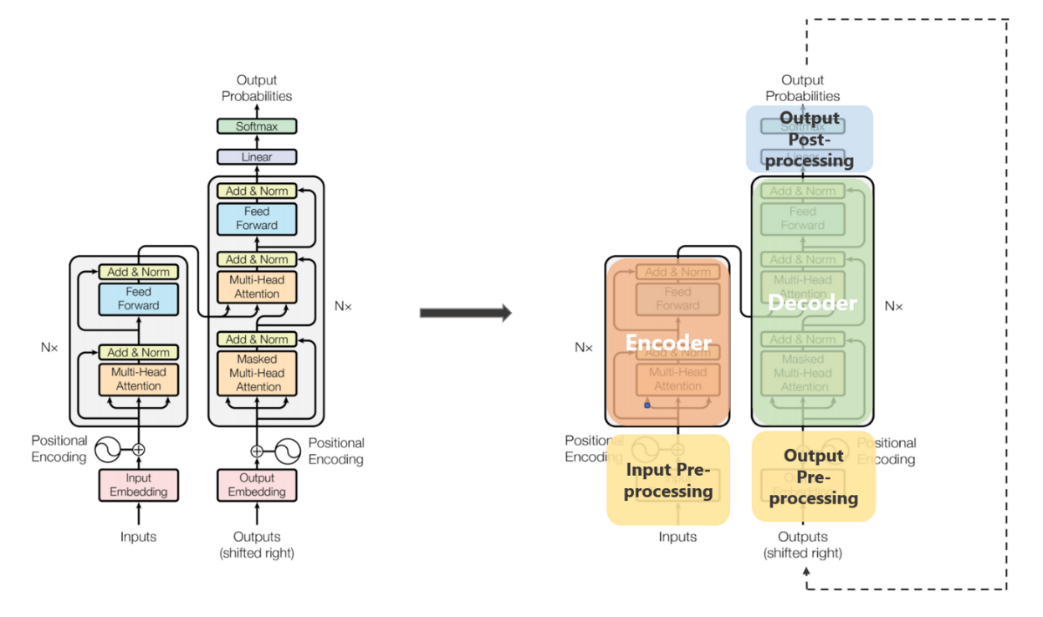
# Transformers

## Transformers

A **transformer** is a [deep learning](https://en.wikipedia.org/wiki/Deep_learning) architecture developed by researchers at [Google](https://en.wikipedia.org/wiki/Google) and based on the multi-head [attention](https://en.wikipedia.org/wiki/Attention_(machine_learning)) mechanism, proposed in a 2017 paper "[Attention Is All You Need](https://en.wikipedia.org/wiki/Attention_Is_All_You_Need)".



The transformer architecture is composed of an encoder and a decoder, each of which is made up of multiple layers of self-attention and feedforward neural networks. The self-attention mechanism is the heart of the transformer, allowing the model to weigh the importance of different words in a sentence based on their affinity with each other. This is similar to how a human might read a sentence, focusing on the most relevant parts of the text rather than reading it linearly from beginning to end.

In addition to self-attention, the transformer also introduces positional bias, which allows the model to keep track of the relative positions of words in a sentence. This is important because the order of words in a sentence can significantly impact its meaning.

## Attention

***Attention allowed us to focus on parts of our input sequence while we predicted our output sequence***

***Self attention****, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.*

In simpler terms, ***self attention helps us create similar connections but within the same sentence.*** Look at the following example:

“I poured water from the *bottle* into the ***cup*** until **it** was ***full***.”  
it => cup“I poured water from the **bottle** into the cup until **it** was ***empty***.”  
it=> bottle

By changing one word “*full*” — > “*empty*” the reference object for “*it*” changed. If we are translating such a sentence, we will want to know the word “*it*” refers to.

**The three kinds of Attention possible in a model:**

1. ***Encoder-Decoder Attention*:**Attention between the input sequence and the output sequence.
2. ***Self attention in the input sequence*:** Attends to all the words in the input sequence.
3. ***Self attention in the output sequence:*** One thing we should be wary of here is that the scope of self attention is limited to the words that occur before a given word. This prevents any information leaks during the training of the model. This is done by masking the words that occur after it for each step. So for step 1, only the first word of the output sequence is NOT masked, for step 2, the first two words are NOT masked and so on.

### Keys, Values, and Queries:

The three random words I just threw at you in this heading are vectors created as abstractions are useful for calculating self attention, more details on each below. These are calculated by multiplying your input vector(*X*) with weight matrices that are learnt while training.

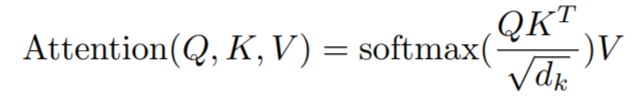
* ***Query Vector***: *q*= *X \* Wq.*Think of this as the current word.
* ***Key Vector****:* *k*= *X \* Wk.*Think of this as an indexing mechanism for Value vector. Similar to how we have key-value pairs in hash maps, where keys are used to uniquely index the values.
* ***Value Vector:*** *v*= *X \* Wv.*Think of this as the information in the input word.

What we want to do is take query *q*and find the most similar key *k*, by doing a dot product for *q*and *k*. The closest query-key product will have the highest value, followed by a softmax that will drive the *q.k*with smaller values close to 0 and *q.k*with larger values towards 1. This softmax distribution is multiplied with *v.*The value vectors multiplied with ~1 will get more attention while the ones ~0 will get less. The sizes of these *q, k*and*v*vectors are referred to as “***hidden size***” by various implementations.

The values represent the index for q, k and i.

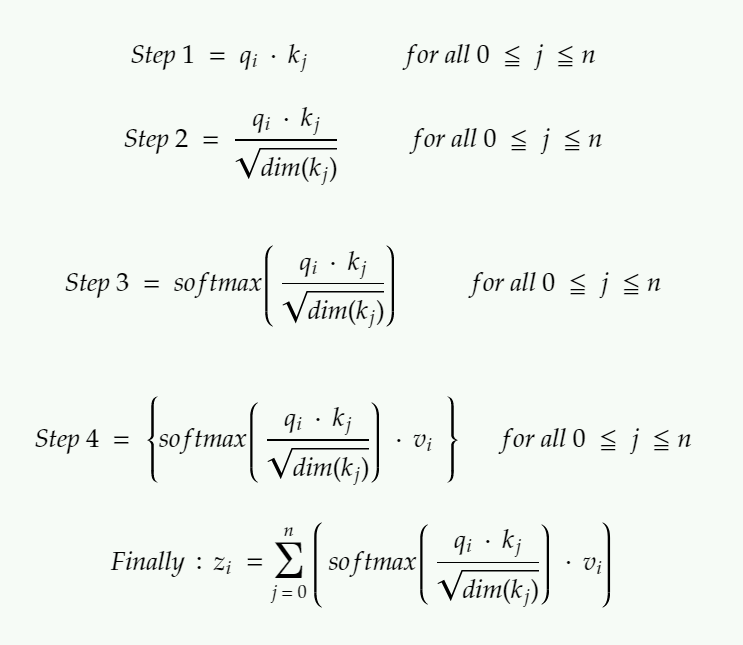
All these matrices *Wq, Wk* and *Wv* are learnt while being jointly trained during the model training.

### Calculating Self attention from q, k and v:



If we are calculating self attention for *#i* input word,

* ***Step 1:*** Multiply qᵢ by the kⱼ key vector of word.
* ***Step 2:*** Then divide this product by the square root of the dimension of key vector.  
  This step is done**for better gradient flow**which is specially important in cases when the value of the dot product in previous step is too big. As using them directly might push the softmax into regions with very little gradient flow.
* ***Step 3:*** Once we have scores for all *j*s, we pass these through a softmax. We get normalized value for each *j*.
* ***Step 4:*** Multiply softmax scores for each *j* with *v*ᵢvector.  
  The idea/purpose here is, very similar attention, to keep preserve only the values *v*of the input word(s) we want to focus on by multiplying them with high probability scores from softmax ~1, and remove the rest by driving them towards 0, i.e. making them very small by multiplying them with the low probability scores ~0 from softmax.



Calculating output of self attention for the ith input word. If you are looking for an analogy between self attention and attention, think of z serving the purpose of context vectors and not global alignment weights.

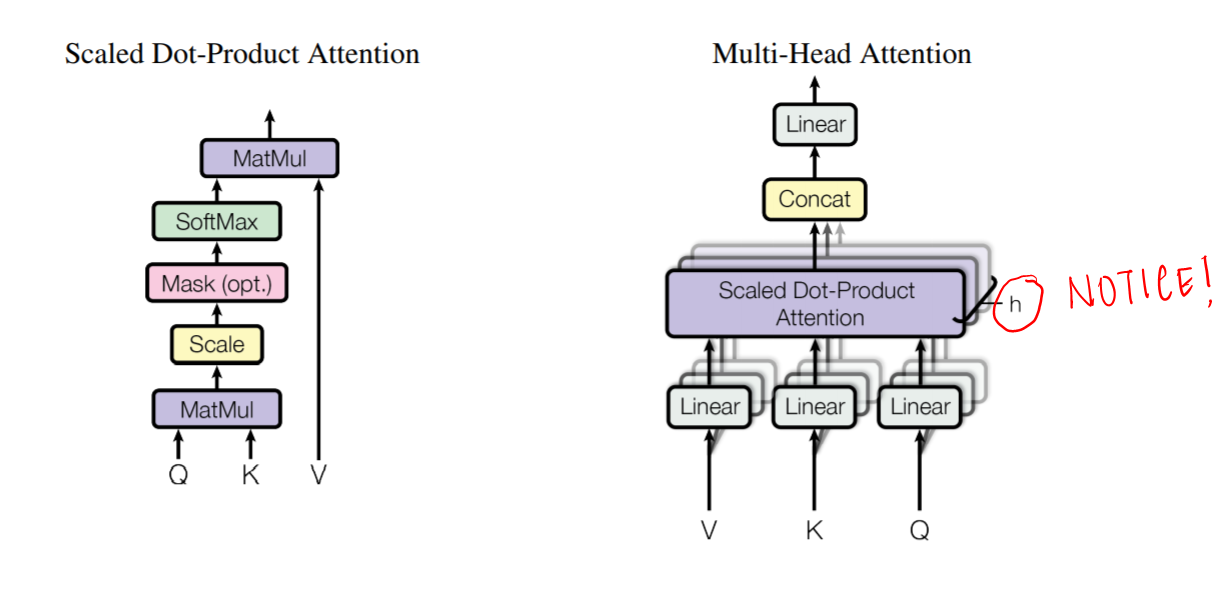
## Multi-Head Attention

The output of each sub-layer needs to be of the same dimension which is 512 in our paper.  
=> zᵢ needs to be of 512 dimensions.  
=> vᵢ needs to be of 512 dimensions as zᵢ are just sort of weighted sums of vᵢs.

Additionally, we want to allow the model to focus on different positions is by **calculating self attention multiple times with different sets** of *q, k*and *v* vectors, then take an average of all those outputs to get our final *z*.

So instead of dealing with these humongous vectors and averaging multiple outputs, we reduce the size of our *k,q* and *v* vectors to some smaller dimension — reduces size of *Wq, Wk*, and *Wv* matrices as well. We keep the multiple sets (*h*) of *k, q* and *v and*refer to each set as an “*attention head”,* hence the name *multi-headed* attention. And lastly, instead of averaging to get final *z*, we concatenate them.

The size of the concatenated vector will be too large to be fed to the next sub-layer, so we scale it down by multiplying it with another learnt matrix *Wo*.



Multiple attention heads allowed the model to jointly attend to information from different representation sub-spaces at different positions which was inhibited by averaging in a single attention head.

## Key-Value Cache

As suggested by its name, Key-Value cache is a technique designed to speedup the autoregressive process by caching and reusing the previous keys and values, rather than re-computing them at each decoding step.

Note that KV cache is typically used only during the inference stage, since in training we still need to process the entire input sequence in parallel.

KV cache is commonly implemented as a rolling buffer. At each decoding step, only the new query Q is computed, while the K and V stored in the cache will be reused, so that the attention will be computed using the new Q and reused K, V. Meanwhile, the new token’s K and V will also be appended to the cache for later use.

However, the speedup achieved by KV cache comes at a cost of memory, since KV cache often scales with **batch size × sequence length × hidden size × number of heads**, leading to a memory bottleneck when we have larger batch size or longer sequences.

That further leads to two techniques aiming at addressing this limitation: Multi-Query Attention and Grouped-Query Attention.

## Multi-Query Attention(MQA)

Multi-Query Attention is a variation of multi-head attention.

**The approach of MQA is to keep the original number of heads for Q, but have only one head for K and V. This means that all the Q heads share the same set of K and V heads, hence the name Multi-Query.**

In MQA, the size of the key and value tensors is b \* k and b \* v, while in MHA, the size of the key and value is b \* h \* k and b \* h \* v, where h represents the number of heads.

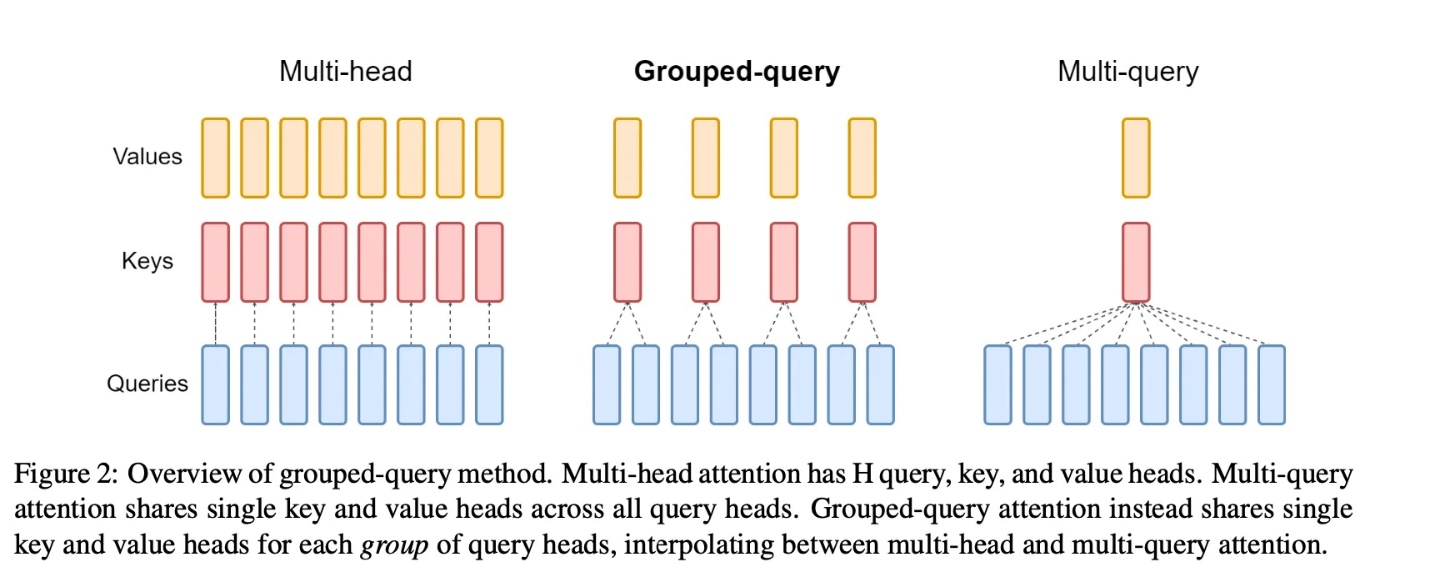
**In general, MQA achieves inference acceleration through the following methods:**

* The KV cache size is reduced by a factor of h(number of heads), which means that the tensors that need to be stored in the GPU memory are also reduced. The space saved can be used to increase the batch size, thereby improving efficiency.
* The amount of data read from memory is reduced, which reduces the waiting time for computational units and improves computational utilization.
* MQA has a relatively small KV cache that can fit into the cache (SRAM). MHA, on the other hand, has a larger KV cache that cannot be entirely stored in the cache and needs to be read from the GPU memory (DRAM), which is time-consuming.

## Grouped Query Attention (GQA)

Grouped Query Attention simplifies how LLMs understand large amounts of text by bundling similar pieces together. This makes the model faster and smarter, as it can focus on groups of words at a time instead of each word individually.

Grouped Query Attention (GQA) is a method that interpolates between multi-query attention (MQA) and multi-head attention (MHA) in Large Language Models (LLMs). It aims to achieve the quality of MHA while maintaining the speed of MQA.



In GQA, query heads are divided into groups, each of which shares a single key head and value head. This approach allows GQA to interpolate between multi-head and multi-query attention, achieving a balance between quality and speed. For instance, GQA with a single group (and therefore a single key and value head) is equivalent to MQA, while GQA with groups equal to the number of heads is equivalent to MHA.

## Multi-head Latent Attention

The basic idea of MLA is to compress the attention input **h\_t** into a low-dimensional latent vector with dimension **d\_c**, where **d\_c** is much lower than the original (**h\_n** · **d\_h**). Later when we need to calculate attention, we can map this latent vector back to the high-dimensional space to recover the keys and values. As a result, only the latent vector needs to be stored, leading to significant memory reduction.

This process can be more formally described with the following equations, where **c^{KV}\_t** is the latent vector, **W^{DKV}** is the compressing matrix that maps **h\_t**‘s dimension from (**h\_n** · **d\_h**) to **d\_c** (here D in the superscript stands for "down-projection", meaning compressing the dimension), while **W^{UK}** and **W^{UV}** are both up-projection matrices that map the shared latent vector back to the high-dimensional space.

Изображение выглядит как текст, Шрифт, белый

Автоматически созданное описание

Similarly, we can also map the queries into a latent, low-dimensional vector and then map it back to the original, high-dimensional space:

Изображение выглядит как Шрифт, текст, белый, линия

Автоматически созданное описание

## Encoder

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

### Positional Encoding

The input sequence is first embedded into a high-dimensional vector space using an embedding layer, which maps each token in the sequence to a vector representation. After that, these vector representations got elementwise added with the positional embedding.

### 2. Multi-Head Attention:

Combined input and positional embedding then fee into the first Encoder block here first combined embedding is passed through the multi-head attention layer.

### 3. Layer Normalization and Residual

Output from the Multi-head attention and initial combined embedding (as a residual) is passed through the layer Normalization layer.

### 4. Feed Forward Neural Network :

Output after layer Normalization is passed through a feed-forward neural network.

### 5. Layer Normalization and Residual

Finally, the output of the Feed Forward Neural Network is added with an output of the first layer normalization, and then the combined vector is passed layer normalization layer.

## Decoder

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

Given *z*, the decoder then generates an output sequence *(y, …, y*ₘ*)* of symbols one element at a time.

### Decoder Input Embeddings & Positional Encoding

The decoder’s starting is very similar to that of the encoder. To obtain positional embeddings, the input passes through layers of positional encoding and embedding. The first multi-head attention layer receives the positional embeddings and uses them to calculate the attention scores for the input from the decoder.

### Decoders First Multi-Headed Attention

The way that this multi-headed attention layer function is a little different. You must stop the decoder from conditioning to upcoming tokens because it generates the sequence word by word and is autoregressive. The word “fine,” for instance, should not be available when calculating attention ratings for the word “am” because it is a future term that was created subsequently. Just the words that come before and after the word “am” should have access to each other. All other words can only pay attention to prior words, and this is true for all other words.

We require a technique to stop attention scores from being calculated for future words. This process is known as masking. You apply a look ahead mask to stop the decoder from seeing tokens in the future. Before figuring out the softmax and after scaling the scores, the mask is added.

### Encoder-Decoder Attention

In encoder-decoder attention, VALUE and KEY are from the output of the encoder, QUERY is from the output of the self-attention layer in the decoder. The model is going to train on the transformation matrix to apply to value, key, and query according. The output of encoder-decoder attention is still the weighted sum of values with the weights as the output from the softmax of the product between key and query.

### Look-Ahead Mask

The mask is a matrix with values of 0 and negative infinities that is the same size as the attention ratings. The upper right triangle of the score matrix is filled with negative infinities when the mask is added to the scaled attention scores.

The mask is used because, when the masked scores are softmaxed, the negative infinities are zeroed out, leaving no attention scores for subsequent tokens. As you can see in the graph below, the attention score for “am” is 0 for the word “fine,” but it has values for all words that come before it. In essence, this instructs the model to ignore those terms.

The first multi-headed attention layer’s calculation of the attention scores only differs in this masking. The mask is still being applied to several heads in this layer before being concatenated and passed through a linear layer for additional processing. A masked output vector with instructions on how the model should pay attention to the decoder’s input is the result of the initial multi-headed attention.

### Point-wise Feed Forward Layer with Second Multi-Headed Decoder

The second attention tier with many heads. **The first multi-headed attention layer’s outputs are the values for this layer, whereas the encoder’s outputs are the queries and keys.** The decoder can choose which encoder input to focus on by using this technique to match the encoder’s input to the decoder’s input. The second multi-headed attention’s output is processed further after passing through a pointwise feedforward layer.

### Final Softmax with a Linear Classifier for Output Probabilities

The output of the last linear layer, which serves as a classifier, is passed through the final pointwise feedforward layer. As many classes as you have determined how large the classifier is. For instance, the output of that classier will be 10,000 words in size if you have 10,000 classes for 10,000 words. The softmax layer receives the classifier’s output and generates probability scores between 0 and 1. Our anticipated word is equivalent to the index with the highest likelihood score.

Once a token is anticipated, the decoder adds the output to its list of inputs and starts the decoding process all over again. In our situation, the final class that is given to the end token is the prediction with the highest probability.

The decoder may alternatively be layered N layers high, with each layer receiving input from the layers above it as well as the encoder. The model can learn to extract and concentrate on various attentional combinations from its attention heads by stacking the layers, potentially improving its prediction power.

## Decoder Stack

Изображение выглядит как текст, снимок экрана, диаграмма

Автоматически созданное описание

The output of the decoder stack at each step is fed back to the decoder in the next time step — pretty similar to how outputs from previous steps in RNNs were used as next hidden states. And just as we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to preserve the position of each word. This positional encoding + word embedding combo is then fed into a masked multi-headed self attention.

This self-attention sub-layer in the decoder stack is modified to prevent positions from attending to subsequent positions — you can’t look at future words. This masking ensures that the predictions for position *i* can depend only on the known outputs at positions less than *i*.

The outputs from the encoder stack are then used as multiple sets of key vectors ***k***and value vectors ***v***, for the “encoder decoder attention” layer. It helps the decoder focus on the contextually relevant parts in the input sequence for that step. (The part similar to global attention vectors.) The ***q***vector comes from the “output self attention” layer.

Once we get the output from the decoder, we do a softmax again to select the final probabilities of words.

## Transformer Encoder-Decoder Architecture

The transformer encoder-decoder architecture is used for tasks like language translation, where the model must take in a sentence in one language and output a sentence in another language. The encoder takes in the input sentence and produces a fixed-size vector representation of it, which is then fed into the decoder to generate the output sentence. The decoder uses both self-attention and cross-attention, where the attention mechanism is applied to the output of the encoder and the input of the decoder.

One of the most popular transformer encoder-decoder models is the [T5](https://ai.googleblog.com/2020/02/exploring-transfer-learning-with-t5.html) (Text-to-Text Transfer Transformer), which was introduced by Google in 2019. The T5 can be fine-tuned for a wide range of NLP tasks, including language translation, question answering, summarization, and more.

Real-world examples of the transformer encoder-decoder architecture include Google Translate, which uses the T5 model to translate text between languages, and Facebook’s M2M-100, a massive multilingual machine translation model that can translate between 100 different languages.

## Transformer Encoder Architecture

The transformer encoder architecture is used for tasks like text classification, where the model must classify a piece of text into one of several predefined categories, such as sentiment analysis, topic classification, or spam detection. The encoder takes in a sequence of tokens and produces a fixed-size vector representation of the entire sequence, which can then be used for classification.

One of the most popular transformer encoder models is [BERT](https://ai.googleblog.com/2018/11/open-sourcing-bert-state-of-art-pre.html) (Bidirectional Encoder Representations from Transformers), which was introduced by Google in 2018. BERT is pre-trained on large amounts of text data and can be fine-tuned for a wide range of NLP tasks.

Unlike the encoder-decoder architecture, the transformer encoder is only concerned with the input sequence and does not generate any output sequence. It applies self-attention mechanism to the input tokens, allowing it to focus on the most relevant parts of the input for the given task.

Real-world examples of the transformer encoder architecture include sentiment analysis, where the model must classify a given review as positive or negative, and email spam detection, where the model must classify a given email as spam or not spam.

## Transformer Decoder Architecture

The transformer decoder architecture is used for tasks like language generation, where the model must generate a sequence of words based on an input prompt or context. The decoder takes in a fixed-size vector representation of the context and uses it to generate a sequence of words one at a time, with each word being conditioned on the previously generated words.

One of the most popular transformer decoder models is the [GPT-3](https://en.wikipedia.org/wiki/GPT-3) (Generative Pre-trained Transformer 3), which was introduced by OpenAI in 2020. The GPT-3 is a massive language model that can generate human-like text in a wide range of styles and genres.

The transformer decoder architecture introduces a technique called triangle masking for attention, which ensures that the attention mechanism only looks at tokens to the left of the current token being generated. This prevents the model from “cheating” by looking at tokens that it hasn’t generated yet.

Real-world examples of the transformer decoder architecture include text generation, where the model must generate a story or article based on a given prompt or topic, and chatbots, where the model must generate responses to user inputs in a natural and engaging way.

In **Encoder-Decoder architectures** (e.g., original Transformer for translation):

* **Encoder provides K, V**: The Decoder’s cross-attention uses K and V from the Encoder’s output.
* **Decoder uses self-attention for Q, K, V**: For its own input (like masked self-attention).

In **Decoder-only models** (e.g., GPT):

* **No Encoder exists**, so all attention is **self-attention** (with masking for autoregression).
* K and V are **never sourced externally**; they’re always projections of the Decoder’s own hidden states.

## Drawbacks of Transformers

The drawbacks of the transformer architecture are:

1. High computational cost due to the attention mechanism, which increases quadratically with sequence length.
2. Difficulty in interpretation and debugging due to the attention mechanism operating over the entire input sequence.
3. Prone to overfitting when fine-tuned on small amounts of task-specific data.

Despite these downsides, the transformer architecture remains a powerful and widely-used tool in NLP, and research is ongoing to mitigate its computational requirements and improve its interpretability and robustness.

## GPT vs BERT: What’s The Difference?

* BERT is a Transformer encoder, which means that, for each position in the input, the output at the same position is the same token (or the [MASK] token for masked tokens), that is the inputs and output positions of each token are the same. Models with only an encoder stack like BERT generate all its outputs at once.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

BERT has two training objectives, and the most important of them is the Masked Language Modeling (MLM) objective. is With the MLM objective, at step the following happens:

* select some tokens  
  (each token is selected with the probability of 15%)
* replace these selected tokens  
  (with the special token **[MASK]** - with p=80%, with a random token - with p=10%, with the original token (remain unchanged) - with p=10%)
* predict original tokens (compute loss).

The illustration below shows an example of a training step for one sentence. You can go over the slides to see the whole process.

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

* GPT is an autoregressive transformer decoder, which means that each token is predicted and conditioned on the previous token. We don't need an encoder, because the previous tokens are received by the decoder itself. This makes these models really good at tasks like language generation, but not good at classification. These models can be trained with unlabeled large text corpora from books or web articles.

