embedding.

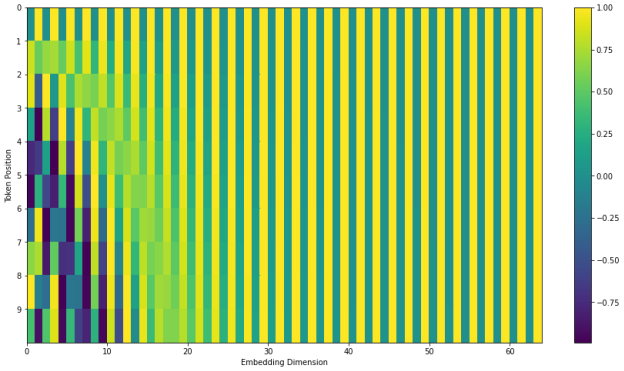

Figure 8: [Vaswani et al., 2017]'s positional encoding visualization for 10 words (rows) with an embedding size of 64 (columns). The method interweaves the two signals. The rows would be added to each token's embedding.

1.4 Inferencing of transformer

The encoder and the decoder are colored in green and red in figure below. Here, we show how the English sentence "I want to buy a car" is translated into German: "Ich will ein Auto kaufen".

To begin with, the encoder processes the complete input sequence into the light green vectors

Next, the input encoding together with the `[BOS]` vector, i.e. $\mathbf{y}_0$, is fed to the decoder (in different part). The decoder processes the inputs $\bar{\mathbf{X}}_{1:7}$ (as $\mathbf{K}, \mathbf{V}$ in cross-attention layer) and $\mathbf{y}_0$ (as input) to the get the first logit $\mathbf{l}_1$ (shown in darker red) to define the conditional distribution of the first target vector $p_\theta(\mathbf{y}|\mathbf{y}_0, \bar{\mathbf{X}}_{1:7})$

Next, the first target vector $\mathbf{y}_1 = \text{Ich}$ is sampled from the distribution (represented by the grey arrows) and can now be fed to the decoder again. The decoder now processes both $p_\theta(\mathbf{y}_2|\text{BOS}, \mathbf{y}_0, \bar{\mathbf{X}}_{1:7})$ to define the conditional distribution of the second target vector $\mathbf{y}_2$

It is important to understand that the encoder is only used in the first forward pass to map input sequence $\mathbf{X}_{1:7}$ to hidden state $\bar{\mathbf{X}}_{1:7}$. As of the second forward pass, the decoder can directly make use of the previously calculated encoding $\bar{\mathbf{X}}_{1:7}$. For clarity, let's illustrate the first and the second forward pass for our example above.

The encoded output vector $\bar{\mathbf{y}}_i$ should be a good representation of the *next* target vector $\bar{\mathbf{y}}_{i+1}$ but not of the input vector itself.

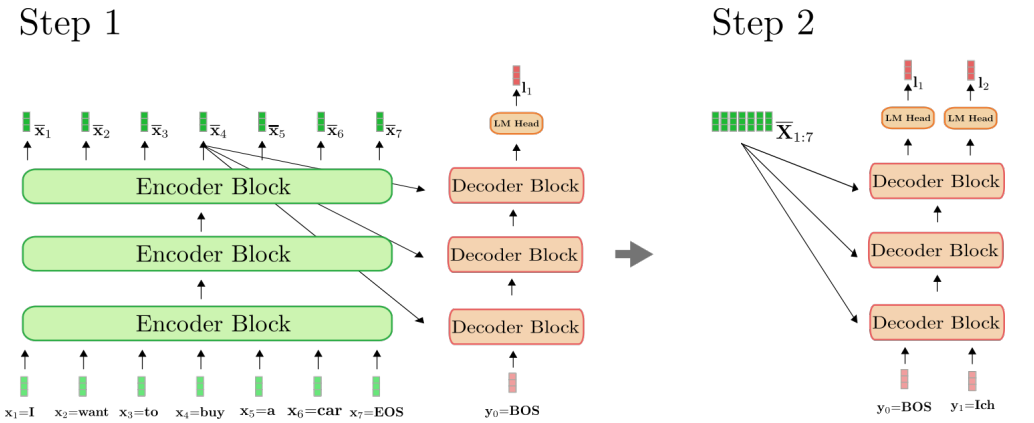

Figure 9: Autoregressive generation pipeline.

Definition 1.9. **Temperature** T is a float that controls the randomness of the sampling:

$$P(i) = \frac{e^{z_i/T}}{\sum_{j=1}^{n} e^{z_j/T}},$$

where $P(i)$ is the probability of selecting the i-th token and $e^{z_i/T}$ is the exponentially transforms the scaled logit.

Remark 1.18. *Meaning behind different temperature:*

- *Higher temperature (> 1): Makes the output more random by flattening the probability distribution (e.g., tokens with lower probabilities are more likely to be sampled).*
- *Temperature $= 1$: No change.*
- *Lower temperature (< 1): Makes the output more deterministic by sharpening the probability distribution (e.g., high-probability tokens are even more likely to be sampled).*
- *Temperature $= 0$: Greedy decoding; always selects the token with the highest probability.*

Definition 1.10. `top_p` is the float that controls the cumulative probability of the top tokens to consider. Must be in $(0, 1]$. Set to 1 to consider all tokens.

Example 4. Let's say the model predicts the following probabilities for the next word:

1. "mat" - 30%
2. "chair" - 25%
3. "sofa" - 20%
4. "table" - 15%
5. "floor" - 5%
6. "roof" - 3%

7. "dog" - 2%

If `top_p = 0.7`, then it would include "mat" (30%), "chair" (25%), and "sofa" (20%), as their sum (75%) exceeds 0.7. The selection is among these three words, maintaining their relative probabilities.

In practice:

- Lower `top_p` values (like 0.5) make the output more focused and predictable.
- Higher values (like 0.9) allow for more diversity but may occasionally include less likely options.
- Setting `top_p = 1` is equivalent to not using top_p at all, considering all options based on their original probabilities.

This method helps in controlling the trade-off between creativity and focus in generated text. It's particularly useful in maintaining coherent outputs while still allowing for some variability.

Table 1: Comparison of the effects of high temperature and high top-p on randomness in text generation.

Aspect	**High Temperature**	**High Top-p**
Distribution Shape	Flattens the entire distribution.	Keeps the shape but truncates the tail.
Token Pool	All tokens are considered.	Only tokens contributing to cumulative p are considered.
Likelihood of Rare Tokens	Rare tokens have a higher probability.	Rare tokens may still be excluded if cumulative probability p is reached.
Effect on Output	Often more chaotic and unpredictable.	More focused diversity while excluding very unlikely tokens.

Definition 1.11. `vLLM` is an open-source Python package designed to optimize serving large language models (LLMs) in real time, with high efficiency, low latency, and scalability. It is particularly built for use cases like generative inference with modern transformer-based models.

Remark 1.19. *PagedAttention in vLLM addresses these challenges by introducing a paging mechanism for the KV cache, similar to how operating systems manage virtual memory:*

- *Instead of preallocating a single contiguous block of memory for the KV cache, PagedAttention divides the KV cache into small, fixed-size "pages"*
- *By breaking the KV cache into smaller pages, PagedAttention avoids memory fragmentation issues common with dynamic memory allocation in GPUs.*
- *Efficient management of KV cache allows the model to handle long sequences and large batch sizes without running into memory bottlenecks.*

Remark 1.20. *Continuous Batching in vLLM solves these problems by dynamically aggregating incoming requests in real-time to form optimal batches on the fly:*

- *Static batching requires padding shorter sequences to match the longest one in the batch, wasting compute resources. Serving systems must wait until a batch is "full" before processing it, which adds latency for real-time inference.*

- *Continuous Batching dynamically combines incoming requests to maximize GPU utilization without unnecessary delays or padding overhead.*
- *New inference requests can be added to the batch continuously, even while the GPU is processing the current batch. This eliminates the need to wait for a full static batch to start processing.*
- *Sequences of different lengths are handled efficiently without requiring padding. This is because vLLM uses the PagedAttention mechanism to handle sparse memory access for KV cache.*

1.5 Training of transformer

During training, an untrained model would go through the exact same forward pass. But since we are training it on a labeled training dataset, we can compare its output with the actual correct output. Once we define our output vocabulary, we can use a vector of the same width to indicate each word in our vocabulary. This also known as one-hot encoding.

We want the output to be a probability distribution indicating the word we expect, but since this model is not yet trained, that's unlikely to happen just yet. We use cross-entropy and Kullback–Leibler divergence to calculate the difference between the predicted and target distribution.

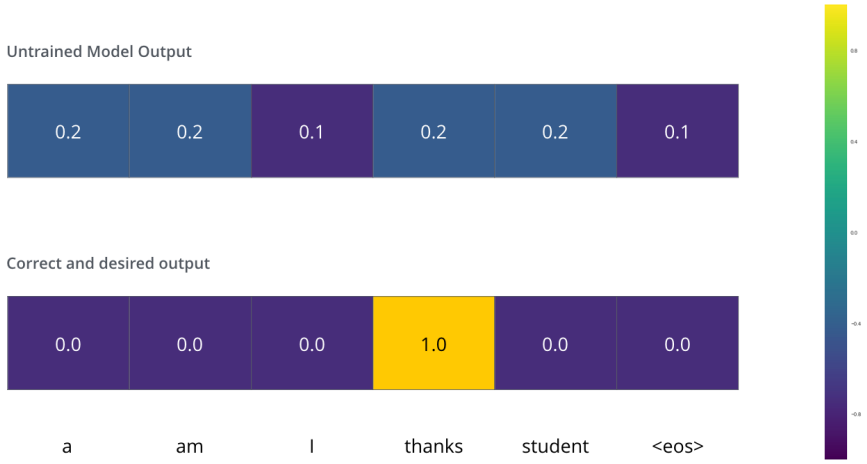

Figure 10: Cross-entropy between ground truth one-hot vector and model's prediction.

1.5.1 Warmup in transformer training

In standard deep learning training, the learning rate typically starts at a predefined value and decays over time. This prevents overshooting and ensures convergence as training progresses.

However for Transformers, training often starts with a warmup phase, where the learning rate gradually increases to a peak value before following a decay schedule (e.g., inverse square root or linear decay).

Transformers rely heavily on layer normalization (LayerNorm) to stabilize training. At the beginning of training, the model's weights are randomly initialized. The outputs of LayerNorm can be very small, leading to vanishing gradients. If the gradients are too small, the Adam optimizer may struggle to make meaningful updates, especially early in training.

2 LLM architecture

2.1 Encoder-decoder models (T5)

Definition 2.1. T5 (Text-to-Text Transfer Transformer) is a transformer-based model developed by Google Research, designed to handle a wide range of natural language processing (NLP) tasks by framing them as a unified text-to-text problem. T5 is pre-trained on a massive dataset called C4 (Colossal Clean Crawled Corpus) and fine-tuned on task-specific datasets. Its encoder-decoder architecture allows it to effectively generate and understand text, making it a powerful model for diverse language generation tasks.

2.2 Encoder-only models (BERT)

Definition 2.2. **Bidirectional encoder representations from transformers (BERT)** is a language model learned by self-supervised learning to represent text as a sequence of vectors, i.e. masked token prediction and next sentence prediction.

Remark 2.1. *BERT is an **encoder-only** transformer architecture. At a high level, BERT consists of 4 modules:*

1. ***Tokenizer** converts a piece of English text into a sequence of tokens.*
2. ***Embedding** converts the sequence of tokens into an array of real-valued vectors representing the tokens. It represents the conversion of discrete token types into a lower-dimensional Euclidean space.*
3. ***Encoder**: a stack of Transformer encoder blocks with self-attention, but without causal masking.*
4. ***Task head** converts the final representation vectors into one-hot encoded tokens again by producing a predicted probability distribution over the token types. It can be viewed as a simple decoder, decoding the latent representation into token types, or as an "un-embedding layer".*

Definition 2.3. In **masked language modeling**, 15% of tokens would be randomly selected for masked-prediction task, and the training objective was to predict the masked token given its context. In more detail, the selected token is

- replaced with a `[MASK]` token with probability 80%;
- replaced with a random word token with probability 10%;
- not replaced with probability 10%.

