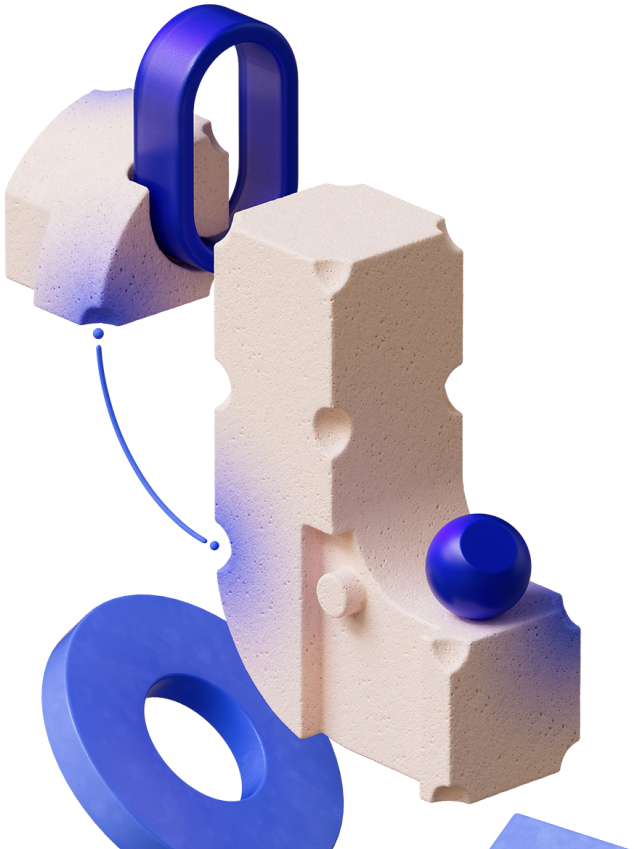
Google DeepMind



Efficiency in LLMs

Nikita Saxena
Research Engineer

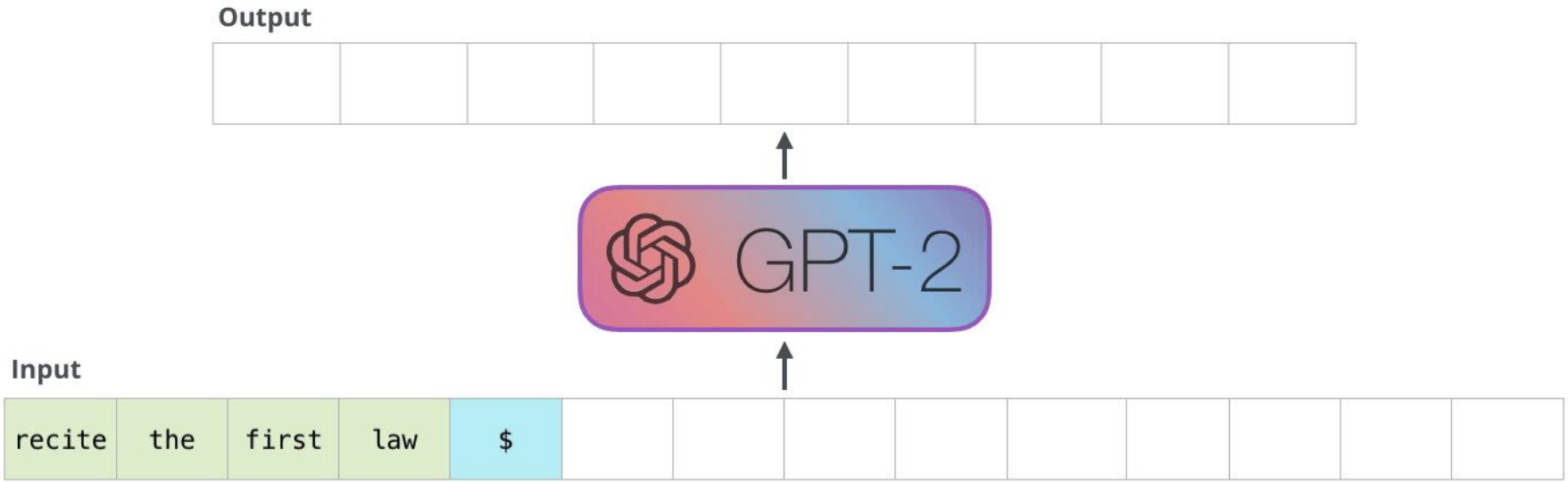
Agenda

Efficient Attention

4

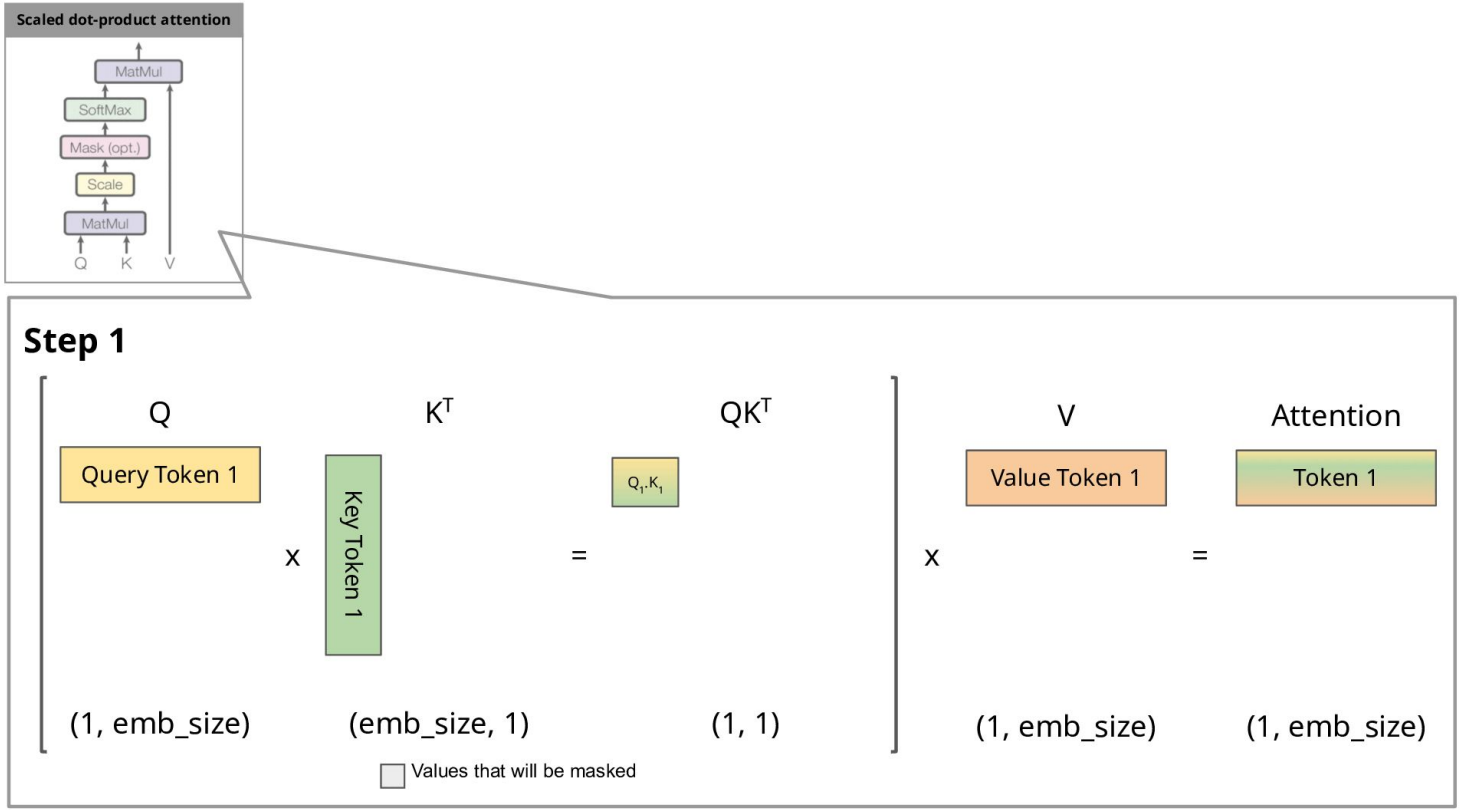
01 The KV Cache

Auto-Regressive Behaviour of Decoders



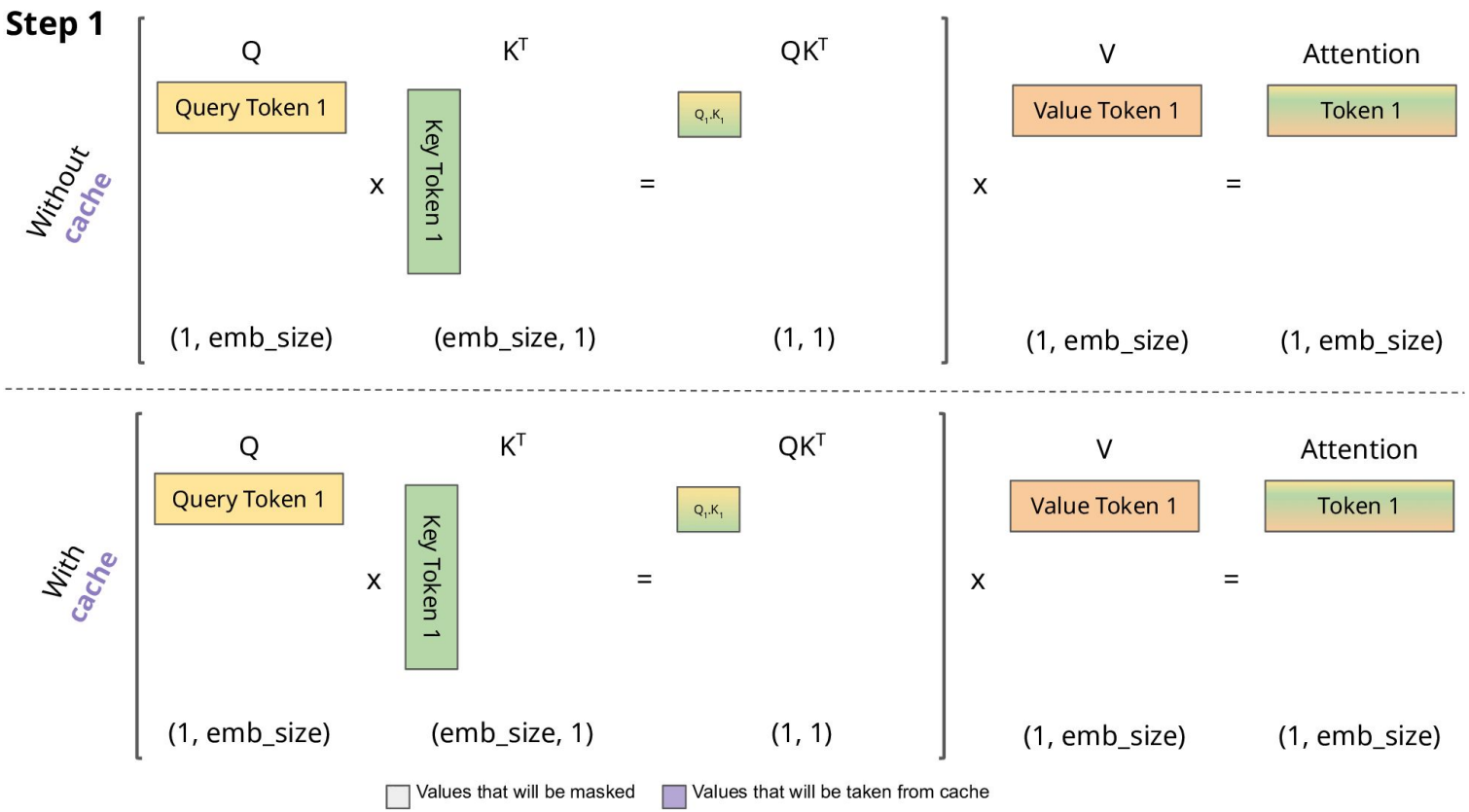
Note that at each step, all previous tokens need to be re-fed into the decoder.