In conclusion, while both GPT and BERT are examples of transformer architectures that have been influencing the field of natural language processing in recent years, they have different strengths and weaknesses that make them suitable for different types of tasks. GPT excels at generating long sequences of text with high accuracy whereas BERT focuses more on the understanding context within given texts in order to perform more sophisticated tasks such as question answering or sentiment analysis. Data scientists, developers, and machine learning engineers should decide which architecture best fits their needs before embarking on any NLP project using either model. Ultimately, both GPT and BERT are powerful tools that offer unique advantages depending on the task at hand.

# Positional Embeddings

## Input and Output Pre-processing in Transformers

The input words are represented using some form of embedding. This is done for both encoder and decoder.

Word embedding on their own lack any positional information which is achieved in RNNs by virtue of their sequential nature. Meanwhile in self-attention, due to softmax, any such positional information is lost.

To preserve the positional information, the transformer injects a vector to individual input embeddings (could be using word embeddings for corresponding to the input words). These vectors follow a specific periodic function (Example: combination of various sines/cosines having different frequency, in short not in sync with each other) that the model learns and is able to **determine the position of individual word with each other**based on the values **.**

This injected vector is called “*positional encoding*” and are added to the input embeddings at the bottoms of both encoder and decoder stacks.

## Absolute Positional Embeddings

In the context of a sentence, suppose we have an embedding representing a word. To encode its position, we use another vector of identical dimensionality, where each vector uniquely represents a position in the sentence. For instance, a specific vector is designated for the second word in a sentence. Thus, each sentence position gets its distinct vector. The input for the Transformer layer is then formed by summing the word embedding with its corresponding positional embedding.

**There are primarily two methods to generate these embeddings:**

1. **Learning from Data:** Here, positional vectors are learned during training, just like other model parameters. We learn a unique vector for each position, say from 1 to 512. However, this introduces a limitation — the maximum sequence length is capped. If the model only learns up to position 512, it cannot represent sequences longer than that.
2. **Sinusoidal Functions:**This method involves constructing unique embeddings for each position using a sinusoidal function. Although the intricate details of this construction are complex, it essentially provides a unique positional embedding for every position in a sequence. Empirical studies have shown that learning from data and using sinusoidal functions offer comparable performance in real-world models.

**Limitations of Absolute Positional Embeddings**

Despite their widespread use, absolute positional embeddings are not without drawbacks:

1. **Limited Sequence Length:** As mentioned, if a model learns positional vectors up to a certain point, it cannot inherently represent positions beyond that limit.
2. **Independence of Positional Embeddings:** Each positional embedding is independent of others. This means that in the model’s view, the difference between positions 1 and 2 is the same as between positions 2 and 500. However, intuitively, positions 1 and 2 should be more closely related than position 500, which is significantly farther away. This lack of relative positioning can hinder the model’s ability to understand the nuances of language structure.

## Relative Positional Embeddings

Article: <https://arxiv.org/pdf/2104.09864>

Rather than focusing on a token’s absolute position in a sentence, relative positional embeddings concentrate on the distances between pairs of tokens. This method doesn’t add a position vector to the word vector directly. Instead, it alters the attention mechanism to incorporate relative positional information.

**Case Study: The T5 Model**

One prominent model that utilizes relative positional embeddings is T5 (Text-to-Text Transfer Transformer). T5 introduces a nuanced way of handling positional information:

* **Bias for Positional Offsets:** T5 uses a bias, a floating-point number, to represent each possible positional offset. For example, a bias B1 might represent the relative distance between any two tokens that are one position apart, regardless of their absolute positions in the sentence.
* **Integration in Self-Attention Layer:**This matrix of relative position biases is added to the product of the query and key matrices in the self-attention layer. This ensures that tokens at the same relative distance are always represented by the same bias, regardless of their position in the sequence.
* **Scalability:**A significant advantage of this method is its scalability. It can extend to arbitrarily long sequences, a clear benefit over absolute positional embeddings.

**Challenges with Relative Positional Embeddings**

Despite their theoretical appeal, relative positional embeddings pose certain practical challenge.

1. **Performance Issues:**Benchmarks comparing T5’s relative embeddings with other types have shown that they can be slower, particularly for longer sequences. This is primarily due to the additional computational step in the self-attention layer, where the positional matrix is added to the query-key matrix.
2. **Complexity in Key-Value Cache Usage:**As each additional token alters the embedding for every other token, this complicates the effective use of key-value caches in Transformers. For those unfamiliar, key-value caches are crucial in enhancing efficiency and speed in Transformer models.

Due to these engineering complexities, relative embeddings haven’t been widely adopted, especially in larger language models.

## Rotary Positional Embeddings

RoPE represents a novel approach in encoding positional information. Traditional methods, either absolute or relative, come with their limitations. Absolute positional embeddings assign a unique vector to each position, which though straightforward, doesn’t scale well and fails to capture relative positions effectively. Relative embeddings, on the other hand, focus on the distance between tokens, enhancing the model’s understanding of token relationships but complicating the model architecture.

RoPE ingeniously combines the strengths of both. It encodes positional information in a way that allows the model to understand both the absolute position of tokens and their relative distances. This is achieved through a rotational mechanism, where each position in the sequence is represented by a rotation in the embedding space. The elegance of RoPE lies in its simplicity and efficiency, enabling models to better grasp the nuances of language syntax and semantics.

Изображение выглядит как текст, снимок экрана, диаграмма, число

Автоматически созданное описание

RoPE introduces a novel concept. Instead of adding a positional vector, it applies a rotation to the word vector. Imagine a two-dimensional word vector for “dog.” To encode its position in a sentence, RoPE rotates this vector. The angle of rotation (θ) is proportional to the word’s position in the sentence. For instance, the vector is rotated by θ for the first position, 2θ for the second, and so on. This approach has several benefits:

1. **Stability of Vectors:** Adding tokens at the end of a sentence doesn’t affect the vectors for words at the beginning, facilitating efficient caching.
2. **Preservation of Relative Positions:** If two words, say “pig” and “dog,” maintain the same relative distance in different contexts, their vectors are rotated by the same amount. This ensures that the angle, and consequently the dot product between these vectors, remains constant

The technical implementation of RoPE involves rotation matrices. In a 2D case, the equation from the paper incorporates a rotation matrix that rotates a vector by an angle of Mθ, where M is the absolute position in the sentence. This rotation is applied to the query and key vectors in the self-attention mechanism of the Transformer.

For higher dimensions, the vector is split into 2D chunks, and each pair is rotated independently. This can be visualized as an n-dimensional corkscrew rotating in space.

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

The rotation is executed through simple vector operations rather than matrix multiplication for efficiency. An important property is that words closer together are more likely to have a higher dot product, while those far apart have a lower one, reflecting their relative relevance in a given context.

# Training

## Training Steps

**Step 1 — Pre-training:** In this phase, Large Language Models (LLMs) like GPT-3 are trained on a massive dataset from the internet to predict the next word in a sequence of text. The data is cleaned, preprocessed, and tokenized, and transformer architectures are commonly used for this purpose. The model learns language patterns but doesn’t yet understand instructions or questions.

**Step 2 — Supervised Fine-Tuning or Instruction Tuning:** In this stage, the model is provided with user messages as input and AI trainer responses as targets. The model learns to generate responses by minimizing the difference between its predictions and the provided responses. It begins to understand instructions and learns to retrieve knowledge based on them.

**Step 3 — Reinforcement Learning from Human Feedback (RLHF)**: RLHF is applied as a second fine-tuning step to align the model with human preferences, focusing on being helpful, honest, and harmless (HHH). This involves two sub-steps:

* **Training Reward Model Using Human Feedback:** Multiple model outputs for the same prompt are generated and ranked by human labelers to create a reward model. This model learns human preferences for HHH content.
* **Replacing Humans with Reward Model for Large-Scale Training:** Once the reward model is trained, it can replace humans in labeling data. Feedback from the reward model is used to further fine-tune the LLM at a large scale.

RLHF helps improve the model’s behavior and alignment with human values, ensuring it provides useful, truthful, and safe responses.

## Pre-training

*In the pre-training phase, the model is trained as the next word predictor on internet scale data.*

In pre-training phase

1. Gather a large and diverse dataset from the internet. This dataset contains text from a wide range of sources to ensure the model learns a broad spectrum of language patterns.
2. Clean and preprocess the data to remove noise, formatting issues, and irrelevant information.
3. Tokenize the cleaned text data into smaller units, such as words or subword pieces (e.g., Byte-Pair Encoding or WordPiece).
4. For LLMs like GPT-3, transformer architectures are commonly used due to their effectiveness in handling sequential data.
5. Pre-training of Large Language Models (LLMs) occurs by training the model to predict the next word in a sequence of text, using a massive dataset, to enable it to understand and generate human-like language.

**Output of model after step 1**

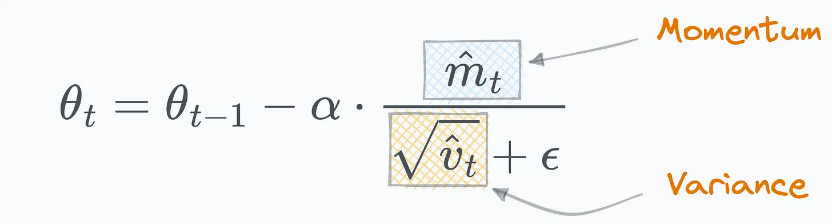
What if we use a model after just pre-training where it has just learned to predict the next word only & does not take input as question or instruction. During training data model might have seen those sequences of questions as some sort of question paper then the model just predicts the next words.

## Memory Consumption during Training

Let’s examine the memory consumption of the current training system. For example, a 1.5B parameter GPT-2 model requires 3GB (1.5B \* 16bit) of memory for its weights (or parameters) in 16-bit precision, yet, it cannot be trained on a single GPU with 32GB memory using Tensorflow or PyTorch. One may wonder where all the memory goes. During model training, most of the memory is consumed by *model states*, i.e., tensors comprising of optimizer states, gradients, and parameters. Besides these model states, the rest of the memory is consumed by activations, temporary buffers and fragmented memory which we call *residual states*. We look at the memory consumption from both in details.

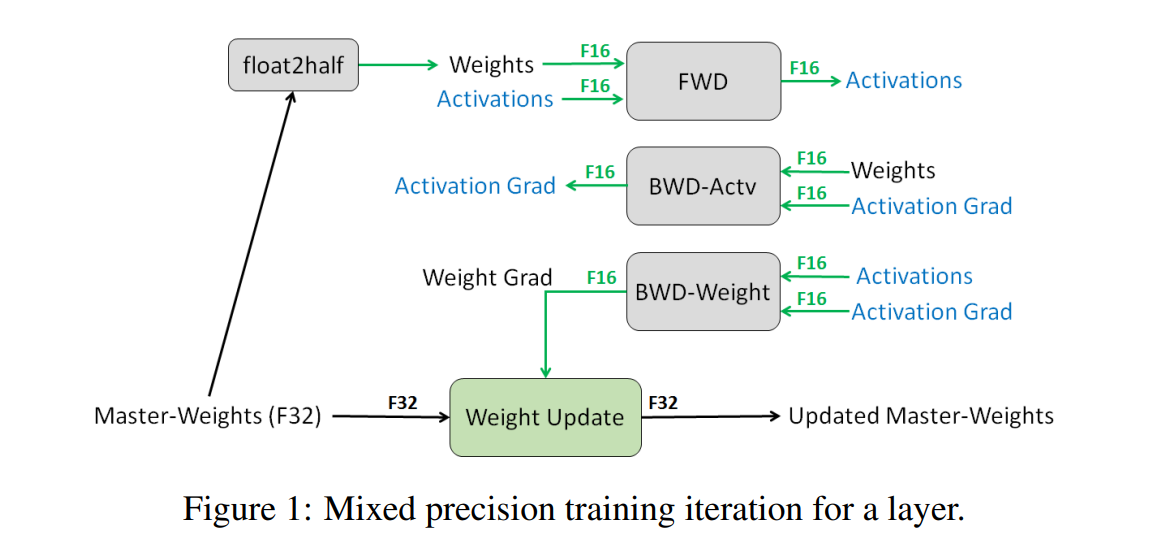
**Model States: Optimizer States, Gradients and Parameters**

Majority of the device memory is consumed by model states during training. Consider for instance, [Adam](https://arxiv.org/abs/1412.6980), one of the most popular optimizers for DL training. Adam requires storing two optimizer states, 1) the time averaged momentum and 2) variance of the gradients to compute the updates.

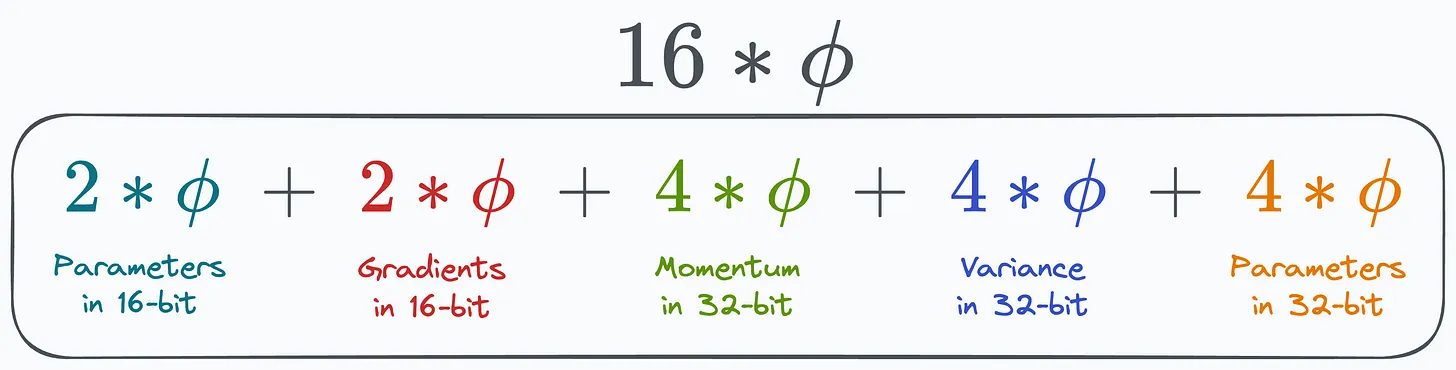


Therefore, to train a model with Adam, there has to be enough memory to hold a copy of both the momentum and variance of the gradients. In addition, there needs to be enough memory to store the gradients and the weights themselves. Of these three types of the parameter-related tensors, the optimizer states usually consume the most memory, specially when mixed-precision training is applied.

**Mixed-Precision Training** The state-of-the-art approach to train large models on the current generation of NVIDIA GPUs is via [mixed precision training](https://arxiv.org/abs/1710.03740), where parameters and activations are stored as fp16, enabling the use of the high throughput tensor core units on these GPUs. During mixed-precision training, both the forward and backward propagation are performed using fp16 weights and activations. However, to effectively compute and apply the updates at the end of the backward propagation, the mixed-precision optimizer keeps an fp32 copy of the parameters as well as an fp32 copy of all the other optimizer states.



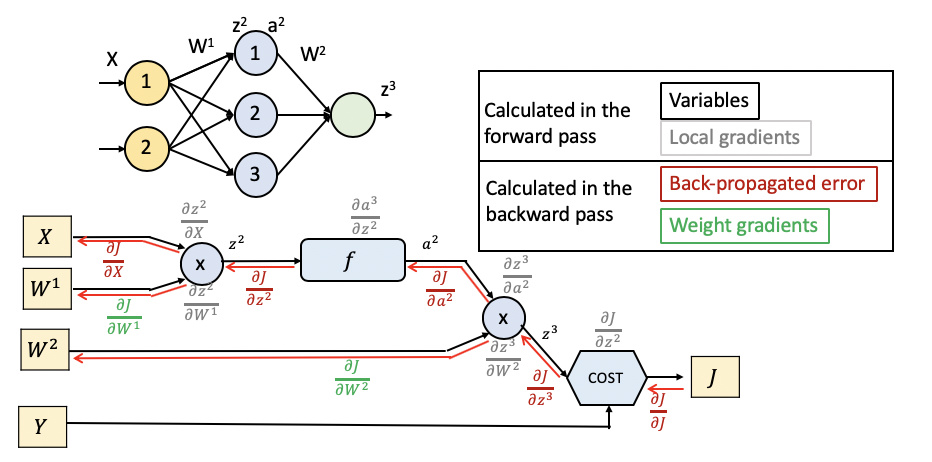
Let’s take Adam as a concrete example. Mixed precision training of a model with Φ parameters using Adam requires enough memory to hold an fp16 copy of the parameters and the gradients, with memory requirements of 2Φ and 2Φ bytes respectively. In addition, it needs to hold the optimizer states: an fp32 copy of the parameters, momentum and variance, with memory requirements of 4Φ, 4Φ, and 4Φ bytes, respectively.



In total, this results 16Φ bytes of memory requirement. For a model such as GPT-2 with 1.5 Billion parameters, this leads to a memory requirement of at least 24 GB, which is significantly higher than the meager 3 GB of memory required to hold the fp16 parameters alone.

**Residual Memory Consumption**

**Activations** can take up a significant amount of memory during training. As a concrete example, the 1.5B parameter GPT-2 model trained with sequence length of 1K and batch size of 32 requires about 60 GB of memory.



The activation memory of a transformer-based model is proportional to the number of *transformer layers* × *hidden dimensions* × *sequence length* × *batch size*.

[**Activation checkpointing**](https://arxiv.org/abs/1604.06174) (or gradient checkpointing) is a common approach to reduce the activation memory by approximately the square root of the total activations at the expense of 33% re-computation overhead. This would reduce the activation memory consumption of this model from 60 GB to about 8 GB.

Despite the significant reduction, the activation memory can grow quite large for bigger models even with activation checkpointing. For example, a GPT-like model with 100 billion parameters requires around 60 GB of memory for batch size 32, even when using activation checkpointing.

**Temporary buffers** used for storing intermediate results consumes non-trivial amount of memory for large models. Operations such as gradient all-reduce, or gradient norm computation tend to fuse all the gradients into a single flattened buffer before applying the operation in an effort to improve throughput. For example, the bandwidth of all-reduce across devices improves with large message sizes. While the gradient themselves are usually stored as fp16 tensors, the fused buffer can be an fp32 tensor depending on the operation. When the size of the model is large, these temporary buffer sizes are non-trivial. For example, for a model with 1.5B parameters, a flattened fp32 buffer would required 6 GB of memory

**Memory Fragmentation**: So far we have discussed the actual memory consumption during training. Additionally, it is possible to run out of usable memory even when there is plenty of available memory. This can happen with memory fragmentation. A request for a memory will fail if there isn’t enough contiguous memory to satisfy it, even if the total available memory is larger than requested. We observe significant memory fragmentation when training very large models, resulting in out of memory issue with over 30% of memory still available in some extreme cases.

## Fine-tuning

Research shows that the pattern-recognition abilities of foundation language models are so powerful that they sometimes require relatively little additional training to learn specific tasks. That additional training helps the model make better predictions on a specific task. This additional training, called [**fine-tuning**](https://developers.google.com/machine-learning/glossary#fine-tuning), unlocks an LLM's practical side.

Fine-tuning trains on examples *specific* to the task your application will perform. Engineers can sometimes fine-tune a foundation LLM on just a few hundred or a few thousand training examples.

Despite the relatively tiny number of training examples, standard fine-tuning is often computationally expensive. That's because standard fine-tuning involves updating the weight and bias of every parameter on each [**backpropagation**](https://developers.google.com/machine-learning/glossary#backpropagation) iteration. Fortunately, a smarter process called [**parameter-efficient tuning**](https://developers.google.com/machine-learning/glossary#parameter-efficient-tuning) can fine-tune an LLM by adjusting only a *subset* of parameters on each backpropagation iteration.

A fine-tuned model's predictions are usually better than the foundation LLM's predictions. However, a fine-tuned model contains the same number of parameters as the foundation LLM. So, if a foundation LLM contains ten billion parameters, then the fine-tuned version will also contain ten billion parameters.

## What is Fine-tuning, and Why is it Important?

Fine-tuning is the process of taking a pre-trained model and further training it on a domain-specific dataset.

Most LLM models today have a very good global performance but fail in specific task-oriented problems. The fine-tuning process offers considerable advantages, including lowered computation expenses and the ability to leverage cutting-edge models without the necessity of building one from the ground up.

Transformers grant access to an extensive collection of pre-trained models suited for various tasks. Fine-tuning these models is a crucial step for improving the model's ability to perform specific tasks, such as sentiment analysis, question answering, or document summarization, with higher accuracy.

Fine-tuning tailors the model to have a better performance for specific tasks, making it more effective and versatile in real-world applications. This process is essential for tailoring an existing model to a particular task or domain.

Whether to engage in fine-tuning hinges on your goals, which typically vary based on the specific domain or task at hand.

Transformers (Audio, Vision, Text) can be fine-tuned in two ways based on the learning algorithms.

### Supervised Fine-Tuning (SFT):

Supervised Fine-tuning is used for task-specific use cases. The model is trained on a labeled dataset. One can implement SFT in the following ways:

* **Full parameter fine-tuning:** fine-tuning the whole model.
* **Parameter-efficient fine-tuning (PEFT):** fine-tuning a specific set of parameters.
* **Instruction fine-tuning:**fine-tuning based on instruction-format dataset.

### Reinforcement Learning Human Feedback(RLHF):

RLHF involves training models through human interaction. It enhances the model to produce accurate and contextual-aware responses. RLHF can be performed in different ways:

* Reward Modeling
* Proximal Policy Optimization (PPO)
* Preference Learning
* \*Direct Preference Optimization (DPO)
* \*Reinforcement Learning AI Feedback (RLAIF)

## Prompt Engineering vs RAG vs Fine tuning.

Let us explore the difference between prompt engineering, RAG, and fine-tuning.

| **Criteria** | **Prompt Engineering** | **RAG** | **Fine-Tuning** |
| --- | --- | --- | --- |
| **Purpose** | Prompt engineering focuses on how to write an effective prompt that can maximize the generation of an optimized output for a given task. | The purpose of[RAG](https://www.geeksforgeeks.org/rag-vs-fine-tuning-for-enhancing-llm-performance/) is to relevant information for a given prompt from an external database. | Fine-tuning focuses on training and adapting a model for a specific task. |
| **Model** | Model weights are not updated. It focuses on building an effective prompt. | Model weights are not updated. It focuses on building context for a given prompt. | Model weights are updated |
| **Complexity** | No technical knowledge required | Compared to fine-tuning it is less complex as it requires skills related to vector databases and retrieval mechanisms only | Technical knowledge required |
| **Compute Cost** | Very less cost. Only costs related to API calls | Cost-effective compared to fine-tuning. | We may need specialized hardware to train the model depending on model size and dataset size |
| **Knowledge** | The model does not learn new data | The prompt is equipped with new data in the form of context | The model learns new data |

**Benefits of Fine Tuning LLMs**

Fine-tuning offers several advantages:

* **Increased Performance:**Fine-tuned models adapt to new data, leading to more accurate and reliable outputs.
* **Efficiency:**Fine-tuning saves computational costs by adapting pre-trained models rather than training a model from scratch.
* **Domain Adaptation:** LLMs can be tailored to specific industries like medical, legal, or financial domains by focusing on relevant terminology and structures.
* **Better Generalization:** Models fine-tuned on task-specific data generalize better to the unique patterns and structures of the task.

## Gradient Accumulation

**Gradient accumulation** is a technique where you can train on bigger batch sizes than your machine would normally be able to fit into memory. This is done by accumulating gradients over several batches and only stepping the optimizer after a certain number of batches have been performed.

For instance, if the gradient accumulation factor is set to 2, the process works as follows: We first calculate the gradient on one batch, which gives us a direction on the [loss function landscape](https://losslandscape.com/). Instead of updating the model weights immediately, we calculate another gradient from the next batch, obtaining a potentially different direction. By adding these two gradients together, we find a more accurate path in the loss landscape. To ensure the final update step is properly scaled, we divide the accumulated gradient by the number of batches, preventing any artificial inflation of the step size.

## How does Gradient Accumulation affect Batch Normalization?

It can interfere with **Batch Normalization (BN)** because BN relies on batch statistics (mean and variance) computed for each mini-batch. When using gradient accumulation, the batch statistics remain based on small batches rather than the effective large batch, potentially leading to suboptimal normalization and training instability.

**Layer Normalization (LN)** does not depend on batch size, as it normalizes values across channels within a single sample instead of across a mini-batch.

## Mixed precision training

**Mixed precision training** is a technique that aims to optimize the computational efficiency of training models by utilizing lower-precision numerical formats for certain variables. Traditionally, most models use 32-bit floating point precision (fp32 or float32) to represent and process variables. However, not all variables require this high precision level to achieve accurate results. By reducing the precision of certain variables to lower numerical formats like 16-bit floating point (fp16 or float16), we can speed up the computations. Because in this approach some computations are performed in half-precision, while some are still in full precision, the approach is called mixed precision training.

Most commonly mixed precision training is achieved by using fp16 (float16) data types, however, some GPU architectures (such as the Ampere architecture) offer bf16 and tf32 (CUDA internal data type) data types.

# Supervised Fine-Tuning

## Supervised fine-tuning or instruction tuning

In the SFT or instruction tuning phase

1. During this process, the model is provided with the user’s message as input and the AI trainer’s response as the target. The model learns to generate responses by minimizing the difference between its predictions and the provided responses.
2. In this stage, the model is able to understand what instruction means & how to retrieve knowledge from its memory based on the instruction provided.

**Output of model after step 2**

So to continue with the previous example, In training data model has seen the example “**what is capital of India?**” & human labeled output for this is “**Delhi**”.

Now model learned the relation between **what is asked from LLM** and **what should be the output**. so if you now ask the question “**What is Capital of France ?**” the model is more likely to say “**Paris**”

## Parameter-efficient fine-tuning (PEFT)

PEFT: To efficiently fine-tune models for various applications without fine-tuning all the model parameters. PEFT can be applied to any model — large language models, small language models, and deep neural networks.

Trade-offs: Parameter Efficiency, Training Speed, Memory Efficiency, Model Performance, Inference Costs.

PEFT can be implemented in a couple of methods:

1. **Selective** — selecting a subset of initial language model parameters to fine-tune.
2. **Additive** — adding trainable layers or parameters to the model. For example, LoRA, Soft Prompt (Prompt Tuning)…

PEFT (Parameter-Efficient Fine-Tuning) is a library for efficiently adapting large pretrained models to various downstream applications without fine-tuning all of a model’s parameters because it is prohibitively costly. PEFT methods only fine-tune a small number of (extra) model parameters - significantly decreasing computational and storage costs - while yielding performance comparable to a fully fine-tuned model. This makes it more accessible to train and store large language models (LLMs) on consumer hardware.

PEFT is integrated with the Transformers, Diffusers, and Accelerate libraries to provide a faster and easier way to load, train, and use large models for inference.

## Prefix Tuning

* **What it tunes**: Learns **soft prefixes** (KV cache) *for every layer* in the transformer.
* **Key points**:
  + Modifies the **key (K) and value (V)** matrices in *all attention layers*.
  + Uses a small **MLP to generate prefixes** (avoiding direct tuning of high-dimensional vectors).
  + More expressive than prompt tuning (layer-specific adaptation).
  + Originally applied to **GPT-2** (autoregressive) and **BART** (seq2seq).
* **Difference from Prompt Tuning**:
  + Prefix tuning affects *all layers*; prompt tuning only affects the input.

Изображение выглядит как текст, снимок экрана, дизайн

Автоматически созданное описание

## Soft Prompts

Soft prompts are created during the process of prompt tuning.

Unlike hard prompts, soft prompts cannot be viewed and edited in text. Prompts consist of an embedding, a string of numbers, that derives knowledge from the larger model.