Remark 2.2. *The reason not all selected tokens are masked is to avoid the dataset shift problem. The dataset shift problem arises when the distribution of inputs seen during training differs significantly from the distribution encountered during inference.*

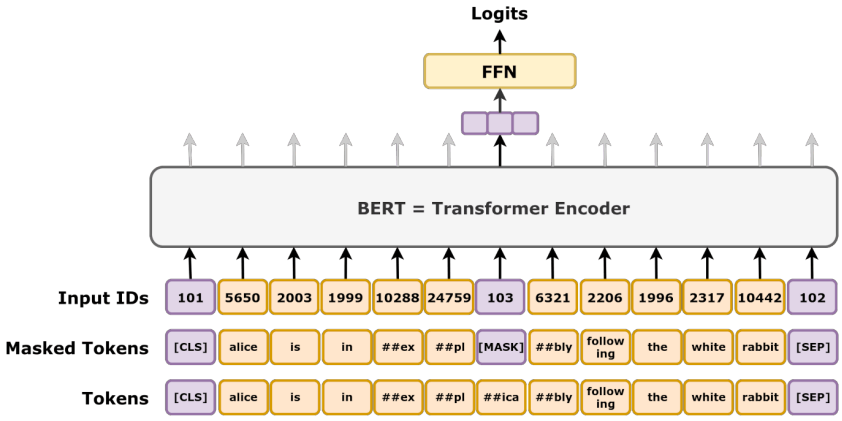

Figure 11: Masked language modeling task.

Definition 2.4. For **next sentence prediction**, given two spans of text, the model predicts if these two spans appeared sequentially in the training corpus, outputting either `[IsNext]` or `[NotNext]`.

The first span starts with a special token `[CLS]` (for "classify"). The two spans are separated by a special token `[SEP]` (for "separate"). After processing the two spans, the 1-st output vector (the vector coding for `[CLS]`) is passed to a separate neural network for the binary classification into `[IsNext]` and `[NotNext]`.

Example 5.

- Given "`[CLS]` my dog is cute `[SEP]` he likes playing" the model should output token `[IsNext]`;
- Given "`[CLS]` my dog is cute `[SEP]` how do magnets work" the model should output token `[NotNext]`.

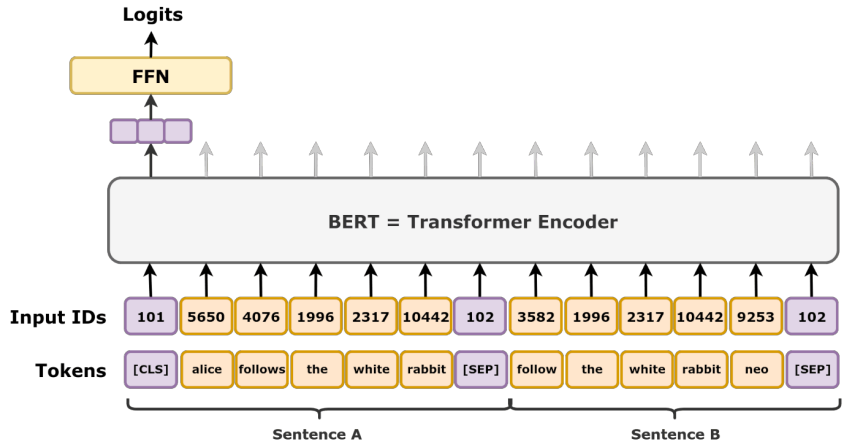

Figure 12: Next sentence prediction task.

Example 6. As an encoder only language model, `BERT-base` has following structure:

```
BertModel(
  (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
```

```
    (position_embeddings): Embedding(512, 768)
    (token_type_embeddings): Embedding(2, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
    (layer): ModuleList(
      (0): BertLayer()
      (1): BertLayer()
      ...
      (11): BertLayer()
    )
  )
  (pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh()
  )
)
```

Example 7. As an encoder layer, each `BertLayer` has following structure:

```
BertLayer(
    (attention): BertAttention(
      (self): BertSelfAttention(
        (query): Linear(in_features=768, out_features=768, bias=True)
        (key): Linear(in_features=768, out_features=768, bias=True)
        (value): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
    (intermediate): BertIntermediate(
      (dense): Linear(in_features=768, out_features=3072, bias=True)
    )
    (output): BertOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
```

2.3 Decoder-only models (Llama/GPT)

2.3.1 Why decoder-only model is now predominant?

Remark 2.3. ***decoder-only architecture theoretically has a stronger expressive ability***[2] *As we known, the attention matrix is typically derived from low-rank decomposition matrices (multiplying an $n \times d$ (where $n \gg d$) matrix by a $d \times n$ matrix) followed by a softmax operation. Due to the low-rank nature of this kind of attention matrix (the maximum rank of $d \times n$ matrix is d), it suffers from a decrease in expressive power, namely cannot fully span the n-dimensional space.*

In contrast, the attention matrix in a decoder-only architecture is a lower triangular matrix. It's important to note that the determinant of a triangular matrix equals the product of its diagonal elements. Given that the softmax operation ensures all diagonal elements are positive, the determinant is also positive. This implies that the attention matrix in a decoder-only architecture is always positive definite, a.k.a. full-rank. Full-rank implies a theoretically stronger expressive capability.

In other words, the attention matrix in a decoder-only architecture theoretically has a stronger expressive ability, and switching to bidirectional attention could actually reduce this capacity.

Remark 2.4. ***Other advantages****:*

- *Next token prediction as an pretraining task is much more difficult than masked language modeling or next sentence prediction for encoder-only model, like BERT.*
- *Causal attention itself is an implicit positional embedding.*
- *Decoder-only model support KV-cache reusage.*

Remark 2.5. ***How does the decoder-only models incorporate information from prompt?*** *In models that are decoder-only, the prompt and generated text are part of the same sequence. That is, in a decoder-only model, everything is handled within the same sequence through self-attention.*

Here's a more detailed breakdown:

1. ***Initial input****: When you first input a prompt, the entire prompt is fed into the language model as a sequence of tokens. The model processes this sequence using its self-attention mechanism to understand the context and relationships between the tokens.*
2. ***First token generation****: The model then generates the first token of the output based on the prompt. This first token is predicted as the most likely continuation of the sequence according to the model's understanding.*
3. ***Appending the generated token****: Once the first token is generated, it is appended to the end of the original prompt. The sequence now consists of the original prompt plus the newly generated token.*
4. ***Repeat process****: The model then processes this updated sequence (original prompt + first generated token) to generate the next token. This process is repeated, with each new token being appended to the sequence and the updated sequence being fed back into the model.*
5. ***Autoregressive generation****: This loop continues, with the model autoregressively generating one token at a time, each time taking into account the entire sequence of the original prompt plus all previously generated tokens.*

Remark 2.6. ***Stopping criteria of LLMs****:*

[2] `https://kexue.fm/archives/9529`

- ***EOS token****: The most common stopping criterion where the model stops generating when it predicts the EOS token.*
- ***Maximum length****: Ensures the output doesn't exceed a predefined token length.*
- ***Repetition penalties/N-gram blocking****: Prevents repetitive outputs and indirectly influences the stopping point.*
- ***Probability threshold****: Stops generation if token probabilities fall below a certain level.*
- ***Heuristics and rules****: Custom criteria based on the specific application.*

Example 8. Below is the snippet of llama decoder layer (which presents the residual mechanism):

```
class LlamaDecoderLayer(nn.Module):
def __init__(self, config: LlamaConfig, layer_idx: int):
    super().__init__()
    self.hidden_size = config.hidden_size

    self.self_attn = LLAMA_ATTENTION_CLASSES[config._attn_implementation](config=config, layer_idx=

    self.mlp = LlamaMLP(config)
    self.input_layernorm = LlamaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)
    self.post_attention_layernorm = LlamaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)

def forward(
    self,
    hidden_states, attention_mask, position_ids, past_key_value,
    output_attentions, use_cache, cache_position, position_embeddings,
    **kwargs,
) -> Tuple[torch.FloatTensor, Optional[Tuple[torch.FloatTensor, torch.FloatTensor]]]:
    residual = hidden_states

    hidden_states = self.input_layernorm(hidden_states)

    # Self Attention
    hidden_states, self_attn_weights, present_key_value = self.self_attn(
        hidden_states=hidden_states,
        attention_mask=attention_mask,
        position_ids=position_ids,
        past_key_value=past_key_value,
        output_attentions=output_attentions,
        use_cache=use_cache,
        cache_position=cache_position,
        position_embeddings=position_embeddings,
        **kwargs,
    )
    hidden_states = residual + hidden_states
```

```
# Fully Connected
residual = hidden_states
hidden_states = self.post_attention_layernorm(hidden_states)
hidden_states = self.mlp(hidden_states)
hidden_states = residual + hidden_states

outputs = (hidden_states,)

if output_attentions:
    outputs += (self_attn_weights,)

if use_cache:
    outputs += (present_key_value,)

return outputs
```

2.3.2 Unique features in Llama

Definition 2.5. **Rotary positional embeddings (RoPE)** applies a rotation matrix to the existing word embeddings, as opposed to adding a separate positional embedding. The figure below credits to youtube.

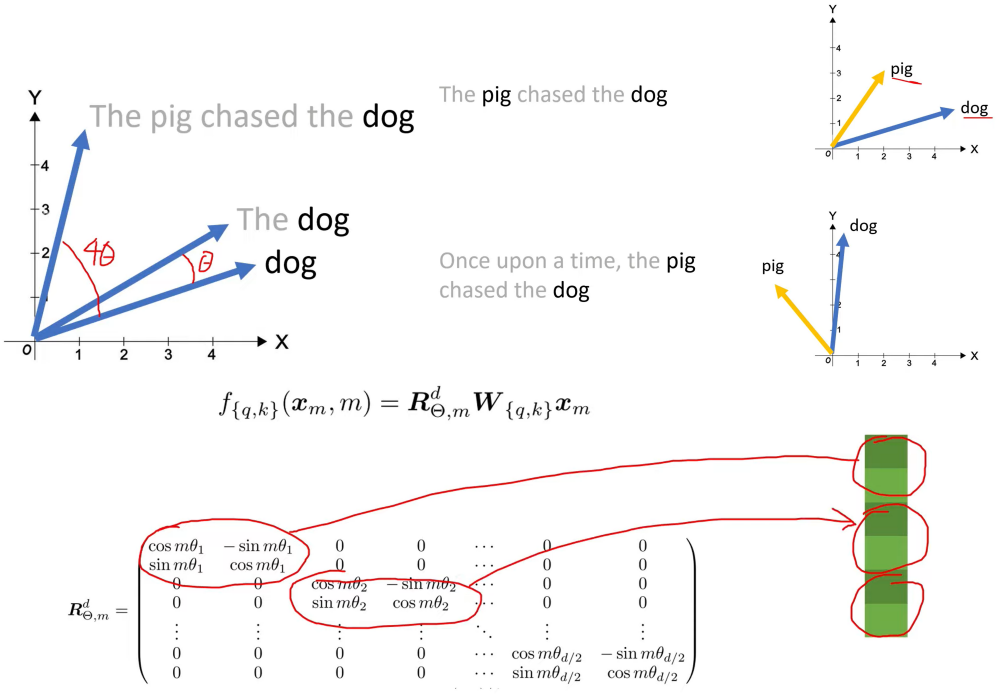

Figure 13: RoPE overview.

Definition 2.6. **Root mean square normalization (RMS norm)** is a normalization technique that normalizes the input data by dividing it by the square root of the mean of the squared values. Mathematically, RMS norm can be

represented as:

$$\text{norm}(x) = \sqrt{\sum_{i}^{n} x_i^2}$$

Remark 2.7. *The Llama model uses RMS Norm instead of the more traditional layer normalization (LN) or batch normalization (BN) for several reasons:*

- ***Stability**: RMS Norm is more stable than LN and BN, which is less sensitive to outliers than standard deviation.*
- ***Computational efficiency**: RMS Norm is computationally more efficient than LN and BN, by removing the need to center the inputs (i.e., subtract the mean). It only normalizes by the root mean square of the input, which can make the computation slightly faster and simpler.*

Remark 2.8. *RMS norm is performed sample-wise.*

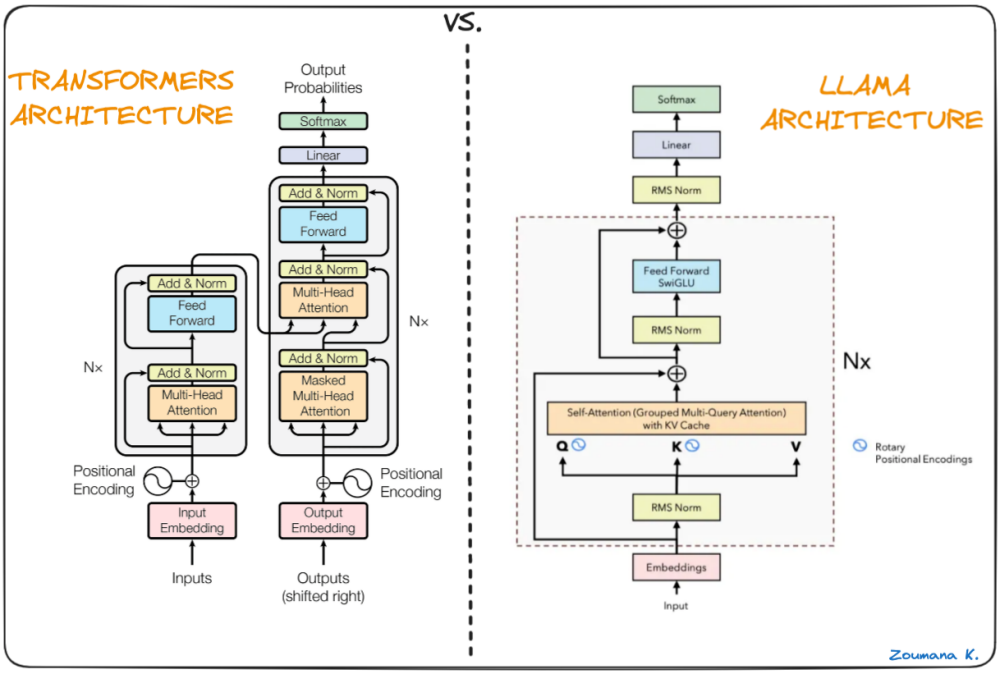

Figure 14: Transformers vs. Llama

Definition 2.7. **Batch normalization** [Ioffe and Szegedy, 2015] is defined as

$$\mu_c = \frac{1}{HWN} \sum_{n}^{N} \sum_{w}^{W} \sum_{h}^{H} x_{cnhw}$$

$$\sigma_c^2 = \frac{1}{HWN} \sum_{n}^{N} \sum_{w}^{W} \sum_{h}^{H} (x_{cnhw} - \mu_c)^2$$

$$\hat{x}_c = \frac{x_c - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$$

where x are the values of input over a mini-batch, C, N, H, W are the channel, batch, height, and width size, respectively. Batch normalization should be performed by each channel.