Recap: Attention

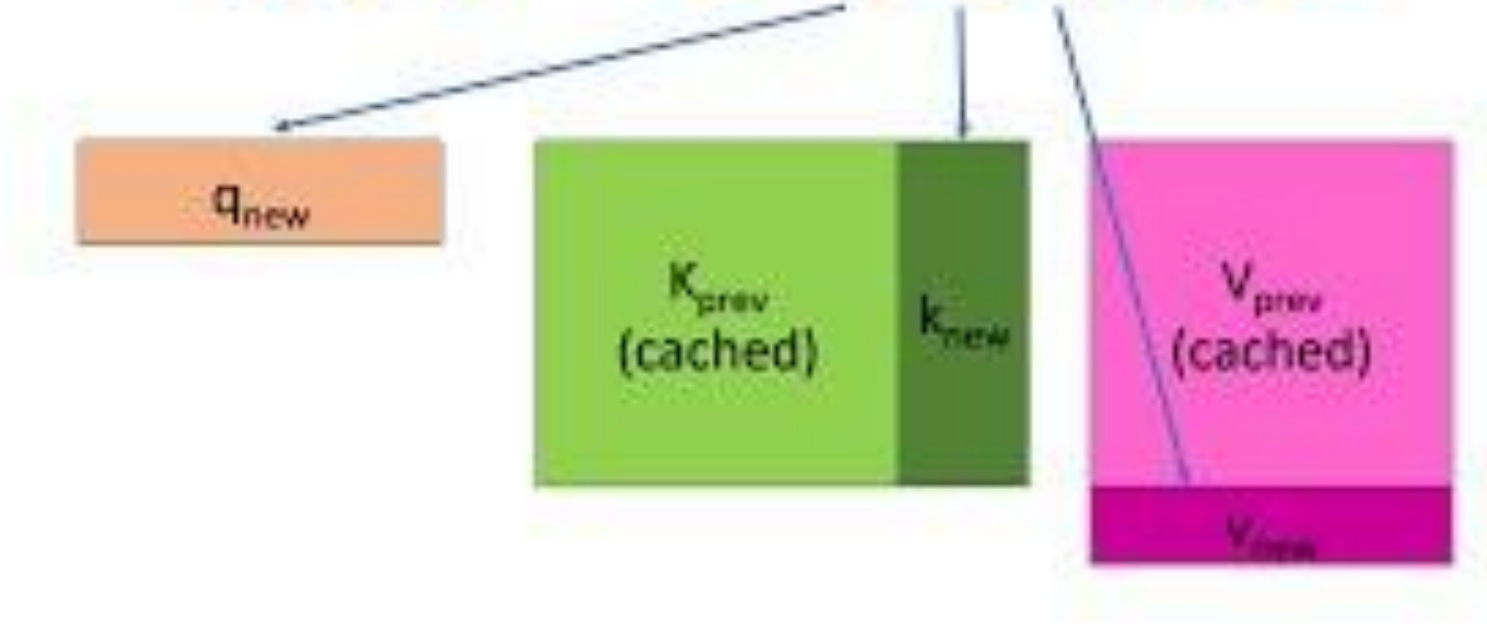


Zoom-in! (simplified without Scale and Softmax)

Introducing KV Cache



KV Cache



The Problem: The Cache is a Memory Hog

The size of the KV Cache is proportional to:

`(batch size) x (sequence length) x (model size)`

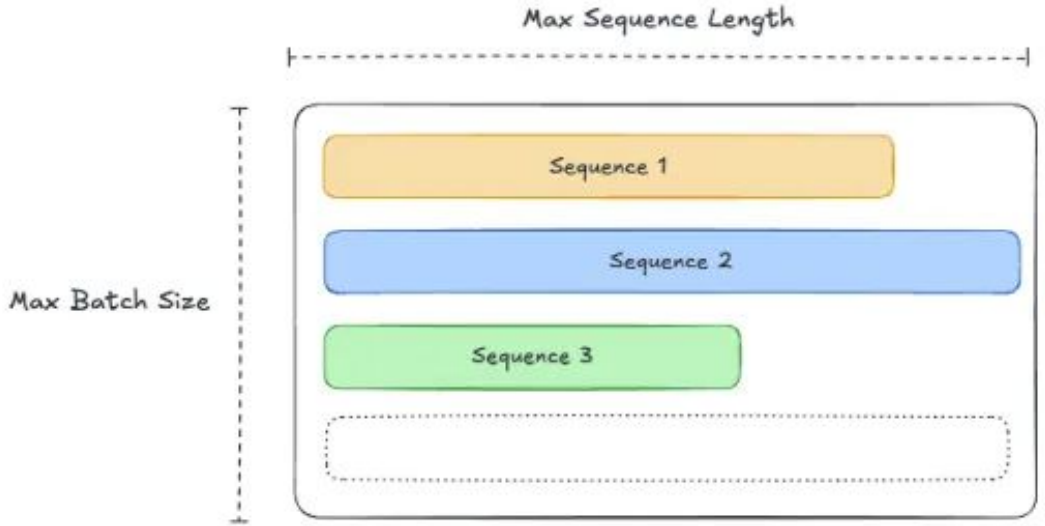
For a model like Llama 13B, a single sequence of 2048 tokens can consume over 1 GB of VRAM.

This memory is in addition to the model's weights!

Wasted Memory

To manage this growing cache, systems pre-allocate one large, contiguous block of memory for each sequence.

Example: We set max_sequence_length = 2048. The system immediately reserves 2048 tokens' worth of cache memory for every new user request.



Wasted Memory

Researchers at Stanford and Berkeley found that allocating a static KV cache for a worst-case scenario—where every incoming sequence is assumed to grow to its maximum length—can waste up to 80% of GPU memory. While this approach ensures we never trigger catastrophic out-of-memory errors, it also causes inefficient resource usage when most sequences terminate well before the maximum length.

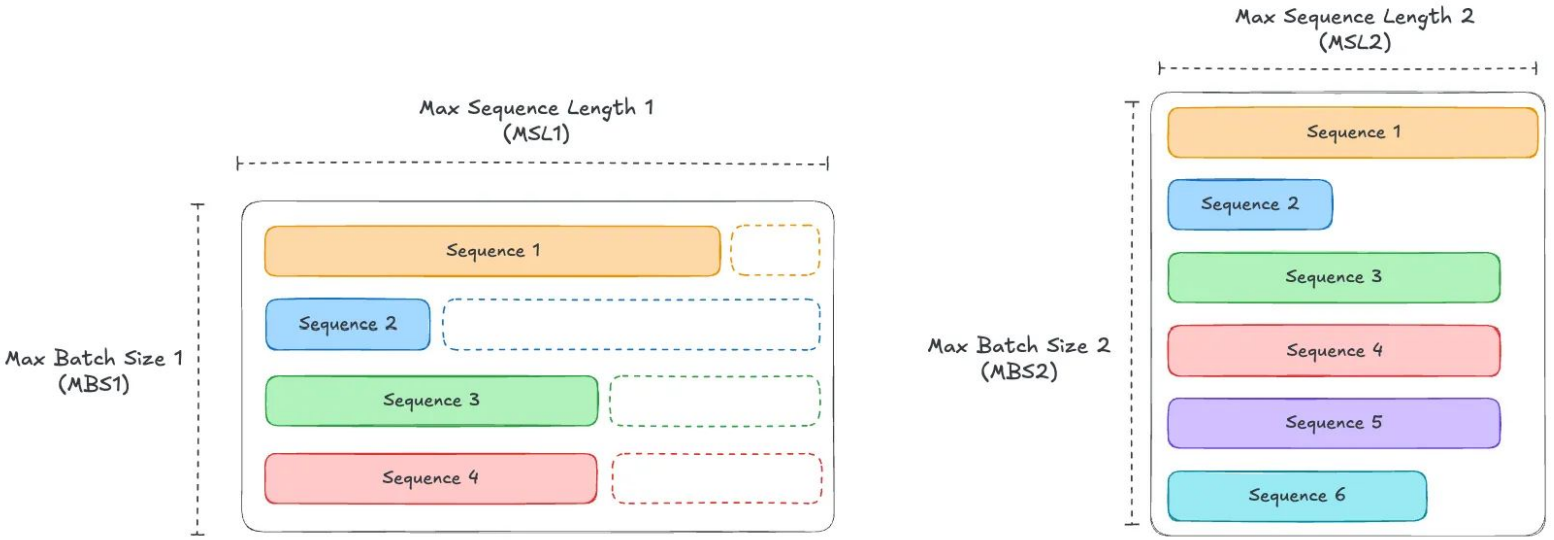
See the below prompt, and imagine setting aside 1GB of GPU memory for requests like these, only to use ~25 out of the reservation of 8192 tokens.

"It was the best of times, it was the worst of times" is the beginning to which novel?

"It was the best of times, it was the worst of times" is the famous opening line of **"A Tale of Two Cities"** by **Charles Dickens**.

Dynamically Trading Sequence Length

This inefficiency is illustrated on the left side of the below figure; for an application where high request concurrency is important, the right side shows a more optimal reorganization of the cache. Instead of changing the total cache size, we simply trade off some maximum sequence length capacity to accommodate more sequences at once, thereby improving throughput.

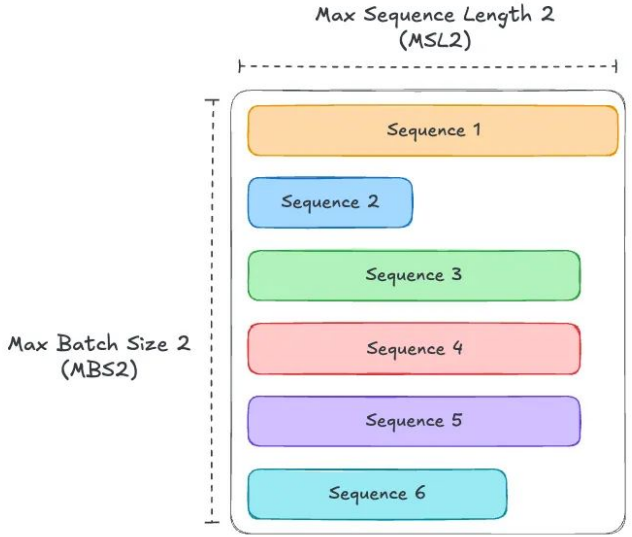


Dynamically Trading Sequence Length

While the concept of dynamically trading sequence length for batch size is appealing, it becomes complex in practice when the KV cache is pre-allocated.

It's like trying to fit a new car into three small, non-adjacent parking spots. This leads to External Fragmentation.

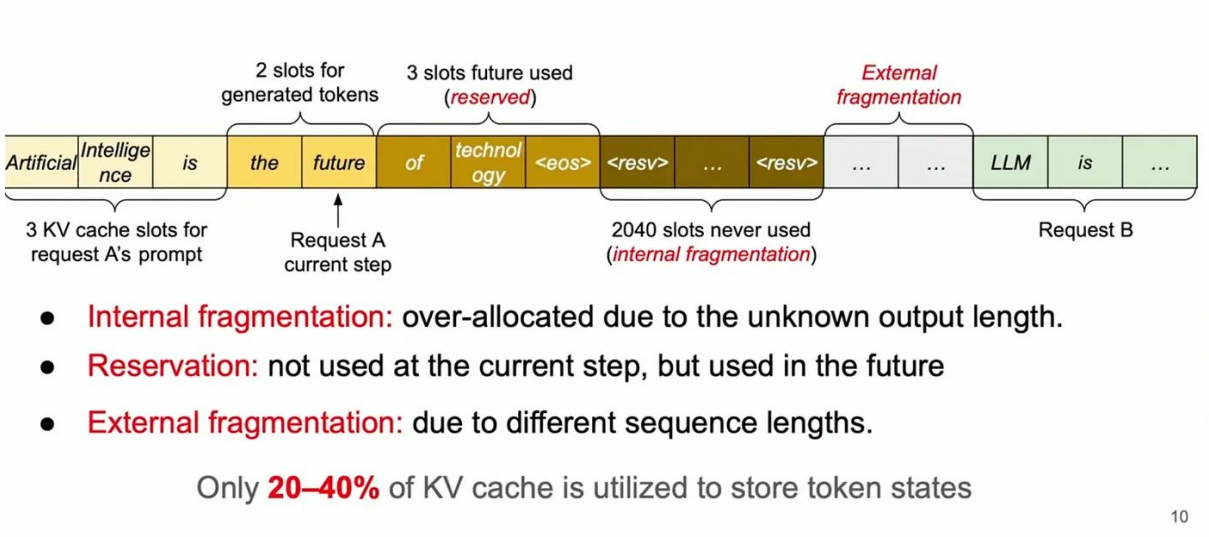
Because re-allocating this memory at runtime can be costly and risky, we need a mechanism that can efficiently manage sequences within a fixed GPU memory block.