So for sure, a disadvantage is the lack of interpretability of soft prompts. The AI discovers prompts relevant for a specific task but can’t explain why it chose those embeddings. Like deep learning models themselves, soft prompts are opaque.

Prompt tuning involves using a small trainable model before using the LLM. The small model is used to encode the text prompt and generate task-specific virtual tokens.

## Prompt Tuning

Prompt tuning is a form of task-specific adaptation where the model is provided with a **trainable prompt**. Unlike Prefix Tuning, which involves adding the prefix to the input sequence, prompt tuning typically focuses on optimizing the **initial part of the input prompt** that is used to condition the model. This can be a trainable set of tokens that guide the model's responses to be more task-specific.

* **What it tunes**: Learns **soft prompt embeddings** (continuous vectors) *concatenated to the input* (not intermediate layers).
* **Key points**:
  + Only the **input embeddings** are modified; the rest of the model is frozen.
  + Soft prompts are *task-specific* but *not layer-specific*.
  + Originally applied to **T5** (sequence-to-sequence models).

Изображение выглядит как текст, снимок экрана, Прямоугольник, прямоугольный

Автоматически созданное описание

## P-tuning

* **What it tunes**: Learns **soft prompts** (like prompt tuning) but uses an **LSTM** (or MLP) to generate them.
* **Key points**:
  + Still operates at the *input level* (not layer-specific).
  + Replaces discrete prompts with *continuous, trainable embeddings*.
  + Originally applied to **GPT-2**, **BERT**, and **RoBERTa**.
* **Why LSTM?**:
  + Models dependencies between soft prompt tokens (unlike static embeddings in prompt tuning).

[P-tuning](https://hf.co/papers/2103.10385) adds trainable prompt embeddings to the input that is optimized by a prompt encoder to find a better prompt, eliminating the need to manually design prompts. The prompt tokens can be added anywhere in the input sequence, and p-tuning also introduces anchor tokens for improving performance.

Изображение выглядит как текст, линия, Шрифт, диаграмма

Автоматически созданное описание

P-Tuning is similar to prompt tuning, but it uses **continuous embeddings** (rather than discrete tokens) as the prompt. These continuous embeddings are trained and optimized in a manner that acts as a dynamic “prompt” to guide the model’s behavior. The embeddings are typically learned through gradient-based optimization and serve as a way to influence the model without needing to rely on specific, pre-defined tokens.

## Adapters

* **Adapters** are small, trainable neural modules that are inserted into certain layers of a pre-trained model, typically between the model’s existing layers. These modules learn the specific task or domain-specific knowledge, while the rest of the pre-trained model's weights remain frozen (i.e., unchanged).
* The main goal of adapters is to allow a pre-trained model to be adapted to new tasks efficiently by **only fine-tuning the adapter modules**. This is in contrast to traditional fine-tuning, where the entire model’s parameters are updated.

**How Do Adapters Work?**

* The typical approach is to insert the adapter modules into **transformer layers** (like the attention or feedforward layers) of the pre-trained model. The adapter modules are small networks (usually consisting of a down-projection, non-linearity, and up-projection) that process the input and adjust the representation at the given layer.
* The model's original weights are kept frozen, and only the parameters of the adapters are updated during training.

## What is LoRa?

[LoRA](https://arxiv.org/abs/2106.09685) is an improved finetuning method where instead of finetuning all the weights that constitute the weight matrix of the pre-trained large language model, two smaller matrices that approximate this larger matrix are fine-tuned. These matrices constitute the LoRA adapter. This fine-tuned adapter is then loaded into the pre-trained model and used for inference.

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

h=W0​x+ΔWx=W0​x+BAx

A neural network contains many dense layers which perform matrix multiplication. The weight matrices in these layers typically have full-rank. When adapting to a specific task, Aghajanyan et al. (2020) shows that the pre-trained language models have a low “instrisic dimension” and can still learn efficiently despite a random projection to a smaller subspace

This means that while training for a broad, complex task, the weight matrices in a neural network have full rank, which minimizes redundancy. However, when fine-tuning this universal model for a specialized task, not all the knowledge from the original model is necessary. Therefore, only a small fraction of the parameters needs to be trained. In simpler terms, the weight matrices can be represented by smaller matrices with fewer parameters. Thus, during full fine-tuning, the weight matrices can be considered low-rank, indicating that full fine-tuning involves some degree of redundancy.

Inspired by this, we hypothesize the updates to the weights also have a low “intrinsic rank” during adaptation.

Given that low-rank weight matrices suffice for full fine-tuning on a downstream task, it's reasonable to assume that the gradient updates themselves can be represented by low-rank matrices.

The most significant benefit comes from the reduction in memory and storage usage. For a large Transformer trained with Adam, we reduce that VRAM usage by up to 2/3 if r≪d as we do not need to store the gradients and optimizer states for the frozen parameters. We also observe a 25% speedup during training on GPT-3 175B compared to full fine-tuning as we do not need to calculate the gradient for the vast majority of the parameters.

## What is Quantized LoRA (QLoRA)?

[QLoRA](https://arxiv.org/abs/2305.14314) uses 4-bit quantization to compress a pretrained language model. The LM parameters are then frozen and a relatively small number of trainable parameters are added to the model in the form of Low-Rank Adapters. During finetuning, QLoRA backpropagates gradients through the frozen 4-bit quantized pretrained language model into the Low-Rank Adapters. The LoRA layers are the only parameters being updated during training.

QLoRA has one storage data type (usually 4-bit NormalFloat) for the base model weights and a computation data type (16-bit BrainFloat) used to perform computations. QLoRA dequantizes weights from the storage data type to the computation data type to perform the forward and backward passes, but only computes weight gradients for the LoRA parameters which use 16-bit bfloat. The weights are decompressed only when they are needed, therefore the memory usage stays low during training and inference.

# Alignment

## AI alignment

Alignment is an emerging field of study where you ensure that an AI system performs exactly what you want it to perform. Think of a framework like Asimov’s Three Laws of Robotics as a rough example. A first google search might lead you to this definition by IBM, “Alignment is the process of encoding human values and goals into large language models to make them as helpful, safe, and reliable as possible. “ In the context of LLMs specifically, alignment is a process that trains an LLM to ensure that the generated outputs align with human values and goals. This is also called as alignment with respect to human preferences, hence “preference optimization”.

## Why do we need Alignment?

There are a couple of examples where the model behaves badly.

If I ask the model how’s the weather outside, it might respond saying it's really good. But is this answer helpful?

Also sometimes the model might respond with which is completely wrong. A Very famous example of this is **Can coughing effectively stop heart attack?** This isn’t true but the model might respond by saying **yes it can**.

*We want our model to be honest and don’t give misleading information which isn’t true.*

Also sometime model can provide answers to which it shouldn’t. **how can I rob a bank?** it definitely should not respond to this. It can create sometimes harmful content as well.

*Helpful, honest, and Harmless is also known as HHH. So we want to align the model with human preferences. RLHF helps us to do this.*

## What are the current methods for LLM alignment?

You will find many alignment methods in research literature, we will only stick to 3 alignment methods for the sake of discussion.

**RLHF (Reinforcement Learning with Human Feedback):**

* Step 1 & 2: Train an LLM (pre-training for the base model + supervised/instruction fine-tuning for a chat variant)
* Step 3: RLHF uses an ancillary language model (it could be much smaller than the main LLM) to learn human preferences. This can be done using a preference dataset — it contains a prompt, and a response/set of responses graded by expert human labelers. This is called a “reward model”.
* Step 4: Use a reinforcement learning algorithm (eg: PPO — proximal policy optimization), where the LLM is the agent, the reward model provides a positive or negative reward to the LLM based on how well it’s responses align with the “human preferred responses”.

In theory, it is as simple as that. However, implementation isn’t that easy — requiring lot of human experts and compute resources. To overcome the “expense” of RLHF, researchers developed DPO

**DPO (Direct Preference Optimization):**

* Step 1&2 remain the same
* Step 4: DPO eliminates the need for the training of a reward model (i.e step 3). How? DPO defines an additional preference loss as a function of it’s policy and uses the language model directly as the reward model. The idea is simple, If you are already training such a powerful LLM, why not train itself to distinguish between good and bad responses, instead of using another model?
* DPO is shown to be more computationally efficient (in case of RLHF you also need to constantly monitoring the behavior of the reward model) and has better performance than RLHF in several settings.

**ORPO (Odds Ratio Preference Optimization):**

* The newest method so far, ORPO combines Step 2, 3 & 4 into a single step — so the dataset required for this method is a combination of a fine-tuning + preference dataset.
* The supervised fine-tuning and alignment/preference optimization is performed in a single step. This is because the fine-tuning step, while allowing the model to specialize to tasks and domains, it can also increase the probability of undesired responses from the model.
* ORPO combines the steps using a single objective function by incorporating an odds ratio (OR) term — reward preferred responses & penalizing rejected responses.

## Reinforcement Learning from Human Feedback

For RLHF you will start with an instruction fine-tuned model. The objective of the RLHF is

*Maximize helpfulness*

*Minimize harm*

*Avoid dangerous topics*

**RLHF steps**

* **Training reward model using Human feedback**

In RLHF, we will generate multiple outputs for the same prompt & ask the human labeler to rank output from best to worst. This data is used to train another NN model which is called the reward model. This reward model is now able to understand human preferences. Think of it as training an intern by experts to identify Helpful, honest, and Harmless content.

* **Replacing humans with a reward model for large-scale training**

Once the reward model is trained, this can be used instead of human beings to label data & feedback on it can be used to further fine-tune LLM at a large scale.

Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Автоматически созданное описание

## Reward Model

Obtaining human feedback is time-consuming and expensive. As a workaround, we can train another model called Reward Model, as a proxy for human feedback. The goal of a reward model is to evaluate the degree of alignment a model response has with human preferences. On a simpler note, a reward model is a model that takes *(prompt, response)*pair as input and outputs a reward/score as output. This can be formulated as a simple regression or classification task. The real challenge in building such a model is good quality dataset. The perception of good/bad differs from person to person and to map it to a scaler quantity is infeasible.

One workaround is to ask labelers to compare two responses and decide which one is better. This kind of dataset is called a comparison dataset, each record comprises (prompt, chosen response, rejected response).

**Training**

To train a reward model the comparison dataset should be in the format of (prompt, chosen response, rejected response ), i.e. the better option comes first. The ordering is crucial because it is the base assumption while designing the loss function for the reward model. Any model that can take in variable length text input and output a scaler value can be used. Typically we use an SFT model which aligns with our task and remove the last de-embedding layer while adding a single neuron in the last layer for the scaler output.

For every epoch, we do two passes of the model. In the first pass, we feed in prompt and chosen response to the Reward Model, the output is Rchosen. In the second pass, we feed in the same prompt along with the rejected response. The output, in this case, is Rrejected. Next, we use the loss function defined below to update the reward model.

Изображение выглядит как Шрифт, текст, белый, типография

Автоматически созданное описание

The intuition behind the loss function is to maximize the gap between chosen response score and rejected response score. For a very high reward score for chosen response and a low reward score for rejected response, the loss would be 0.

## Proximal Policy Optimization (PPO)

PPO is a widely used reinforcement learning algorithm known for its balance between simplicity and performance. It employs a clipped surrogate objective function to limit the size of policy updates, ensuring stable training. PPO is computationally efficient, requires minimal hyperparameter tuning, and performs well across various tasks. It is frequently used to fine-tune large language models (LLMs) by maximizing rewards based on human feedback, ensuring stable and reliable policy updates.

**How PPO Works**

1. **Rollout**:

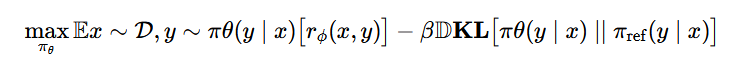
* **Process**: The language model generates a response or continuation based on a given query, which could be the start of a sentence or another prompt.
* **Objective**: Create query/response pairs that will be evaluated in the next step.

**2. Evaluation**:

* **Process**: The query and response pairs are evaluated using a function, model, human feedback, or a combination of these methods.
* **Objective**: Produce a scalar value (reward) for each query/response pair. This reward reflects the quality and alignment of the response with human preferences.

**3. Optimization**:

* **Process**: The optimization step involves calculating the log-probabilities of the tokens in the sequences using both the trained model and a reference model (usually the pre-trained model before fine-tuning).
* **KL-Divergence**: The Kullback-Leibler (KL) divergence between the outputs of the active model and the reference model is used as an additional reward signal. This ensures that the generated responses do not deviate too far from the reference language model.
* **Training**: The active language model is trained using PPO, incorporating both the primary reward signal and the KL-divergence penalty.



**Challenges of PPO**

* **Complexity**: The optimization step is intricate and requires careful handling of log-probabilities and KL-divergence.
* **Slow Iteration**: PPO involves numerous iterations, debugging, and fine-tuning to achieve optimal performance.
* **Debugging Difficulty**: As an RL algorithm, PPO can be slow and challenging to debug, especially when ensuring the correct application of human feedback and reward signals.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

## RLAIF: Reinforcement Learning from Human Feedback with AI Feedback

Reinforcement learning from human feedback (RLHF) is effective at aligning large language models (LLMs) to human preferences, but gathering high quality human preference labels is a key bottleneck. We conduct a head-to-head comparison of RLHF vs. RL from AI Feedback (RLAIF) - a technique where preferences are labeled by an off-the-shelf LLM in lieu of humans, and we find that they result in similar improvements. On the task of summarization, human evaluators prefer generations from both RLAIF and RLHF over a baseline supervised fine-tuned model in ~70% of cases. Furthermore, when asked to rate RLAIF vs. RLHF summaries, humans prefer both at equal rates. These results suggest that RLAIF can yield human-level performance, offering a potential solution to the scalability limitations of RLHF.

## Direct Preference Optimization (DPO)

**Overview**

Direct Preference Optimization (DPO) is an alternative to Proximal Policy Optimization (PPO) for fine-tuning models with human alignment. Unlike PPO, DPO avoids complex normalization terms that can be difficult to optimize. Instead, DPO uses an updated or re-parameterized cross-entropy loss function that incorporates preference data. The training loss function penalizes the model for generating less-preferred outputs and increases the log probability of preferred responses.

**Advantages of DPO**

* **Simplicity**: DPO does not require the complex normalization terms used in PPO, making it easier to optimize.
* **Direct Optimization**: Uses a straightforward approach to maximize the log-likelihood of preferred responses, simplifying the fine-tuning process.
* **Efficiency**: Faster and easier to implement compared to PPO, requiring fewer iterations and less debugging.

**How DPO Works**

1. **Train the SFT Model**:

* **Objective**: Ensure the data used for training is appropriate and in-distribution for the DPO algorithm.
* **Process**: Begin with supervised fine-tuning (SFT) to align the model with initial user expectations.

**2. Data Collection**:

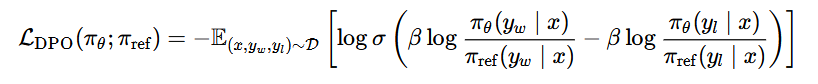
* **Process**: Gather a preference dataset consisting of positive and negative selected pairs of generated responses for given prompts.
* **Objective**: Create a dataset that clearly distinguishes between preferred and less-preferred outputs based on human feedback.

**3. Optimization**:

* **Process**: Maximize the log-likelihood of the DPO loss directly. The loss function is designed to penalize the model for producing less-preferred outputs while increasing the log probability of preferred responses.
* **Objective**: Directly optimize the model to align its outputs with human preferences using the collected preference data.

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание



## Odds Ratio Preference Optimization (ORPO)

**Overview**

Odds Ratio Preference Optimization (ORPO) is a new fine-tuning technique that integrates Supervised Fine-Tuning (SFT) and preference alignment into a single process. By leveraging preference data, ORPO posits that a minor penalty for disfavored outputs, combined with a strong adaptation signal for the chosen response via a simple log odds ratio term appended to the Negative Log-Likelihood (NLL) loss, is sufficient for preference-aligned SFT. This approach eliminates the need for an additional preference alignment phase, saving computational resources and memory.

**Key Features of ORPO**

* **Reference Model-Free**: ORPO does not require a separate reference model for preference alignment, streamlining the fine-tuning process.
* **No Reward Model Needed**: Unlike PPO, ORPO eliminates the need for a reward model, simplifying the architecture.
* **Integrated Loss Function**: Combines the benefits of PPO and DPO through a loss function that incorporates cross-entropy and odds ratio preference optimization, penalizing disfavored responses while emphasizing preferred ones.
* **Efficient and Effective**: ORPO widens the distance between chosen and rejected responses, ensuring better alignment with human preferences without additional computational overhead.

**How ORPO Works**

1. **Train the SFT Model**:

* **Objective**: Ensure the initial data is appropriate for training, aligning the model with user expectations through supervised fine-tuning.

**2. Data Collection**:

* **Process**: Gather preference data consisting of prompts and their respective chosen and rejected responses.
* **Objective**: Create a dataset that clearly distinguishes between preferred and less-preferred outputs based on human feedback.

**3. Optimization**:

* **Process**: Apply a loss function that combines cross-entropy loss with an odds ratio preference optimization term.
* **Objective**: Penalize disfavored responses and strongly adapt to preferred ones, using a single integrated process for fine-tuning.

**ORPO Loss Function**

* **Cross-Entropy Loss**: Measures the difference between the predicted probability distribution and the true distribution of responses.
* **Odds Ratio Preference Optimization (ORPO) Term**: Increases the log probability of preferred responses and decreases the log probability of disfavored responses.

*Loss = Cross-Entropy Loss + Odds Ratio Preference Optimization*

This integrated loss function helps fine-tune the model to produce outputs that align more closely with human preferences by directly penalizing less-preferred responses and enhancing the probability of chosen responses.

## Kahneman-Tversky Optimization (KTO)

**Kahneman-Tversky Optimization (KTO)** is an approach to AI alignment inspired by **prospect theory**, developed by **Daniel Kahneman and Amos Tversky**. It focuses on making AI systems more aligned with **human decision-making biases** and risk assessments.

Изображение выглядит как текст, линия, снимок экрана, Шрифт

Автоматически созданное описание

**Key Ideas of KTO Alignment:**

1. **Humans Are Not Perfectly Rational**
   * Traditional AI optimization assumes people make rational choices (like in expected utility theory).
   * Kahneman & Tversky showed that humans often make **irrational but predictable** decisions due to cognitive biases.
2. **Loss Aversion & Risk Perception**
   * People **fear losses more than they value equivalent gains** (e.g., losing $100 feels worse than gaining $100 feels good).
   * AI using KTO takes this into account, optimizing not just for **expected value** but also for **perceived risk**.
3. **Framing Effects & Decision Context**
   * The way choices are **framed** influences decisions (e.g., "90% survival" vs. "10% mortality").
   * AI aligned with KTO ensures that recommendations and interactions consider **human psychological framing**.

Изображение выглядит как текст, Шрифт, рукописный текст, снимок экрана

Автоматически созданное описание

## DPO vs PPO vs KTO

| **Algorithm** | **Type** | **Key Idea** | **Strengths** | **Weaknesses** |
| --- | --- | --- | --- | --- |
| **PPO** (Proximal Policy Optimization) | Policy Gradient RL | Optimizes a policy while limiting drastic updates via clipping | Stable, sample-efficient, widely used in RL (e.g., robotics, games) | Struggles with preference-based learning, needs well-defined rewards |
| **DPO** (Direct Preference Optimization) | Preference-based Learning | Trains a model directly from human preference rankings instead of reward functions | Works well in **alignment** tasks (e.g., LLM fine-tuning), avoids explicit reward modeling | Needs high-quality human preference data |
| **KTO** (Kahneman-Tversky Optimization) | Human-aligned RL | Inspired by **Prospect Theory**, adjusts AI decisions to match human **risk perception** and biases | More human-aligned decisions, considers loss aversion and framing effects | More complex to implement, requires psychological modeling |

**When to Use Each?**

* **PPO** → Best for traditional **reinforcement learning** with clear rewards (e.g., robotics, games).
* **DPO** → Best for aligning AI models with **human preferences** (e.g., fine-tuning LLMs).
* **KTO** → Best for AI systems needing **human-like risk perception** (e.g., finance, medical AI).

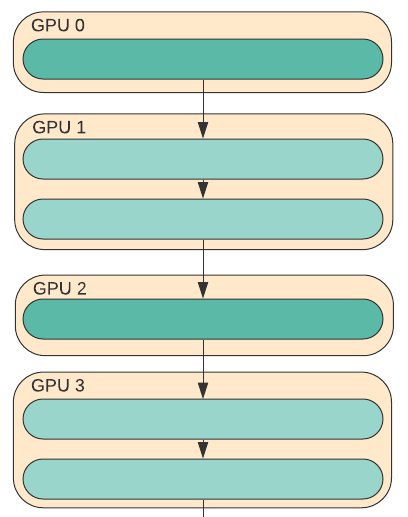
# MultiGpu Training

## Model Parallelism, Tensor Parallelism, Pipeline Parallelism

When a model does not fit in the device memory, model parallelism split the model among processes, in vertical or horizontal way.

### Naive Model Parallelism

This approach involves distributing groups of model layers across multiple GPUs by assigning specific layers to specific GPUs. As data flows through these layers, it is moved to the same GPU as the layer, while the other layers remain untouched.

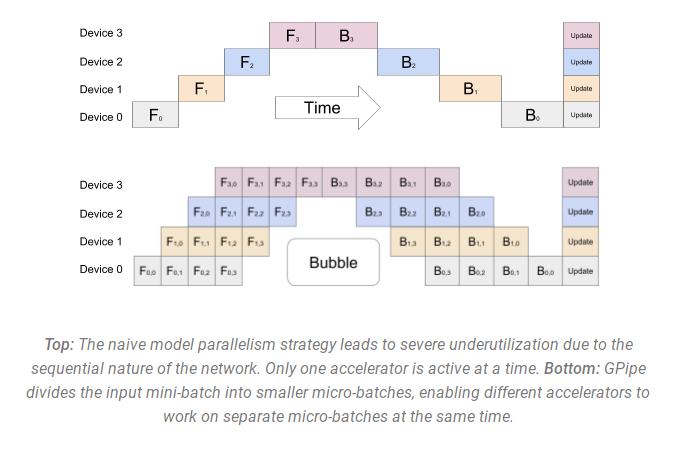
[](https://cdn-uploads.huggingface.co/production/uploads/660710b03ef451aa2bab8971/MrYIdXnXDTQNGbL3_8DMC.png)

In this example, when data moves through layers within one GPU, it’s no different from regular forward pass. However, moving data between layers on different GPUs results in a communication overhead. If the participating GPUs are on the same compute node (e.g. same physical machine) this copying is fast, but if the GPUs are distributed across different compute nodes (e.g. multiple machines), the communication overhead could be substantially greater.

The main problem with Naive Model Parallelism is that **аll but one GPU are idle at any given moment**, which is very inefficient.

### Pipeline Parallelism

PP is almost identical to a naive MP, but it solves the GPU idling problem by chunking the incoming batch into micro-batches and artificially creating a pipeline, which allows different GPUs to concurrently participate in the computation process.

[](https://cdn-uploads.huggingface.co/production/uploads/660710b03ef451aa2bab8971/8B1qNEhiSN8EcAb_jgxlo.png)

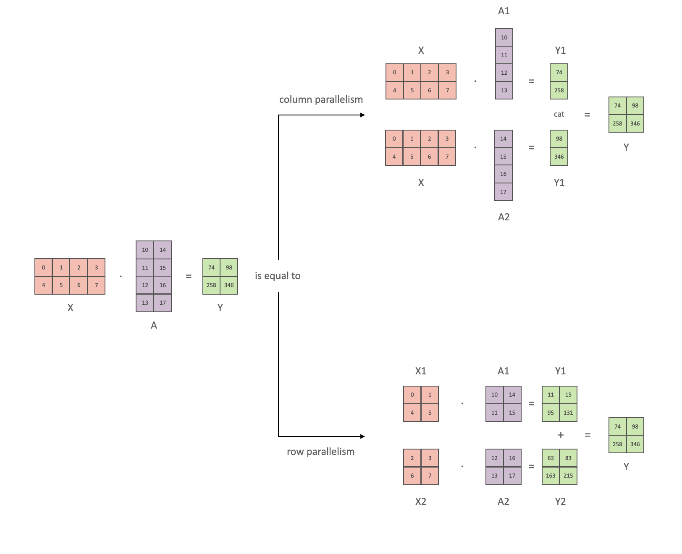
But this comes at the expense of a great deal of technical complication.

### Tensor Parallelism

In Tensor Parallelism, each GPU processes a slice of a tensor and only aggregates the full tensor for operations requiring it. So, unlike Model Parallelism (MP), we don't have to wait for the previous GPUs to finish processing the previous layers of the model. This allows for more efficient processing and reduced idle time.

The main building block of any transformer is a fully connected **nn.Linear** followed by a nonlinear activation **GeLU**. The dot dot-product part of it, following the [Megatron’s paper](https://arxiv.org/abs/2104.04473) notation, can be written as **Y = GeLU(XA)**, where **X** is an input vector, **Y** is the output vector, and **A** is the weight matrix.

If we look at the computation in matrix form, you can see how the matrix multiplication can be split between multiple GPUs:

[](https://cdn-uploads.huggingface.co/production/uploads/660710b03ef451aa2bab8971/oOGpgNvibHgLg7ST7jyH1.png)

### Data Parallelism + Pipeline Parallelism + Tensor Parallelism

To get an even more efficient training a 3D parallelism is used where PP is combined with TP and DP.

## Collective Operations

Before diving into distributed training, it’s beneficial to first understand the basic operations involved in multi-GPU and multi-node communication.

For this purpose, we'll focus on the [NVIDIA NCCL](https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html)

The NVIDIA Collective Communication Library (NCCL) implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and Networking. NCCL provides routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter as well as point-to-point send and receive that are optimized to achieve high bandwidth and low latency over PCIe and NVLink high-speed interconnects within a node and over NVIDIA Mellanox Network across nodes.

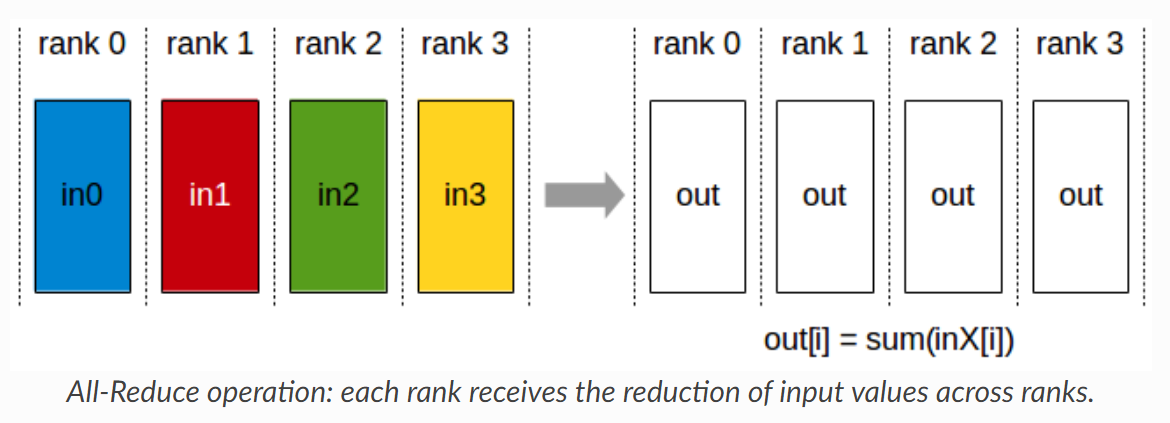
Leading deep learning frameworks such as Caffe2, Chainer, MxNet, PyTorch and TensorFlow have integrated NCCL to accelerate deep learning training on multi-GPU multi-node systems.