Remark 2.9. *The inventors of batch normalization postulated informally that this drift in the distribution of such variables could hamper the convergence of the network. Intuitively, we might conjecture that if one layer has variable values that are 100 times that of another layer, this might necessitate compensatory adjustments in the learning rates.*

Remark 2.10. ***Batch size matters.*** *When applying batch normalization, the choice of batch size may be even more significant than without batch normalization. In some preliminary research, [Teye et al., 2018] and [Luo et al., 2018] relate the properties of batch normalization to Bayesian priors and penalties respectively. This sheds some light on the puzzle of why batch normalization works best for moderate minibatch sizes in the* $[50, 100]$ *range.*

Remark 2.11. ***BN in training and testing differentiates slightly.*** *Typically, during the training, we use the statistics (mean and variance) from the specific mini-batch. During inference time, the statistics of the population is instead used in the batch normalization. Recall that dropout also exhibits this characteristic. Hence always remember to enable evaluation mode of the model during the inference time via* `model.eval()`.

Remark 2.12. *BN is typically implemented after the convolutional or fully-connected layer and before the activation function.*

Definition 2.8. **Layer normalization** is defined as

$$\mu_n = \frac{1}{HWC} \sum_{c}^{C} \sum_{w}^{W} \sum_{h}^{H} x_{cnhw}$$
$$\sigma_n^2 = \frac{1}{HWC} \sum_{c}^{C} \sum_{w}^{W} \sum_{h}^{H} (x_{cnhw} - \mu_n)^2$$
$$\hat{x}_n = \frac{x_n - \mu_n}{\sqrt{\sigma_n^2 + \epsilon}}$$

where x are the values of input over a mini-batch, C, N, H, W are the channel, batch, height, and width size, respectively. Layer normalization should be performed by each sample in a mini-batch.

Remark 2.13. *When batch size is small or dealing with sequential data, layer norm is a better choice than batch norm. When batch size is small, the batch norm would be relatively unstable given the small sample size.*

In NLP tasks, the length of sentences often varies. This makes it uncertain what the correct normalization constant would be if using batch normalization. Different batches would have different normalization constants, which can lead to instability during training.

2.4 Mixture of expert model (Mixtral)

Definition 2.9. **Mixtral** is a large language model (LLM) using Mixture of Experts.

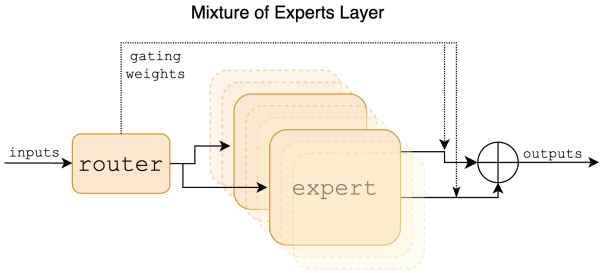

Figure 15: Mixture of Experts Layer. Each input vector is assigned to 2 of the 8 experts by a router. The layer's output is the weighted sum of the outputs of the two selected experts. In Mixtral, an expert is a standard feedforward block as in a vanilla transformer architecture.

Mixtral uses a Mixture of Experts (MoE) design, where only a subset of the model's parameters is activated for each forward pass. Mixtral-8x7B consists of 8 experts, each of which is a 7B parameter dense model. During inference or training, only 2 out of the 8 experts are activated for each token. As a result, although the total parameter count is 56B (8 x 7B), only 14B parameters are used at any given time.

2.4.1 How Mixtral inferences

When a sequence of tokens is fed into the model, each token passes through a routing mechanism. The routing mechanism is typically a lightweight, trainable neural network that calculates a score for each expert based on the token's input representation. These scores determine which 2 experts out of the 8 will be activated for each token.

After the top-2 experts are chosen, the token input is passed to the two selected experts. Each expert processes the input independently (in parallel), producing its own output (hidden representation).

The outputs of the two activated experts are combined using the scores produced by the routing mechanism. The combination is typically a weighted sum of the two outputs. The combined output from the two experts becomes the token's representation, which is then passed to the next layer of the transformer model. This process repeats for each token at each layer of the model.

Each MoE layer in the transformer has its own routing network. The routing network at each MoE layer is responsible for selecting the top-2 experts for every token at that specific layer. This means if there are N MoE layers in the transformer, there are N routing networks.

The routing network is typically a small, trainable projection layer (e.g., a linear layer or MLP) that computes the routing scores for all experts. In practice, not all transformer layers are MoE layers.

2.4.2 How is Mixtral trained

The 8 experts in Mixtral (or most Mixture of Experts (MoE) models) are not trained individually on different sources of data. Instead, they are trained jointly on the same dataset as part of the overall model training process.

For each token in the training data, the routing network dynamically selects the top-2 experts based on the routing scores. Only the selected experts' parameters (weights) are updated during that forward pass. The non-selected experts do not participate in the computation for that token and do not receive gradients.

3 LLM training

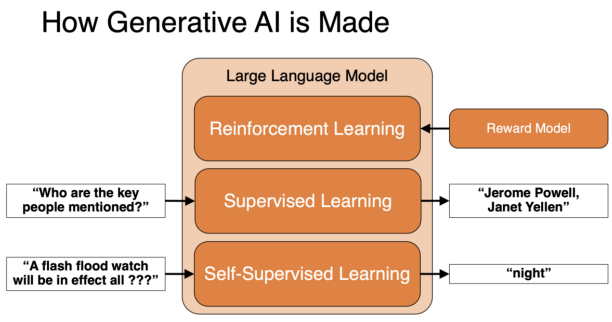

Figure 16: The three stages of LLM training.

3.1 Pre-training

Pretraining a LLM, like GPT, is a self-supervised learning procedure:

- Data collection: Gather a large corpus of unlabeled text data, such as articles, books, and web pages. This serves as the training dataset.
- Data preparation: Process the text data into sequences of tokens. The input is a string of tokens, and the target output is the next token immediately following the input sequence.
- Training objective: Train the model to predict the next token using a classification objective, typically cross-entropy loss. This loss measures the difference between the model's predicted probability distribution over the vocabulary and the actual target token. The process aims to minimize the loss, improving the model's ability to generate coherent text.

Definition 3.1. **Colossal Clean Crawled Corpus (C4)** is a colossal, cleaned version of Common Crawl's web crawl corpus dataset which is two orders of magnitude larger than Wikipedia. It is widely used for LLMs like T5 (Text-to-Text Transfer Transformer).

Remark 3.1. *To accurately measure the effect of scaling up the amount of pre-training, one needs a dataset that is not only high quality and diverse, but also massive. Existing pre-training datasets don't meet all three of these criteria — for example, text from Wikipedia is high quality, but uniform in style and relatively small for our purposes, while the Common Crawl web scrapes are enormous and highly diverse, but fairly low quality.*

3.2 Post-training

3.2.1 Supervised fine-tuning (SFT)

Definition 3.2. **Supervised fine-tuning (SFT)** involves adapting a pre-trained model to a specific task or domain by training it on a labeled dataset. The model learns to map inputs to the correct outputs based on this labeled data.

Remark 3.2. *The author of [Ouyang et al., 2022] fine-tune GPT-3 on the labeler demonstrations using supervised learning. Their SFT models overfit on validation loss after 1 epoch; however, they find that training for more epochs helps both the RM score and human preference ratings, despite this overfitting.*

Remark 3.3. *SFT vs. RLHF:*

- *Supervised fine-tuning is a training process where we provide a prompt to the model along with the desired output, and the model is trained to predict the expected tokens exactly as specified in the labeled data. This aligns the model's behavior with the supervised examples we provide.*
- *Reinforcement learning with human feedback (RLHF), on the other hand, uses a reward model to evaluate the quality of the generated output based on how well it meets our standards or preferences. If the generated output does not meet the desired criteria, the model receives a high penalty (or large negative reward) during training, encouraging it to adjust its future behavior accordingly.*

Definition 3.3. **Instruction fine-tuning** involves training a model on datasets that consist of instructions (commands or tasks written in natural language) paired with desired outputs.

Example 9.

- Input: `"Summarize the following article: [text]"`
- Output: `"[summary of the article]"`

Definition 3.4. InstructGPT [Ouyang et al., 2022] is both a dataset and the GPT model finetuned on the dataset.

Definition 3.5. FLAN (Fine-Tuned LAnguage Net) [Wei et al., 2022] is a dataset released by Google, which focus on multi-task fine-tuning to improve instruction generalization. FLAN-T5 is the T5 model finetuned on the dataset.

Example 10. Creating a dataset of instructions from scratch to fine-tune the model would take a considerable amount of resources. Therefore, FLAN instead make use of templates to transform existing datasets into an instructional format.

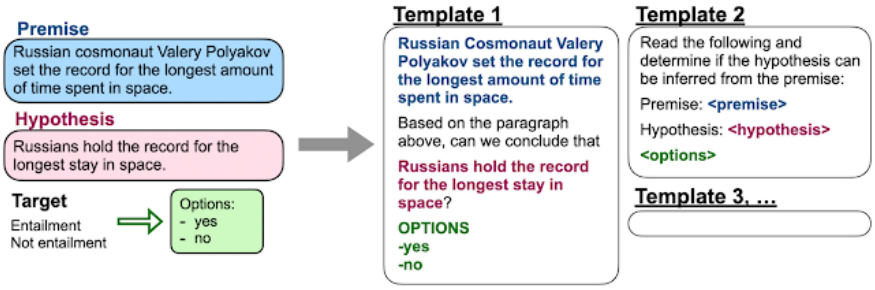

Figure 17: Example templates for a FLAN dataset.

3.2.2 Reinforcement learning

Definition 3.6. **Proximal policy optimization (PPO)** in LLM is a reinforcement learning algorithm used to fine-tune LLMs to align their outputs with human preferences or specific criteria. It works by: [Ouyang et al., 2022]

1. **Collect demonstration data, and train a supervised policy (model)**: the labelers provide demonstrations of the desired behavior on the input prompt distribution;

2. **Collect comparison data, and train a reward model**: collecting a dataset of comparisons between model outputs, where labelers indicate which output they prefer for a given input, and then training a reward model to predict the human-preferred output;

3. **Optimize a policy against the reward model using PPO**: using the output of the reward model as a scalar reward and fine-tunning the supervised policy to optimize this reward.