Recap: Memory Waste in KV Cache

Internal Fragmentation
Caused by over-allocation due to unknown output length.
Solution: More precise output length estimation or dynamic allocation strategies

Reservation Waste
Memory reserved for future token generation.
Solution: Better prediction of required future slots



External Fragmentation
Results from handling multiple requests with different sequence lengths. Creates memory gaps between different requests.
Solution: Memory defragmentation and intelligent request batching

02

Paged Attention

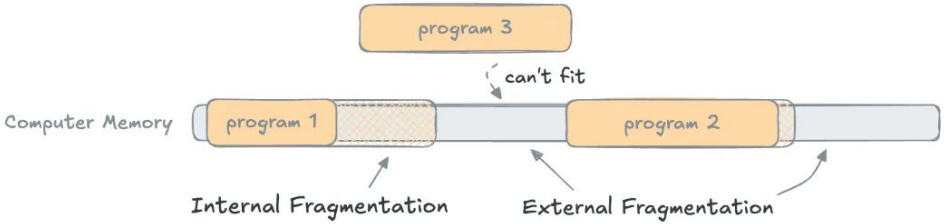
Memory Fragmentation

Memory Fragmentation refers to the condition where available memory becomes broken into numerous small, noncontiguous blocks, making it difficult or impossible to allocate large contiguous memory segments even if sufficient total free space exists.

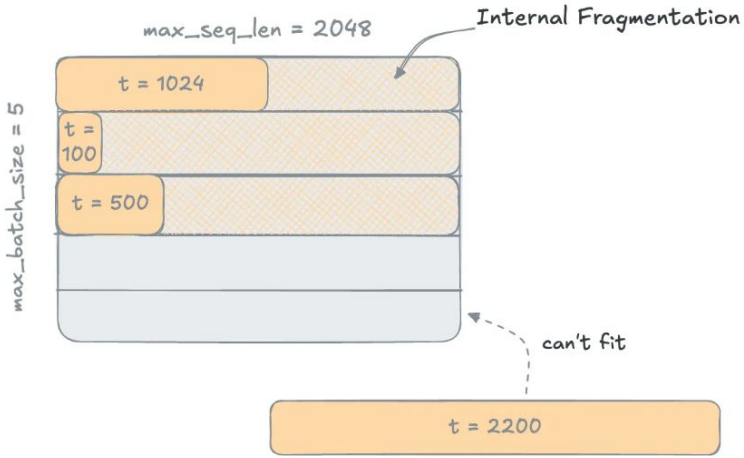
1. Internal Memory Fragmentation: occurs when programs are given all the memory they will potentially need before starting. They may or may not use this memory eventually, but until they use it or the program ends, it is reserved and idle.

2. External Memory Fragmentation: when programs are allocated and deallocated in contiguous memory, gaps of memory are left as they are evicted (see Fig 8, left side). Our restriction on contiguous memory means that future programs may not be able to fit in available gaps, despite the total free memory available being enough to accommodate them. As we don't know the lifetime and therefore order of termination of our programs in advance, it is difficult to pack our program memory to avoid these gaps from occuring.

Memory Fragmentation



a) In operating systems



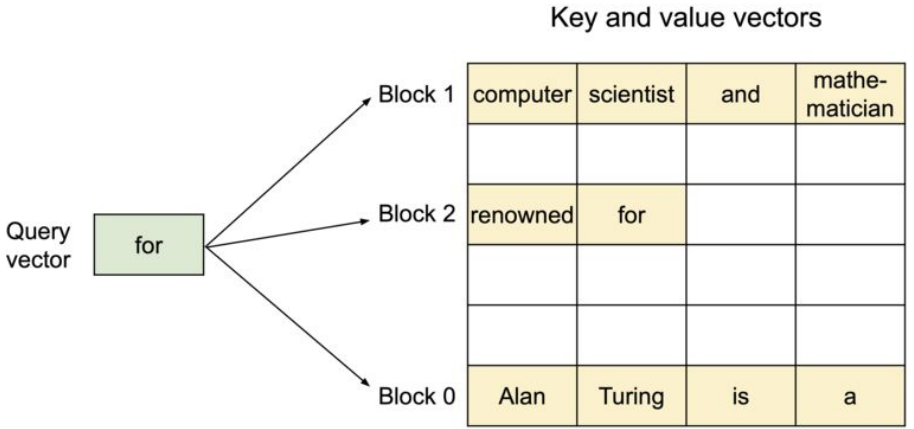
b) In our KV Cache

Implementing Paged Attention with the KV Cache

Unlike the traditional attention algorithms, PagedAttention allows storing continuous keys and values in non-contiguous memory space.

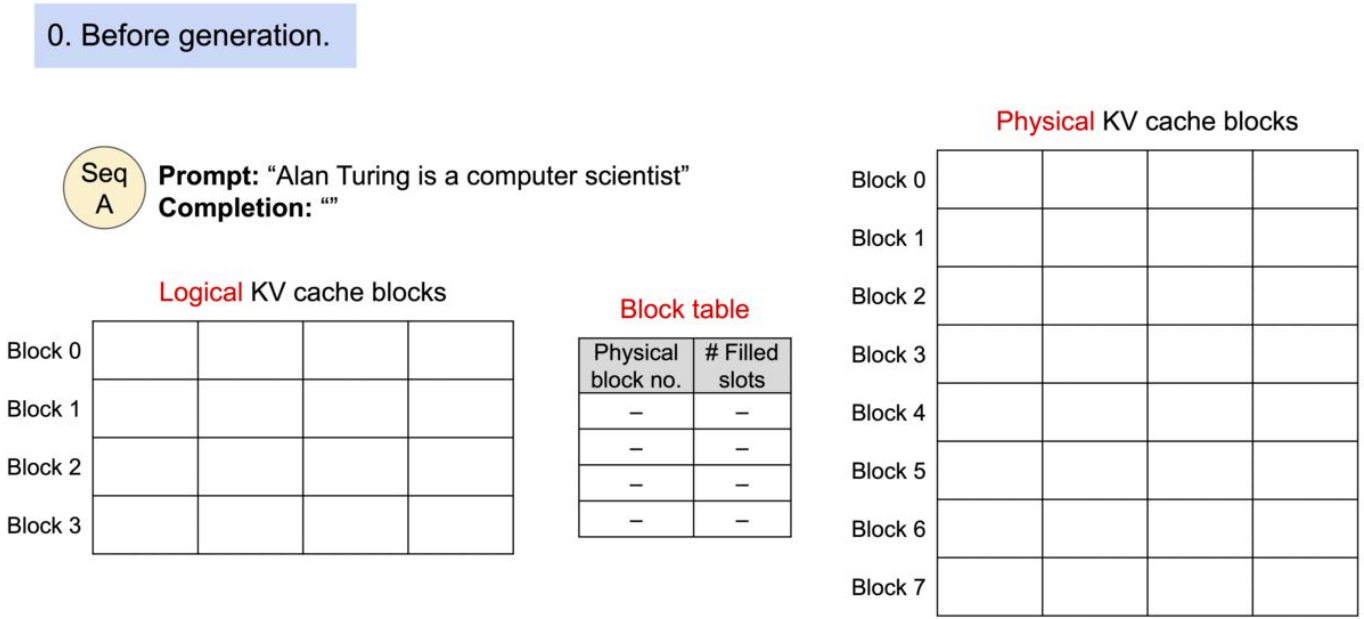
Specifically, PagedAttention partitions the KV cache of each sequence into blocks, each block containing the keys and values for a fixed number of tokens.

During the attention computation, the PagedAttention kernel identifies and fetches these blocks efficiently.



To implement this approach, we reorder the KV cache dimensions to:

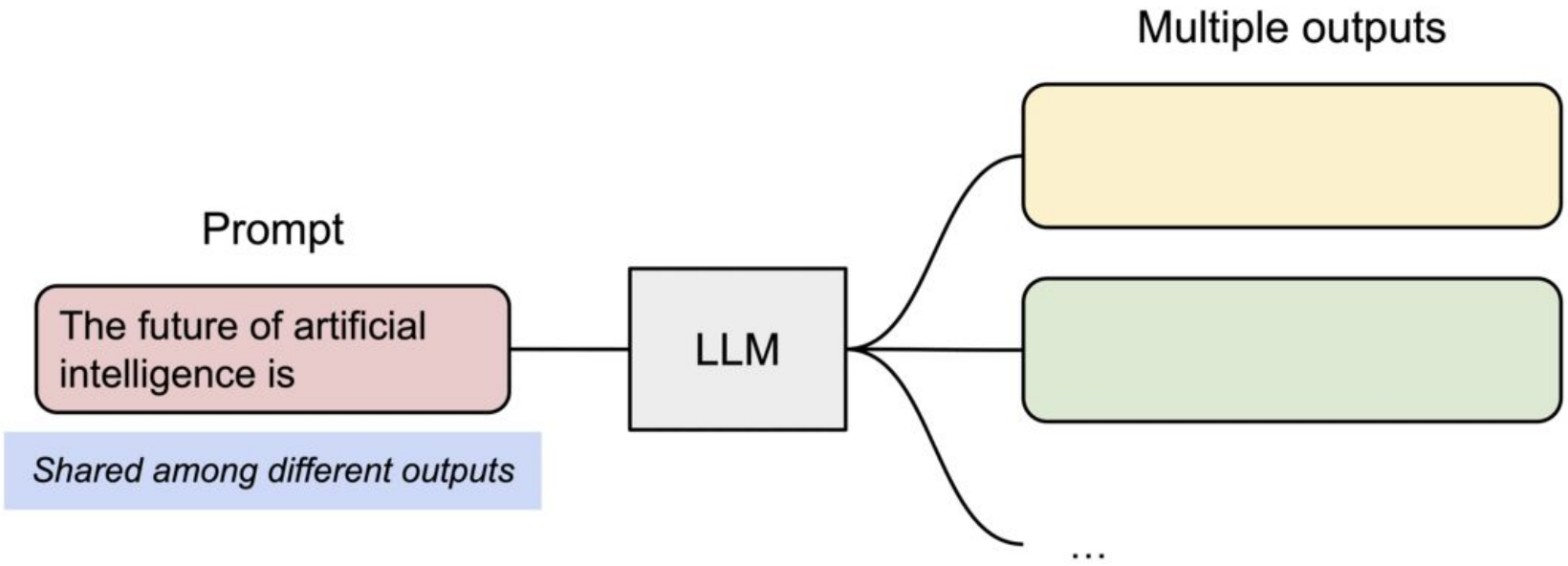
```
[
    total_blocks
    ×block_size
    ×hidden_dims
].
```



For each sequence, we construct a Block Table that tracks which blocks in the paged KV cache belong to which sequence. A scheduler then manages these sequence fragments—allocating new blocks as needed—to efficiently handle growing sequences.