Collective operations have to be called for each rank (hence CUDA device) to form a complete collective operation. Failure to do so will result in other ranks waiting indefinitely.

### AllReduce

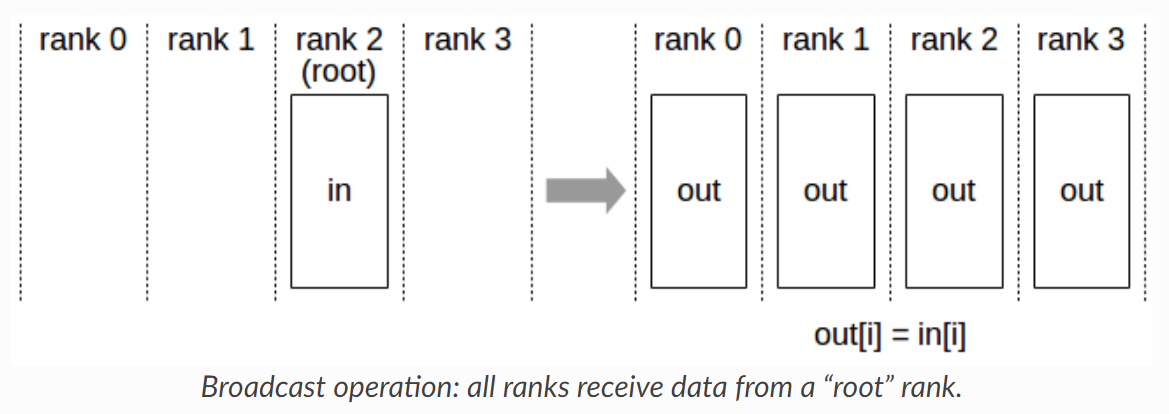
The AllReduce operation performs reductions on data (for example, sum, min, max) across devices and stores the result in the receive buffer of every rank.

In a sum allreduce operation between k ranks, each rank will provide an array in of N values, and receive identical results in array out of N values, where out[i] = in0[i]+in1[i]+…+in(k-1)[i].

[](https://cdn-uploads.huggingface.co/production/uploads/660710b03ef451aa2bab8971/fEKFr4jK-NCawJmD4ZbPq.png)

### Broadcast

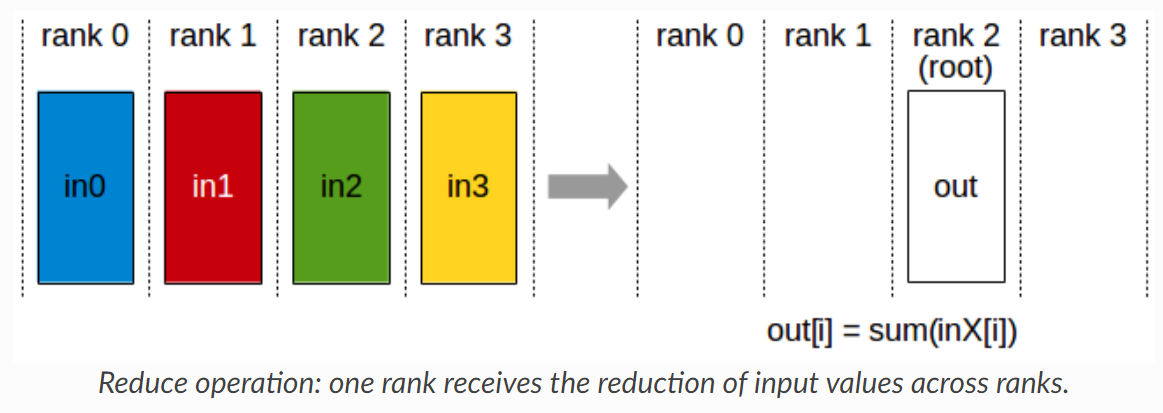
The Broadcast operation copies an N-element buffer from the root rank to all the ranks.

[](https://cdn-uploads.huggingface.co/production/uploads/660710b03ef451aa2bab8971/PSZn_fdIu5oAiQhx2us4X.png)

Important note: The root argument is one of the ranks, not a device number, and is therefore impacted by a different rank to device mapping.

### Reduce

The Reduce operation performs the same operation as AllReduce, but stores the result only in the receive buffer of a specified root rank.

[](https://cdn-uploads.huggingface.co/production/uploads/660710b03ef451aa2bab8971/tXisKmL7_Q9MqEjJA2twc.png)

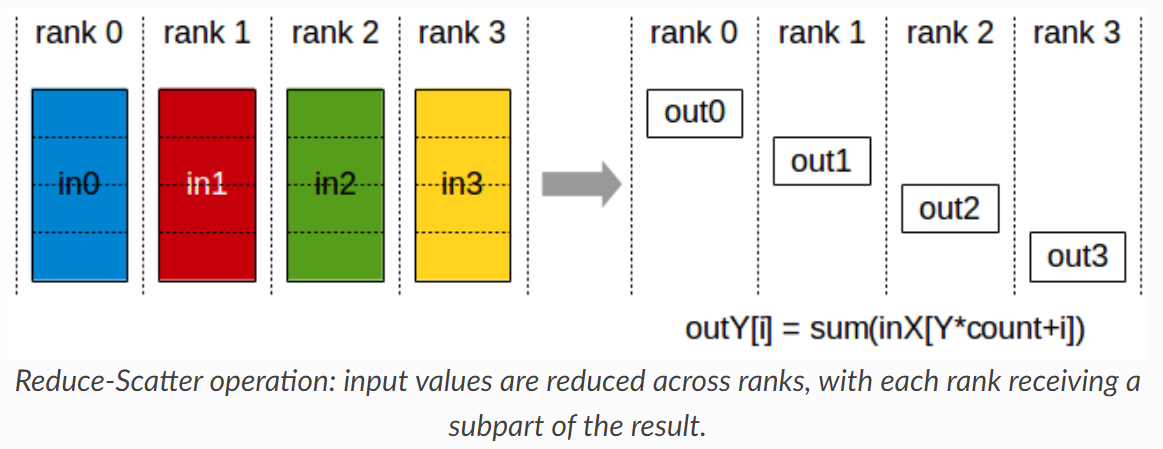
Important note: The root argument is one of the ranks (not a device number), and is therefore impacted by a different rank to device mapping.

Note: A Reduce, followed by a Broadcast, is equivalent to the AllReduce operation.

### ReduceScatter

The ReduceScatter operation performs the same operation as Reduce, except that the result is scattered in equal-sized blocks between ranks, each rank getting a chunk of data based on its rank index.

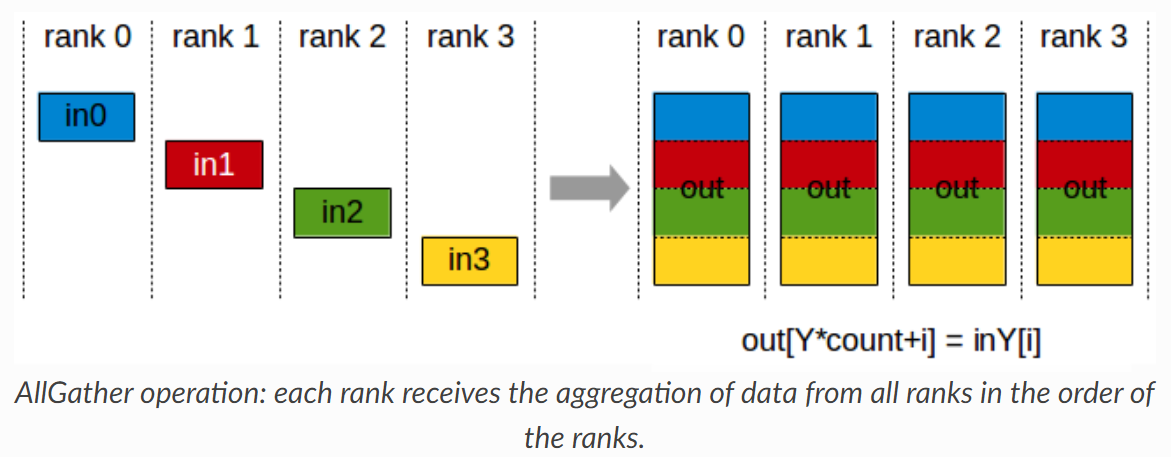
The ReduceScatter operation is impacted by a different rank to device mapping since the ranks determine the data layout.

[](https://cdn-uploads.huggingface.co/production/uploads/660710b03ef451aa2bab8971/Z6wffjJ3IPcpyyfpyzcHD.png)

### AllGather

The AllGather operation gathers N values from k ranks into an output buffer of size k\*N, and distributes that result to all ranks.

The output is ordered by the rank index. The AllGather operation is therefore impacted by a different rank to device mapping.

[](https://cdn-uploads.huggingface.co/production/uploads/660710b03ef451aa2bab8971/fvBzt8G7NsjKQTDvJanDK.png)

Note: Executing ReduceScatter, followed by AllGather, is equivalent to the AllReduce operation.

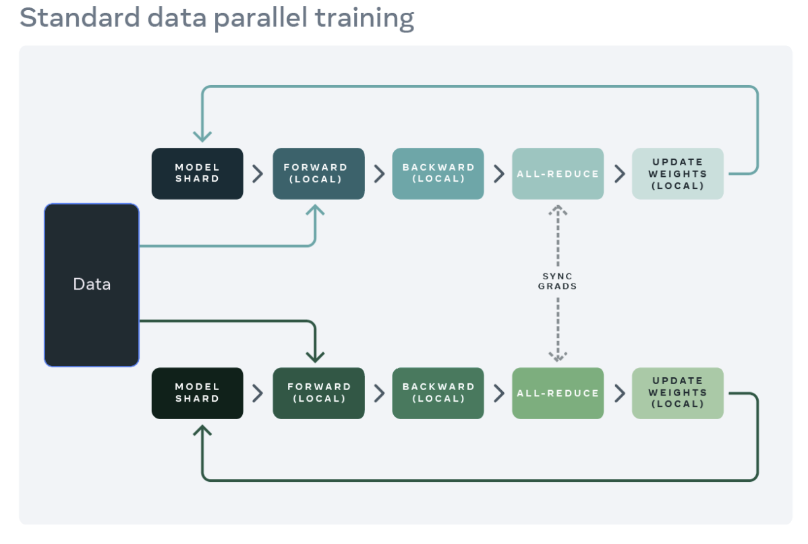
## Distributed Data Parallel

[DistributedDataParallel](https://pytorch.org/docs/stable/nn.html#module-torch.nn.parallel) (DDP) is a powerful module in PyTorch that allows you to parallelize your model across multiple machines, making it perfect for large-scale deep learning applications. To use DDP, you’ll need to spawn multiple processes and create a single instance of DDP per process.

But how does it work? DDP uses collective communications from the [torch.distributed](https://pytorch.org/tutorials/intermediate/dist_tuto.html) package to synchronize gradients and buffers across all processes. This means that each process will have its own copy of the model, but they’ll all work together to train the model as if it were on a single machine.

To make this happen, DDP registers an autograd hook for each parameter in the model. When the backward pass is run, this hook fires and triggers gradient synchronization across all processes. This ensures that each process has the same gradients, which are then used to update the model.

The recommended way to use DDP is to spawn one process for each model replica. The model replica can span multiple devices. DDP processes can be placed on the same machine or across machines. Note that GPU devices cannot be shared across DDP processes (i.e. one GPU for one DDP process).

[](https://cdn-uploads.huggingface.co/production/uploads/660710b03ef451aa2bab8971/uZ60Sbc6Q7ZrvzQSIR1xX.png)

## FSDP - Fully Sharded Data Parallel

[FSDP](https://arxiv.org/abs/2304.11277) expands upon distributed data parallel, by parallelizing not just data, but the model parameters, the optimizer states and gradients associated with the model. Specifically - each GPU only stores a subset of the entire model and the associated subset of optimizer states and gradients.

**FSDP Units**

FSDP breaks down a model instance into smaller units and then flattens and shards all of the parameters within each unit. The sharded parameters are communicated and recovered on-demand before computations, and then they are immediately discarded afterwards. This approach ensures that FSDP only needs to materialize parameters from one unit at a time, which significantly reduces peak memory consumption

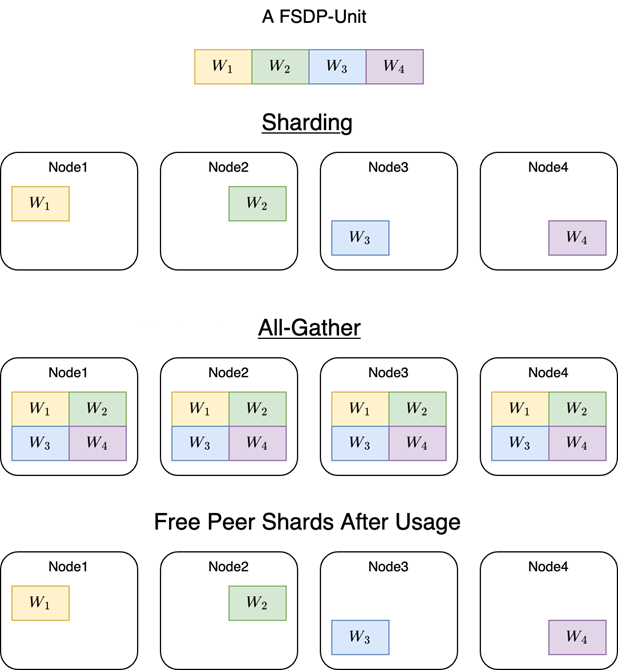
**FSDP Workflow**

Let us consider FSDP **unit1** that contains **[layer1,layer2]** to explain this process.

**Forward pass**:

1. Before forward computation enters **layer1**, FSDP collects the unsharded parameters for **layer1** and **layer2** by gathering shards from other peer ranks.
2. With the unsharded parameters, FSDP runs the local computation of those layers
3. Then frees the peer shards it just collected to reduce memory footprint

Therefore, during the entire forward pass, FSDP only needs to fully materialize one unit at a time, while all other units can stay sharded.

[](https://cdn-uploads.huggingface.co/production/uploads/660710b03ef451aa2bab8971/7zvopsLHueGXc0FmKGi28.png)

**Backward pass**:

1. Similarly, during the backward computation, FSDP **unit1** recovers the unsharded parameters for **layer1** and **layer2** before backward reaches **layer2**
2. When the autograd engine finishes the backward computation of these two layers, FSDP frees the peer shards and launches ReduceScatter to reduce and shard gradients.
3. Hence, after backward computation, each rank only keeps a shard of both parameters and gradients

While FSDP significantly optimizes memory usage by sharding parameters, it introduces some communication overhead due to the frequent need to gather and scatter parameters and gradients across GPUs. This overhead is a trade-off for the reduced memory footprint, and its impact can vary depending on the network bandwidth and latency between GPUs. Efficient implementation of the gather and scatter operations, along with optimizations such as overlapping communication with computation, can help mitigate this overhead to maintain high training throughput.

## ZeRO

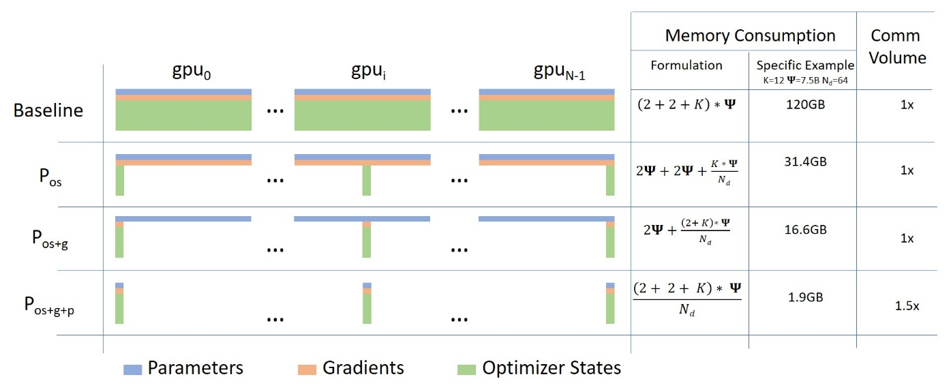
ZeRO is a memory-optimization strategy developed as part of **DeepSpeed**. It **reduces memory redundancy** in distributed training by partitioning model states (weights, gradients, and optimizer states) across multiple GPUs instead of replicating them.

**ZeRO Stages**

ZeRO is implemented in **three stages**, progressively reducing memory usage:

1. **ZeRO-1 (Optimizer State Sharding)**
   * Each GPU stores only a portion of the optimizer states instead of a full copy.
   * Reduces memory usage but still keeps full model weights on each GPU.
2. **ZeRO-2 (Optimizer + Gradient Sharding)**
   * Further shards gradients across GPUs, reducing redundancy even more.
   * Model weights are still fully replicated on each GPU.
3. **ZeRO-3 (Full Model Sharding)**
   * The most aggressive stage where **model weights, gradients, and optimizer states are fully sharded** across all GPUs.
   * This allows training models **much larger than a single GPU’s memory** but comes with increased communication overhead.

🔹 **ZeRO is part of DeepSpeed and is well-optimized for mixed precision training (e.g., ZeRO + FP16).**



**Reducing Activation Memory:**

* Managing Temporary buffers
* Managing fragmented Memory
* **FSDP** is PyTorch-native and simpler, but **ZeRO** offers more memory savings.
* **ZeRO-3** is the most memory-efficient but requires DeepSpeed.
* For models >50B parameters, **ZeRO + offload** is often the only viable option.

## Strategies

Begin by estimating how much vRAM is required to train your model. For models hosted on the 🤗 Hub, use our [Model Memory Calculator](https://huggingface.co/spaces/hf-accelerate/model-memory-usage), which gives you accurate calculations within a few percent margin.

**Parallelization strategy for a single Node / multi-GPU setup**

When training a model on a single node with multiple GPUs, your choice of parallelization strategy can significantly impact performance. Here’s a breakdown of your options:

### Case 1: Your model fits onto a single GPU

If your model can comfortably fit onto a single GPU, you have two primary options:

1. DDP - Distributed DataParallel
2. [Zero Redundancy Optimizer (ZeRO)](https://arxiv.org/abs/1910.02054) - depending on the situation and configuration used, this method may or may not be faster, however, it’s worth experimenting with it.

### Case 2: Your model doesn’t fit onto a single GPU:

If your model is too large for a single GPU, you have several alternatives to consider:

1. PipelineParallel (PP)
2. [ZeRO](https://arxiv.org/abs/1910.02054)
3. [TensorParallel](https://huggingface.co/docs/transformers/perf_train_gpu_many#tensor-parallelism) (TP)

With very fast inter-node connectivity (e.g., NVLINK or NVSwitch) all three strategies (PP, ZeRO, TP) should result in similar performance. However, without these, PP will be faster than TP or ZeRO. The degree of TP may also make a difference. It’s best to experiment with your specific setup to determine the most suitable strategy.

TP is almost always used within a single node. That is TP size <= GPUs per node.

### Case 3: Largest layer of your model does not fit onto a single GPU

1. If you are not using ZeRO, you have to use TensorParallel (TP), because PipelineParallel (PP) alone won’t be sufficient to accommodate the large layer.
2. If you are using ZeRO, additionally adopt techniques from the [Methods and tools for efficient training on a single GPU](https://huggingface.co/docs/transformers/perf_train_gpu_one).

### Parallelization strategy for a multi-Node / multi-GPU setup

* When you have fast inter-node connectivity (e.g., NVLINK or NVSwitch) consider using one of these options:
  1. ZeRO - as it requires close to no modifications to the model
  2. A combination of PipelineParallel (PP) with TensorParallel(TP) and DataParallel(DP) - this approach will result in fewer communications, but requires significant changes to the model
* When you have slow inter-node connectivity and still low on GPU memory:
  1. Employ a combination of DataParallel(DP) with PipelineParallel(PP), TensorParallel(TP), and ZeRO.

# RAG

## What is RAG?

LLMs, although capable of generating text that is both meaningful and grammatically correct, these LLMs suffer from a problem called hallucination. Hallucination in LLMs is the concept where the LLMs confidently generate wrong answers, that is they make up wrong answers in a way that makes us believe that it is true. This has been a major problem since the introduction of the LLMs. These hallucinations lead to incorrect and factually wrong answers. Hence Retrieval Augmented Generation was introduced.

In RAG, we take a list of documents/chunks of documents and encode these textual documents into a numerical representation called vector embeddings, where a single vector embedding represents a single chunk of document and stores them in a database called vector store. The models required for encoding these chunks into embeddings are called encoding models or bi-encoders. These encoders are trained on a large corpus of data, thus making them powerful enough to encode the chunks of documents in a single vector embedding representation.

## Explain Semantic Chunking

In order to abide by the context window of the LLM , we usually break text into smaller parts / pieces which is called chunking.

Different chunking methods:

* Fixed size chunking
* Recursive Chunking
* Document Specific Chunking
* Semantic Chunking
* Agentic Chunking

Semantic chunking involves taking the embeddings of every sentence in the document, comparing the similarity of all sentences with each other, and then grouping sentences with the most similar embeddings together. By focusing on the text’s meaning and context, Semantic Chunking significantly enhances the quality of retrieval. It’s a top-notch choice when maintaining the semantic integrity of the text is vital.

The hypothesis here is we can use embeddings of individual sentences to make more meaningful chunks. Basic idea is as follows :-

1. *Split the documents into sentences based on separators(.,?,!)*
2. *Index each sentence based on position.*
3. *Group: Choose how many sentences to be on either side. Add a buffer of sentences on either side of our selected sentence.*
4. *Calculate distance between group of sentences.*
5. *Merge groups based on similarity i.e. keep similar sentences together.*
6. *Split the sentences that are not similar.*

## Keyword-Based Retrieval

This retrieval type uses a keyword-based retriever, also known as a sparse retriever. An example of such a retriever is [BM25Retriever](https://docs.cloud.deepset.ai/docs/retriever#keyword-based-retrievers).

Sparse retrievers work with keywords, looking for words shared between the document and the query. They operate on a bag-of-words level and don’t consider the order of words or their contextual meanings, which means they may not capture semantic nuances as effectively as dense retrievers.

These retrievers don’t need any training and are fast and effective. They can work on any language and any domain.

## Vector-Based Retrieval

This retrieval type relies on vector-based, or dense, retrievers, such as [EmbeddingRetriever](https://docs.cloud.deepset.ai/docs/retriever#vector-based-retrievers). Dense retrievers use a model to transform both the documents and the query into numerical vectors (embeddings). Then, they compare both embeddings and, based on that, fetch the documents most similar to the query.

Dense retrievers are very good at capturing nuances in queries and documents, recognizing similarities that go beyond keyword matching. They can recognize contextual and semantic information about words and their relationships within a sentence.

Unlike sparse retrievers, dense retrievers need to be trained. This means they perform best on the domain and language they were trained on. They’re also more computationally expensive than keyword-based retrievers.

## Hybrid Retrieval

Sparse retrievers are fast and can quickly reduce the number of candidate documents. Dense retrievers are better at capturing semantic nuances, thus improving the relevance of search results.

For example, when searching for product IDs, keyword search is best. When given the query “P12642”, a sparse retriever would fetch “Miura climbing shoes” as a result. Dense retrievers would be thrown off by such a query since they can return results with a similar product ID.

On the other hand, a query like “What are EVs?” would be easier for vector-based retrievers. They would retrieve results like “Electric cars are..”, while sparse retrievers would look for the exact keyword match.

Combining both retrieval methods in one system makes it more robust to different kinds of queries and documents.

Once the retrievers fetch the most relevant documents, you can use a combination strategy to produce the final ranking and return the top documents as search results.

A good use case for hybrid retrieval is when your documents are from a niche domain, and it’s unlikely the model was trained on it. Hybrid retrieval saves you the time and money you’d need to train or fine-tune a model and it’s a good trade-off between speed and accuracy.

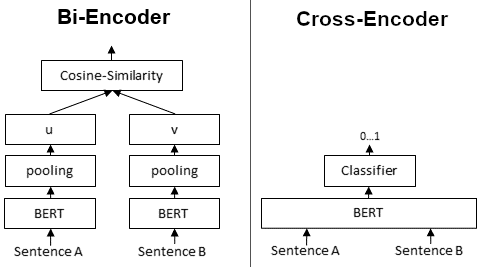
## Cross-Encoders

Cross-Encoders are used for sentence pair scoring and sentence pair classification tasks.

**Bi-Encoder vs. Cross-Encoder**

First, it is important to understand the difference between Bi- and Cross-Encoder.

**Bi-Encoders** produce for a given sentence a sentence embedding. We pass to a BERT independently the sentences A and B, which result in the sentence embeddings u and v. These sentence embedding can then be compared using cosine similarity:



In contrast, for a **Cross-Encoder**, we pass both sentences simultaneously to the Transformer network. It produces then an output value between 0 and 1 indicating the similarity of the input sentence pair:

A **Cross-Encoder does not produce a sentence embedding**. Also, we are not able to pass individual sentences to a Cross-Encoder.

As detailed in our [paper](https://arxiv.org/abs/1908.10084), Cross-Encoder achieve better performances than Bi-Encoders. However, for many application they are not practical as they do not produce embeddings we could e.g. index or efficiently compare using cosine similarity.

**When to use Cross- / Bi-Encoders?**

Cross-Encoders can be used whenever you have a pre-defined set of sentence pairs you want to score. For example, you have 100 sentence pairs and you want to get similarity scores for these 100 pairs.

Bi-Encoders are used whenever you need a sentence embedding in a vector space for efficient comparison. Applications are for example Information Retrieval / Semantic Search or Clustering. Cross-Encoders would be the wrong choice for these application: Clustering 10,000 sentence with CrossEncoders would require computing similarity scores for about 50 Million sentence combinations, which takes about 65 hours. With a Bi-Encoder, you compute the embedding for each sentence, which takes only 5 seconds. You can then perform the clustering.

## What is Reciprocal Rank Fusion?

Reciprocal Rank Fusion is a rank aggregation method that combines rankings from multiple sources into a single, unified ranking. In the context of RAG, these sources typically use different retrieval models or approaches.

**The RRF Formula**

The core of RRF is captured in its formula:

RRF(d) = Σ(r ∈ R) 1 / (k + r(d))

*Where:  
- d is a document  
- R is the set of rankers (retrievers)  
- k is a constant (typically 60)  
- r(d) is the rank of document d in ranker r*

**How RRF Works in RAG**

Let’s break down the process of using RRF in a RAG system:

1. **User Query**: The process begins when a user inputs a question or query.

2. **Multiple Retrievers**: The query is sent to multiple retrievers. These could be different retrieval models (e.g., dense, sparse, hybrid).

3. **Individual Rankings**: Each retriever produces its own ranking of relevant documents.

4. **RRF Fusion**: The rankings from all retrievers are combined using the RRF formula.

5. **Final Ranking**: A unified ranking is produced based on the RRF scores.

6. **Generation**: The generative model uses the top-ranked documents to produce the final answer.

**Mathematical Intuition Behind RRF**

Understanding the mathematical intuition behind RRF helps explain why it’s effective:

**1. Reciprocal Ranking**

Using 1/(rank + k), RRF gives more weight to higher ranks (lower rank numbers). This ensures that documents ranked highly by multiple retrievers are favoured in the final ranking.

**2. Diminishing Returns**

The contribution to the score decreases non-linearly as rank increases. This model shows the intuition that the difference in relevance between ranks 1 and 2 is likely larger than between ranks 100 and 101.