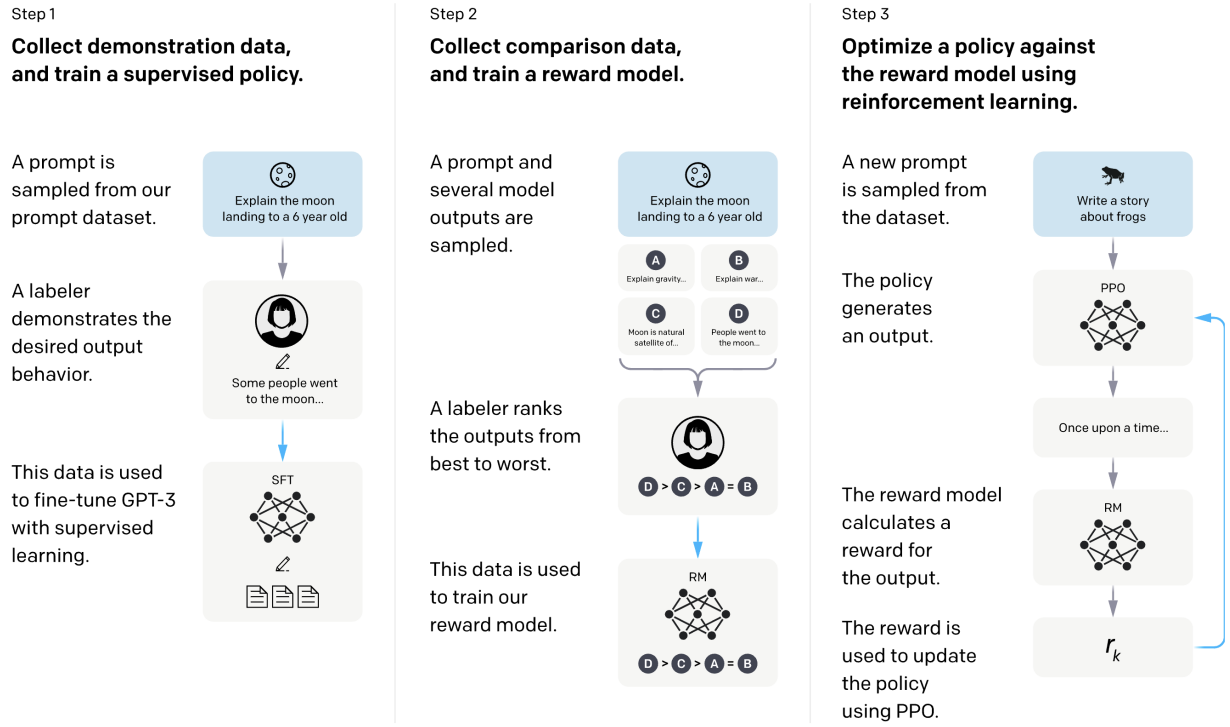

Figure 18: A diagram illustrating the three steps of PPO in RLHF: (1) supervised fine-tuning (SFT), (2) reward model (RM) training, and (3) reinforcement learning via proximal policy optimization (PPO) on this reward model.

Definition 3.7. **Reward model** in PPO aims to minimize following loss function:

$$\mathcal{L}(\theta) = -\frac{1}{\binom{K}{2}} \mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}} \left[\log\left(\sigma\left(r_\theta(x,y_w) - r_\theta(x,y_l)\right)\right)\right]$$

where $r_\theta(x, y_w)$ is the scalar output of the reward model for prompt x and completion y with parameters θ, y_w is the preferred completion out of the pair of y_w and y_l, and D is the dataset of human comparisons. K is the response for labeler to rank, which produces $\binom{K}{2}$ comparisons for each prompt shown to a labeler. $\sigma(\cdot)$ is the sigmoid function.

Remark 3.4. *Starting from the SFT model with the final unembedding layer removed, [Ouyang et al., 2022] trained a model to take in a prompt and response, and output a scalar reward. In the paper they only use 6B RMs, as this saves a lot of compute, and they found that 175B RM training could be unstable and thus was less suitable to be used as the value function during RL.*

Definition 3.8. **Reinforcement learning from human feedback (RLHF)** leverages human feedback to guide the model's learning process, making it more effective at generating desirable outputs.

Example 11. How RLHF Works

1. **Initial model training**: The process typically starts with a pre-trained language model (like GPT-3) that has been trained on a large corpus of text using supervised learning.
2. **Collecting human feedback**: Human evaluators are shown pairs or sets of outputs generated by the model. They provide feedback by ranking these outputs or selecting which one they prefer. This feedback is used to create a dataset of human preferences.

3. **Training the reward model**: A reward model is trained on the dataset of human preferences. The reward model learns to predict the human preference for new outputs generated by the model. Essentially, it assigns a "reward score" to each output based on how well it aligns with human preferences.
4. **Reinforcement learning with the reward model**: The language model is then fine-tuned using reinforcement learning, where the reward model's scores guide the training. The model's objective is to maximize the expected reward, meaning it should generate outputs that are more likely to be preferred by humans according to the reward model.
5. **Iterative improvement**: The process is often iterative. After each round of training, new outputs are generated, and human feedback is collected again. This feedback is used to further improve the reward model and refine the language model's policy.
6. **Deployment and monitoring**: Once the model's performance is satisfactory, it can be deployed. However, continuous monitoring and occasional retraining with updated human feedback may be necessary to ensure the model remains aligned with human values over time.

Definition 3.9. **Direct preference optimization (DPO)** [Rafailov et al., 2024] is a reinforcement learning algorithm used to fine-tune LLMs to align their outputs with human preferences or specific criteria with the aim of

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta\log\frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\right)\right].$$

where x is the given prompt, β is a parameter controlling the deviation from the base reference policy, π_{ref} is the base reference policy and π_θ is the language model policy.

Remark 3.5. *The human feedback or preference in DPO is incorporated into the loss function via the assignment of y_w, y_l with in the data point.*

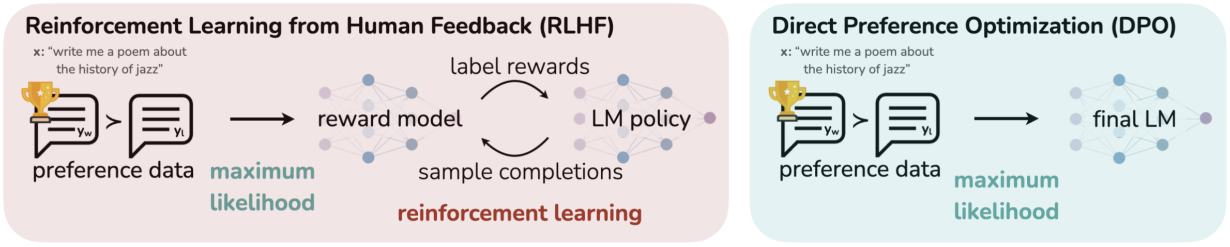

Figure 19: DPO directly optimizes for the policy best satisfying the preferences with a simple classification objective, fitting an implicit reward model whose corresponding optimal policy can be extracted in closed form.

Remark 3.6. *DPO uses a reference model (usually the initial pre-trained model) to regularize the training process. This helps prevent the model from deviating too far from its initial behavior.*

Remark 3.7. *Comparison between PPO and DPO:*

- *PPO is generally stable but requires careful tuning of hyperparameters. DPO is potentially more stable due to direct optimization;*
- *DPO is more sample-efficient since it directly uses preference data.*

3.2.3 Efficient post-training

Definition 3.10. **Low-Rank Adaptation (LoRA)** is a parameter-efficient fine-tuning technique that decomposes weight update matrices $\Delta W \in \mathbb{R}^{d\times d}$ into two smaller matrices $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{d\times r}$, while the original model

weights W are "frozen" and not directly updated. The new weights $W' = W + \Delta W = W + AB^T$. The figure below credits to this blog.

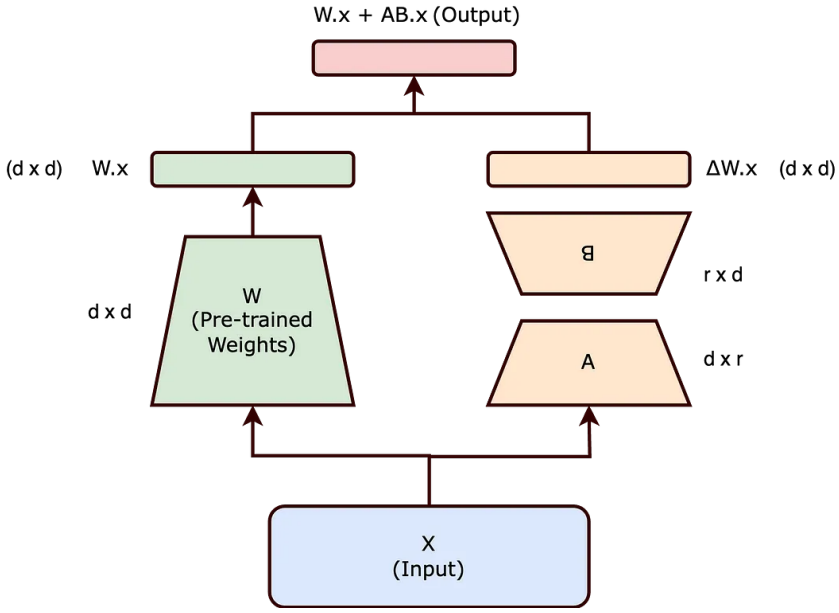

Figure 20: LoRA overview.

Remark 3.8. *By choosing matrices A and B to have a lower rank r, the number of trainable parameters is significantly reduced. For example, if W is a $d \times d$ matrix, traditionally, updating W would involve d^2 parameters. However, with B and A of sizes $d \times r$ and $d \times r$ respectively, the total number of parameters reduces to $2dr$, which is much smaller when $d \gg r$.*

Remark 3.9. ***What are some approaches to reduce the computational cost of LLMs?***

- ***Model pruning:*** *Removing less important weights or neurons from the model to reduce its size and computational requirements.*
- ***Quantization:*** *Converting the model weights from higher precision (e.g., 32-bit floating-point) to lower precision (e.g., 8-bit integer) reduces memory usage and speeds up inference.*
- ***Distillation****: Training a smaller model (student) to mimic the behavior of a larger, pre-trained model (teacher) to achieve similar performance with fewer resources.*

4 LLM inference

Definition 4.1. **Key-Value Cache (KV Cache)** is a basic optimization technique in decoder-only Transformers which reduces compute at the expense of increased memory utilization.

Example 12. Conceptually, during generation, one token is generated for each forward pass, appended to the input, and then the input (now with +1 sequence length) is passed back into the model for another forward pass. However doing it this naive way is wasteful because:

- It recomputes previous key, value, and attention rows which are not needed.

- The other parts of the network which are per-token also get used again, wasting computation.

To make it more intuitive, in Figure 21, we can see that the $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ matrices just have an additional row, and previous rows were unaffected. Also note how the attention matrix just has one additional row, and consequently, the output also only has one extra row. All previous rows were unaffected due to the self-attention mask. The figure below credits to this blog.

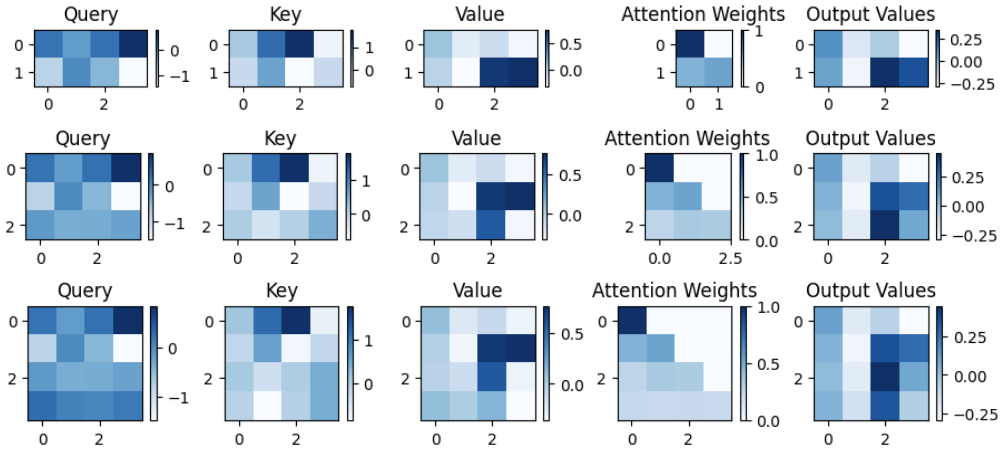

Figure 21: What the $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, attention matrix, and output values look like after passing an 2/3/4 tokens to single attention head.