Efficient Memory Sharing: Copy on Write

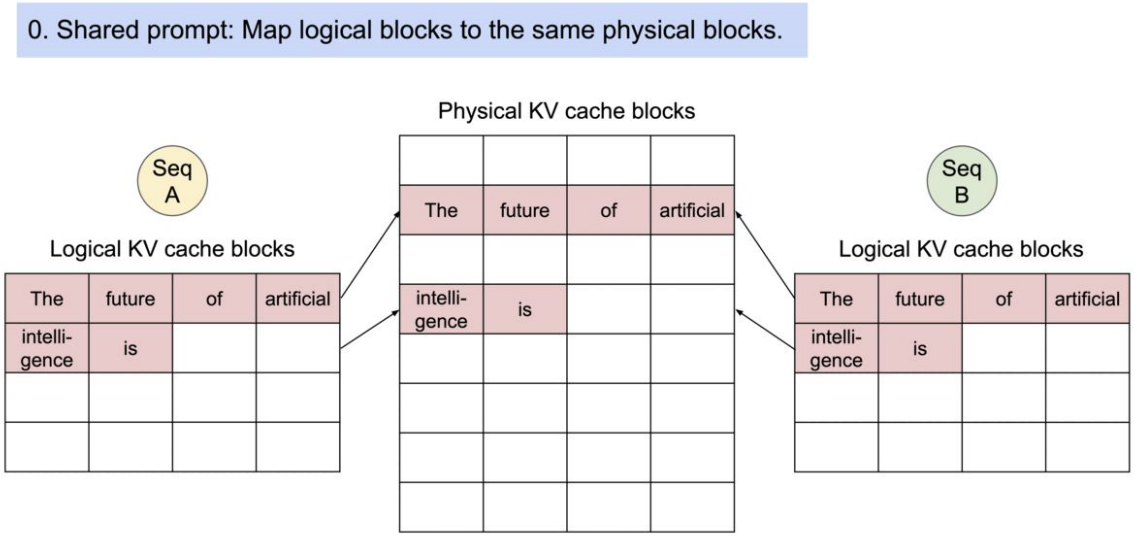
In KV cache, in the case of parallel sampling (where we generate multiple possible outputs from the same prompt), the prompt "The future of artificial intelligence is" would be duplicated three times in memory, one for each generation. This is extremely wasteful.



Efficient Memory Sharing: Copy on Write

Mechanism

1. **Sharing**: Multiple sequences can have their block tables point to the same physical memory blocks for the shared prompt portion.
2. **Copy-on-Write**: The moment a sequence generates a new, unique token, a new block is allocated for it, and its block table is updated. The original shared blocks remain untouched.



Impact

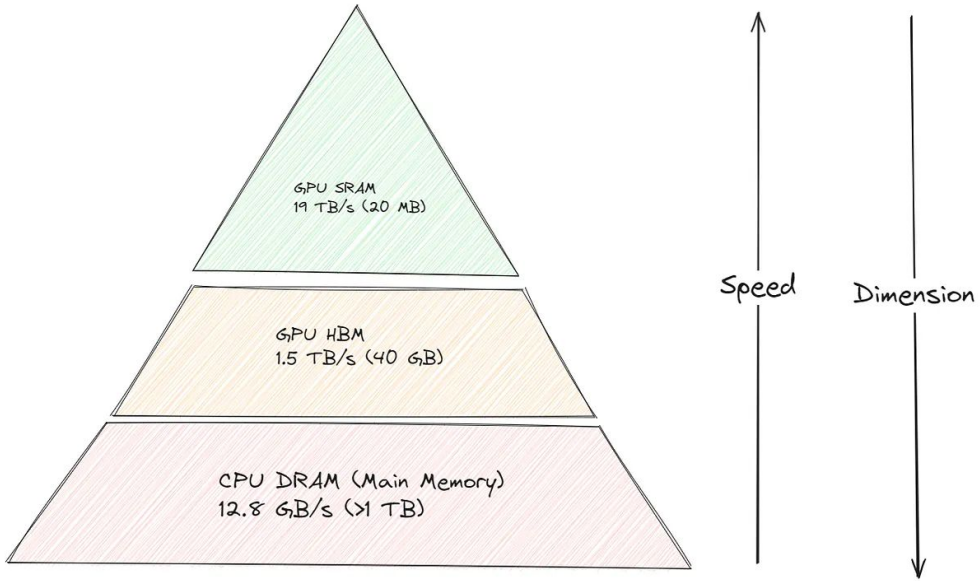
1. **Massive Memory Savings**
   For complex search algorithms like beam search or parallel sampling, memory usage for the prompt is reduced by a factor of the number of parallel generations.
2. **Up to 22x Higher Throughput**
   By combining the elimination of fragmentation with efficient memory sharing, PagedAttention allows for much larger batch sizes, dramatically improving GPU throughput.

02

Flash Attention

GPU Hierarchy

Before diving into FlashAttention, let's understand the GPU memory hierarchy, as this is key to appreciating why FlashAttention works so well



Memory Type	Location	Speed & Size	Primary Use
SRAM (Static Random-Access Memory)	Inside GPU cores (registers and shared memory)	Very fast, but very small (~KBs per core)	Temporary computation values during kernel execution
HBM (High Bandwidth Memory)	L2 Cache and high-speed memory near GPU cores	Extremely fast (~TB/s bandwidth)	Frequently accessed tensors, reducing memory transfer overhead (e.g., activations)
DRAM (Global Memory - HBM2/GDDR)	Typically off-chip	Largest size, slowest speed (~GB/s bandwidth)	Model parameters, large input tensors, and activations

The Standard Attention Method (The Inefficient Chef)

1. Chef walks to the warehouse (HBM) to get the entire Q matrix.
2. Chef walks back to the workbench (SRAM).
3. Chef walks to the warehouse to get the entire K matrix.
4. Chef walks back.
5. Chef computes S = Q @ K.T on the workbench.
6. Chef writes the entire intermediate S matrix back to the warehouse (HBM). (This is the key inefficiency).
7. Chef walks to the warehouse to get S.
8. Chef walks to the warehouse to get V.
9. Chef computes Softmax and final result O.

Too many trips to the warehouse.

The massive intermediate result S doesn't fit on the workbench and must be stored!

The FlashAttention Method (The Efficient Chef)

1. Chef fetches a small block of Q and a small block of K from the warehouse (HBM). (These small blocks fit on the workbench).
2. Chef computes the partial result for just that block on the workbench (SRAM).
3. Chef fetches the corresponding small block of V.
4. Chef computes the partial final output for that block without ever writing the intermediate S block back to the warehouse.
5. Repeat for all blocks, intelligently combining the partial results on-chip.

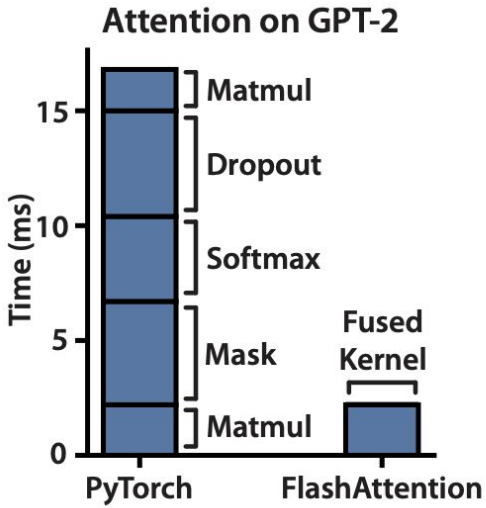
Work on small tiles that fit on the workbench.

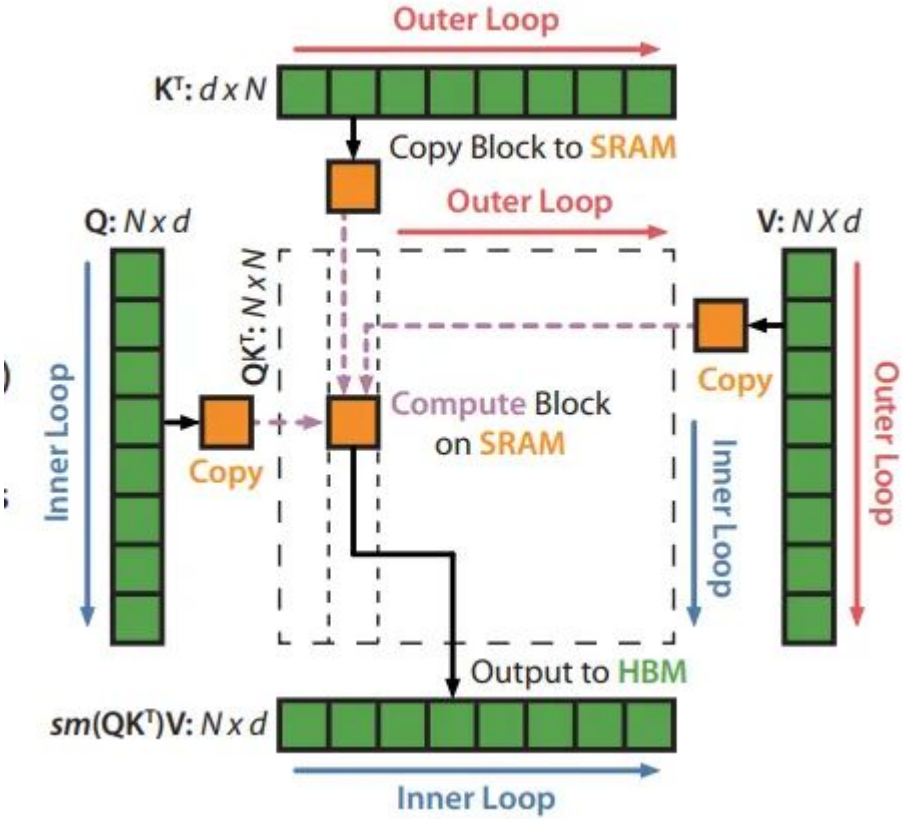
Never write intermediate results back to the slow warehouse


GPUs go brrrrrrrr
FLASH

FlashAttention: Key Techniques

1. Tiling and Block-Sparse Attention
2. Online Softmax
3. Fused CUDA Kernels





Tiling and Block-Sparse Attention

Instead of computing the entire N×N attention matrix at once, FlashAttention divides it into smaller tiles or blocks

Standard Attention:

```
Compute full QK^T matrix (N×N)

Store entire matrix in HBM/DRAM
```

FlashAttention:

```
Divide Q, K, V into blocks of size B

Process blocks sequentially in fast shared memory

Never materialize the full N×N matrix
```

Online Softmax: Computing Block by Block

Rather than waiting until all logits are in, it does a rolling calculation using just the current block.

For each row i:

1. Track the current maximum value m_i
2. Track the current sum of exponentials l_i
3. For each new block of logits x_i^t
   a. Update max:

$$m_i^{new} = \max(m_i, \max(x_i^t))$$

   b. Update sum:

$$l_i^{new} = l_i \exp(m_i - m_i^{new}) + \sum\{\exp(x_i^t - m_i^{new})\}$$

4. Normalize partial outputs:

$$\text{softmax}(x_i^t) = \exp(x_i^t - m_i^{new}) / l_i^{new}$$

```
Block 1: [1.0, 2.0]

max_1 = 2.0

sum_1 = exp(1.0-2.0) + exp(2.0-2.0) = 0.368 + 1.0 = 1.368



Block 2: [3.0, 0.5]

max_2 = max(2.0, 3.0) = 3.0

sum_2 = 1.368 * exp(2.0-3.0) + exp(3.0-3.0) + exp(0.5-3.0)
= 0.503 + 1.0 + 0.082 = 1.585
```

With online softmax:

```
[exp(1.0-3.0)/1.585, exp(2.0-3.0)/1.585,
exp(3.0-3.0)/1.585, exp(0.5-3.0)/1.585] = [0.050, 0.137,
0.631, 0.052]
```

Without online softmax:

```
softmax([1.0,2.0,3.0,0.5] )=: [0.085, 0.232, 0.631, 0.052]
```

Fused CUDA Kernels

FlashAttention merges several GPU operations into one kernel. No more unnecessary memory shuffling.

Each kernel launch incurs overhead, and data must move between GPU global memory and registers. Fusing these operations eliminates these inefficiencies.