**3. Rank Aggregation**

By summing the reciprocal ranks across all retrievers, RRF effectively combines evidence from multiple sources. This makes the final ranking more robust and less susceptible to the quirks or biases of any single retriever.

**4. Normalization**

The constant k acts as a smoothing factor. It prevents any single retriever from dominating the results and helps handle ties more gracefully, especially among lower-ranked items.

**The Mystery of k = 60**

One aspect of RRF that often raises questions is the choice of k = 60. While this value isn’t set in stone, it’s commonly used due to several factors:

**1. Empirical Performance**

Studies have shown that k = 60 performs well across various datasets and retrieval tasks.

**2. Balancing Influence**

It provides a good balance between the influence of top-ranked and lower-ranked items. For example:  
- For rank 1: 1/(1+60) ≈ 0.0164  
- For rank 10: 1/(10+60) ≈ 0.0143  
- For rank 100: 1/(100+60) ≈ 0.00625

**3. Effective Tie-Breaking**

k = 60 helps break ties effectively, especially for lower-ranked items where small differences in the original rankings might not be significant.

**4. Robustness**

This value has shown to be robust across different types of retrieval systems and data distributions.

It’s worth noting that while k = 60 is common, the optimal value can vary depending on the specific application and data characteristics. Some systems may benefit from tuning this parameter.

# Similarity Search

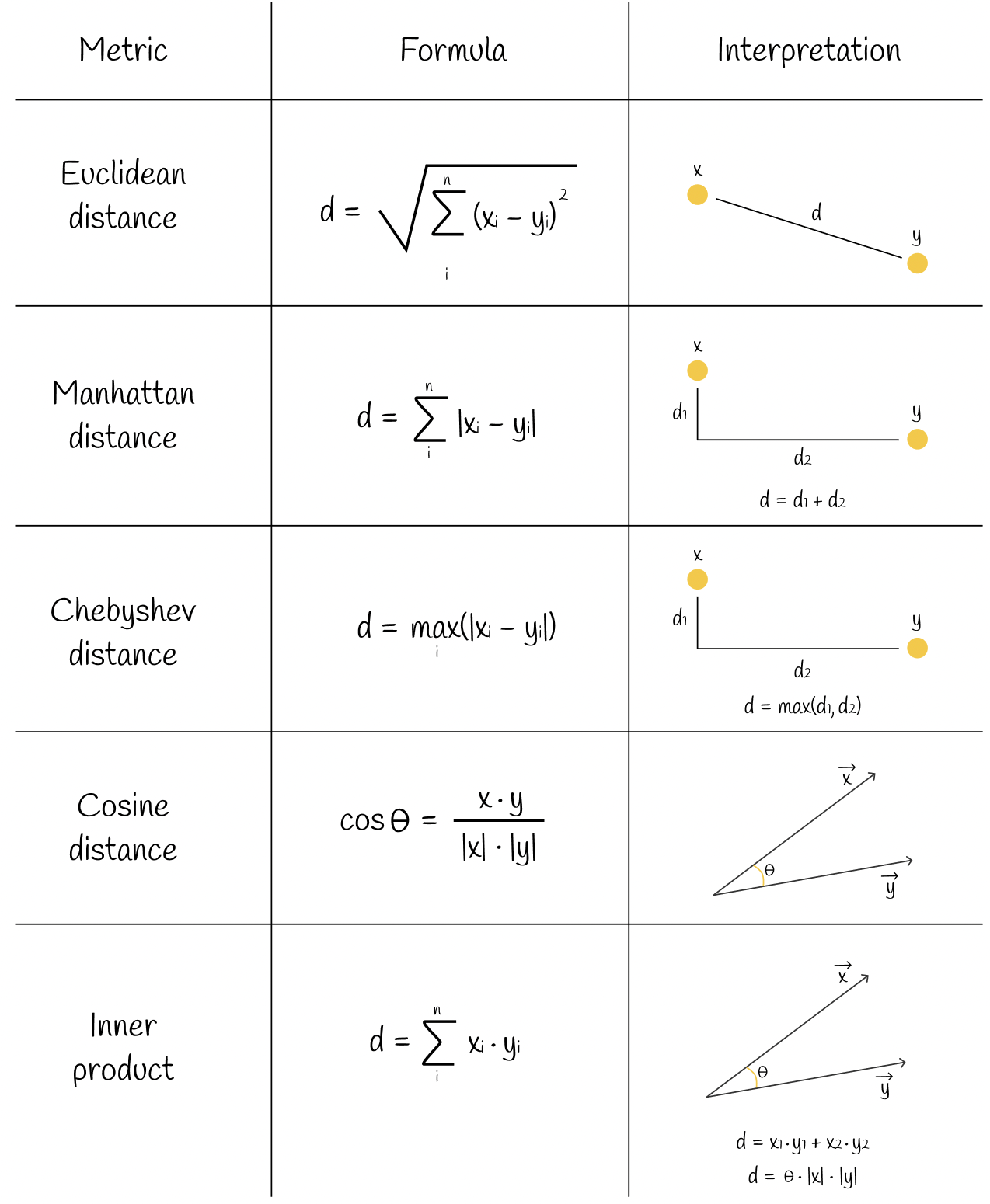
## Shingling

**Shingling** is the process of collecting *k*-grams on given texts. ***k*-gram** is a group of *k* sequential tokens. Depending on the context, tokens can be words or symbols. The ultimate goal of shingling is by using collected *k*-grams to encode each document. We will be using one-hot encoding for this. Nevertheless, other encoding methods can also be applied.

Firstly, unique *k*-grams for each document are collected. Secondly, to encode each document, a vocabulary is needed which represents a set of unique *k*-grams in all documents. Then for each document, a vector of zeros with the length equal to the size of the vocabulary is created. For every appearing k-gram in the document, its position in the vocabulary is identified and a *"1"* is placed at the respective position of the document vector. Even if the same *k*-gram appears several times in a document, it does not matter: the value in the vector will always be 1.

## kNN

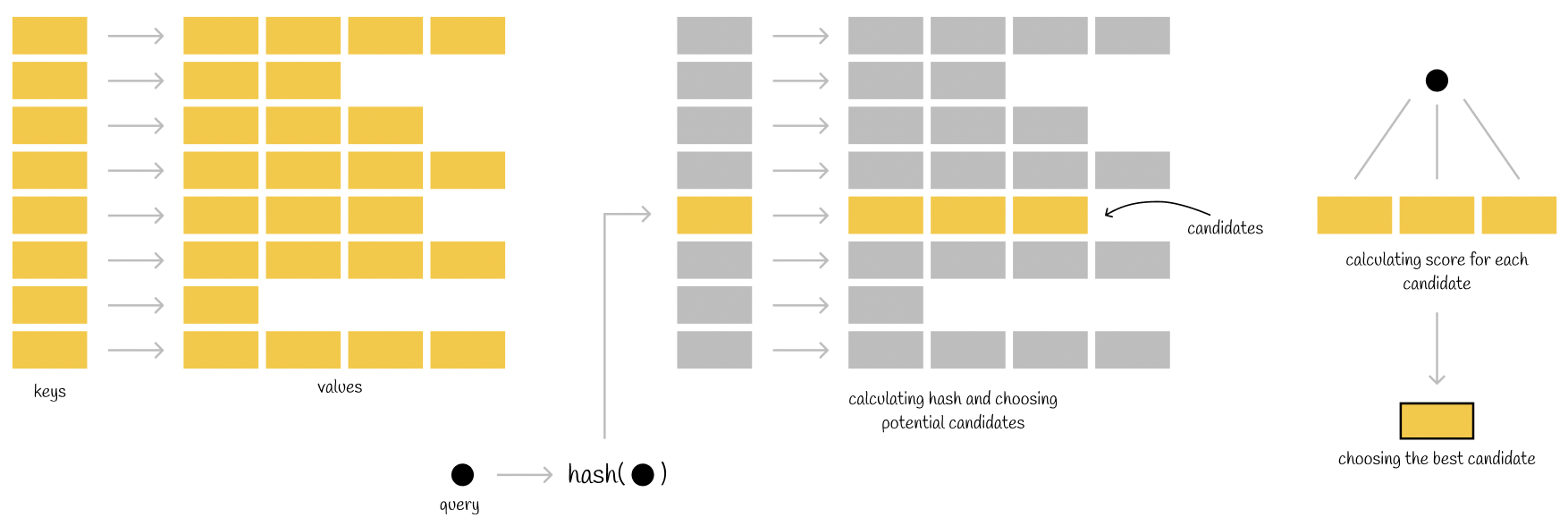
**kNN** is the simplest and the most naive algorithm for similarity search. Consider a dataset of vectors and a new query vector *Q*. We would like to find the top *k* dataset vectors which are the most similar to *Q*. The first aspect to think about is how to measure a similarity (distance) between two vectors. In fact, there are several similarity metrics to do it. Some of them are illustrated in the figure below.



## Inverted File Index

**Inverted index** (also referred to as a **postings list**, **postings file**, or **inverted file**) is a database index storing a mapping from content, such as words or numbers, to its locations in a table, or in a document or a set of documents.

When performing a query, the hash function of the query is computed and mapped values from the hash table are taken. Each of these mapped values contains its own set of potential candidates which then are fully checked on a condition to be the nearest neighbour for the query. By doing so, the search scope of all database vectors is reduced.



There are different implementations of this index depending on how hash functions are computed. The implementation we are going to look at is the one that uses **Voronoi diagrams** (or **Dirichlet tessellation**).

**Training**

The idea of the algorithm is to create several non-intersecting regions to which each dataset point will belong. Each region has its own centroid which points to the center of that region.

Sometimes **Voronoi regions** are referred to as **cells** or **partitions**.



The main property of Voronoi diagrams is that the distance from a centroid to any point of its region is less than the distance from that point to another centroid.

## Faiss

FAISS (Facebook AI Similarity Search) is a library that helps in searching for vectors in high-dimensional spaces efficiently. It is widely used for tasks involving nearest neighbor search and allows for searching in massive datasets by optimizing both memory and search time. The key benefit of FAISS is that it provides a variety of indexing strategies, each suited to different data sizes and search precision requirements.

**Key Index Types in FAISS**

FAISS provides multiple indexing options. Below are three commonly used types:

**1. Flat Index (IndexFlat)**

* **Overview**: IndexFlat is the most basic FAISS index type, performing exact nearest neighbor search.
* **Characteristics**: Simple and easy to use, but not efficient for large datasets due to its linear search…
* **Application**: Best suited for small datasets where precision is a top priority.

**2. Inverted File Index (IndexIVF)**

* **Overview**: The Inverted File Index (IVF) divides the data space into clusters, and searches are performed within these clusters.
* **Characteristics**: Faster than Flat Index, making it ideal for medium-sized datasets. IndexIVF trades off some accuracy for higher speed.
* **Application**: Recommended when working with moderately large datasets that require fast search speeds without sacrificing too much accuracy.

**3. Product Quantization Index (IndexPQ)**

* **Overview**: Product Quantization (PQ) further compresses the data, allowing it to operate efficiently over massive datasets by approximating the search.

**Algorithm**

* **Vector Quantization**: PQ compresses high-dimensional vectors by dividing each vector into M*M* subvectors of lower dimensionality each of which is quantized separately. This reduces the dimensionality and makes it more efficient to store and search for vectors.
* **Codebooks**: Each sub-vector is treated as an individual vector and k-means clustering is used to generate the codebook for each sub-vector. The quantizer assigns each subvector to the closest centroid in its corresponding codebook. The combination of the indices of these centroids forms the compressed representation of the vector. The original vector x is now represented by the indices of the centroids in each codebook, which requires significantly less memory than storing the full high-dimensional vector. This compressed representation is much smaller, as you are only storing the index of the centroid for each sub-vector instead of the full sub-vector itself.
* **Search**: When querying, you also divide the query vector into the same number of sub-vectors and find the closest centroids from the codebooks. Once quantized, the search process involves comparing the quantized representations of vectors (which are indices) rather than the original high-dimensional vectors, making it faster.
* **Accuracy**: PQ introduces some approximation errors due to quantization, but can be tuned to find a good balance between memory and accuracy.  
  **Use case**: Useful for very large datasets with constraints on memory usage.

1. **Hierarchical Navigable Small World**

HNSW algorithm creates a multi-level graph to store vectors, enabling fast nearest neighbor searches by traversing connected nodes.

**Algorithm**

* **Graph based approach**: HNSW builds a multi-layered graph where each node represents a vector and edges are formed based on proximity. Higher layers contain fewer nodes and connect vectors that are further apart, while lower layers contain more nodes and connect vectors that are closer together.
* **Search Process**: The algorithm starts from the top layer of the graph, navigating down to lower layers progressively narrowing the search area. This allows for fast nearest neighbor retrieval.

**Accuracy**: HNSW is quite accurate and scales well for large datasets.  
**Use case**: Ideal for high-dimensional data and large-scale applications where fast retrieval is important.

neighbours = 16 # Number of neighbors  
index = faiss.IndexHNSWFlat(dimension, neighbours)  
index.add(vectors) # Add the vectors to the index  
distances, indices = index.search(query, k)

1. **Inverted File with Product Quantization**

First the dataset is partitioned into clusters (as in IVF) and then each cluster is compressed using PQ.

**Algorithm**

* **Cluster the Data (Inverted File)**: The result is an Inverted File Index, where each cluster corresponds to a bucket containing a set of vectors. This means that when querying, you only need to search a subset of the clusters rather than the entire dataset.
* **Apply Product Quantization (PQ)**: Each vector is divided into sub-vectors (or blocks). These sub-vectors are independently quantized using separate codebooks (a set of centroids). Each codebook contains a set of centroids representing the possible values for that sub-vector. The vectors are then represented by indices pointing to the nearest centroid in the codebook for each sub-vector.
* **Index Construction**: The centroids of each cluster are stored, and each vector in the cluster is represented by the indices of the centroids of its sub-vectors.
* **Search**: The query vector is compared to the centroids of the clusters and the closest cluster/s are identified. The query vector is also divided into sub-vectors, and each sub-vector is quantized using the same codebooks that were used during indexing. The quantized representations of the query vector and the vectors in the clusters are compared to find the nearest neighbors.

**Accuracy**: The accuracy improves with a higher number of clusters and centroids per sub-vector, but this comes at the cost of increased memory usage and longer search times. Therefore, these parameters must be fine-tuned to strike a balance between speed and accuracy.  
**Use case**: Ideal for large-scale high-dimensional vector search tasks where both speed and memory efficiency are essential.

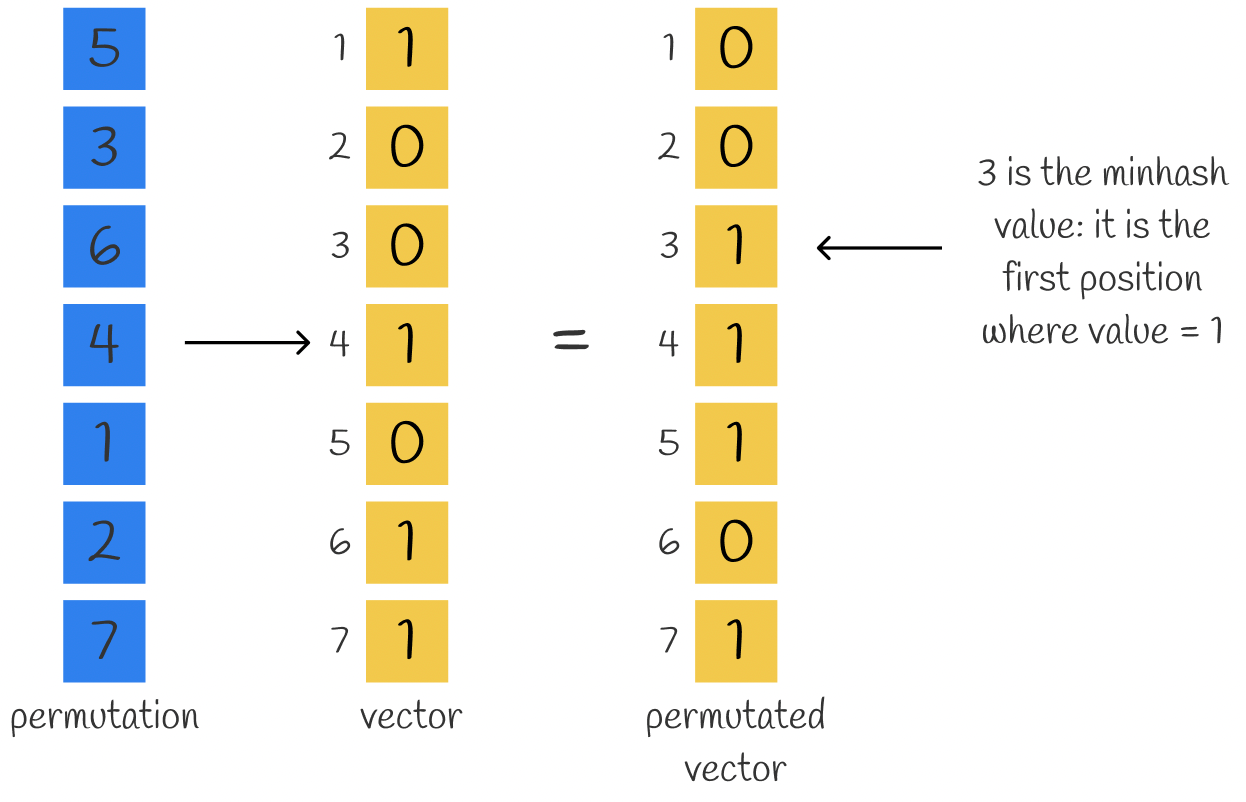
## MinHashing

At this stage, initial texts have been vectorised. The similarity of vectors can be compared via **Jaccard index**. Remember that Jaccard index of two sets is defined as the number of common elements in both sets divided by the length of all the elements.

If a pair of encoded vectors is taken, the intersection in the formula for Jaccard index is the number of rows that both contain 1 (i.e. *k*-gram appears in both vectors) and the union is the number of rows with at least one 1 (*k*-gram is presented at least in one of the vectors).

The current problem right now is the sparsity of encoded vectors. Computing a similarity score between two one-hot encoded vectors would take a lot of time. Transforming them to a dense format would make it more efficient to operate on them later. Ultimately, the goal is to design such a function that will transform these vectors to a smaller dimension preserving the information about their similarity. The method that constructs such a function is called MinHashing.

**MinHashing** is a hash function that permutes the components of an input vector and then returns the first index where the permutated vector component equals 1.



For getting a dense representation of a vector consisting of *n* numbers, *n* minhash functions can be used to obtain *n* minhash values which form a **signature**.

It may not sound obvious at first but several minhash values can be used to approximate Jaccard similarity between vectors. In fact, the more minhash values are used, the more accurate the approximation is.



This is just a useful observation. It turns out that there is a whole theorem behind the scenes. Let us understand why Jaccard index can be calculated by using signatures.

## Locality Sensitive Hashing

At the current moment, we can transform raw texts into dense signatures of equal length preserving the information about similarity. Nevertheless, in practice, such dense signatures still usually have high dimensions and it would be inefficient to directly compare them.

Consider *n = 10⁶* documents with their signatures of length 100. Assuming that a single number of a signature requires 4 bytes to store, then the whole signature would require 400 bytes. For storing *n = 10⁶* documents, 400 MB of space is needed which is doable in reality. But comparing each document with each other in a brute-force manner would require approximately 5 \* 10¹¹ comparisons which is too much, especially when *n* is even larger.

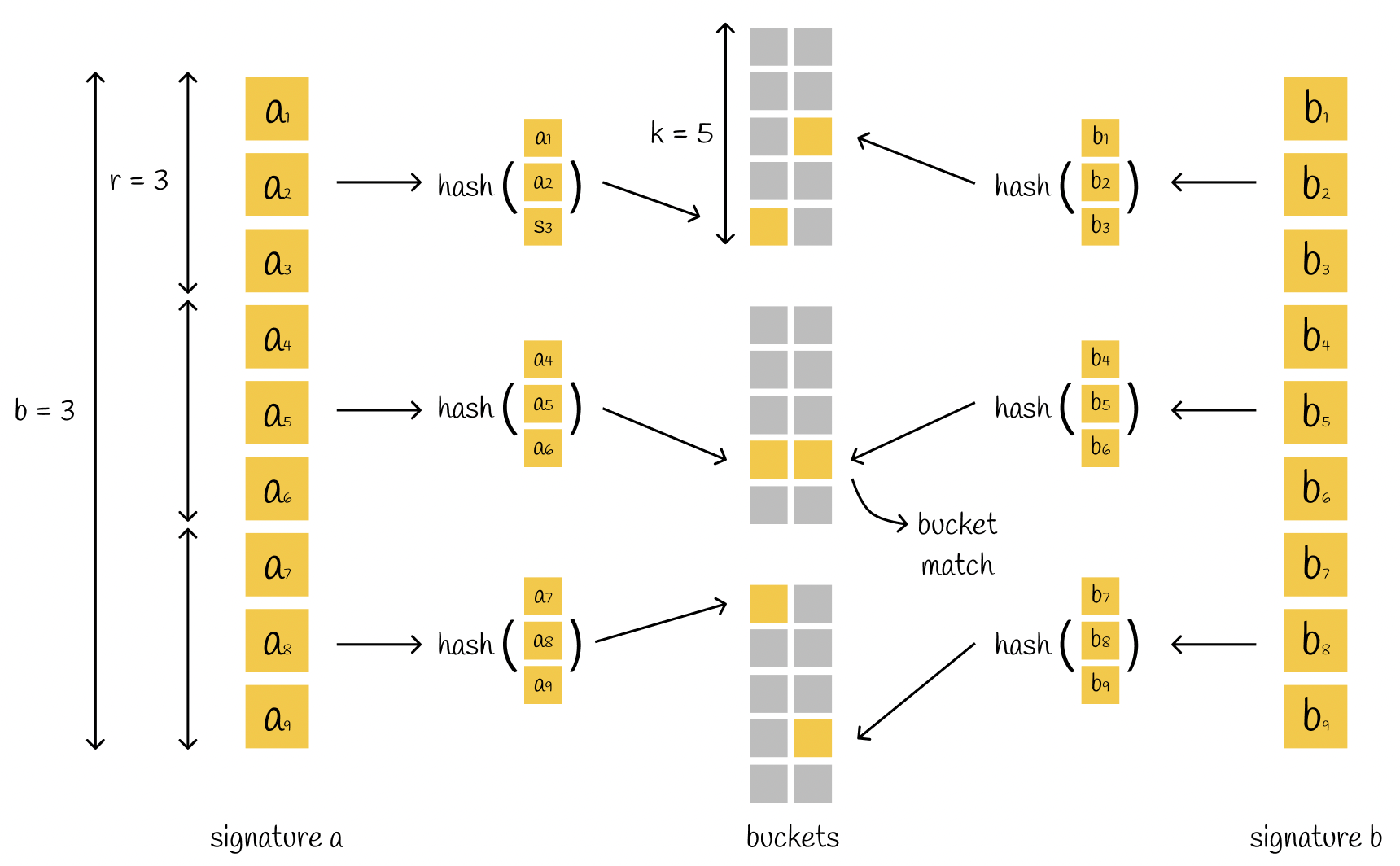
Изображение выглядит как текст, Шрифт, линия, белый

Автоматически созданное описание

To avoid the problem, it is possible to build a hash table to accelerate search performance but even if two signatures are very similar and differ only in 1 position, they are still likely to have a different hash (because vector remainders are likely to be different). However, we normally want them to fall into the same bucket. This is where LSH comes to the rescue.

**LSH** mechanism builds a hash table consisting of several parts which puts a pair of signatures into the same bucket if they have at least one corresponding part.

LSH takes a signature matrix and horizontally divides it into equal *b* parts called **bands** each containing *r* **rows**. Instead of plugging the whole signature into a single hash function, the signature is divided by *b* parts and each subsignature is processed independently by a hash function. As a consequence, each of the subsignatures falls into separate buckets.

Example of using LSH. Two signatures of length 9 are divided into b = 3 bands each containing r = 3 rows. Each subvector is hashed into one of k possible buckets. Since there is a match in the second band (both subvectors have the same hash value), we consider a pair of these signatures as candidates to be the nearest neighbours.

If there is at least one collision between corresponding subvectors of two different signatures, the signatures are considered candidates. As we can see, this condition is more flexible since for considering vectors as candidates they do not need to be absolutely equal. Nevertheless, this increases the number of false positives: a pair of different signatures can have a single corresponding part but in overall be completely different. Depending on the problem, it is always better to optimize parameters *b*, *r* and *k*.

# Sampling

## Top-*k* and Top-*p* Sampling

When you opt for sampling rather than greedy decoding, you’ll have an additional two hyperparameters with which to influence a model’s output: top-*k* and top-*p* sampling values.

The Top-*k* sampling value is an integer that ranges from 1 to 100 (with a default value of 50) that specifies that the tokens sampled by the model should be those with the highest probabilities until the set value is reached. To better illustrate how top-*k* sampling works, let’s use a brief example.

Let’s say you have the sentence “*I went to meet a friend…”.*Now, out of the vast number of ways to end this sentence, let’s look at the five examples provided below – each beginning with a different token:

1. *at the library*
2. *for a brief work lunch*
3. *to discuss our shared homework assignment*
4. *in the centre of the city*
5. *on the other side of town*

From there, let’s assign each of the initial tokens for each sentence a probability.

Now, if we set the top-*k* sampling value to 2, it will only add *at* and *for* to the sampling sunset from which it selects an output token. Setting it to 5, by contrast, would mean all options could be considered. So, in short, the higher the *k*-sampling value, the greater the potential variety in output.

Alternatively, the Top-*p* sampling value is a decimal number in the range of 0.0 to 1.0, that configures a model to sample the tokens with the highest probabilities until the sum of those probabilities reaches the set value.

Returning to the above table, if the top-*p* sampling value is set to 0.7, once again, at and for will be the only tokens included in the subset, as their combined probabilities are 0.55 (0.30 + 0.25). As *at*, *for*, and *to* have a cumulative probability of 0.77 (0.30 + 0.25 + 0.22), this breaches the set threshold of 0.7 and *to* is excluded from the subset as a result. As with top-k sampling, the higher the value, the more varied the output.

Lastly, in the event both sampling values are set, top-*k* takes precedence – with all probabilities outside the set threshold set to 0.

## Temperature

Temperature performs a similar function to the above-described top-*k* and top-*p* sampling values, providing a way to vary the range of possible output tokens and influence the model’s “creativity”. It is represented by a decimal number between 0.0 (which is effectively the same as greedy decoding, whereby the token with the highest probability is added to the output) and 2.0 (maximum creativity).

The temperature hyperparameter influences output by changing the shape of the token probability distribution. For low temperatures, the difference between probabilities is amplified, so tokens with higher probabilities become even more likely to be output compared to less-likely tokens. Consequently, you should set a lower temperature value when you want your model to generate more predictable or dependable responses.

In contrast, high temperatures cause token probabilities to converge closer to one another, so less likely or unusual tokens receive an increased chance of being output. In light of this, you should set a higher temperature value when you want to increase the randomness and creativity of responses.

## Stop Sequences

Aside from the max output tokens hyperparameter, the other way to influence the length of an LLM’s response is by specifying a stop sequence, i.e., a string composed of one or more characters, which automatically stops a model’s output. A common example of a stop sequence is a period (full stop).

Alternatively, you can specify the end of a sequence by setting a stop token limit – which is an integer value rather than a string. For instance,  if the stop token limit is set to 1, the generated output will stop at a sentence. If it’s set to 2, on the other hand, the response will be constrained to a paragraph.