Remark 4.1. *Another perspective to see why only* $\mathbf{K}, \mathbf{V}$ *need to be cached:*

We write $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ *in following form:*

$$\mathbf{K} = \begin{bmatrix} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \vdots \\ \mathbf{k}_T \end{bmatrix}, \quad \mathbf{Q} = \begin{bmatrix} \mathbf{q}_1 \\ \mathbf{q}_2 \\ \vdots \\ \mathbf{q}_T \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_T \end{bmatrix}, \quad \mathbf{k}_i, \mathbf{q}_i \in \mathbb{R}^{1 \times d_k}, \, \mathbf{v}_i \in \mathbb{R}^{1 \times d_v}, \, i = 1, 2, \ldots, T$$

where d_k *is the dimension of* $\mathbf{Q}, \mathbf{K}$*, and* d_v *is the dimension of* $\mathbf{V}$*,* T *is the sequence length. Then we have*

$$\mathbf{Q}\mathbf{K}^\top = \begin{bmatrix} \mathbf{q}_1 \\ \mathbf{q}_2 \\ \vdots \\ \mathbf{q}_T \end{bmatrix} \begin{bmatrix} \mathbf{k}_1^\top & \mathbf{k}_2^\top & \cdots & \mathbf{k}_T^\top \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1 \cdot \mathbf{k}_1 & \mathbf{q}_1 \cdot \mathbf{k}_2 & \cdots & \mathbf{q}_1 \cdot \mathbf{k}_T \\ \mathbf{q}_2 \cdot \mathbf{k}_1 & \mathbf{q}_2 \cdot \mathbf{k}_2 & \cdots & \mathbf{q}_2 \cdot \mathbf{k}_T \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{q}_T \cdot \mathbf{k}_1 & \mathbf{q}_T \cdot \mathbf{k}_2 & \cdots & \mathbf{q}_T \cdot \mathbf{k}_T \end{bmatrix}$$

After applying softmax function S*, we have*

$$S(\mathbf{Q}\mathbf{K}^\top)\mathbf{V} = \begin{bmatrix} S_1(\mathbf{q}_1 \cdot \mathbf{k}_1) & S_1(\mathbf{q}_1 \cdot \mathbf{k}_2) & \cdots & S_1(\mathbf{q}_1 \cdot \mathbf{k}_T) \\ S_2(\mathbf{q}_2 \cdot \mathbf{k}_1) & S_2(\mathbf{q}_2 \cdot \mathbf{k}_2) & \cdots & S_2(\mathbf{q}_2 \cdot \mathbf{k}_T) \\ \vdots & \vdots & \ddots & \vdots \\ S_T(\mathbf{q}_T \cdot \mathbf{k}_1) & S_T(\mathbf{q}_T \cdot \mathbf{k}_2) & \cdots & S_T(\mathbf{q}_T \cdot \mathbf{k}_T) \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_T \end{bmatrix}$$
$$= \begin{bmatrix} S_1(\mathbf{q}_1 \cdot \mathbf{k}_1)\mathbf{v}_1 + S_1(\mathbf{q}_1 \cdot \mathbf{k}_2)\mathbf{v}_2 + \cdots + S_1(\mathbf{q}_1 \cdot \mathbf{k}_T)\mathbf{v}_T \\ S_2(\mathbf{q}_2 \cdot \mathbf{k}_1)\mathbf{v}_1 + S_2(\mathbf{q}_2 \cdot \mathbf{k}_2)\mathbf{v}_2 + \cdots + S_2(\mathbf{q}_2 \cdot \mathbf{k}_T)\mathbf{v}_T \\ \vdots \\ S_T(\mathbf{q}_T \cdot \mathbf{k}_1)\mathbf{v}_1 + S_T(\mathbf{q}_T \cdot \mathbf{k}_2)\mathbf{v}_2 + \cdots + S_T(\mathbf{q}_T \cdot \mathbf{k}_T)\mathbf{v}_T \end{bmatrix}$$

Due to the introduction of $\mathbf{V}$*, the* $\mathbf{Q}, \mathbf{K}$ *are no longer interchangeable. And we can see that when calculating each new row (model is generating token and append the newly generated token to the input, like the growing output in Figure 21), we still need the old values in* $\mathbf{V}, \mathbf{K}$*, while the new* $\mathbf{q}$ *will be used immediately. That is why we need cache* $\mathbf{V}, \mathbf{K}$*, but not* $\mathbf{Q}$ *cache.*

We then apply mask to attention:

$$S(\mathbf{Q}\mathbf{K}^\top)\mathbf{V} = \begin{bmatrix} S_1(\mathbf{q}_1 \cdot \mathbf{k}_1) & 0 & \cdots & 0 \\ S_2(\mathbf{q}_2 \cdot \mathbf{k}_1) & S_2(\mathbf{q}_2 \cdot \mathbf{k}_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ S_T(\mathbf{q}_T \cdot \mathbf{k}_1) & S_T(\mathbf{q}_T \cdot \mathbf{k}_2) & \cdots & S_T(\mathbf{q}_T \cdot \mathbf{k}_T) \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_T \end{bmatrix}$$
$$= \begin{bmatrix} S_1(\mathbf{q}_1 \cdot \mathbf{k}_1)\mathbf{v}_1 \\ S_2(\mathbf{q}_2 \cdot \mathbf{k}_1)\mathbf{v}_1 + S_2(\mathbf{q}_2 \cdot \mathbf{k}_2)\mathbf{v}_2 \\ \vdots \\ S_T(\mathbf{q}_T \cdot \mathbf{k}_1)\mathbf{v}_1 + S_T(\mathbf{q}_T \cdot \mathbf{k}_2)\mathbf{v}_2 + \cdots + S_T(\mathbf{q}_T \cdot \mathbf{k}_T)\mathbf{v}_T \end{bmatrix}$$

The new row of $S(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$ *would only use the current* $\mathbf{q}$*, but all the precedent* $\mathbf{k}, \mathbf{v}$*.*

4.1 Parallelism

Definition 4.2. **Tensor parallelism** splits the computations for a single operation (like a matrix multiplication) across multiple devices. The idea is to divide the tensors themselves so that each device handles part of the computation.

Remark 4.2.
- *Useful when the computation within a layer is too large to fit into a single GPU's memory.*
- *Requires frequent communication between GPUs during every forward and backward pass to share intermediate results.*

Definition 4.3. **Pipeline parallelism** splits the layers of a model across multiple devices, and the computations are executed sequentially, like an assembly line.

Example 13. If a model has 6 layers and there are 3 GPUs, each GPU can process 2 layers. During training, one GPU processes the first set of layers, then passes the outputs (activations) to the next GPU for the subsequent layers.

Remark 4.3.
- *Effective when the entire model doesn't fit on a single GPU, but individual layers do.*
- *Introduces pipeline "bubbles" (idle time) because GPUs must wait for their turn in the pipeline. Requires careful batch scheduling to minimize these idle times.*

5 LLM prompting

Definition 5.1. **In-context learning (ICL)** refers to a mechanism used in large language models where the model learns to perform a task based on the context provided within the input itself, without requiring explicit training on that specific task.

Example 14. Below is an example input with in-context learning:

```
Classify the sentiment of the following sentences:
"I love this product!"
Sentiment: Positive

"This is the worst experience I've ever had."
Sentiment:
```

This is the answer from the LLM:

```
Negative
```

Definition 5.2. **Chain of thoughts (CoT)** prompting is a method guide the model to explicitly articulate intermediate reasoning steps when answering a question, rather than providing a direct answer.

Remark 5.1. *CoT prompting makes the reasoning process visible, allowing humans (or other systems) to follow along and catch mistakes in intermediate steps.*

Example 15.

- Direct Prompt: Question:

  ```
  What is 23 times 47?
  ```

  Model Answer:

  ```
  1081
  ```

- Chain of Thought Prompt: Question:

  ```
  What is 23 times 47? Let's think step by step.
  First, we multiply 23 by 40, which gives us 920.
  Next, we multiply 23 by 7, which is 161.
  Finally, we add 920 and 161 together to get 1081.


  1081
  ```

Definition 5.3. **Context size (context window)** in large language models refers to the maximum number of tokens that the model can process once as input and consider when generating a response.

Example 16.

Model	Context Size
GPT 3.5	4,096
GPT 4	8,192
GPT 4-32k	32,768
Llama 1	2,048
Llama 2	4,096
Gemini	326,914

Definition 5.4. **Corpus size** is the total number of words or tokens in a collection of texts (corpus) used for training language models. Note that it is not vocabulary size.

Definition 5.5. **Retrieval-augmented generation (RAG)** is the process of optimizing the output of a large language model, so it references an authoritative knowledge base outside of its training data sources before generating a response.

1. **Query processing**: When a user submits a query, the system first processes it to understand the information need.
2. **Information retrieval**: The retriever component searches the knowledge base for relevant information related to the query. This often uses semantic search techniques to find contextually similar content.
3. **Context augmentation**: The retrieved information is then used to augment the original query, providing additional context for the LLM.
4. **Generation**: The LLM, acting as the generator, uses the augmented query (original query + retrieved context) to produce a response. This allows the model to incorporate both its pre-trained knowledge and the freshly retrieved information.
5. **Response Delivery**: The final generated response is presented to the user.

Example 17.

1. **Dense retriever**: RAG typically uses a dense retriever like DPR (Dense Passage Retrieval). This involves:
   (a) Encoding the input query and corpus documents into dense vectors using a bi-encoder (query encoder and passage encoder).
   (b) Calculating similarity scores (often cosine similarity) between the query vector and document vectors.
   (c) Retrieving the top k documents based on these similarity scores.
2. **Document encoder**: The retrieved documents are then encoded using a document encoder. This can be the same encoder used in the retrieval step or a different one. In a typical RAG implementation, both the query and the retrieved documents are passed through a transformer-based encoder (e.g., BERT).
3. **Generator**: The encoded representations of the retrieved documents are fed into a sequence-to-sequence generative model like BART or T5. The generator processes the concatenated representations of the query and each retrieved document to generate the output text. The generator can be designed to either generate the text based on all documents collectively (by concatenating them) or treat each document separately and then fuse the results.

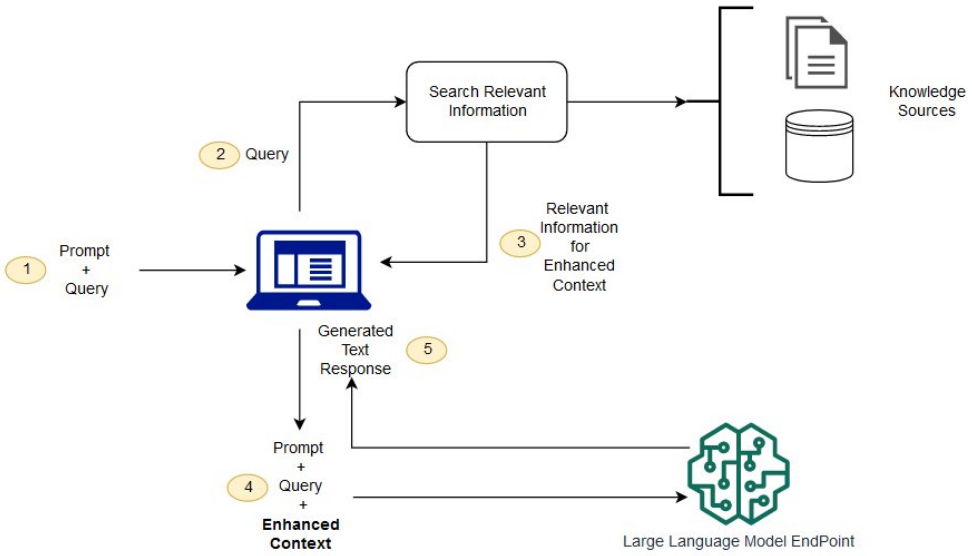

Figure 22: Overview of RAG.

Example 18. Constructing vector databases:

1. **Collect and load data**: For example, we will use President Biden's State of the Union Address from 2022 as additional context.
2. **Chunk documents**: Because the document is too long to fit into the LLM's context window, we need to chunk it into smaller pieces.
3. **Embed and store the chunks**: To enable semantic search across the text chunks, we need to generate the vector embeddings for each chunk and then store them together with their embeddings. To generate the vector embeddings, we can use the OpenAI embedding model, and to store them, we can use the Weaviate vector database.

Definition 5.6. **Sparse retrieval methods** are based on traditional information retrieval techniques, such as the Bag-of-Words (BoW) model. They rely on keyword matching and count-based statistics.

Example 19. BM25 (Best Matching 25) is one of the most widely used sparse retrieval algorithms. It scores documents based on the frequency of query terms within them, with adjustments for term frequency, document length, and other factors.

Remark 5.2. *Pros and cons of sparse retrieval methods:*

1. *Term-Based Representation: Documents and queries are represented as high-dimensional sparse vectors, where each dimension corresponds to a specific word or token from the vocabulary. The vector entries are typically binary (indicating the presence or absence of a term) or term frequency-inverse document frequency (TF-IDF) scores.*
2. *Exact Matching: Retrieval is largely based on exact term matches between the query and documents. If a query term is not present in a document, that document will not be retrieved, even if it might be relevant. But this property also makes it easier to understand why a document was retrieved.*

3. *Vocabulary Mismatch: Sparse retrievers are sensitive to variations in wording. If a query uses different words than those in the relevant documents (e.g., synonyms), those documents may not be retrieved.*

4. *High Dimensionality: Sparse vectors are usually high-dimensional, with most of the dimensions being zero, which can be inefficient in certain contexts.*

Definition 5.7. **Dense retrieval methods** rely on dense vector representations of queries and documents, typically generated by neural networks. These vectors are designed to capture semantic meanings beyond simple keyword matching.

Example 20. Dense Passage Retrieval (DPR) uses a bi-encoder architecture where both queries and documents are encoded into dense vectors using pre-trained transformer models like BERT. Retrieval is performed by finding the documents with the closest vector representations to the query vector.