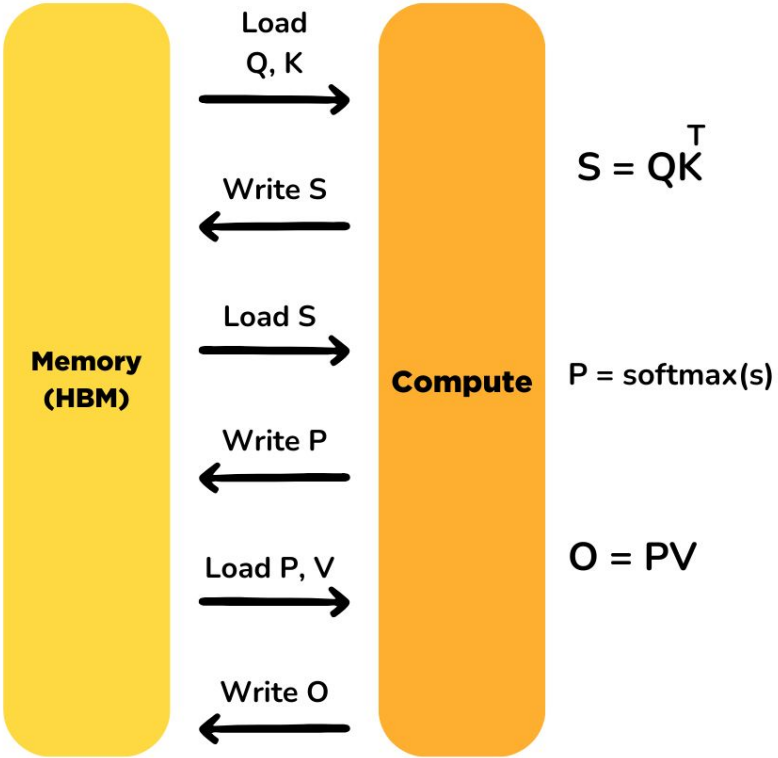
Standard implementation (multiple kernel launches)

```
scores = torch.matmul(Q, K.transpose(-2, -1))  #
Kernel 1

scores = scores / math.sqrt(d)  # Kernel 2

attn = F.softmax(scores, dim=-1)  # Kernel 3

output = torch.matmul(attn, V)  # Kernel 4
```

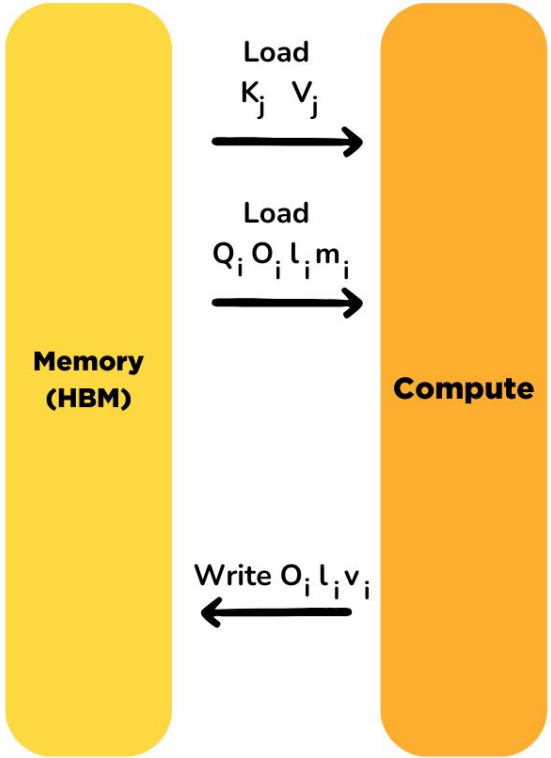
FlashAttention (single fused kernel)

```
output = flash_attention(Q, K, V)  # All operations
in one kernel!
```

Standard Attention Implementation



Flash Attention

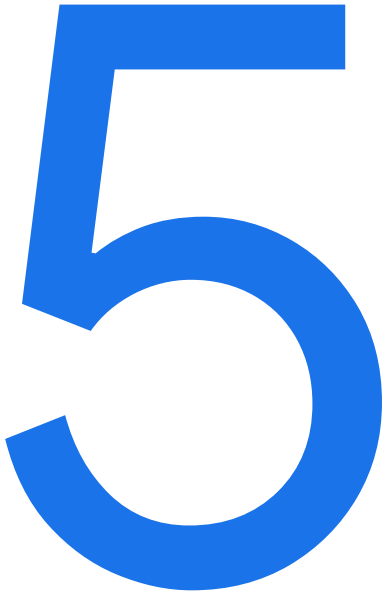


Kernel operations fused together, reducing reads & writes

$S_{ij} = Q_i K_j^T$
$\tilde{m}$ = rowmax of S
P = exp(s - m)
l = rowsum of P
$m = \max(\tilde{m}_{ij}, m)$
calculate O from l & m

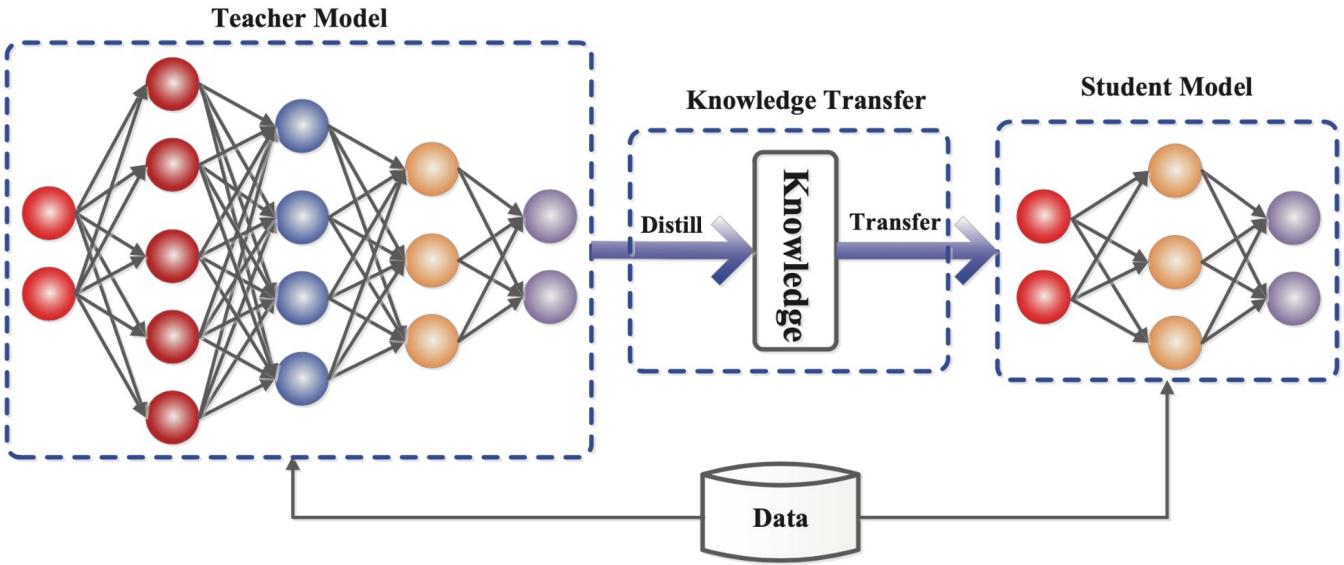
Initialize O, l and m matrices with zeroes. m and l are used to calculate cumulative softmax. Divide Q, K, V into blocks (due to SRAM's memory limits) and iterate over them, for i is row & j is column.

Training Method: Knowledge Distillation



Introduction

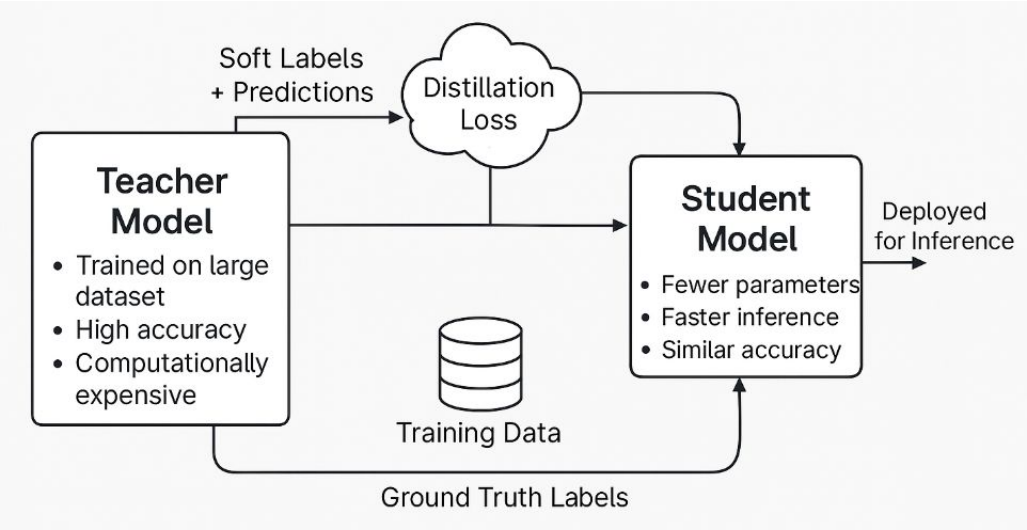
Model distillation operates by transferring knowledge from a large, complex language model—the "teacher"—to a smaller, more efficient model known as the "student."



Introduction

Distillation was referred to the process of transferring knowledge from a large, complex AI model or ensemble to a smaller, faster AI model, called the distilled model.

Instead of just training the smaller model on correct answers, researchers proposed to give it the probability distribution from the large model. **This helps the smaller model learn not just what the right answer is, but also how confident the big model is about each option.** This training concept is closely connected to the softmax function.



Step-by-Step Overview

Step 1: Choose Your Models

- The Teacher is powerful but slow/expensive.
- The Student is small, fast, and un-trained.

Step 2: Prepare a Transfer Dataset

- This is the "curriculum" or "textbook" for the Student.
- It should be high-quality and representative of the task the Student needs to learn (e.g., summarization, sentiment analysis).

Step 3: Generate Teacher's "Soft Labels"

- Feed the transfer dataset through the frozen Teacher model.
- Instead of just saving the final answer (the "hard label," e.g., "Positive"), we save the Teacher's full probability distribution over all possible outputs. These are the "soft labels."
- Example: {"Positive": 0.9, "Negative": 0.09, "Neutral": 0.01}. This reveals the Teacher "thinks" the sentence is overwhelmingly positive, but with a tiny hint of negativity.

Step-by-Step Overview

Step 4: Train the Student

- Feed the same transfer dataset to the Student model.
- The Student's goal is to minimize a combined loss function:

  `Total Loss = α * (Distillation Loss) + (1-α) * (Student Loss)`

  - `Distillation Loss`: Match its output distribution to the Teacher's "soft labels". (Learn how the teacher thinks).
  - `Student Loss` (Optional but common): Match the true "hard labels" from the dataset. (Learn the correct answer).

The Outcome:

A small, fast Student model that has "internalized" the reasoning patterns of the much larger Teacher.

Components

I. Knowledge,
II. Distillation algorithm,
III. Teacher-student architecture

01 Knowledge

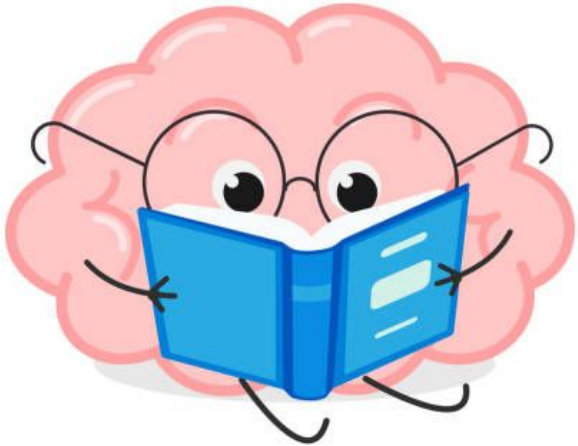
Knowledge

In a neural network, knowledge typically refers to the learned weights and biases.

At the same time, there is a rich diversity in the sources of knowledge in a large deep neural network.