A reason you might set a stop sequence or stop token limit is that, similar to the max output tokens parameter, you have greater control over inference, which may be a concern if budget is a consideration.

## Frequency and Presence Penalties

A frequency, or repetition, penalty, which is a decimal between -2.0 and 2.0, is a an LLM hyperparameter that indicates to a model that it should refrain from using the same tokens too often. It works by lowering the probabilities of tokens that were recently added to a response, so they’re less likely to be repeated to produce a more diverse output.  
  
The presence penalty works in a similar way but is only applied to tokens that have been used at least once – while the frequency is applied *proportionally* to how often a specific token has been used. In other words, the frequency penalty affects output by preventing repetition, while the presence penalty encourages a wider assortment of tokens.

# Inference

## Compute-Bound, Memory-Bound, and Overhead-Bound Regimes

When optimizing machine learning and deep learning workloads (or any computational tasks), performance bottlenecks usually fall into **three categories**:

**1. Compute-Bound Regime (CPU/GPU Limited)**

* The system spends most of its time performing computations.
* The bottleneck is **processing power (FLOPs, tensor operations, etc.)**, not memory or communication.
* Common in **dense matrix multiplications, deep learning forward/backward passes**.

**2. Memory-Bound Regime (Bandwidth Limited)**

* The system spends most of its time **waiting for data to be fetched** from memory (VRAM, RAM).
* Computation itself is **not the bottleneck**, but **memory bandwidth** is.
* Common in **large models with high memory access needs (LLMs, CNNs, attention layers with large KV Cache)**.

**3. Overhead-Bound Regime (Latency Limited)**

* Performance is limited by **non-computational overheads**, such as:
  + Kernel launch latency.
  + CPU-GPU synchronization delays.
  + Framework inefficiencies (Python GIL, scheduling delays).
  + Communication bottlenecks in distributed training (e.g., **network latency** in multi-GPU setups).

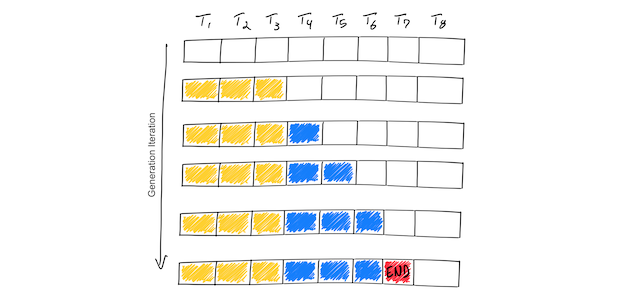
| **Regime** | **Bottleneck** | **Example Workloads** | **Optimizations** |
| --- | --- | --- | --- |
| **Compute-Bound** | GPU/CPU FLOPs | Matrix multiplications, dense neural networks, Transformer training | Use mixed precision (FP16/BF16), optimize kernels, use Tensor Cores |
| **Memory-Bound** | Memory bandwidth | Large LLM inference (KV Cache), CNN activations | Operator Fusion |
| **Overhead-Bound** | Communication, latency | Small batch inference, multi-GPU training | Tracing, Operator Fusion |

Link: <https://horace.io/brrr_intro.html>

## LLM inference

For each request:

1. You start with a sequence of tokens (called the "prefix" or "prompt").
2. The LLM produces a sequence of completion tokens, stopping only after producing a stop token or reaching a maximum sequence length.



This toy example shows a hypothetical model which supports a maximum sequence length of 8 tokens (T1, T2, …, T8). Starting from the prompt tokens (yellow), the iterative process generates a single token at a time (blue). Once the model generates an end-of-sequence token (red), the generation loop stops. This example shows a batch of only one input sequence, so the batch size is 1.

1. The initial ingestion (“prefill”) of the prompt "What is the capital of California: " takes about as much time as the generation of each subsequent token. This is because the [prefill phase](https://github.com/huggingface/text-generation-inference/tree/f59fb8b630844c2ad2cd80e689202de89d45c37e/router#prefill-decode-and-past-key-values) pre-computes [some inputs](https://kipp.ly/transformer-inference-arithmetic/#kv-cache) of the attention mechanism that remain constant over the lifetime of the generation. This prefill phase efficiently uses the GPU’s parallel compute because these inputs can be computed independently of each other.
2. LLM inference is [memory-IO bound](https://en.wikipedia.org/wiki/Memory_bandwidth), not compute bound. In other words, it currently takes more time to load 1MB of data to the GPU’s compute cores than it does for those compute cores to perform LLM computations on 1MB of data. This means that LLM inference throughput *is largely determined by how large a batch you can fit into high-bandwidth GPU memory*.
3. The amount of GPU memory consumed scales with the base model size + the length of the token sequence. In [*Numbers every LLM developer should know*](https://github.com/ray-project/llm-numbers#1-mb-gpu-memory-required-for-1-token-of-output-with-a-13b-parameter-model), it’s estimated that a 13B parameter model consumes nearly 1MB of state for each token in a sequence. On a higher-end A100 GPU with 40GB RAM, back-of-the-envelope math suggests that since 14 GB are left after storing the 26GB of model parameters, ~14k tokens can be held in memory at once. This may seem high but is actually quite limiting; if we limit our sequence lengths to 512, we can process at most ~28 sequences in a batch. The problem is worse for higher sequence lengths; a sequence length of 2048 means our batch size is limited to 7 sequences. Note that this is an upper bound since it doesn’t leave room for storing intermediate computations.

| **Feature** | **Classification LLM** | **Generation LLM** |
| --- | --- | --- |
| **Computation Type** | Single forward pass | Autoregressive (iterative) |
| **KV Cache Needed?** | ❌ No | ✅ Yes |
| **Attention Mask** | Square | Triangle |
| **Forward passes** | 1 | 1 per token |
| **Multiplication** | GeMM | GeMV |
| **Batching Efficiency** | ✅ High | ⚠ Difficult (due to varying output lengths) |
| **Latency** | ✅ Low (fixed time) | ⚠ High (depends on output length) |
| **Regime** | Computation-bound | Memory-bound |
| **Optimizations** | Batch inference, quantization, distillation | KV Cache optimization, speculative decoding, continuous batching |

## Metrics for LLM Serving

1. **Time To First Token (TTFT)**: How quickly users start seeing the model's output after entering their query. Low waiting times for a response are essential in real-time interactions, but less important in offline workloads. This metric is driven by the time required to process the prompt and then generate the first output token.
2. **Time Per Output Token (TPOT)**: Time to generate an output token for *each* user that is querying our system. This metric corresponds with how each user will perceive the "speed" of the model. For example, a TPOT of 100 milliseconds/tok would be 10 tokens per second per user, or ~450 words per minute, which is faster than a typical person can read.
3. **Latency**: The overall time it takes for the model to generate the full response for a user. Overall response latency can be calculated using the previous two metrics: latency = *(TTFT)* + *(TPOT)* \* (the number of tokens to be generated).
4. **Throughput**: The number of output **tokens per second** an inference server can generate across all users and requests.
5. **Requests Per Second (RPS):** This is the average number of requests that can be successfully completed by the system in a 1-second period.

Our goal? The fastest time to first token, the highest throughput, and the quickest time per output token. In other words, we want our models to generate text as fast as possible for as many users as we can support.

Notably, there is a tradeoff between throughput and time per output token: if we process 16 user queries concurrently, we'll have *higher* throughput compared to running the queries sequentially, but we'll take *longer* to generate output tokens for each user.

If you have overall inference latency targets, here are some useful heuristics for evaluating models:

* **Output length dominates overall response latency:** For average latency, you can usually just take your expected/max output token length and multiply it by an overall average time per output token for the model.
* **Input length is not significant for performance but important for hardware requirements:** The addition of 512 *input* tokens increases latency less than the production of 8 additional *output* tokens in the MPT models. However, the need to support long inputs can make models harder to serve. For example, we recommend using the A100-80GB (or newer) to serve MPT-7B with its maximum context length of 2048 tokens.
* **Overall latency scales sub-linearly with model size:** On the same hardware, larger models are slower, but the speed ratio won't necessarily match the parameter count ratio. MPT-30B latency is ~2.5x that of MPT-7B latency. Llama2-70B latency is ~2x that of Llama2-13B latency.

## LLM Optimization Techniques

Optimizing LLM inference requires reducing latency, memory usage, and computational overhead. Here are key techniques:

**1. Efficient Computation Techniques**

* **Kernel Fusion** – Combines multiple GPU operations into one to reduce memory bottlenecks (e.g., fused attention, fused layer normalization).
* **Quantization** – Reduces precision (e.g., FP16, INT8, 4-bit quantization) to lower memory usage and speed up computation.
* **Tensor Parallelism** – Splits large matrix multiplications across multiple GPUs.
* **Speculative Decoding** – Uses a smaller model to predict multiple tokens in advance, reducing inference time.

**2. Memory Optimization**

* **Paged Attention (vLLM)** – Efficient memory management for KV cache, avoiding redundant memory copies.
* **FlashAttention** – Optimized attention mechanism that minimizes memory reads/writes, improving speed.
* **KV Cache Optimization** – Reuses previously computed key-value pairs to speed up autoregressive decoding.

**3. Model Optimization**

* **LoRA (Low-Rank Adaptation)** – Fine-tunes only a small subset of model parameters, reducing computation.
* **Mixture of Experts (MoE)** – Activates only a subset of model parameters per input, saving computation.
* **Pruning** – Removes less important model weights to reduce size and inference cost.

**4. Parallelism & Distributed Computing**

* **Pipeline Parallelism** – Splits model layers across multiple GPUs to balance computation.
* **Sequence Parallelism** – Divides long input sequences across multiple GPUs for better efficiency.
* **Triton / TensorRT-LLM** – Uses optimized GPU execution engines for high-performance inference.

**5. Engineering Optimizations**

* **Batching Requests** – Processes multiple queries at once to maximize GPU utilization.
* **Token Merging (DeepSpeed-Mii)** – Dynamically merges redundant tokens for faster processing.
* **Continuous Batching (vLLM)** – Dynamically merges new and ongoing requests for improved throughput.

## Kernel Fusion

**Kernel fusion** is an optimization technique that combines multiple GPU operations (kernels) into a single, more efficient kernel to reduce memory access overhead and improve performance.

**How it's used in LLM optimization:**

1. **Reduces memory bottlenecks** – Instead of writing intermediate results to memory and reading them back, fused kernels keep computations in GPU registers.
2. **Improves throughput** – Fewer kernel launches mean lower scheduling overhead.
3. **Optimizes common operations** – In LLMs, fusion is used for matrix multiplications, activation functions, and layer norm computations.
4. **Used in frameworks** – Libraries like **FlashAttention**, **TensorRT-LLM**, and **vLLM** leverage fusion to speed up inference.

This helps LLMs run faster and use less memory, making them more efficient for real-time applications.

## Quantization

Quantization is a technique to reduce the computational and memory costs of running inference by representing the weights and activations with low-precision data types like 8-bit integer (int8) instead of the usual 32-bit floating point (float32).

Reducing the number of bits means the resulting model requires less memory storage, consumes less energy (in theory), and operations like matrix multiplication can be performed much faster with integer arithmetic. It also allows to run models on embedded devices, which sometimes only support integer data types.

Let’s consider a float x in [a, b], then we can write the following quantization scheme, also called the *affine quantization scheme*:

x = S \* (x\_q - Z)

where:

* x\_q is the quantized int8 value associated to x
* S and Z are the quantization parameters
  + S is the scale, and is a positive float32
  + Z is called the zero-point, it is the int8 value corresponding to the value 0 in the float32 realm. This is important to be able to represent exactly the value 0 because it is used everywhere throughout machine learning models.

The quantized value x\_q of x in [a, b] can be computed as follows:

x\_q = round(x/S + Z)

### Per-tensor and per-channel quantization

Depending on the accuracy / latency trade-off you are targetting you can play with the granularity of the quantization parameters:

* Quantization parameters can be computed on a *per-tensor* basis, meaning that one pair of (S, Z) will be used per tensor.
* Quantization parameters can be computed on a *per-channel* basis, meaning that it is possible to store a pair of (S, Z) per element along one of the dimensions of a tensor. For example for a tensor of shape [N, C, H, W], having *per-channel* quantization parameters for the second dimension would result in having C pairs of (S, Z). While this can give a better accuracy, it requires more memory.

### Quantization Methods

1. Weights-only (wNa16 – N bits weights, 16 bits activations)
   1. Reduces memory footprint
   2. Speedups memory-bound
   3. Keeps activations in higher precision to preserve accuracy.

#### 1.1 GPT-Q

A post-training quantization technique where each row of the weight matrix is quantized independently to find a version of the weights that minimizes error. These weights are quantized to int4, stored as int32 (int4 x 8) and dequantized (restored) to fp16 on the fly during inference. This can save memory by almost 4x because the int4 weights are often dequantized in a fused kernel. You can also expect a substantial speedup in inference due to lower bandwidth requirements for lower bitwidth.

→ W4A16

→ 3,25x (1,5x–2x actually)

→ decoding only

→ Code and CUDA kernels published

→ Highly popular in open source

→ LLaMa.cpp CPU inference

1. wNaM (w4a8, w8a8)
   1. also reduces compute
   2. speedups compute-bound
   3. harder because of outliers

#### 2.1 FP8

* Static amax-scaling quantization
* Several formats (E5M2, E4M3)
* Per-tensor/per-token
* Most current GPUs (e.g., NVIDIA H100, AMD MI300) support FP8 acceleration, but older hardware (Ampere, Turing) does not.

#### 2.2 SmoothQuant

* W8a8
* Per-channel scale is accurate but not efficient
* Since weights are easy to quantize while activations are not, SmoothQuant smooths the activation outliers by offline *migrating* the quantization difficulty from activations to weights with a mathematically equivalent transformation.

#### 2.3 AWQ

[Activation-aware Weight Quantization (AWQ)](https://hf.co/papers/2306.00978) (w4a8) doesn’t quantize all the weights in a model, and instead, it preserves a small percentage of weights that are important for LLM performance. This significantly reduces quantization loss such that you can run models in 4-bit precision without experiencing any performance degradation.

### Calibration

The section above described how quantization from float32 to int8 works, but one question remains: how is the [a, b] range of float32 values determined? That is where calibration comes in to play.

Calibration is the step during quantization where the float32 ranges are computed. For weights it is quite easy since the actual range is known at *quantization-time*. But it is less clear for activations, and different approaches exist:

1. Post training **dynamic quantization**: the range for each activation is computed on the fly at *runtime*. While this gives great results without too much work, it can be a bit slower than static quantization because of the overhead introduced by computing the range each time. It is also not an option on certain hardware.
2. Post training **static quantization**: the range for each activation is computed in advance at *quantization-time*, typically by passing representative data through the model and recording the activation values. In practice, the steps are:
   1. Observers are put on activations to record their values.
   2. A certain number of forward passes on a calibration dataset is done (around 200 examples is enough).
   3. The ranges for each computation are computed according to some *calibration technique*.
3. **Quantization aware training**: the range for each activation is computed at *training-time*, following the same idea than post training static quantization. But “fake quantize” operators are used instead of observers: they record values just as observers do, but they also simulate the error induced by quantization to let the model adapt to it.

For both post training static quantization and quantization aware training, it is necessary to define calibration techniques, the most common are:

* Min-max: the computed range is [min observed value, max observed value], this works well with weights.
* Moving average min-max: the computed range is [moving average min observed value, moving average max observed value], this works well with activations.
* Histogram: records a histogram of values along with min and max values, then chooses according to some criterion:
  + Entropy: the range is computed as the one minimizing the error between the full-precision and the quantized data.
  + Mean Square Error: the range is computed as the one minimizing the mean square error between the full-precision and the quantized data.
  + Percentile: the range is computed using a given percentile value p on the observed values. The idea is to try to have p% of the observed values in the computed range. While this is possible when doing affine quantization, it is not always possible to exactly match that when doing symmetric quantization. You can check [how it is done in ONNX Runtime](https://github.com/microsoft/onnxruntime/blob/2cb12caf9317f1ded37f6db125cb03ba99320c40/onnxruntime/python/tools/quantization/calibrate.py#L698) for more details.

### Advantages and Disadvantages

Advantages:

* Lesser memory consumption: Lower bit width results in less memory for storage
* Fast Inference: This is due to efficient computation due to its lower memory bandwidth requirements
* Less energy consumption: Larger model need more data movement and storage resulting in more energy consumption. Hence a smaller model results in compartively lesser energy usage.
* Smaller models: Can quantize to suit the need and deploy to device with samller hadware specifications.

Disadvantages

* Potential loss in accuracy due to squeezing of high precision weight to lower precision.

## KV Caching

KV caching occurs during multiple token generation steps and only happens in the decoder (i.e., in decoder-only models like GPT, or in the decoder part of encoder-decoder models like T5). Models like BERT are not generative and therefore do not have KV caching.

Since the decoder is causal (i.e., the attention of a token only depends on its preceding tokens), at each generation step we are recalculating the same previous token attention, when we actually just want to calculate the attention for the new token.

This is where KV comes into play. By caching the previous Keys and Values, we can focus on only calculating the attention for the new token.

Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Автоматически созданное описание

Why is this optimization important? As seen in the picture above, **the matrices obtained with KV caching are way smaller, which leads to faster matrix multiplications**. The only downside is that it needs more GPU VRAM (or CPU RAM if GPU is not being used) to cache the Key and Value states.

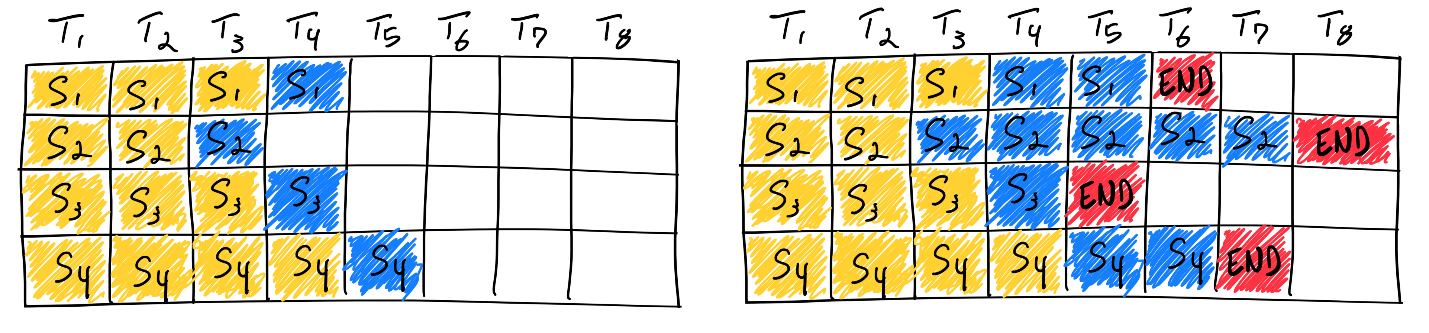
## LLM batching

GPUs are massively-parallel compute architectures, with compute rates (measured in floating-point operations per second, or flops) in the teraflop (A100) or even petaflop (H100) range. Despite these staggering amounts of compute, LLMs struggle to achieve saturation because so much of the chip’s memory bandwidth is spent loading model parameters.

Batching is one way to improve the situation; instead of loading new model parameters each time you have an input sequence, you can load the model parameters once and then use them to process many input sequences. This more efficiently uses the chip’s memory bandwidth, leading to higher compute utilization, higher throughput, and cheaper LLM inference.

### Naive batching / static batching

We call this traditional approach to batching static batching, because the size of the batch remains constant until the inference is complete. Here’s an illustration of static batching in context of LLM inference:

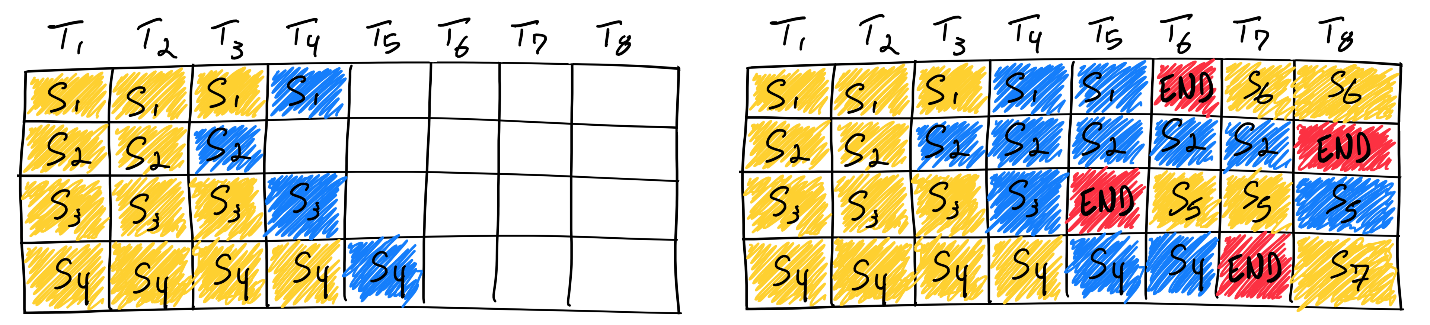


Unlike traditional deep learning models, batching for LLMs can be tricky due to the iterative nature of their inference. Intuitively, this is because requests can "finish" earlier in a batch, but it is tricky to release their resources and add new requests to the batch that may be at different completion states. This means that as the GPU is underutilized as generation lengths of different sequences in a batch differ from the largest generation length of the batch. In the figure on the right above, this is illustrated by the white squares after end-of-sequence tokens for sequences 1, 3, and 4.

How often does static batching under-utilize the GPU? It depends on the generation lengths of sequences in a batch. Without restrictive assumptions on user input and model output, unoptimized production-grade LLM systems simply can’t serve traffic without underutilizing GPUs and incurring unnecessarily high costs. We need to optimize how we serve LLMs for their power to be broadly accessible.

### Continuous (Inflight) batching

The industry recognized the inefficiency and came up with a better approach. Orca: A Distributed Serving System for Transformer-Based Generative Models is a paper presented in OSDI ‘22 which is the first to our knowledge to tackle this problem. Instead of waiting until every sequence in a batch has completed generation, Orca implements iteration-level scheduling where the batch size is determined per iteration. The result is that once a sequence in a batch has completed generation, a new sequence can be inserted in its place, yielding higher GPU utilization than static batching.

Reality is a bit more complicated than this simplified model: since the prefill phase takes compute and has a different computational pattern than generation, it cannot be easily batched with the generation of tokens. Continuous batching frameworks currently manage this via hyperparameter: waiting\_served\_ratio, or the ratio of requests waiting for prefill to those waiting end-of-sequence tokens.

Speaking of frameworks, Hugging Face has productionized continuous batching in their Rust- and Python-based text-generation-inference LLM inference server. We use their implementation to understand the performance characteristics of continuous batching in our benchmarks below.

## Paged Attention

PagedAttention is a new attention mechanism implemented in [vLLM](https://blog.vllm.ai/2023/06/20/vllm.html" \t "_blank) ([GitHub](https://github.com/vllm-project/vllm/tree/main#easy-fast-and-cheap-llm-serving-for-everyone)). It takes inspiration from traditional OS concepts such as [paging](https://en.wikipedia.org/wiki/Memory_paging) and [virtual memory](https://en.wikipedia.org/wiki/Virtual_memory). They allow the KV cache (what is computed in the “prefill” phase, discussed above) to be non-contiguous by allocating memory in fixed-size “pages”, or blocks. The attention mechanism can then be rewritten to operate on block-aligned inputs, allowing attention to be performed on non-contiguous memory ranges.

Изображение выглядит как текст, диаграмма, линия, снимок экрана

Автоматически созданное описание

This means that buffer allocation can happen just-in-time instead of ahead-of-time: when starting a new generation, the framework does not need to allocate a contiguous buffer of size maximum\_context\_length. Each iteration, the scheduler can decide if it needs more room for a particular generation, and allocate on the fly without any degradation to PagedAttention’s performance. This doesn’t guarantee perfect utilization of memory ([their blog](https://blog.vllm.ai/2023/06/20/vllm.html) says the wastage is now limited to under 4%, only in the last block), but it significantly improves upon wastage from ahead-of-time allocation schemes used widely by the industry today.

## Flash Attention

Scaling the transformer architecture is heavily bottlenecked by the self-attention mechanism, which has quadratic time and memory complexity. Recent developments in accelerator hardware mainly focus on enhancing compute capacities and not memory and transferring data between hardware. This results in attention operation having a memory bottleneck. **Flash Attention** is an attention algorithm used to reduce this problem and scale transformer-based models more efficiently, enabling faster training and inference.

Standard attention mechanism uses High Bandwidth Memory (HBM) to store, read and write keys, queries and values. HBM is large in memory, but slow in processing, meanwhile SRAM is smaller in memory, but faster in operations. In the standard attention implementation, the cost of loading and writing keys, queries, and values from HBM is high. It loads keys, queries, and values from HBM to GPU on-chip SRAM, performs a single step of the attention mechanism, writes it back to HBM, and repeats this for every single attention step. Instead, Flash Attention loads keys, queries, and values once, fuses the operations of the attention mechanism, and writes them back.

## Chunked-prefills

SARATHI utilizes “chunked-prefills,” which divide a prefill request into equal-sized chunks, along with “decode-maximal batching,” which forms a batch by combining one prefill chunk with additional decode requests. During inference, the prefill chunk fully utilizes GPU resources, while the decode requests piggyback, significantly reducing computational costs compared to processing decodes independently. This method enables the creation of multiple decode-maximal batches from a single prefill request, optimizing the handling of decode requests. Additionally, the consistent compute load of these batches mitigates imbalances across micro-batches, effectively reducing pipeline inefficiencies.

**Chunked-prefills** is a mechanism for splitting the prefill phase of large language model inference, based on two key insights.

* ***First, there is a point of diminishing returns in throughput when increasing the number of prefill tokens for a given model and GPU*.**
* **Second, in practical applications, the prefill size is often large (1K–4K tokens in production), making it feasible to split the prefill request into smaller compute units.** Implementing chunked-prefills requires careful attention to setting the attention masks. For example, if a 1K token input prompt is split into four chunks of 256 tokens each, the attention masks must be adjusted for each subsequent chunk to ensure that each query token can access all preceding tokens but not those that follow. This approach ensures that the chunked-prefill computation is mathematically equivalent to processing the full prefill in one go.

## Speculative decoding

**Speculative Decoding** is an **LLM inference optimization** technique that speeds up text generation by first using a **smaller, faster model** (draft model) to generate multiple token candidates. The main, larger model then verifies and corrects these predictions in parallel, reducing the number of sequential steps needed.

**How It Works:**

1. **Drafting** → A lightweight model predicts a few tokens ahead.
2. **Verification** → The larger LLM processes these tokens in parallel, accepting correct ones and adjusting incorrect ones.
3. **Acceleration** → This reduces the number of sequential forward passes, improving efficiency without degrading quality.

**Key Benefits:**

* **Faster inference** (reduces sequential processing)
* **Lower latency** (especially for high-throughput systems).
* **Can work with existing models** without retraining.

It's commonly used in **vLLM, TGI, and NVIDIA's TRT-LLM** for optimizing LLM responses.

Изображение выглядит как текст, диаграмма, План, линия

Автоматически созданное описание

### EAGLE (Extrapolation Algorithm for Greater Language-model Efficiency)

* **Drafter is a head instead of model**
* **Frozen base model’s body**
* **High AccRate**
* **Head – 1-layer transformer over hiddens of base model**
* **Tree of hypotheses**

We train a lightweight plugin, called Auto-regression Head,  in conjunction with the Original LLM's frozen embedding layer, to predict the next feature based on the current feature sequence from the second-top-layer of the Original model. The token is then derived using the frozen classification head of the Original LLM, which maps features to tokens. Features, being more abstract and clearer than token sequences, make the task of regressing features considerably simpler than regressing tokens. In summary, EAGLE extrapolates at the feature level using a small auto-regressive head and then employs the frozen classification head to generate the predicted sequence of tokens.