Remark 5.3. *Pros and cons of dense retrieval methods:*

1. *Semantic Representation: Queries and documents are represented as lower-dimensional dense vectors, where each dimension encodes abstract semantic information learned by the neural network.*

2. *Similarity Matching: Retrieval is based on the similarity (often cosine similarity or dot product) between the query vector and document vectors. This allows the retriever to capture semantic similarities even if the exact terms don't match.*

3. *Generalization: Dense retrievers can generalize better across different wordings, synonyms, and paraphrases, making them more robust to vocabulary mismatches.*

4. *Neural Networks: Dense retrievers typically use pre-trained transformer models like BERT, RoBERTa, or specialized retriever models for generating the dense vectors.*

5. *Explainability: Dense retrievers are often less interpretable than sparse retrievers, as it's harder to understand why certain documents were retrieved based on their dense vector representations.*

Remark 5.4. *The "sparse" or "dense" in sparse or dense retrieval refers to whether the embedding vector of the document is represented by high dimensional sparse vector or low dimensional dense vector.*

6 LLM evaluation

6.1 Benchmarking datasets

Definition 6.1. **Massive Multitask Language Understanding (MMLU)** [Hendrycks et al., 2020] is a benchmark dataset containing multiple-choice questions across 57 diverse subjects, created to evaluate the multitask knowledge and reasoning capabilities of language models.

Example 21. The MMLU benchmark dataset consists of 57 `csv` files and is structured like below:

```
abstract_algebra_val.csv
anatomy_val.csv
astronomy_val.csv
business_ethics_val.csv
clinical_knowledge_val.csv
...
```

```
security_studies_val.csv
sociology_val.csv
us_foreign_policy_val.csv
virology_val.csv
world_religions_val.csv
```

Each `csv` file is structured as follows:

Question	A	B	C	D	Correct Answer
Which is not a nonstate actor that poses a threat to the United States?	Terrorists	Organized crime	Drug traffickers	China	D
Who was the first American president to visit communist China?	Nixon	H. W. Bush	Carter	Reagan	A
...	...	...	...	...	...

Definition 6.2. **MATH** [Hendrycks et al., 2020] benchmark dataset is a dataset designed to evaluate the problem-solving and mathematical reasoning capabilities of LLMs — requiring exact matches to the reference solution (verbatim, strict evaluation).

Example 22. The MATH benchmark dataset consists of 7 folders:

```
algebra
counting_and_probability
geometry
intermediate_algebra
number_theory
prealgebra
precalculus
```

Within each folder, there are thousands of `json` files, where each json file reads like this

```
{
    "problem": "How many vertical asymptotes does the graph of $y=\\frac{2}{x^2+x-6}$ have?",
    "level": "Level 3",
    "type": "Algebra",
    "solution": "The denominator of the rational function factors into $x^2+x-6=(x-2)(x+3)$..."
}
```

Definition 6.3. **Multi-turn Benchmark (MT-bench)** [Zheng et al., 2023] is a benchmark specifically designed to evaluate the multi-turn conversational capabilities of LLMs. It consists of 80 carefully crafted questions spanning from writing, roleplay, reasoning, math, coding, extraction, stem, and humanities.

Example 23. The MT-bench is a stand alone `jsonl` file consists of 80 dictionary objects like below:

```
{"question_id": 81,
"category": "writing",
"turns": [  "Compose an engaging travel blog post about a recent trip to Hawaii,
            highlighting cultural experiences and must-see attractions.",
            "Rewrite your previous response. Start every sentence with the letter A."]}
```

6.2 Metrics

Definition 6.4. **LLM-as-a-Judge** [Zheng et al., 2023] refers to a framework using strong LLM to evaluate and compare outputs from other candidate LLMs or systems on various tasks.

Definition 6.5. The **win-tie-lose rate** is a metric used in pairwise comparisons of LLMs to evaluate their performance on shared tasks, based on human preference. For N total tasks, the win-tie-rate for model A is calculate via

$$\text{win-tie-rate} = \frac{\text{\# of wins} + 0.5 \times \text{\# of ties}}{N}$$

Definition 6.6. **Term frequency-inverse document frequency (TF-IDF)** is a statistical measure used to evaluate the importance of a word within a document relative to a collection of documents, known as a corpus.

$$TF - IDF(t, d) = TF(t, d) \times IDF(t)$$

where t, d stand for term and document, and

$$TF(t, d) = \frac{\text{\# of times term } t \text{ appears in document } d}{\text{total \# of terms in document } d}$$
$$IDF(t) = \log\left(\frac{\text{total \# of document}}{\text{\# of documents containing the term } t}\right)$$

Example 24. If a term appears 25 times in a document of 10,000 words, the TF would be $\frac{25}{10,000} = 0.0025$. If there are 10,000 documents in total and a term appears in 500 of them, the IDF would be: $\log\left(\frac{10,000}{500}\right)$.

Remark 6.1. *Higher IDF indicates the term appears only in a few documents and higher TF indicates the term appears a lot in the certain one documents. A higher TF-IDF score indicates that the term is more relevant to the document, while a lower score suggests it is less significant.*

Definition 6.7. **Reciprocal rank** for a single query is the reciproal of the rank position of the first relevant result.

Example 25. If the first relevant result returned in a list appears at position k, then the reciprocal rank is $\frac{1}{k}$.

Definition 6.8. **Mean reciprocal rank (MRR)** is the average of the reciprocal ranks over a set of queries.

Remark 6.2. *MRR is widely used in following tasks:*

- ***Recommendation systems**: in e-commerce and content platforms, MRR helps evaluate how well recommendation algorithms present relevant items to users.*
- ***Question-answering systems**: MRR is used to assess the performance of systems that provide answers to user queries, measuring how quickly they return a correct response.*

Definition 6.9. **Average precision** is a metric that quantifies the quality of a retrieval system's ranked results for a single query. It can be calculated as

$$\text{AP} = \frac{\sum_{k=1}^{n} P(k) \cdot \text{rel}(k)}{\text{Total \# of relevant documents}}$$

where $P(k)$ is the precision at cut off k, rel is a function tells you if document k is relevant (1) or not (0).

Remark 6.3. *Average precision is basically the area under Precision-Recall curve.*

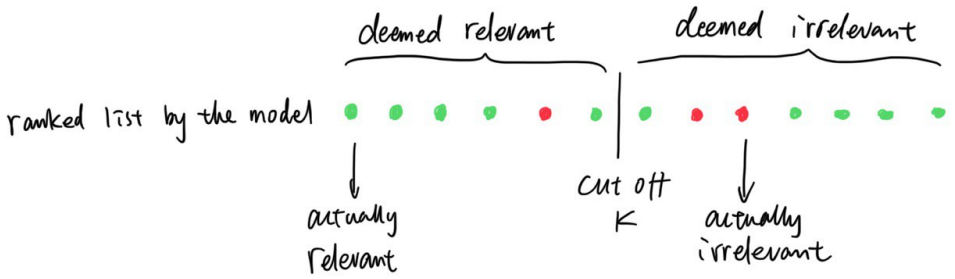

Figure 23: Average precision overview.

Definition 6.10. **Mean average precision (MAP)** is the average precision averaged across a set of query

$$\text{MAP} = \frac{\sum_q^{|Q|} \text{AP}(q)}{|Q|}$$

where Q is the query set, and q is a query.

Definition 6.11. **Perplexity** in language models is calculated as

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log_2 P(w_i|w_1,...,w_{i-1})\right) \in [1,+\infty)$$

where N is the total number of words in the sequence, $P(w_i|w_1,...,w_{i-1})$ is the probability assigned by the model to word w_i given the previous words.

Remark 6.4. *A lower perplexity indicates better prediction (less surprise). Higher perplexity suggests poorer performance or more uncertainty in predictions. A perplexity of 1 (minimum value) would indicate perfect prediction, namely* $P(w_i|w_1,...,w_{i-1}) = 1$.

Definition 6.12. **BLEU (Bilingual Evaluation Understudy) score** is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another.

Example 26. Steps to Calculate the BLEU Score:

1. **Tokenization**: Split the candidate and reference texts into words or tokens.
2. **Count N-grams**: Extract and count all n-grams (sequences of 1, 2, 3, 4, etc., words) from both the candidate and reference texts. Start with unigrams (1-grams), then bigrams (2-grams), trigrams (3-grams), and so on.
3. **Match N-grams**: For each n-gram size, find the number of matching n-grams between the candidate and the reference texts. Limit the count of matching n-grams to the maximum number found in any single reference for that n-gram.
4. **Precision calculation**: Calculate the precision for each n-gram size by dividing the number of matching n-grams by the total number of n-grams in the candidate text. For example,

$$\text{Precision} = \frac{\text{Number of matched n-grams}}{\text{Total number of n-grams in candidate text}}.$$

5. **Compute the geometric mean of precision scores**: Take the geometric mean of the precision scores for all n-gram sizes considered. For instance, if you consider 1-grams to 4-grams, the geometric mean is calculated as:

$$\text{Geometric Mean} = \left(\prod_{i=1}^{4} \text{Precision}_i \right)^{\frac{1}{4}}.$$

6. **Apply a brevity penalty (BP)**: Calculate the brevity penalty to penalize short candidate translations. The brevity penalty is calculated as:

$$\text{BP} = \begin{cases} 1 & \text{if length of candidate} > \text{length of reference}, \\ e^{\left(1 - \frac{\text{length of reference}}{\text{length of candidate}}\right)} & \text{otherwise.} \end{cases}$$

7. **Calculate the BLEU Score**: Multiply the geometric mean of the precision scores by the brevity penalty:

$$\text{BLEU} = \text{BP} \times \text{Geometric Mean}.$$

8. **Interpret the BLEU score**: The BLEU score ranges from 0 to 1, where 1 indicates a perfect match between the candidate and reference texts.

Definition 6.13. **Recall-oriented understudy for gisting evaluation (ROUGE)** score is a set of metrics used for evaluating the quality of summaries or machine-generated text by comparing them to reference summaries or texts.

Example 27.

1. **Tokenization**: First, tokenize both the candidate summary and the reference summary into words or tokens. This involves splitting the text into individual words or other meaningful units.
2. **Generate N-grams**: Next, generate all n-grams from the tokenized candidate and reference summaries. An n-gram is a contiguous sequence of *n* items from the given text. For example, in the case of ROUGE-1, the n-gram is a unigram (1-gram); for ROUGE-2, it is a bigram (2-gram), and so on.
3. **Count N-grams in reference summary**: Count the frequency of each n-gram in the reference summary. This will be used as the denominator when calculating recall.
4. **Match N-grams**: For each n-gram in the candidate summary, check if it appears in the reference summary. Count how many of these n-grams from the candidate summary match those in the reference summary. This will be used as the numerator.
5. **Calculate ROUGE-N recall**: Calculate the recall for ROUGE-N by dividing the number of matched n-grams by the total number of n-grams in the reference summary:

$$\text{ROUGE-N} = \frac{\sum_{\text{ngram} \in \text{reference}} \text{Count}_{\text{match}}(\text{ngram})}{\sum_{\text{ngram} \in \text{reference}} \text{Count}(\text{ngram})}.$$

   Here, $\text{Count}_{\text{match}}(\text{ngram})$ is the number of times an n-gram from the candidate summary matches an n-gram in the reference summary, and $\text{Count}(\text{ngram})$ is the total count of that n-gram in the reference summary.
6. **Interpret the ROUGE-N score**: A higher ROUGE-N score indicates that the candidate summary has a higher overlap with the reference summary in terms of n-grams, which implies better content recall. The score ranges from 0 to 1, where 1 represents a perfect match.

Remark 6.5. *Unlike the BLEU score, which focuses on precision, ROUGE emphasizes recall, making it well-suited for tasks like summarization where it's important to capture as much relevant content as possible.*

Remark 6.6. ***How do you evaluate the effectiveness of a prompt?***

- ***Output quality:*** *Assessing the relevance, coherence, and accuracy of the model's responses.*
- ***Consistency:*** *Checking if the model consistently produces high-quality outputs across different inputs.*
- ***Task-specific metrics:*** *Using task-specific evaluation metrics, such as BLEU for translation or ROUGE for summarization, to measure performance.*
- ***Human evaluation:*** *Involving human reviewers to provide qualitative feedback on the model's outputs.*
- ***A/B testing:*** *Comparing different prompts to determine which one yields better performance.*

Example 28.

- **SafetyBench**: This framework includes over 11,000 diverse samples across various safety categories, allowing for a comprehensive assessment through multiple-choice questions.
- **Holistic Evaluation of Language Models (HELM)**: comprehensive benchmarking framework that evaluates large language models across multiple scenarios and metrics to provide a holistic assessment of their capabilities, limitations, and risks, aiming to improve transparency in AI language technologies.
- **AlpacaEval**: an automated benchmarking tool that evaluates instruction-following capabilities of large language models by comparing their responses to a set of 805 diverse prompts against a baseline model (currently GPT-4 Turbo), using another instance of GPT-4 Turbo as an auto-evaluator to determine win rates and rank models on a public leaderboard.