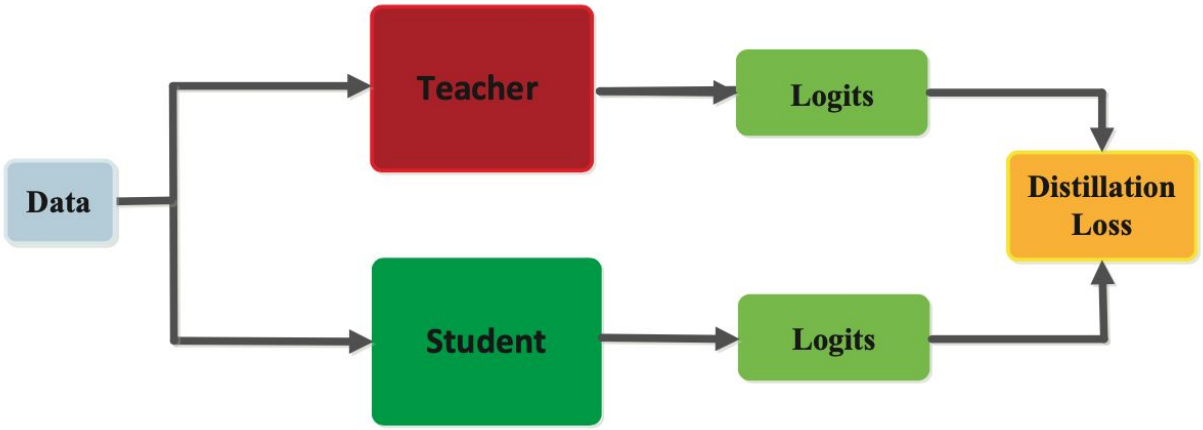
Typical knowledge distillation uses the logits as the source of teacher knowledge, whilst others focus on the weights or activations of intermediate layers.

Other kinds of relevant knowledge include the relationship between different types of activations and neurons or the parameters of the teacher model themselves.



Response-based knowledge

- Focuses on the final output layer of the teacher model.
- Hypothesis is that the student model will learn to mimic the predictions of the teacher model.



- Achieved by using a loss function, termed the distillation loss, that captures the difference between the logits of the student and the teacher model respectively. As this loss is minimized over training, the student model will become better at making the same predictions as the teacher.

Avoiding Spiky Distributions

But there's a problem. A very confident Teacher model might produce an extremely 'spiky' distribution. For example, it might be 99.99% sure the answer is 'cat', and the probabilities for all other classes are nearly zero. This distribution is not much better than a hard label; it doesn't give the Student much information about the relationships between other classes.

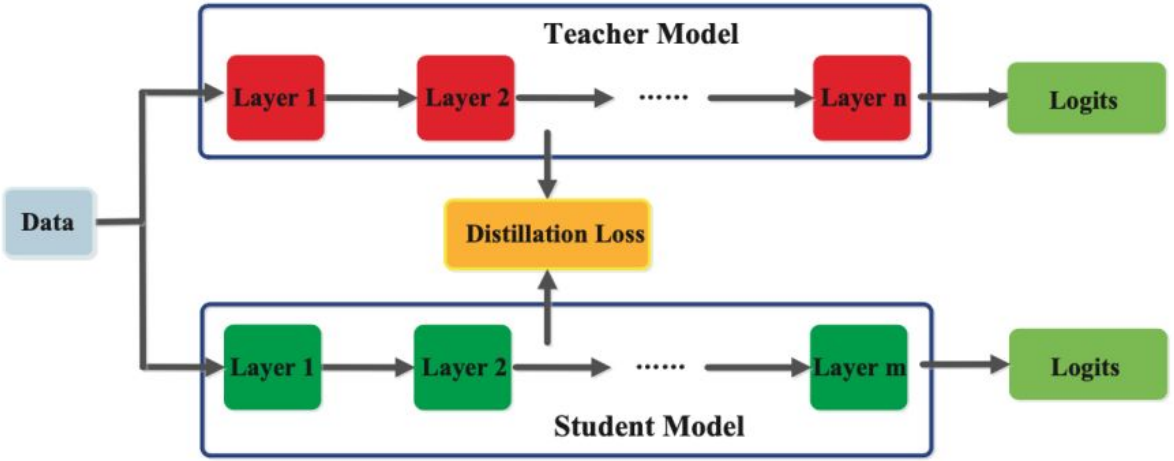
To solve this, the original paper divided the softmax logits by “temperature”.

- A high temperature (T > 1) makes the distribution softer and more uniform. It forces the Teacher to 'reveal' more of its knowledge by assigning more significant probabilities to less likely classes. This gives the Student a richer, more informative signal to learn from.
- A low temperature (T = 1) is just the standard softmax.

"So, by tuning the temperature, we can control how much 'dark knowledge' the Teacher reveals to the Student, making the distillation process more effective.

Feature-based knowledge

- A trained teacher model also captures knowledge of the data in its intermediate layers, which is especially pertinent for deep neural networks.
- The intermediate layers learn to discriminate specific features and this knowledge can be used to train a student model.

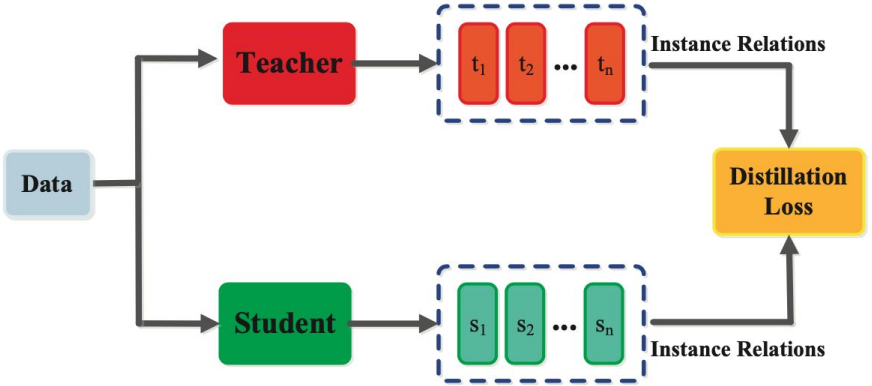


As shown in the figure, the goal is to train the student model to learn the same feature activations as the teacher model. The distillation loss function achieves this by minimizing the difference between the feature activations of the teacher and the student models

Relation-based knowledge

Knowledge that captures the relationship between feature maps can also be used to train a student model.

This relationship can be modeled as correlation between feature maps, graphs, similarity matrix, feature embeddings, or probabilistic distributions based on feature representations.



02

Distillation Algorithms

Offline

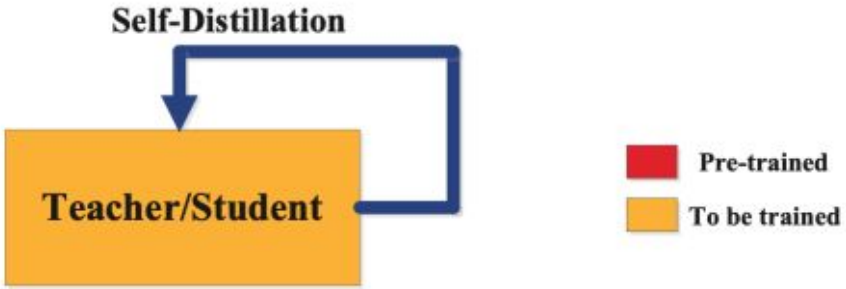
- A pre-trained teacher model is used to guide the student model.
- The teacher model is first pre-trained on a training dataset, and then knowledge from the teacher model is distilled to train the student model.

Online

- Both the teacher and student models are updated simultaneously in a single end-to-end training process.
- Online distillation can be operationalized using parallel computing thus making it a highly efficient method.

Self-Distillation

- In self-distillation, the same model is used for the teacher and the student models.
- For instance,
  - knowledge from deeper layers of a deep neural network can be used to train the shallow layers.
  - Knowledge from earlier epochs of the teacher model can be transferred to its later epochs to train the student model.



Distillation Type	Teacher Model	Training Approach	Advantages	Challenges
Offline Distillation	**Pre-trained & fixed**	Teacher is trained first, then knowledge is transferred to the student	Simple, efficient, and reusable teacher	The student may struggle to reach teacher-level performance due to a **knowledge gap**
Online Distillation	**Learns alongside student**	Teacher and student train together in real-time	Adaptive learning, **no need for a separate pre-trained teacher**	Computationally expensive and harder to set up
Self-Distillation	**No separate teacher**	The model teaches itself using its own deeper layers	No additional teacher model required, improves efficiency	Might not be as effective as learning from a stronger external teacher

03 Architecture

Architecture Design

The most common architectures for knowledge transfer include a student model that is:

- a shallower version of the teacher model with fewer layers and fewer neurons per layer,
- a quantized version of the teacher model,
- a smaller network with efficient basic operations,
- a smaller networks with optimized global network architecture,
- the same model as the teacher.

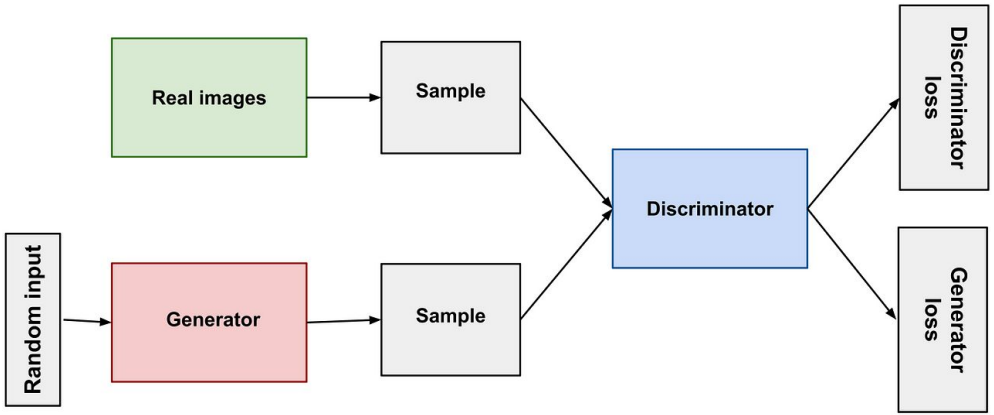
In addition to the above methods, recent advances like neural architecture search can also be employed for designing an optimal student model architecture given a particular teacher model.

04 Algorithms

Adversarial Distillation

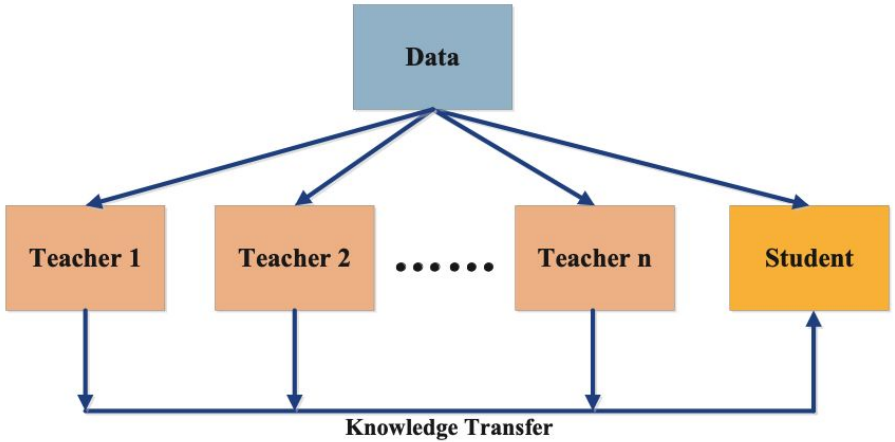
Approaches:

1. **Data Augmentation:** Train a generator model to obtain synthetic training data to use as such or to augment the original training dataset.
2. **Student Model tries to Fool the Discriminator:** Focuses on a discriminator model to differentiate the samples from the student and the teacher models based on either logits or feature maps. This method helps the student mimic the teacher well.
3. **Online Distillation:** The student and the teacher models are jointly optimized.