During the forward process of the Original LLM, we collect the features from the second-top-layer. Beginning with these features and a token generated in this forward pass by the Original LLM, the FeatExtrapolator commences its "guessing" process. The FeatExtrapolator integrates embeddings and generates the next feature in an auto-regressive manner. Subsequently, the distribution of tokens is determined using the frozen LM head, enabling us to sample from this distribution. By repeating this sampling multiple times, we conduct a tree-like generation process, as depicted on the right side of the figure. In this example, three forward passes of the FeatExtrapolator "guess" a tree of 10 tokens.

## Knowledge Distillation

Most fine-tuned LLMs contain enormous numbers of parameters. Consequently, foundation LLMs require enormous computational and environmental resources to generate predictions. Note that large swaths of those parameters are typically irrelevant for a specific application.

[**Distillation**](https://developers.google.com/machine-learning/glossary#distillation) creates a smaller version of an LLM. The distilled LLM generates predictions much faster and requires fewer computational and environmental resources than the full LLM. However, the distilled model’s predictions are generally not quite as good as the original LLM’s predictions. Recall that LLMs with more parameters almost always generate better predictions than LLMs with fewer parameters.

The most common form of distillation uses bulk inference to label data. This labeled data is then used to train a new, smaller model (known as the student model) that can be more affordably served. The labeled data serves as a channel by which the larger model (known as the teacher model) funnels its knowledge to the smaller model.

* **Teacher** p(y|x): usually a LLM, e.g. GPT-3 (175B)
  + Achieves SOTA quality
  + Doesn't fit inference computational budget
* **Student** q(y|x): small LM, e.g. T5 XL (3B)
  + Unable to reach teacher’s quality by ordinary training
  + Fits inference computational budget

### Hard- and Soft-Label KD

The overall training loss LLL is typically a weighted sum of:

* **Hard Loss:** The cross-entropy between the student’s predictions and the true labels.

Изображение выглядит как текст, рукописный текст, Шрифт, каллиграфия

Автоматически созданное описание

* **Soft Loss:** The cross-entropy between the student’s predictions (with temperature scaling) and the teacher’s softened outputs. Cross-Entropy loss.

Изображение выглядит как текст, Шрифт, рукописный текст, каллиграфия

Автоматически созданное описание

### KL KD

**KL Divergence** measures how one probability distribution diverges from another:

Изображение выглядит как Шрифт, текст, типография, каллиграфия

Автоматически созданное описание

The KL-divergence is nicer as a loss since it will equal 0 when the student network matches the teacher on all labels. In contrast, if we use X-entropy, then the loss will fluctuate even when the student and teacher output the exact same thing, and it will fluctuate according to the batch

KLDiv(P||Q) = entropy(P) + crossentropy(P, Q).

### Speculative KD

* Student generates
* Teacher monitors
* Teacher’s top-k
* No backprop through sampling

## Offline inference

The number of parameters in an LLM is sometimes so large that [**online inference**](https://developers.google.com/machine-learning/glossary#online-inference) is too slow to be practical for real-world tasks like regression or classification. Consequently, many engineering teams rely on [**offline inference**](https://developers.google.com/machine-learning/glossary#offline-inference) (also known as *bulk inference* or *static inference*) instead. In other words, rather than responding to queries at serving time, the trained model makes predictions in advance and then caches those predictions.

It doesn't matter if it takes a long time for an LLM to complete its task if the LLM only has to perform the task once a week or once a month.

For example, Google Search [used an LLM](https://blog.google/products/search/how-mum-improved-google-searches-vaccine-information/) to perform offline inference in order to cache a list of over 800 synonyms for Covid vaccines in more than 50 languages. Google Search then used the cached list to identify queries about vaccines in live traffic.

## Optimization Frameworks

* **TRT-LLM** → Best for NVIDIA GPU users who need extreme performance via TensorRT optimizations.
* **vLLM** → Best for high-throughput applications (e.g., chatbots) due to **PagedAttention** & **continuous batching**.
* **SGLang** → Good for **serverless** or multi-cloud inference, focusing on **scalability & efficiency**.
* **TGI** → Hugging Face’s production-ready LLM inference API, great for **deploying Hugging Face models at scale**.

## Triton

Triton is an open-source framework developed by OpenAI for writing high-performance GPU code with a Python-like syntax. It simplifies writing custom GPU kernels while achieving performance comparable to handwritten CUDA code.

* **Triton** is a language for programming **GPUs**.
  + More convenient than **CUDA**.
  + Allows writing **Python-like code** that compiles to **PTX** (Parallel Thread Execution).
    - PTX is the same intermediate representation used by CUDA.
* **Triton Compiler**:
  + **Optimizes code** by rearranging it for better performance without changing its meaning.
  + Targets the **same hardware** as CUDA.

### When to Use Triton

- Optimization Steps:

1. Use torch.compile():

- Start by using torch.compile() to optimize your code.

2. Adapt Your Code:

- Rewrite code to be more suitable for torch.compile().

- E.g., eliminate graph breaks to enable CUDA graphs.

3. Profile and Identify Bottlenecks:

- Find slow parts of your code using profiling tools.

- Write custom Triton kernels for these parts.

4. Consider CUDA:

- If still not fast enough, write custom CUDA kernels.

**Rough Edges in Triton**

- New-ish Project:

- Contains rough edges; code may not behave as expected.

- Expected to become more polished over time.

**Recommendation:**

- Debugging is important; use “simulator mode” when possible.

- Be aware of limitations on older GPUs or with certain operations.

Link: <https://christianjmills.com/posts/cuda-mode-notes/lecture-014/#auto-tuning>

# LLMOps

## What is LLMOps (large language model operations)?

LLMOps, or large language model operations, refers to the practices and processes involved in managing and operating large language models (LLMs). LLMs are artificial intelligence (AI) models trained on vast datasets of text and code, enabling them to perform various language-related tasks, such as text generation, translation, and question answering.

LLMOps involves a comprehensive set of activities, including:

* **Model deployment and maintenance**: deploying and managing LLMs on cloud platforms or on-premises infrastructure
* **Data management**: curating and preparing training data, as well as monitoring and maintaining data quality
* **Model training and fine-tuning**: training and refining LLMs to improve their performance on specific tasks
* **Monitoring and evaluation**: tracking LLM performance, identifying errors, and optimizing models
* **Security and compliance**: ensuring the security and regulatory compliance of LLM operations

### What is the difference between LLMOps and MLOps?

LLMOps is a specialized subset of MLOps (machine learning operations), which focuses specifically on the challenges and requirements of managing LLMs. While MLOps covers the general principles and practices of managing machine learning models, LLMOps addresses the unique characteristics of LLMs, such as their large size, complex training requirements, and high computational demands.

### How does LLMOps work?

LLMOps involves a number of different steps, including:

**Data collection and preparation**: LLMs require large amounts of data to train. This data must be collected and prepared in a way that is suitable for training the model.

**Model development**: LLMs are developed using a variety of techniques, including unsupervised learning, supervised learning, and reinforcement learning.

**Model deployment**: Once a LLM has been developed, it must be deployed to a production environment. This involves setting up the necessary infrastructure and configuring the model to run on a specific platform.

**Model management**: LLMs require ongoing management to ensure that they are performing as expected. This includes monitoring the model's performance, retraining the model as needed, and making sure that the model is secure.

### Benefits of LLMOps

LLMOps (large language model operations) offers numerous benefits for organizations looking to manage and deploy LLMs (large language models) effectively. These benefits include:

**Performance**

LLMOps tools and techniques help organizations optimize the performance of their LLMs by identifying and resolving bottlenecks, fine-tuning model parameters, and implementing efficient deployment strategies. This can lead to improved accuracy, faster response times, and better overall user experiences.

**Scalability**

LLMOps provides a scalable and flexible framework for managing LLMs, enabling organizations to easily adapt to changing demands and requirements.

**Risk reduction**

LLMOps helps organizations mitigate risks associated with deploying and operating LLMs. By implementing robust monitoring systems, establishing disaster recovery plans, and conducting regular security audits, LLMOps reduces the likelihood of outages, data breaches, and other disruptions. This proactive approach minimizes the impact of potential risks and ensures the continuous availability and reliability of LLMs.

**Efficiency**

LLMOps streamlines the entire life cycle of LLMs, from data preparation and model training to deployment and monitoring. Automated tools and standardized processes improve efficiency by reducing manual tasks, optimizing resource utilization, and minimizing the time required for model development and deployment.

### Best practices for LLMOps

LLMOps (large language model operations) best practices are a set of guidelines and recommendations that help organizations manage and deploy LLMs (large language models) effectively and efficiently. These best practices cover various aspects of the LLMOps life cycle, including data management, model training, deployment, and monitoring.

**Data management**

* **Use high-quality data**: LLMs require large amounts of high-quality data to train effectively. Organizations should ensure that the data used for training is clean, accurate, and relevant to the desired use case.
* **Manage data efficiently:** LLMs can generate vast amounts of data during training and inference. Organizations should implement efficient data management strategies, such as data compression and data partitioning, to optimize storage and retrieval.
* **Establish data governance**: Clear data governance policies and procedures should be established to ensure the secure and responsible use of data throughout the LLMOps life cycle.

**Model training**

* **Choose the right training algorithm**: Different training algorithms are suitable for different types of LLMs and tasks. Organizations should carefully evaluate the available training algorithms and select the one that best aligns with their specific requirements.
* **Optimize training parameters**: Hyperparameter tuning is important for optimizing LLM performance. Experiment with different training parameters, such as learning rate and batch size, to find the optimal settings for your models.
* **Monitor training progress**: Regular monitoring of training progress is essential to identify potential issues and make necessary adjustments. Organizations should implement metrics and dashboards to track key training indicators, such as loss and accuracy.

**Deployment**

* **Choose the right deployment strategy**: LLMs can be deployed in various ways, such as cloud-based services, on-premises infrastructure, or edge devices. Carefully consider their specific requirements and choose the deployment strategy that best meets their needs.
* **Optimize deployment performance**: Once deployed, LLMs should be monitored and optimized for performance. This may involve scaling resources, adjusting model parameters, or implementing caching mechanisms to improve response times.
* **Ensure security**: Strong security measures should be implemented to protect LLMs and the data they process. This includes access controls, data encryption, and regular security audits.

**Monitoring**

* **Establish monitoring metrics**: Key performance indicators (KPIs) should be established to monitor the health and performance of LLMs. These metrics may include accuracy, latency, and resource utilization.
* **Implement real-time monitoring**: Real-time monitoring systems should be implemented to detect and respond to any issues or anomalies that may arise during operations.
* **Analyze monitoring data**: Monitoring data should be regularly analyzed to identify trends, patterns, and potential areas for improvement. This analysis helps optimize LLMOps processes and ensure the continuous delivery of high-quality LLMs.

# Prompting

## Prompting an LLM

You can achieve a lot with simple prompts, but the quality of results depends on how much information you provide it and how well-crafted the prompt is. A prompt can contain information like the *instruction* or *question* you are passing to the model and include other details such as *context*, *inputs*, or *examples*. You can use these elements to instruct the model more effectively to improve the quality of results.

Let's get started by going over a basic example of a simple prompt:

*Prompt*

The sky is

*Output:*

blue.

## Zero-Shot Prompting

Large language models (LLMs) today, such as GPT-3.5 Turbo, GPT-4, and Claude 3, are tuned to follow instructions and are trained on large amounts of data. Large-scale training makes these models capable of performing some tasks in a "zero-shot" manner. Zero-shot prompting means that the prompt used to interact with the model won't contain examples or demonstrations. The zero-shot prompt directly instructs the model to perform a task without any additional examples to steer it.

We tried a few zero-shot examples in the previous section. Here is one of the examples (ie., text classification) we used:

*Prompt:*

*Classify the text into neutral, negative or positive.*

*Text: I think the vacation is okay.*

*Sentiment:*

*Output:*

*Neutral*

Note that in the prompt above we didn't provide the model with any examples of text alongside their classifications, the LLM already understands "sentiment" -- that's the zero-shot capabilities at work.

## Few-Shot Prompting

Let's demonstrate few-shot prompting via an example that was presented in [Brown et al. 2020](https://arxiv.org/abs/2005.14165). In the example, the task is to correctly use a new word in a sentence.

*Prompt:*

*A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is:*

*We were traveling in Africa and we saw these very cute whatpus.*

*To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:*

*Output:*

*When we won the game, we all started to farduddle in celebration.*

We can observe that the model has somehow learned how to perform the task by providing it with just one example (i.e., 1-shot). For more difficult tasks, we can experiment with increasing the demonstrations (e.g., 3-shot, 5-shot, 10-shot, etc.).

Following the findings from [Min et al. (2022)](https://arxiv.org/abs/2202.12837), here are a few more tips about demonstrations/exemplars when doing few-shot:

* "the label space and the distribution of the input text specified by the demonstrations are both important (regardless of whether the labels are correct for individual inputs)"
* the format you use also plays a key role in performance, even if you just use random labels, this is much better than no labels at all.
* additional results show that selecting random labels from a true distribution of labels (instead of a uniform distribution) also helps.

## Chain-of-Thought Prompting

Introduced in [Wei et al. (2022)](https://arxiv.org/abs/2201.11903), chain-of-thought (CoT) prompting enables complex reasoning capabilities through intermediate reasoning steps. You can combine it with few-shot prompting to get better results on more complex tasks that require reasoning before responding.

*Prompt:*

*The odd numbers in this group add up to an even number: 4, 8, 9, 15, 12, 2, 1.*

*A: Adding all the odd numbers (9, 15, 1) gives 25. The answer is False.*

*The odd numbers in this group add up to an even number: 17, 10, 19, 4, 8, 12, 24.*

*A: Adding all the odd numbers (17, 19) gives 36. The answer is True.*

*The odd numbers in this group add up to an even number: 16, 11, 14, 4, 8, 13, 24.*

*A: Adding all the odd numbers (11, 13) gives 24. The answer is True.*

*The odd numbers in this group add up to an even number: 17, 9, 10, 12, 13, 4, 2.*

*A: Adding all the odd numbers (17, 9, 13) gives 39. The answer is False.*

*The odd numbers in this group add up to an even number: 15, 32, 5, 13, 82, 7, 1.*

*A:*

*Output:*

*Adding all the odd numbers (15, 5, 13, 7, 1) gives 41. The answer is False.*

One recent idea that came out more recently is the idea of [zero-shot CoT](https://arxiv.org/abs/2205.11916) (Kojima et al. 2022) that essentially involves adding "Let's think step by step" to the original prompt.

*Prompt:*

*I went to the market and bought 10 apples. I gave 2 apples to the neighbor and 2 to the repairman. I then went and bought 5 more apples and ate 1. How many apples did I remain with?*

*Let's think step by step.*

*Output:*

*First, you started with 10 apples.*

*You gave away 2 apples to the neighbor and 2 to the repairman, so you had 6 apples left.*

*Then you bought 5 more apples, so now you had 11 apples.*

*Finally, you ate 1 apple, so you would remain with 10 apples.*

**Automatic Chain-of-Thought (Auto-CoT)**

When applying chain-of-thought prompting with demonstrations, the process involves hand-crafting effective and diverse examples. This manual effort could lead to suboptimal solutions. [Zhang et al. (2022)](https://arxiv.org/abs/2210.03493) propose an approach to eliminate manual efforts by leveraging LLMs with "Let's think step by step" prompt to generate reasoning chains for demonstrations one by one. This automatic process can still end up with mistakes in generated chains. To mitigate the effects of the mistakes, the diversity of demonstrations matter. This work proposes Auto-CoT, which samples questions with diversity and generates reasoning chains to construct the demonstrations.

Auto-CoT consists of two main stages:

* Stage 1): **question clustering**: partition questions of a given dataset into a few clusters
* Stage 2): **demonstration sampling**: select a representative question from each cluster and generate its reasoning chain using Zero-Shot-CoT with simple heuristics

The simple heuristics could be length of questions (e.g., 60 tokens) and number of steps in rationale (e.g., 5 reasoning steps). This encourages the model to use simple and accurate demonstrations.

# Agents

## LLM Agents

LLM based agents, hereinafter also referred to as LLM agents for short, involve LLM applications that can execute complex tasks through the use of an architecture that combines LLMs with key modules like planning and memory. When building LLM agents, an LLM serves as the main controller or "brain" that controls a flow of operations needed to complete a task or user request. The LLM agent may require key modules such as planning, memory, and tool usage.

To better motivate the usefulness of an LLM agent, let's say that we were interested in building a system that can help answer the following question:

*What's the average daily calorie intake for 2023 in the United States?*

The question above could potentially be answered using an LLM that already has the knowledge needed to answer the question directly. If the LLM doesn't have the relevant knowledge to answer the question, it's possible to use a simple RAG system where an LLM has access to health related information or reports. Now let's give the system a more complex question like the following:

*How has the trend in the average daily calorie intake among adults changed over the last decade in the United States, and what impact might this have on obesity rates? Additionally, can you provide a graphical representation of the trend in obesity rates over this period?*

To answer such a question, just using an LLM alone wouldn't be enough. You can combine the LLM with an external knowledge base to form a RAG system but this is still probably not enough to answer the complex query above. This is because the complex question above requires an LLM to break the task into subparts which can be addressed using tools and a flow of operations that leads to a desired final response. A possible solution is to build an LLM agent that has access to a search API, health-related publications, and public/private health database to provide relevant information related to calorie intake and obesity.

In addition, the LLM will need access to a "code interpreter" tool that helps take relevant data to produce useful charts that help understand trends in obesity. These are the possible high-level components of the hypothetical LLM agent but there are still important considerations such as creating a plan to address the task and potential access to a memory module that helps the agent keep track of the state of the flow of operations, observations, and overall progress.

## LLM Agent Framework

Изображение выглядит как текст, диаграмма, снимок экрана, Шрифт

Автоматически созданное описание

Generally speaking, an LLM agent framework can consist of the following core components:

* User Request - a user question or request
* Agent/Brain - the agent core acting as coordinator
* Planning - assists the agent in planning future actions
* Memory - manages the agent's past behaviors

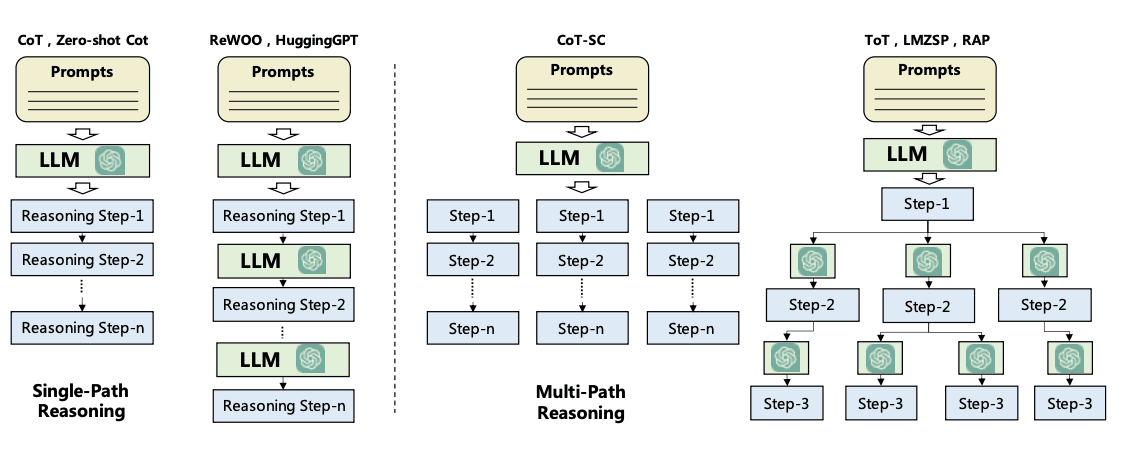
## Agent

A large language model (LLM) with general-purpose capabilities serves as the main brain, agent module, or coordinator of the system. This component will be activated using a prompt template that entails important details about how the agent will operate, and the tools it will have access to (along with tool details).

While not mandatory, an agent can be profiled or be assigned a persona to define its role. This profiling information is typically written in the prompt which can include specific details like role details, personality, social information, and other demographic information. According to [Wang et al. 2023], the strategies to define an agent profile include handcrafting, LLM-generated or data-driven.

## Planning Without Feedback

The planning module helps to break down the necessary steps or subtasks the agent will solve individually to answer the user request. This step is important to enable the agent to reason better about the problem and reliably find a solution. The planning module will leverage an LLM to decompose a detailed plan which will include subtasks to help address the user question. Popular techniques for task decomposition include [Chain of Thought(opens in a new tab)](https://www.promptingguide.ai/techniques/cot) and [Tree of Thoughts(opens in a new tab)](https://www.promptingguide.ai/techniques/tot) which can be categorized as single-path reasoning and multi-path reasoning, respectively. Below is a figure comparing different strategies as formalized in [Wang et al., 2023(opens in a new tab)](https://arxiv.org/abs/2308.11432):



## Planning With Feedback

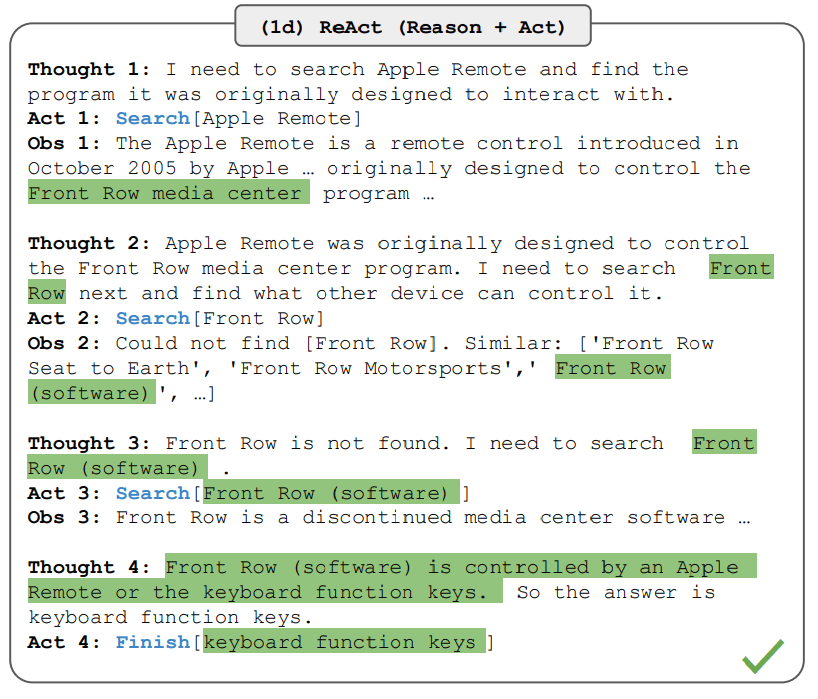
The planning modules above don't involve any feedback which makes it challenging to achieve long-horizon planning to solve complex tasks. To address this challenge, you can leverage a mechanism that enables the model to iteratively reflect and refine the execution plan based on past actions and observations. The goal is to correct and improve on past mistakes which helps to improve the quality of final results. This is particularly important in complex real-world environments and tasks where trial and error are key to completing tasks. Two popular methods for this reflection or critic mechanism include [ReAct](https://www.promptingguide.ai/techniques/react" \t "_blank) and [Reflexion](https://arxiv.org/abs/2303.11366" \t "_blank).

## ReAct

ReAct is inspired by the synergies between "acting" and "reasoning" which allow humans to learn new tasks and make decisions or reasoning.

Chain-of-thought (CoT) prompting has shown the capabilities of LLMs to carry out reasoning traces to generate answers to questions involving arithmetic and commonsense reasoning, among other tasks [(Wei et al., 2022)(opens in a new tab)](https://arxiv.org/abs/2201.11903). But its lack of access to the external world or inability to update its knowledge can lead to issues like fact hallucination and error propagation.

ReAct is a general paradigm that combines reasoning and acting with LLMs. ReAct prompts LLMs to generate verbal reasoning traces and actions for a task. This allows the system to perform dynamic reasoning to create, maintain, and adjust plans for acting while also enabling interaction to external environments (e.g., Wikipedia) to incorporate additional information into the reasoning. The figure below shows an example of ReAct and the different steps involved to perform question answering.



## Memory

The memory module helps to store the agent's internal logs including past thoughts, actions, and observations from the environment, including all interactions between agent and user. There are two main memory types that have been reported in the LLM agent literature:

* **Short-term memory** - includes context information about the agent's current situations; this is typically realized by in-context learning which means it is short and finite due to context window constraints.
* **Long-term memory** - includes the agent's past behaviors and thoughts that need to be retained and recalled over an extended period of time; this often leverages an external vector store accessible through fast and scalable retrieval to provide relevant information for the agent as needed.

Hybrid memory integrates both short-term memory and long-term memory to improve an agent's ability for long-range reasoning and accumulation of experiences.

There are also different memory formats to consider when building agents. Representative memory formats include natural language, embeddings, databases, and structured lists, among others. These can also be combined such as in Ghost in the Minecraft ([GITM(opens in a new tab)](https://arxiv.org/abs/2305.17144)) that utilizes a key-value structure where the keys are represented by natural language and values are represented by embedding vectors.

Both the planning and memory modules allow the agent to operate in a dynamic environment and enable it to effectively recall past behaviors and plan future actions.

## Tools

Tools correspond to a set of tool/s that enables the LLM agent to interact with external environments such as Wikipedia Search API, Code Interpreter, and Math Engine. Tools could also include databases, knowledge bases, and external models. When the agent interacts with external tools it executes tasks via workflows that assist the agent to obtain observations or necessary information to complete subtasks and satisfy the user request. In our initial health-related query, a code interpreter is an example of a tool that executes code and generates the necessary chart information requested by the user.

# Frameworks

## LangChain

**LangChain** is a framework designed to simplify the development of applications powered by **large language models (LLMs)** like GPT-3, GPT-4, Llama 2, and others. It provides modular components and tools to integrate LLMs into workflows, manage memory, interact with external data, and build complex AI-driven applications.

**Key Features of LangChain**

**1. LLM Integration & Standardized Interfaces**

* Supports multiple **LLM providers** (OpenAI, Anthropic, Hugging Face, Google Gemini, etc.) via a unified API.
* Simplifies switching between models (e.g., from GPT-4 to Claude 3).
* Includes wrappers for **chat models** (e.g., ChatOpenAI) and **text completion models**.

**2. Prompt Management & Templating**

* **Prompt templates** allow dynamic input generation (e.g., inserting user queries into predefined structures).
* Example:

python

* from langchain.prompts import PromptTemplate
* prompt = PromptTemplate.from\_template("Explain {topic} like I'm 5.")
* prompt.format(topic="quantum physics")
* Supports **few-shot learning** with example-based prompts.

**3. Chains (Sequential Workflows)**

* **Chains** combine multiple steps (e.g., prompt → LLM → output parsing).
* Prebuilt chains for common tasks (e.g., LLMChain, SequentialChain).
* Custom chains for complex workflows (e.g., summarize → translate → answer).

**4. Memory & State Management**

* Maintains **conversation history** across interactions.
* Supports:
  + **ConversationBufferMemory** (raw chat history).
  + **ConversationSummaryMemory** (compressed summaries).
  + **Entity Memory** (track specific details like user preferences).