7 LLM safety

Definition 7.1. **Safety** in LLMs generally refers to minimizing harmful behavior, such as

- generating offensive (harmful) content;
- propagating misinformation;
- making biased or unfair decisions.

Remark 7.1. *How to evaluate the safety of the LLM?*

- ***Toxicity detection****: Use toxicity detection models or tools like the Perspective API to evaluate whether the LLM generates content that could be considered toxic, offensive, or harmful.*
- ***Bias detection****: Analyze model outputs for indications of bias against specific demographic groups, as discussed in fairness evaluations.*
- ***Fact-checking****: Implement fact-checking tools or frameworks to assess the accuracy of information generated by the model. This includes comparing the model's outputs to verified sources of truth.*
- ***Consistency checks****: Test the model for consistency in its outputs across similar queries. Inconsistent answers might indicate issues with reliability and potential for misinformation.*

Remark 7.2. *How to assure the safety of the LLM?*

- ***Data curation:*** *Carefully curate training data to remove or minimize harmful content, misinformation, and biased data. Ensuring that the training data is diverse and representative can help reduce unsafe behavior.*

- ***Fine-tuning for safety:*** *Fine-tune the model on specific datasets designed to encourage safe behavior. This might involve datasets that include examples of correct behavior or penalize harmful content.*
- ***Output filtering:*** *Implement real-time filtering mechanisms that analyze the model's outputs before they are delivered to the user. This can include filtering for offensive language, misinformation, or biased content.*

8 LLM fairness

Definition 8.1. **Bias** in LLMs generally refers to

- Social biases: gender stereotypes, racial and ethnic bias, culture bias, age bias, religious bias, sexuality bias.
- Representation biases: over and under-representation (English vs. Arabic)
- Algorithmic biases: sampling bias (skewed distribution of the training data), temporal bias (reflecting time period of the training data).

Definition 8.2. **Demographic parity** checks whether the model's decisions are independent of a sensitive attribute (e.g., gender, race). For instance, the probability of a positive outcome (e.g., being recommended for a job) should be the same across different demographic groups. Ideally, we should have

$$\Pr(\hat{y} = 1|A = a) = \Pr(\hat{y} = 1|A = b)$$

where A is a sensitive attribute, $\hat{y}$ is the prediction.

Definition 8.3. **Equalized odds** evaluates whether the model's accuracy, both in terms of true positive rate (TPR) and false positive rate (FPR), is the same across different groups. Ideally, we should have

$$\Pr(\hat{y} = 1|y = 1, A = a) = \Pr(\hat{y} = 1|y = 1, A = b)$$
$$\Pr(\hat{y} = 1|y = 2, A = a) = \Pr(\hat{y} = 1|y = 0, A = b)$$

where A is a sensitive attribute, $\hat{y}$ is the prediction and y is the ground truth.

Definition 8.4. **Equal opportunity** is a relaxation of Equalized Odds, this metric focuses only on the true positive rate (TPR), ensuring that individuals in different demographic groups who should receive a positive outcome have equal chances of doing so. Ideally, we should have

$$\Pr(\hat{y} = 1|y = 1, A = a) = \Pr(\hat{y} = 1|y = 1, A = b)$$

where A is a sensitive attribute, $\hat{y}$ is the prediction and y is the ground truth.

Remark 8.1. *How to assure the fairness of the LLM?*

- ***Data curation:*** *Carefully curate training data that is diverse and representative can help reduce unsafe behavior.*
- ***Fine-tuning for safety:*** *Fine-tune the model on specific datasets designed to encourage unbiased behavior. This might involve fine-tuning models on more balanced datasets, using adversarial debiasing techniques, or employing fairness-aware algorithms during training.*
- ***Output filtering:*** *Implement real-time filtering mechanisms that analyze the model's outputs before they are delivered to the user. This can include filtering biased content.*

9 NLP generals

9.1 Data augmentation

Definition 9.1. **Data augmentation** in NLP is aiming to artificially increase the size and diversity of the training dataset, which can help improve model robustness and performance.

Example 29.

- Synonym Replacement

```
Original: "The cat sat on the mat."
Augmented: "The feline sat on the mat."
```

- Back-Translation

```
Original (English): "The quick brown fox jumps over the lazy dog."
Translated to French: "Le renard brun rapide saute par-dessus le chien paresseux."
Back to English: "The fast brown fox leaps over the lazy dog."
```

- Random Insertion

```
Original: "The cat sat on the mat."
Augmented: "The fluffy cat sat quietly on the mat."
```

- Random Deletion

```
Original: "The cat sat on the mat."
Augmented: "The cat on the mat."
```

- Random Swap

```
Original: "The cat sat on the mat."
Augmented: "The sat cat on the mat."
```

- Text Shuffling

```
Original: "The weather is nice today. Let's go for a walk."
Augmented: "Let's go for a walk. The weather is nice today."
```

- Sentence Paraphrasing

```
Original: "I am going to the store."
Augmented: "I'm heading to the shop."
```

- Word Embedding Augmentation

```
Original: "The cat sat on the mat."
Augmented: "The kitten rested on the rug."
```

- Noise Injection

```
Original: "The cat sat on the mat."
Augmented: "Teh cat satted on the mat."
```

- Contextual Word Replacement

```
Original: "The cat sat on the mat."
Augmented: "The dog sat on the carpet."
```

- Entity Replacement

```
Original: "John went to New York."
Augmented: "Alice went to Paris."
```

- Sentence Splitting and Merging

```
Original: "John went to the store. He bought some milk."
Augmented: "John went to the store and bought some milk."
```

- Adversarial Augmentation

```
Original: "The stock market is rising."
Augmented: "The stock market is increasing."
```

9.2 Tokenization

Definition 9.2. **Byte-pair encoding (BPE)** is a subword tokenization algorithm commonly used in natural language processing, especially for large language models. Each time, the pair with highest frequency would be merged. This algorithm produces one of the most commonly used tokenizers.

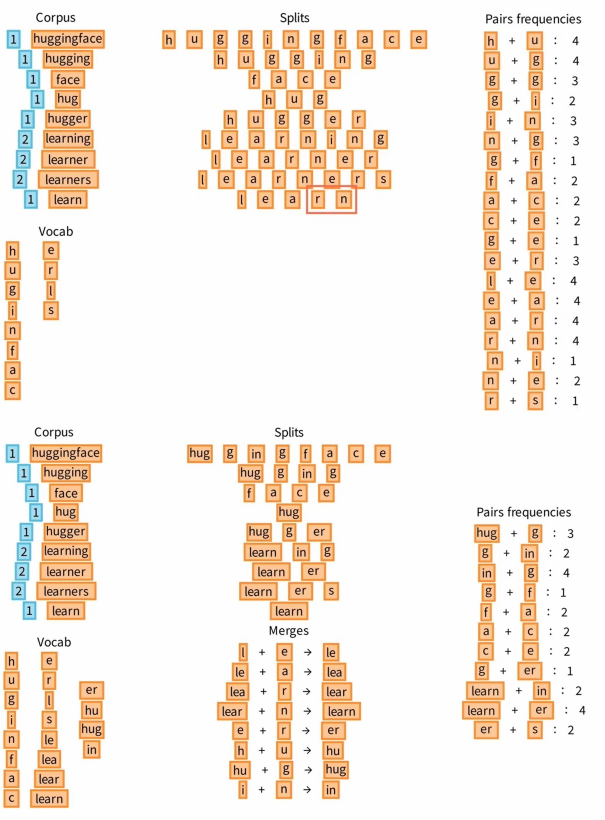

Figure 24: Byte-Pair Encoding (BPE), courtesy of HuggingFace

Example 30. As a very simple example, let's say our corpus uses these five words:

"hug", "pug", "pun", "bun", "hugs"

The base vocabulary will then be [b, g, h, n, p, s, u]. If an example you are tokenizing uses a character that is not in the training corpus, that character will be converted to the unknown token.

After getting this base vocabulary, we add new tokens until the desired vocabulary size is reached by learning merges, which are rules to merge two elements of the existing vocabulary together into a new one. So, at the beginning these merges will create tokens with two characters, and then, as training progresses, longer subwords.

At any step during the tokenizer training, the BPE algorithm will search for the most frequent pair of existing tokens (by "pair" here we mean two consecutive tokens in a word). That most frequent pair is the one that will be merged, and we rinse and repeat for the next step. This process continues until a predefined vocabulary size is reached. The final vocabulary consists of characters, common substrings, and some whole words.

Remark 9.1. *The GPT-2 and RoBERTa tokenizers (which are pretty similar) have a clever way to deal with this: they don't look at words as being written with Unicode characters, but with bytes. This way the base vocabulary has a small size (256), but every character you can think of will still be included and not end up being converted to the unknown token. This trick is called byte-level BPE.*

Definition 9.3. **WordPiece** is the tokenization algorithm Google developed to pretrain BERT. It is similar to BPE in merging rules, but use a difference way to calculate the a score for each pair, using the following formula:

$$\text{score} = \frac{\text{frequency of the pair}}{\text{frequency of the first element} \times \text{frequency of the second element}}$$

Each time, the pair with highest score would be merged.

Remark 9.2. *Starting from the word to tokenize, WordPiece finds the longest subword that is in the vocabulary, then splits on it. The vocabulary includes whole words, subwords, and characters.*

Remark 9.3. *When the tokenization gets to a stage where it's not possible to find a subword in the vocabulary, the whole word is tokenized as unknown — so, for instance, "mug" would be tokenized as* `[UNK]`*, rather than* `[m, ##u, [UNK]]`*.*

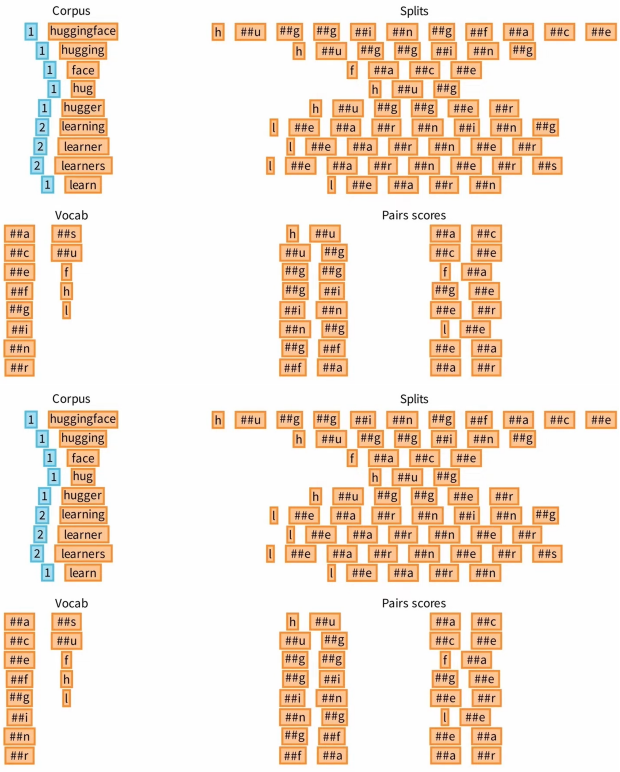

Figure 25: WordPiece, courtesy of HuggingFace

Definition 9.4. **SentencePiece** is an unsupervised text tokenizer which tokenizes raw text directly into subword units without requiring pre-tokenization. It treats the input as a sequence of Unicode characters, including spaces and special characters.

SentencePiece implements subword segmentation algorithms like BPE and unigram language model.

Remark 9.4. *Advantage of SentencePiece:*

- *No Pre-tokenization: Eliminates the need for language-specific pre-tokenization, making it particularly useful for languages without clear word boundaries (e.g., Japanese, Chinese).*
- *Uniform Character Treatment: Treats all characters, including spaces, equally, allowing it to capture a broader range of linguistic patterns.*
- *Reversibility: Provides lossless tokenization, meaning the original text can be perfectly reconstructed from the tokenized representation.*
- *Handling Rare Words: Effectively manages rare or out-of-vocabulary words by breaking them down into subword units*

Remark 9.5. ***Tokenizer usage in the popular models:***

- *BPE is used in models like GPT and RoBERTa.*
- *WordPiece is used in models like BERT.*
- *SentencePiece is used in models like T5, XLM-R, and mBERT.*

9.3 Embedding

Definition 9.5. **Bag-of-words (BoW) model** is an word embedding algorithm that represents text data by treating a document as an unordered collection of words, disregarding grammar and word order, and focusing solely on word frequency.

Example 31. Consider two documents:

1. "John likes to watch movies. Mary likes movies too."
2. "Mary also likes to watch football games."

The vocabulary might be: ["John", "likes", "to", "watch", "movies", "Mary", "too", "also", "football", "games"]. The BoW representation would count the frequency of each word in each document, resulting in vectors that capture these frequencies.

Definition 9.6. **Continous bag-of-words (CBOW) model** is an word embedding algorithm that use a shallow neural network (input layer, a hidden layer, and an output layer) to predict a target word based on its surrounding context words in a text.