Multi-Teacher Distillation

- Student model acquires knowledge from several different teacher models.
- Using an ensemble of teacher models can provide the student model with distinct kinds of knowledge that can be more beneficial than knowledge acquired from a single teacher model.
- The knowledge from multiple teachers can be combined as the average response across all models.
- The type of knowledge that is typically transferred from teachers is based on logits and feature representations.
- Multiple teachers can transfer different kinds of knowledge

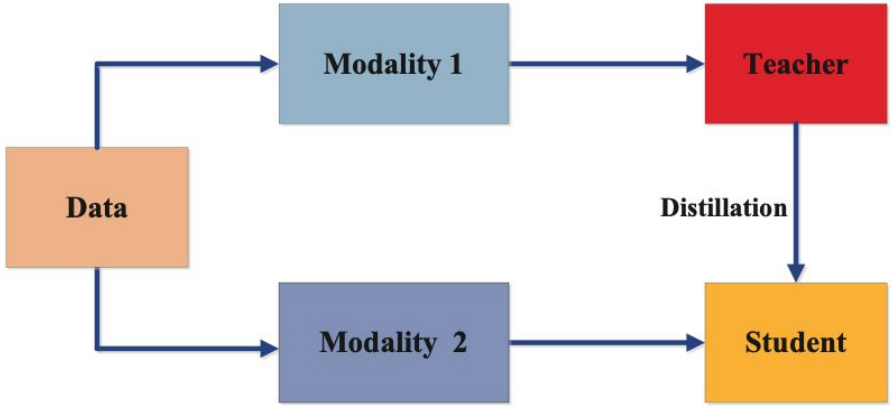


Cross-Modal Distillation

The teacher is trained in one modality and its knowledge is distilled into the student that requires knowledge from a different modality.

This situation arises when data or labels are not available for specific modalities either during training or testing thus necessitating the need to transfer knowledge across modalities.

Cross-modal distillation is used most commonly in the visual domain. For example, the knowledge from a teacher trained on labeled image data can be used for distillation for a student model with an unlabeled input domain like optical flow or text or audio.



Other

1. **Graph-based distillation** captures intra-data relationships using graphs instead of individual instance knowledge from the teacher to the student. Graphs are used in two ways – as a means of knowledge transfer, and to control transfer of the teacher's knowledge. In graph-based distillation, each vertex of the graph represents a self-supervised teacher which may be based on response-based or feature-based knowledge like logits and feature maps respectively.
2. **Attention-based distillation** is based on transferring knowledge from feature embeddings using attention maps.
3. **Data-free distillation** is based on synthetic data in the absence of a training dataset due to privacy, security or confidentiality reasons. The synthetic data is usually generated from feature representations of the pre-trained teacher model. In other applications, GANs are also used to generate synthetic training data.
4. **Quantized distillation** is used to transfer knowledge from a high-precision teacher model (e.g. 32-bit floating point) to a low-precision student network (e.g. 8-bit).
5. **Lifelong distillation** is based on the learning mechanisms of continual learning, lifelong learning and meta-learning where previously learnt knowledge is accumulated and transferred into future learning.
6. **Neural architecture search-based distillation** is used to identify suitable student model architectures that optimize learning from the teacher models.

More on Distillation for LLMs

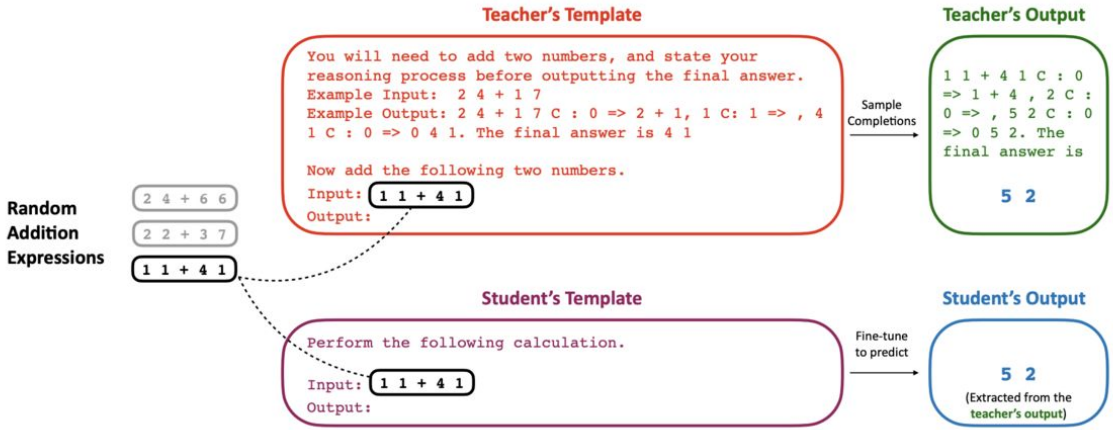
01

Additional Techniques

Context Distillation

Algorithm

1. Build heavily-engineered prompts that ended with simple questions, such as “add these numbers.”
2. Strip the prompt of its engineering and reduce the response to only its final answer to create a new data set used to fine-tune the model.

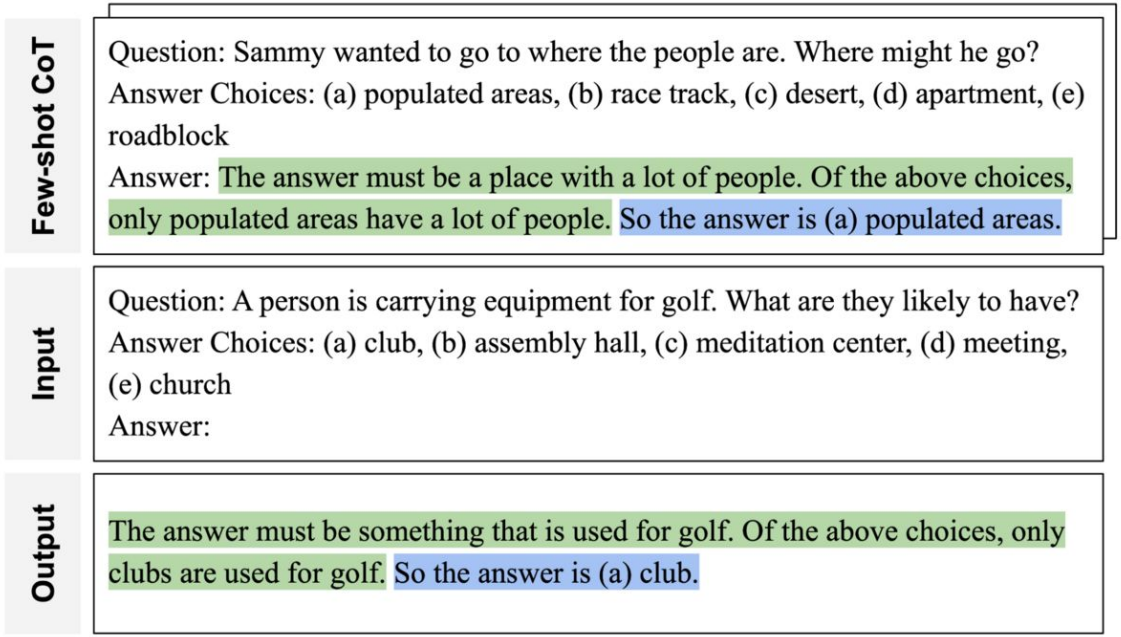


When the fine-tuned models the same questions with no added context, they found that the rate at which the model answered correctly increased.

Step by Step Distillation

The approach works by asking the teacher model to return not only its answer but also the rationale behind its answer.

The training pipeline then directs the student model to do the same—to yield both a final response and reasoning for that response. The pipeline updates the model's weights according to both portions of its output.



02

Distillation Scaling Laws

Overview

We have explored different ways to transfer knowledge from a larger model to a smaller one. But can we predict how effective this knowledge distillation will be? How will the distilled model perform, and what factors will it depend on?

This is where Apple and the University of Oxford have made a significant contribution to this vast topic. They developed distillation scaling laws and identified key trends in model behavior after distillation.

In the next few slides, we will go over there key results.

Result 1: Distillation scaling law predicts how well a student model will perform based on three key factors:

- Student model's size
- The number of training tokens
- The teacher's size and quality

This law follows a "power law" relationship – performance improves in a predictable way but only to a point. After this point adding more resources won't improve the model.

This can be used as the following idea: If the student is small, you can use a smaller teacher to save compute. If the student is large, you need a better and larger teacher for optimal performance. As we increase compute, the best teacher size grows initially, but then it plateaus because using a very large teacher becomes too expensive.

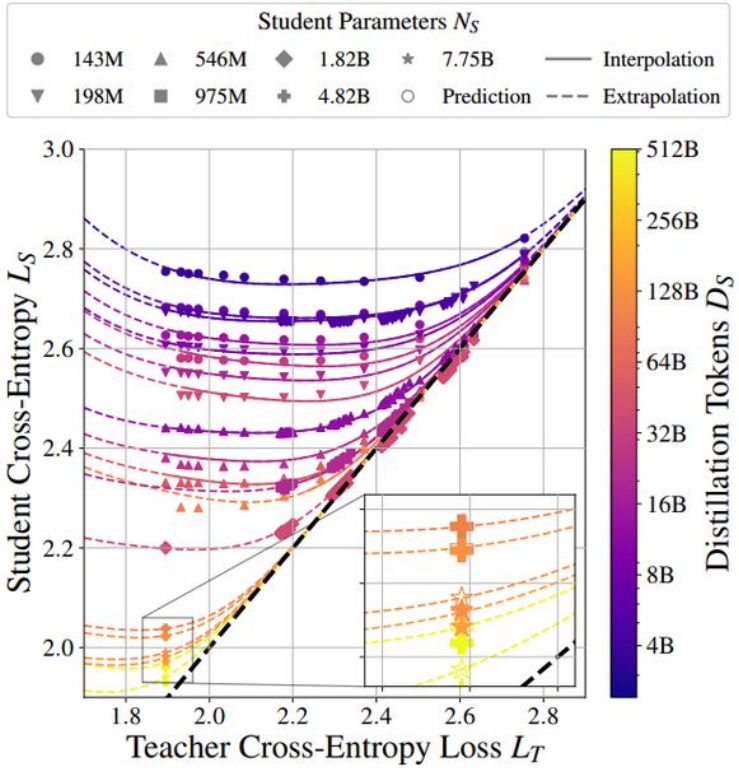
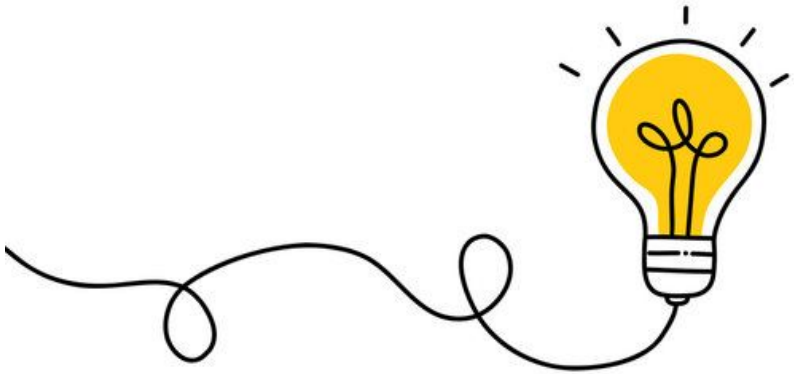


Figure 1. **Extrapolations of the Distillation Scaling Law.** The

Result 2: A good teacher doesn't always mean a better student. If a teacher is too strong, the student might struggle to learn from it, leading to worse performance. This is called the capacity gap — it is when the student isn't powerful enough to properly mimic the teacher.



Result 3: Sometimes a student model can even outperform its teacher. This phenomenon is called weak-to-strong generalization.

Result 4: Distillation is most efficient when:

- The student is small enough that supervised training would be too expensive.
- A teacher model already exists, so the cost of training one is not required, or it can be used beyond training just a single student model. Researchers recommend to use supervised learning instead of distillation:
    - If both the teacher and student need to be trained, because teacher training costs outweigh distillation benefits.
    - If enough compute and data are available, because supervised learning always outperforms distillation at high compute or data budgets.

03

Benefits of Distillation

- **Faster inference:** The distilled student model is typically smaller and requires fewer computations, leading to lower latency and faster predictions, which is crucial for real-time applications.
- **Improved generalization:** The student model often learns a more generalized and distilled version of the knowledge, potentially reducing overfitting and improving performance on unseen data.
- **Training stability:** The student model benefits from the structured knowledge of the teacher model, leading to smoother and more stable training, especially in cases where data is limited or noisy.