**5. Data Augmentation (RAG - Retrieval-Augmented Generation)**

* Integrates with **vector databases** (FAISS, Pinecone, Chroma) for semantic search.
* Loads external data via **document loaders** (PDFs, web pages, SQL DBs).
* Splits text efficiently with **text splitters**.

**6. Agents & Tool Use**

* **Agents** enable LLMs to use external tools (APIs, calculators, web searches).
* Example:

python

* from langchain.agents import load\_tools, initialize\_agent
* tools = load\_tools(["serpapi", "llm-math"], llm=llm)
* agent = initialize\_agent(tools, llm, agent="zero-shot-react-description")
* agent.run("What's the GDP of France divided by its population?")

**7. Output Parsing & Structured Responses**

* Converts LLM outputs into structured formats (JSON, Pydantic models).
* Example:

python

* from langchain.output\_parsers import StructuredOutputParser
* parser = StructuredOutputParser.from\_response\_schemas(schema)
* model\_output = llm(prompt)
* parsed\_result = parser.parse(model\_output)

**8. Callbacks & Logging**

* Tracks events (e.g., token usage, latency) with **callback handlers**.
* Integrates with **LangSmith** for debugging and monitoring.

**9. Multimodal Capabilities (Vision, Audio, etc.)**

* Supports **image-to-text** (e.g., GPT-4V) and **speech-to-text** models.

**10. Ecosystem & Community**

* Open-source with extensive **documentation** and tutorials.
* Integrates with **FastAPI**, **Streamlit**, **Django**, etc.

**Use Cases**

✅ **Chatbots** (with memory)  
✅ **Document QA** (RAG over PDFs/websites)  
✅ **Code Generation & Analysis**  
✅ **AI Agents** (autonomous workflows)  
✅ **Data Extraction & Summarization**

## LlamaIndex

**LlamaIndex** (formerly known as **GPT Index**) is a powerful framework designed to enhance the capabilities of **large language models (LLMs)** like GPT-4, Llama 2, and others by enabling efficient **data indexing, retrieval, and augmentation**. It is particularly useful for building **RAG (Retrieval-Augmented Generation)** applications, where external data is used to improve LLM responses.

**🔹 Key Features of LlamaIndex**

**1. Data Connectors & Ingestion**

* Supports **100+ data sources** (APIs, PDFs, SQL databases, Notion, Slack, etc.).
* Built-in **document loaders** for:
  + **Files** (PDFs, Word, Excel, Markdown, HTML).
  + **Databases** (PostgreSQL, MySQL, SQLite).
  + **Web** (RSS, sitemaps, raw URLs).

Example:

python

from llama\_index.core import SimpleDirectoryReader

documents = SimpleDirectoryReader("./data").load\_data() # Loads files from a directory

**2. Indexing & Retrieval**

* **Converts unstructured data into searchable indexes** (vector, keyword-based, or hybrid).
* Supports multiple index types:
  + **VectorStoreIndex** (for semantic search via embeddings).
  + **ListIndex** (for summarization).
  + **TreeIndex** (for hierarchical data).
  + **KeywordTableIndex** (for keyword-based lookup).

*from llama\_index.core import VectorStoreIndex*

*index = VectorStoreIndex.from\_documents(documents) # Creates a vector index*

**3. Query Engines & Retrieval-Augmented Generation (RAG)**

* **Retrieves relevant context** before passing it to an LLM.
* Supports:
  + **Semantic search** (using embeddings like OpenAI, Hugging Face).
  + **Hybrid search** (combining vectors + keywords).
  + **Multi-document queries**.

*query\_engine = index.as\_query\_engine()*

*response = query\_engine.query("What is the capital of France?")*

*print(response) # Returns LLM answer with retrieved context*

**4. Advanced Retrieval Techniques**

* **Node-based chunking**: Splits documents into smaller chunks for better retrieval.
* **Embedding optimizations**: Supports **Cohere**, **OpenAI**, **Hugging Face**, and local models.
* **Re-ranking**: Improves results by reordering retrieved chunks (e.g., using **Cohere Rerank**).

*from llama\_index.core.node\_parser import SentenceSplitter*

*splitter = SentenceSplitter(chunk\_size=512) # Splits text into 512-token chunks*

*nodes = splitter.get\_nodes\_from\_documents(documents)*

**5. Integration with LLMs & Agents**

* Works with **OpenAI**, **Anthropic**, **Llama 2**, **Gemini**, and more.
* Supports **agentic workflows** (e.g., multi-step reasoning with tools).

*from llama\_index.llms import OpenAI*

*llm = OpenAI(model="gpt-4-turbo")*

*query\_engine = index.as\_query\_engine(llm=llm)*

**6. Storage & Caching**

* **Persists indexes** (saves to disk or databases like **Postgres**, **Redis**).
* **Caches embeddings** to reduce API costs.

Example:

python

index.storage\_context.persist(persist\_dir="./storage") # Saves index to disk

**7. Evaluation & Observability**

* **Tracks retrieval accuracy** (hit rate, MRR).
* **Logs queries/responses** for debugging.

from llama\_index.core.evaluation import RetrieverEvaluator

evaluator = RetrieverEvaluator.from\_metric\_names(["hit\_rate"])

results = evaluator.evaluate(retriever)

**🔹 Use Cases**

✅ **Document Q&A** (Chat with PDFs, websites, databases).  
✅ **Enterprise search** (Internal wiki, Slack, Notion).  
✅ **AI Agents** (Dynamic data fetching for LLMs).  
✅ **Knowledge Graphs** (Structured data exploration).

## Haystack

**Haystack** (by deepset) is an open-source framework designed for **building search systems and NLP applications** with a focus on **question answering, retrieval-augmented generation (RAG), and semantic search**. It integrates seamlessly with **vector databases, transformers, and LLMs** to create powerful AI-powered search pipelines.

**🔹 Key Features of Haystack**

**1. Modular Pipelines for Search & NLP**

* Build **custom workflows** using components like:
  + **Retrievers** (BM25, Dense Passage Retrieval, Embedding-based).
  + **Readers** (LLMs for answer extraction, e.g., OpenAI, Hugging Face).
  + **Classifiers** (for filtering/reranking).
  + **Generators** (for RAG with LLMs like GPT-4, Llama 2).

Example of a **RAG pipeline**:

python

from haystack import Pipeline

from haystack.nodes import BM25Retriever, FARMReader

retriever = BM25Retriever(document\_store, top\_k=5)

reader = FARMReader(model\_name\_or\_path="deepset/roberta-base-squad2")

pipeline = Pipeline()

pipeline.add\_node(component=retriever, name="Retriever", inputs=["Query"])

pipeline.add\_node(component=reader, name="Reader", inputs=["Retriever"])

**2. Document Stores & Vector Databases**

* Supports **multiple databases**:
  + **Elasticsearch** (for keyword + vector hybrid search).
  + **FAISS**, **Milvus**, **Weaviate**, **Pinecone** (for vector search).
  + **InMemoryDocumentStore** (for prototyping).

Example:

python

from haystack.document\_stores import FAISSDocumentStore

document\_store = FAISSDocumentStore(faiss\_index\_factory\_str="Flat")

**3. Advanced Retrieval Methods**

* **Hybrid Search**: Combines **BM25 (keyword) + Dense (vector) retrieval**.
* **Semantic Search**: Uses **embeddings** (OpenAI, Sentence-BERT, etc.).
* **Reranking**: Improves results with **Cross-Encoders** (e.g., BAAI/bge-reranker).

Example:

python

from haystack.nodes import EmbeddingRetriever

retriever = EmbeddingRetriever(

document\_store=document\_store,

embedding\_model="sentence-transformers/all-mpnet-base-v2"

)

**4. LLM Integration & RAG**

* Works with **OpenAI, Anthropic, Llama 2, and local models**.
* Supports **prompt engineering** and **few-shot learning**.

Example (using GPT-4):

python

from haystack.nodes import PromptNode

prompt\_node = PromptNode(

model\_name\_or\_path="gpt-4",

api\_key="YOUR\_OPENAI\_KEY",

default\_prompt\_template="deepset/question-answering"

)

**5. Preprocessing & Data Handling**

* **Text cleaning, splitting, and chunking**.
* **PDF, HTML, and database connectors**.
* **Annotation tools** (for labeling training data).

Example:

python

from haystack.nodes import PreProcessor

processor = PreProcessor(

clean\_empty\_lines=True,

split\_by="word",

split\_length=200

)

processed\_docs = processor.process(documents)

**6. Evaluation & Monitoring**

* **Metrics for retrieval/reader performance** (Recall, Precision, F1).
* **Integration with MLflow & Weights & Biases (W&B)**.

Example:

python

from haystack.eval import EvalDocuments

eval\_retriever = EvalDocuments()

eval\_retriever.eval(retriever.run(query="What is AI?"), gold\_documents)

**🔹 Use Cases**

✅ **Enterprise Search** (Internal docs, Slack, Confluence).  
✅ **Question Answering** (Chatbots, customer support).  
✅ **Semantic Search** (E-commerce, research papers).  
✅ **RAG Systems** (LLMs with real-time data).

## Haystack vs. LlamaIndex vs. LangChain

| **Feature** | **Haystack** | **LlamaIndex** | **LangChain** |
| --- | --- | --- | --- |
| **Primary Use** | Search & QA pipelines | RAG-focused indexing | General LLM workflows |
| **Retrieval** | ✅ Hybrid (BM25 + Vector) | ✅ Vector-based | ✅ Via third-party integrations |
| **LLM Support** | ✅ (OpenAI, Hugging Face, etc.) | ✅ (Same + local models) | ✅ (Broadest LLM support) |
| **Ease of Use** | Medium (Pipeline-centric) | High (RAG-optimized) | High (Flexible but comple |

## Vector databases

A vector database is a specialized type of database designed to store, manage, and query high-dimensional vectors, which are numerical representations of data points. These vectors are used to capture semantic meaning and relationships within data, especially for unstructured data like text, images, and audio. Vector databases are crucial for AI applications, particularly in areas like similarity search, recommendation systems, and retrieval-augmented generation (RAG).

* **Vectors:**

Data points are represented as vectors (arrays of numbers) in a multi-dimensional space. Each dimension corresponds to a specific feature or property of the data.

  **Vector Embeddings:**

These are numerical representations of data, allowing machine learning models to process and understand complex data like text, images, and audio.

  **Similarity Search:**

Vector databases excel at finding vectors that are "close" to each other in the vector space, which corresponds to finding semantically similar data points.

  **Approximate Nearest Neighbor (ANN) Search:**

Vector databases often use ANN algorithms to speed up similarity searches, as finding the absolute nearest neighbor in high-dimensional spaces can be computationally expensive.

How it works:

1. **1. Data is converted to vectors:**

Unstructured data (text, images, etc.) is converted into vector embeddings using machine learning models.

  **2. Vectors are stored in the database:**

The vector embeddings are stored in the vector database.

  **3. Queries are also converted to vectors:**

When a user submits a query, it's also converted into a vector using the same embedding model.

  **4. Similarity search is performed:**

The database searches for vectors that are similar to the query vector, retrieving relevant data points.

Benefits of Vector Databases:

* **Efficiently handle unstructured data:**

They are designed to work with data that doesn't fit neatly into rows and columns.

  **Support AI applications:**

They are crucial for applications like search, recommendations, and text generation.

  **Enable similarity search:**

They can find data points that are similar to a given query, even if they are not identical.

  **Scalability:**

Many vector databases are designed to scale to handle large datasets and high query loads.

Examples of use cases:

* **Recommendation systems:** Suggesting products or content based on user preferences.

  **Semantic search:** Finding documents or information based on meaning rather than keywords.

  **Image and video search:** Finding similar images or videos based on visual features.

  **Retrieval-augmented generation (RAG):** Augmenting the knowledge of large language models (LLMs) with external data.

**1. FAISS (Facebook AI Similarity Search)**

* **Developed by**: Meta (Facebook)
* **Type**: Library (not a standalone server)
* **Strengths**:
  + Blazing-fast **GPU-accelerated** similarity search.
  + Ideal for **static datasets** (no real-time updates).
* **Weaknesses**:
  + No native support for **metadata filtering** or **scalability** (requires manual sharding).
* **Best for**: Research, batch processing, and applications where raw speed matters.

**2. Qdrant**

* **Developed by**: Qdrant team
* **Type**: Standalone server (cloud/self-hosted)
* **Strengths**:
  + Built-in **metadata filtering** (e.g., search with color=red AND price<100).
  + **Hybrid search** (combines vectors + keywords).
  + Horizontal scaling **for large datasets**.
* **Weaknesses**:
  + Slightly higher latency than FAISS for pure vector search.
* **Best for**: Production apps needing **filtering, scalability, and real-time updates**.

**3. Chroma**

* **Developed by**: Chroma team
* **Type**: Lightweight server/embedded
* **Strengths**:
  + **Simplest setup** (good for prototyping).
  + Native **LangChain/LlamaIndex integration**.
  + **Automatic embedding management**.
* **Weaknesses**:
  + Less mature for **large-scale deployments**.
  + Limited advanced features (e.g., no hybrid search).
* **Best for**: Quick RAG prototypes and small-to-medium projects.

# Miscellaneous

## MTP — Multi-Token Prediction

Meta introduces a new training paradigm that alters the overall model architecture. This approach, called multi-token prediction, changes the traditional method by having the model predict several future words simultaneously instead of just one. At each position in a sentence, the model uses multiple prediction pathways, or “heads,” to forecast the next several words all at once, working collaboratively to improve efficiency and coherence.

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

In simple terms, we modify the LLM to predict the next four words instead of just the next one. To achieve this, we add more output heads to the model.

However, this doesn’t mean we predict 16 tokens in total. Each of the four heads produces four tokens, but we only use the first token from each head, discarding the rest (denoted as words 5, 6, 7, and 8 in the explanation).

The core idea of multi-token prediction is to train the model to predict a sequence of future words from each position in the training data, rather than just the next word. This method uses a shared underlying structure called a transformer trunk to understand the context and then employs multiple independent prediction heads to guess future words in parallel.

## If we trained BERT on sequences of 512 tokens, what will happen if we input text with 1024 tokens into the model?

* If BERT is trained on sequences of 512 tokens, inputting 1024 tokens will exceed the model's input limit. The text will either be truncated to 512 tokens, or you might need to split the input into smaller segments and process them separately.
* If truncation is enabled, the model will automatically cut off the input at the maximum token length (512 tokens). This means any tokens beyond that limit will be discarded, and only the first 512 tokens will be processed.

 **Increase the model's maximum token length**: You can fine-tune a version of BERT (or use a larger model like Longformer, BigBird, etc.) that supports longer sequences, such as 1024 tokens. This would allow you to process longer texts without truncation.

 **Split the input into smaller chunks**: If the model can’t handle longer sequences, you can break the 1024-token input into two parts (e.g., the first 512 tokens and the next 512 tokens), process them separately, and then combine the results, depending on the task.

 **Use sliding window technique**: This involves breaking the input into overlapping chunks (e.g., 512 tokens with a 256-token overlap) and then processing each chunk, allowing the model to capture more context across the sequence.

## Different Types of RAM

**What is RAM?**

Random Access Memory, is a type of computer memory that allows data to be read and written randomly, meaning that the computer can access any location in the memory directly rather than having to read the data in a specific order. This makes RAM an essential component of a computer system, as it enables the CPU to access data quickly and efficiently.

RAM is volatile in nature, which means if the power goes off, the stored information is lost. RAM is used to store the data that is currently processed by the CPU. Most of the programs and data that are modifiable are stored in RAM.

Mainly RAM have 2types

* SRAM (Static RAM)
* DRAM (Dynamic RAM)

### **SRAM Memory Cell**

Static memories(SRAM) are memories that consist of circuits capable of retaining their state as long as power is on. Thus this type of memory is called [**volatile memory**](https://www.geeksforgeeks.org/what-is-volatile-memory/). The below figure shows a cell diagram of SRAM. A latch is formed by two inverters connected as shown in the figure. Two transistors T1 and T2 are used for connecting the latch with two-bit lines. The purpose of these [transistors](https://www.geeksforgeeks.org/what-is-transistor/) is to act as switches that can be opened or closed under the control of the word line, which is controlled by the address decoder. When the word line is at 0-level, the transistors are turned off and the latch remains its information. SRAM does not require refresh time. For example, the cell is at state 1 if the logic value at point A is 1 and at point, B is 0. This state is retained as long as the word line is not activated.

### What is DRAM?

DRAM stores the binary information in the form of electric charges applied to capacitors. The stored information on the capacitors tends to lose over a period of time and thus the capacitors must be periodically recharged to retain their usage. DRAM requires refresh time. The main memory is generally made up of DRAM chips.

**Difference Between SRAM and DRAM**

The below table lists some of the differences between SRAM and DRAM.

| **SRAM** | **DRAM** |
| --- | --- |
| SRAM stands for Static Random Access Memory. | DRAM stands for Dynamic Random Access Memory. |
| Uses a flip-flop circuit to store data | Uses a capacitor and a transistor to store data |
| SRAM has a lower access time, so it is faster compared to DRAM. | DRAM has a higher access time, so it is slower than SRAM. |
| SRAM has long data life. | DRAM has short data life. |
| SRAM has a storage capacity of 1 MB to 16 MB in most cases. | DRAM, which is often found in tablets and smartphones, has a capacity of 1 GB to 2 GB |
| SRAM is costlier than DRAM. | DRAM costs less compared to SRAM. |
| SRAM provides faster speed of data read/write. | DRAM provides slower speed of data read/write. |
| SRAM requires a constant power supply, which means this type of memory consumes more power. | DRAM offers reduced power consumption due to the fact that the information is stored in the capacitor. |
| Good choice for applications that may be exposed to extreme temperatures. | Not suitable for such applications. |
| Due to complex internal circuitry, less storage is available compared to the same physical size of a DRAM memory chip. | Due to the small internal circuitry in the one-bit memory cell of DRAM, a large storage capacity is available. |
| SRAM has low packaging capacity. | DRAM has a high packaging density. |
| SRAM is used in cache memories. | DRAM is used in main memories. |
| SRAM does not require refresh time. | DRAM requires periodic refresh time. |
| SRAMs are used as cache memory in computer and other computing devices. | DRAMs are used as main memory in computer systems. |

## Matrix-Matrix Multiplication

GEMMs (General Matrix Multiplications) are a fundamental building block for many operations in neural networks, for example fully-connected layers, recurrent layers such as RNNs, LSTMs or GRUs, and convolutional layers. In this guide, we describe GEMM performance fundamentals common to understanding the performance of such layers.

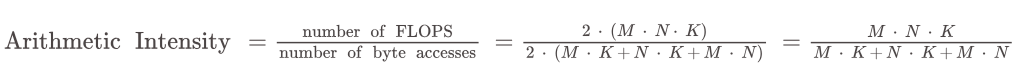
GEMM is defined as the operation C = αAB + βC, with *A* and *B* as matrix inputs, α and β as scalar inputs, and *C* as a pre-existing matrix which is overwritten by the output. A plain matrix product *AB* is a GEMM with α equal to one and β equal to zero. For example, in the forward pass of a fully-connected layer, the weight matrix would be argument *A*, incoming activations would be argument *B*, and α and β would typically be 1 and 0, respectively. β can be 1 in some cases, for example, if we’re combining the addition of a skip-connection with a linear operation.

## Math And Memory Bounds

Following the convention of various linear algebra libraries (such as BLAS), we will say that matrix A is an M x K matrix, meaning that it has M rows and K columns. Similarly, B and C will be assumed to be K x N and M x N matrices, respectively.

The product of A and B has M x N values, each of which is a dot-product of K-element vectors. Thus, a total of M \* N \* K fused multiply-adds (FMAs) are needed to compute the product. Each FMA is 2 operations, a multiply and an add, so a total of 2 \* M \* N \* K FLOPS are required. For simplicity, we are ignoring the α and β parameters for now; as long as K is sufficiently large, their contribution to arithmetic intensity is negligible.

To estimate if a particular matrix multiply is math or memory limited, we compare its arithmetic intensity to the ops:byte ratio of the GPU, as described in [Understanding Performance](https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf). Assuming an NVIDIA® V100 GPU and Tensor Core operations on FP16 inputs with FP32 accumulation, the FLOPS:B ratio is 138.9 if data is loaded from the GPU’s memory.



As an example, let’s consider a M x N x K = 8192 x 128 x 8192 GEMM. For this specific case, the arithmetic intensity is 124.1 FLOPS/B, lower than V100’s 138.9 FLOPS:B, thus this operation would be memory limited. If we increase the GEMM size to 8192 x 8192 x 8192 arithmetic intensity increases to 2730, much higher than FLOPS:B of V100 and therefore the operation is math limited. In particular, it follows from this analysis that matrix-vector products (general matrix-vector product or GEMV), where either M=1 or N=1, are always memory limited; their arithmetic intensity is less than 1.

It is worth keeping in mind that the comparison of arithmetic intensity with the ops:byte ratio is a simplified rule of thumb, and does not consider many practical aspects of implementing this computation (such as non-algorithm instructions like pointer arithmetic, or the contribution of the GPU’s on-chip memory hierarchy).

## Memory Hierarchy

In GPU programming (e.g., with CUDA or OpenCL), **memory hierarchy** is a key concept for optimizing performance. The three most important types of memory are **global memory**, **shared memory**, and **local memory**. Each has different characteristics in terms of **scope**, **latency**, **bandwidth**, and **visibility**.

**1. Global Memory**

* **Scope:** Accessible by all threads across all blocks.
* **Location:** On-device DRAM (external to cores, large size).
* **Latency:** High (400–600 cycles).
* **Speed:** Slowest among the three.
* **Use Case:** For storing data that must be shared across blocks or persist between kernel launches.
* **Optimization Tip:** Access coalescing (threads accessing contiguous addresses) is essential for performance.

**2. Shared Memory**

* **Scope:** Shared among all threads within the same block.
* **Location:** On-chip (close to cores, small size, e.g., 48KB/block).
* **Latency:** Low (comparable to registers or L1 cache).
* **Speed:** Much faster than global memory.
* **Use Case:** For intra-block data sharing and caching global memory values to reduce access latency.
* **Optimization Tip:** Avoid bank conflicts by aligning memory access.

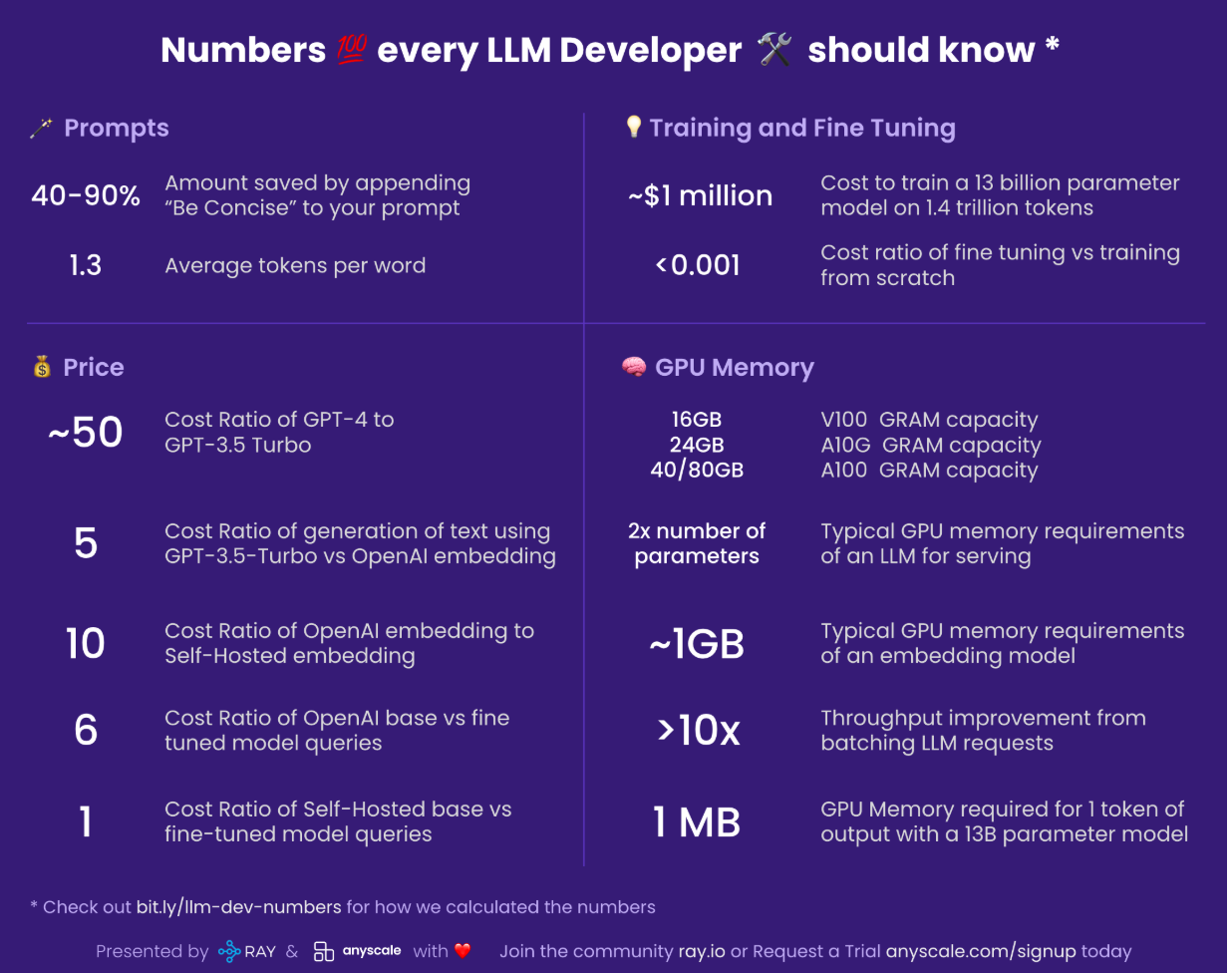
**3. Local Memory**

* **Scope:** Private to each thread.
* **Name Confusion:** Despite the name, it is not on-chip like registers.
* **Location:** Typically stored in **global memory** if register space is insufficient.
* **Latency:** Same as global memory (high).
* **Use Case:** Automatic variables that don't fit into registers (e.g., large arrays per thread).
* **Optimization Tip:** Minimize usage; prefer registers if possible.

# Links

## Links

* **Building effective agents:** <https://www.anthropic.com/research/building-effective-agents>
* **Efficient Deep Learning: A Comprehensive Overview of Optimization Techniques 👐 📚:**
* [**https://huggingface.co/blog/Isayoften/optimization-rush**](https://huggingface.co/blog/Isayoften/optimization-rush)
* **Numbers every LLM Developer should know:** [**https://github.com/ray-project/llm-numbers?tab=readme-ov-file#1-mb-gpu-memory-required-for-1-token-of-output-with-a-13b-parameter-model**](https://github.com/ray-project/llm-numbers?tab=readme-ov-file#1-mb-gpu-memory-required-for-1-token-of-output-with-a-13b-parameter-model)

****

* **LLM Deployment: Best Practices and Tips:** [**https://medium.com/arize-ai/best-practices-for-llm-deployment-81937af82a57**](https://medium.com/arize-ai/best-practices-for-llm-deployment-81937af82a57)
* **Making Deep Learning Go Brrrr From First Principles:** [**https://horace.io/brrr\_intro.html**](https://horace.io/brrr_intro.html)
* **A Visual Guide to Quantization:** [**https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization**](https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization)
* **Matrix Multiplication Background User's Guide:** [**https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html**](https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html)
* **GPU Performance Background User's Guide:** [**https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf**](https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf)
* On-demand NVIDIA GPU instances & clusters for AI training & inference: <https://lambda.ai>