Example 32. The working mechanism of CBOW can be mathematically represented as follows:

1. Assume V the vocabulary size, N the hidden layer dimension, n the window size, $w_{i-n/2}, ..., w_{i+n/2}$ be the context words, and w be a one-hot encoded vector;
2. Let W of shape $V \times N$ be the weight matrix connecting the input layer to the projection layer, and W' of shape $N \times V$ be the weight matrix connecting the projection layer to the output layer;
3. Let h be the projection layer, which is the average of the projected input vectors, $h = \frac{1}{n} \times (W \times w_{i-n/2} + \cdots + W \times w_{i+n/2})$;
4. Let y be the output layer, which is the probability distribution over the vocabulary, $y = \mathrm{softmax}(W' \times h)$;
5. We use the cross-entropy loss to update the network and the target word is selected as the word with the highest probability in y.

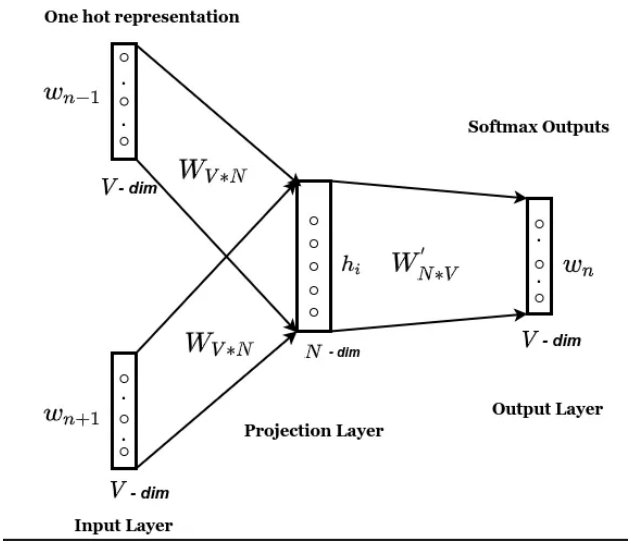

Figure 26: CBOW

Remark 9.6. *In CBOW, the trainable parameters are W, W', and no activation function is used in projection layer.*

Definition 9.7. **Skip-gram** is an word embedding algorithm that aims to predict context words given a target word, which is the inverse of the CBOW.

Example 33. The working mechanism of skip-gram is very similar to the one of CBOW's:

1. Assume V the vocabulary size, N the hidden layer dimension, n the window size, $w_{i-n/2}, ..., w_{i+n/2}$ be the context words, and w be a one-hot encoded vector;
2. Let W of shape $V \times N$ be the weight matrix connecting the input layer to the projection layer, and W' of shape $N \times V$ be the weight matrix connecting the projection layer to the output layer;
3. Let h be the projection layer, which is the average of the projected input vectors, $h = \frac{1}{n} \times (W \times w_i)$;
4. Let y be the output layer, which is the probability distribution over the vocabulary, $y = \mathrm{softmax}(W' \times h)$;
5. We use the cross-entropy loss $L = \frac{1}{n} \sum CE(y, w_c)$ to update the network, where all the context word w_c is still represented by one-hot vector.

Remark 9.7. *The difference between CBOW and skip-gram:*

- ***Workflow:*** *While CBOW predicts the target word from context, Skip-gram predicts context words given a target word;*
- ***Strength:*** *skip-gram tends to perform better on infrequent words because it focuses on learning the representation of a single word at a time, which allows it to capture more nuanced relationships. While CBOW tends to perform better on frequent words because it averages the context words, which can smooth out the noise.*
- ***Training samples:*** *skip-gram generates more training examples because it treats each (target, context) pair as a separate instance. While CBOW generates fewer examples because it treats the entire context as one instance.*
- ***Efficiency:*** *skip-gram is typically slower to train because of the higher number of examples and the need to predict multiple context words for each target word. While CBOW is faster to train due to fewer examples and the use of a single prediction per context.*

- ***Use case:*** *skip-gram is often preferred when the focus is on learning representations for less frequent words or when the context window is large. While CBOW is often preferred for computational efficiency or when the context window is small.*

Definition 9.8. **Word2Vec** is an word embedding algorithm to produce vector representations of words, via CBOW or skip-gram algorithm.

Remark 9.8. *Difference between Bag-of-Words (BoW) model and the Continuous Bag-of-Words (CBOW):*

- *The Bag-of-Words model and the Continuous Bag-of-Words model are both techniques used in natural language processing to represent text in a computer-readable format, but they differ in how they capture context.*
- *The BoW model represents text as a collection of words and their frequency in a given document or corpus. It does not consider the order or context in which the words appear, and therefore, it may not capture the full meaning of the text. The BoW model is simple and easy to implement, but it has limitations in capturing the meaning of language.*
- *In contrast, the CBOW model is a neural network-based approach that captures the context of words. It learns to predict the target word based on the words that appear before and after it in a given context window. By considering the surrounding words, the CBOW model can better capture the meaning of a word in a given context.*

Definition 9.9. **GloVe (Global Vectors for Word Representation)** is an unsupervised learning algorithm for obtaining vector representations for words. The objective function of GloVe can be written as

$$L = \sum_{i,j=1}^{V} f(X_{ij}) \left(\mathbf{w}_i^T \cdot \mathbf{w}_j + b_i + b_j - \log(P_{ij})\right)^2$$

$$f(X_{ij}) = \begin{cases} \left(\frac{X_{ij}}{X_{\text{max}}}\right)^{\alpha} & \text{if } X_{ij} < X_{\text{max}} \\ 1 & \text{otherwise} \end{cases}$$

where

- X_{ij} is the co-occurrence count (be the number of times that the word j appears in the context of the word i over the entire corpus);
- $X_i = \sum_j X_{ij}$;
- $P_{ij} = \frac{X_{ij}}{X_i}$;
- $\mathbf{w}_i$ and $\mathbf{w}_j$ are the word vectors for words i and j;
- b_i and b_j are bias terms;
- $f(X_{ij})$ in the objective helps to handle the fact that not all co-occurrences are equally informative;
- X_{max} and α are hyperparameters.

Remark 9.9. *GloVe aims to learn word vectors such that the exponential of dot product of two word vectors approximates the probability of their co-occurrence:*

$$\exp\left(\mathbf{w}_i^T \cdot \mathbf{w}_j\right) \approx P_{ij}$$

10 Transformers in CV

10.1 Vision Transformer (encoder-only model)

Example 34. Below is the framework of the data efficient vision transformer (DeViT) [Touvron et al., 2021]:

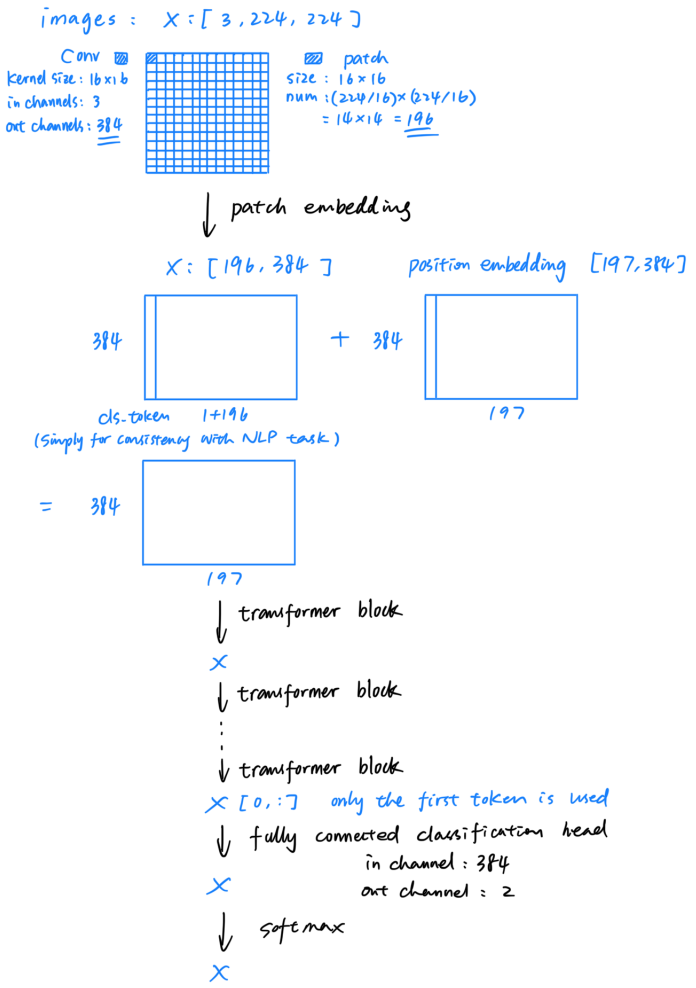

Figure 27: Pipeline of DeViT.

The `cls_token` is not essential in the vision transformer, the only reason for keeping this is to align with the design of the transformers in NLP[3].

[3] `https://github.com/google-research/vision_transformer/issues/61#issuecomment-802233921`

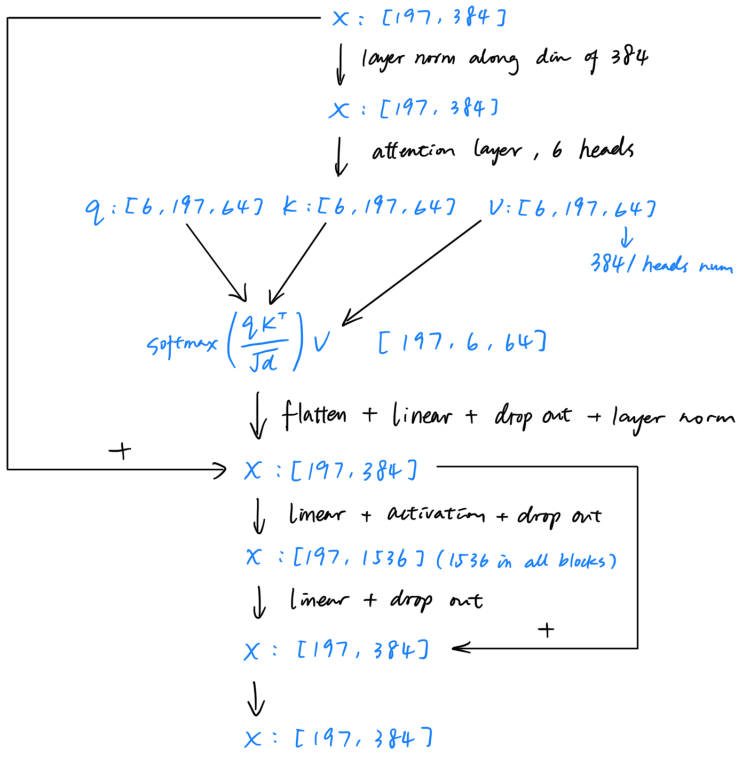

Figure 28: Structure within a transformer block.

DeViT small model consists of 12 transformer blocks like below:

```
Block(
  (norm1): LayerNorm((384,), eps=1e-05, elementwise_affine=True)
  (attn): Attention(
    (qkv): nn.Linear(in_features=384, out_features=1152, bias=True)
    (attn_drop): Dropout(p=0.0, inplace=False)
    (proj): nn.Linear(in_features=384, out_features=384, bias=True)
    (proj_drop): Dropout(p=0.0, inplace=False)
  )
  (drop_path): Identity()
  (norm2): LayerNorm((384,), eps=1e-05, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): nn.Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (fc2): nn.Linear(in_features=1536, out_features=384, bias=True)
    (drop): Dropout(p=0.0, inplace=False)
  )
)
```

Remark 10.1. *Note that the layer norm (`LayerNorm((384,), eps=1e-05, elementwise_affine=True)`) in transformer block only normalize along the token embedding dimension, not including the sequence spatial dimension.*

This is because the sequential dimension represents the relationships between tokens in a sequence, and normalizing it could potentially lose important information.

References

[Chatterjee, 2022] Chatterjee, S. (2022). *Answering Topical Information Needs Using Neural Entity-Oriented Information Retrieval and Extraction*. PhD thesis, University of New Hampshire.

[Hendrycks et al., 2020] Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. (2020). Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France. PMLR.

[Luo et al., 2018] Luo, P., Wang, X., Shao, W., and Peng, Z. (2018). Towards understanding regularization in batch normalization.

[Ouyang et al., 2022] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. (2022). Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

[Rafailov et al., 2024] Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. (2024). Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36.

[Teye et al., 2018] Teye, M., Azizpour, H., and Smith, K. (2018). Bayesian uncertainty estimation for batch normalized deep networks.

[Touvron et al., 2021] Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., and Jégou, H. (2021). Training data-efficient image transformers distillation through attention.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[Wei et al., 2022] Wei, J., Bosma, M., Zhao, V., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. (2022). Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*.

[Zheng et al., 2023] Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. (2023). Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.