- **Transfer of specialized, diverse and multi-task knowledge:** A student model can be trained with insights from multiple teacher models, allowing it to inherit knowledge from diverse architectures or domains and perform well across different tasks.
- **Energy efficiency:** With reduced computation, knowledge-distilled models consume less energy, making them environmentally and economically viable for large-scale AI deployments.

04 Drawbacks

- **Increased training complexity:** Distillation requires training two models, the teacher and the student. This adds an additional step compared to directly training a smaller model from scratch.
- **Loss of information:** The student model may not capture all the nuances, fine-grained knowledge, or complex reasoning capabilities of the larger teacher model.
- **Performance trade-off:** While the student model aims to retain most of the teacher's performance, there is often a trade-off between size and accuracy.
- If the student model is too small, it might not have enough capacity to effectively learn from the teacher. This happens especially when the teacher model is too strong for the student.

- **Dependence on teacher model quality:** If the teacher model is biased or contains errors, these issues will likely be transferred to the student.
- **Sensitivity to temperature and hyperparameters:** The effectiveness of KD heavily depends on the choice of temperature parameter and loss function. Improper tuning can lead to poor knowledge transfer or bad students performance.
- **Energy and computational costs:** While the student model is efficient, the distillation process itself can be computationally expensive, especially for large-scale models.

05

The Controversy

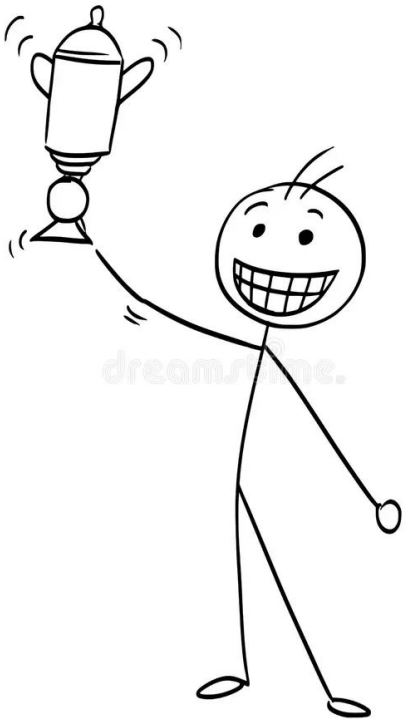
The Technical Achievement

Goal: Distill the advanced reasoning capabilities of a large model (DeepSeek-R1) into smaller, more accessible open-source models (Llama, Qwen).

Method: Fine-tuned smaller models on 800,000 high-quality instruction-following examples generated by the Teacher.

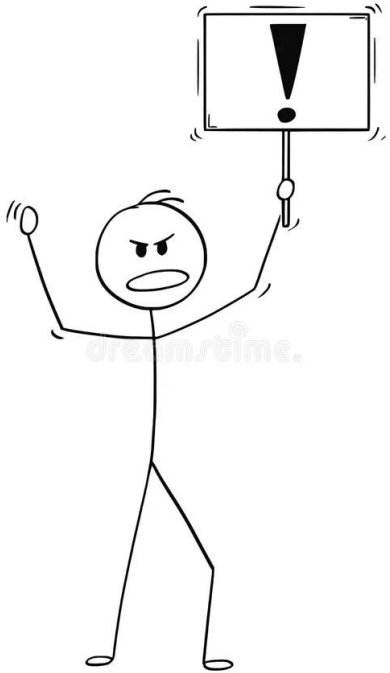
Results:

- The DeepSeek-R1-Distill-Qwen-7B model outperformed a model 4x its size (Qwen-32B) on reasoning benchmarks.
- Larger distilled versions (32B, 70B) set new records for open-source AI reasoning.
- Demonstrated that distillation can be more effective than reinforcement learning for improving reasoning in smaller models.



The Ethical Controversy

The Allegation (from OpenAI):
DeepSeek may have augmented its training data with outputs from proprietary models like ChatGPT.



The Evidence:

- DeepSeek's models sometimes identified themselves as "ChatGPT".
- Unusual data extraction activity linked to DeepSeek was reportedly detected by OpenAI/Microsoft.
- Some model responses closely resembled those from OpenAI systems.

The Fallout:

- Ignited a major industry debate on the ethical boundaries of knowledge distillation and AI intellectual property.
- Raised questions about "knowledge laundering"—using proprietary AI outputs to build a competing open-source model.

The DeepSeek case is a perfect example of both the immense potential of KD to democratize AI power and the critical challenges in defining ethical data sourcing.

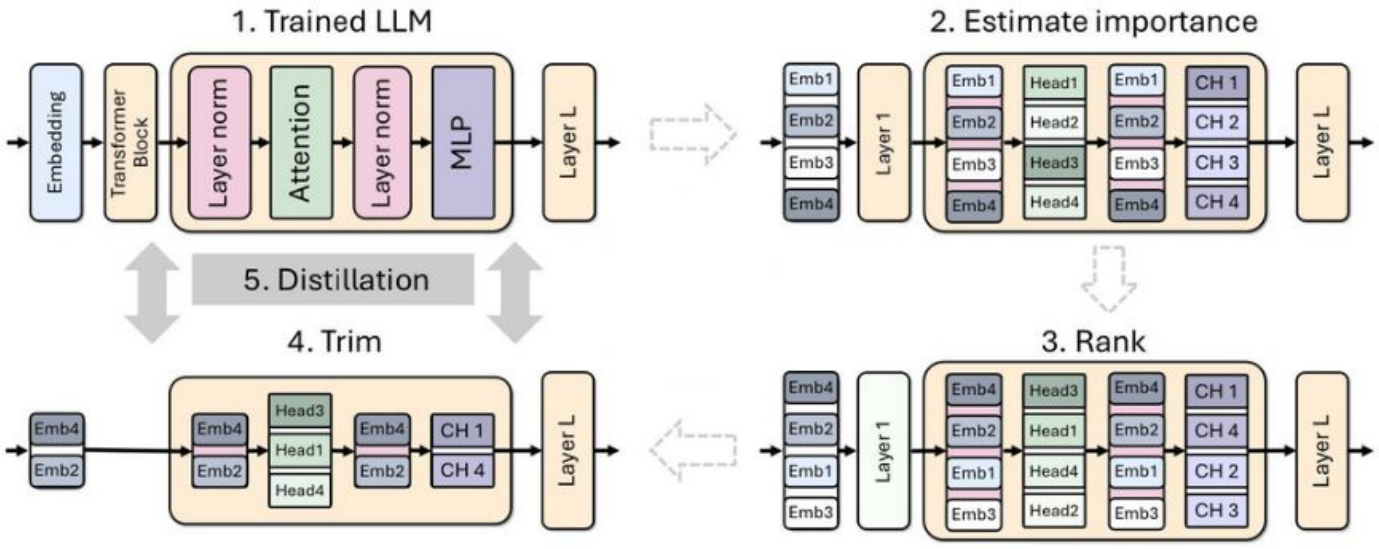
Google DeepMind

Practical: Knowledge Distillation

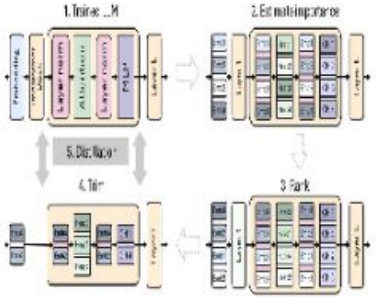
Colab Notebook

tme: 5 mins

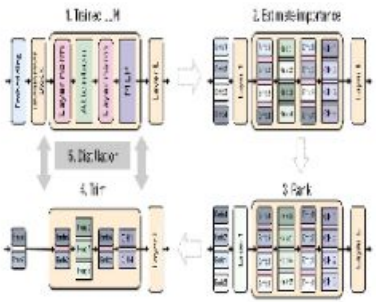
Model Pruning and Distillation in LLMs



15B => 8B



8B => 4B



Advanced

7

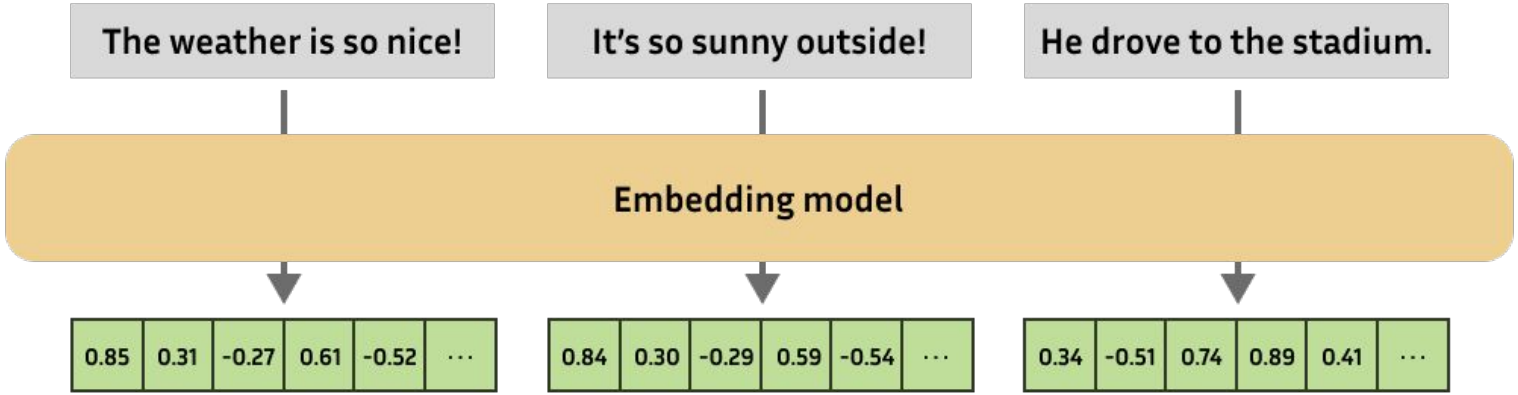
I. Matroyksha Embeddings
II. Speculative Decoding

01

Matryoksha Embeddings

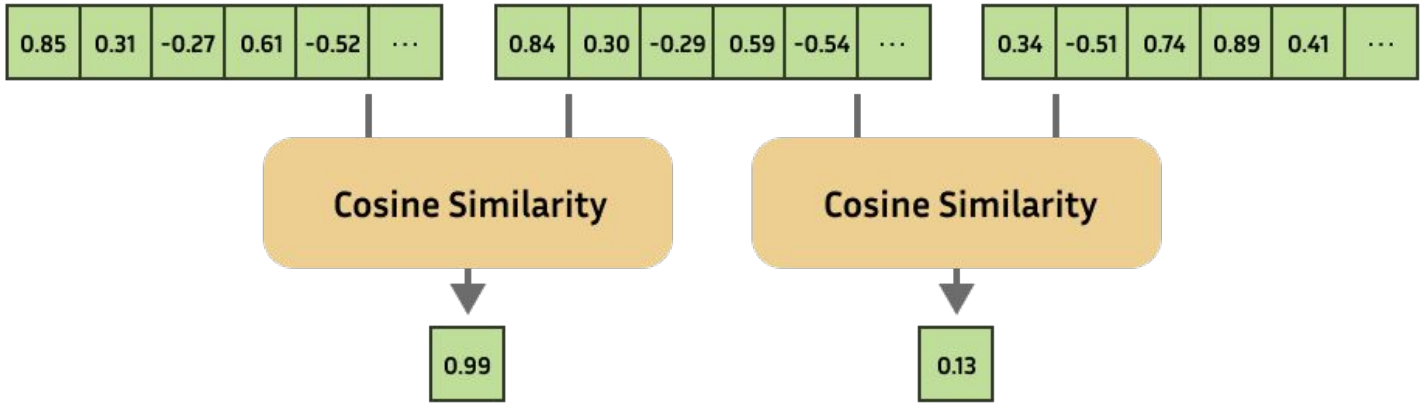
Embeddings

The embedding model will always produce embeddings of the same fixed size.



Embedding Similarity

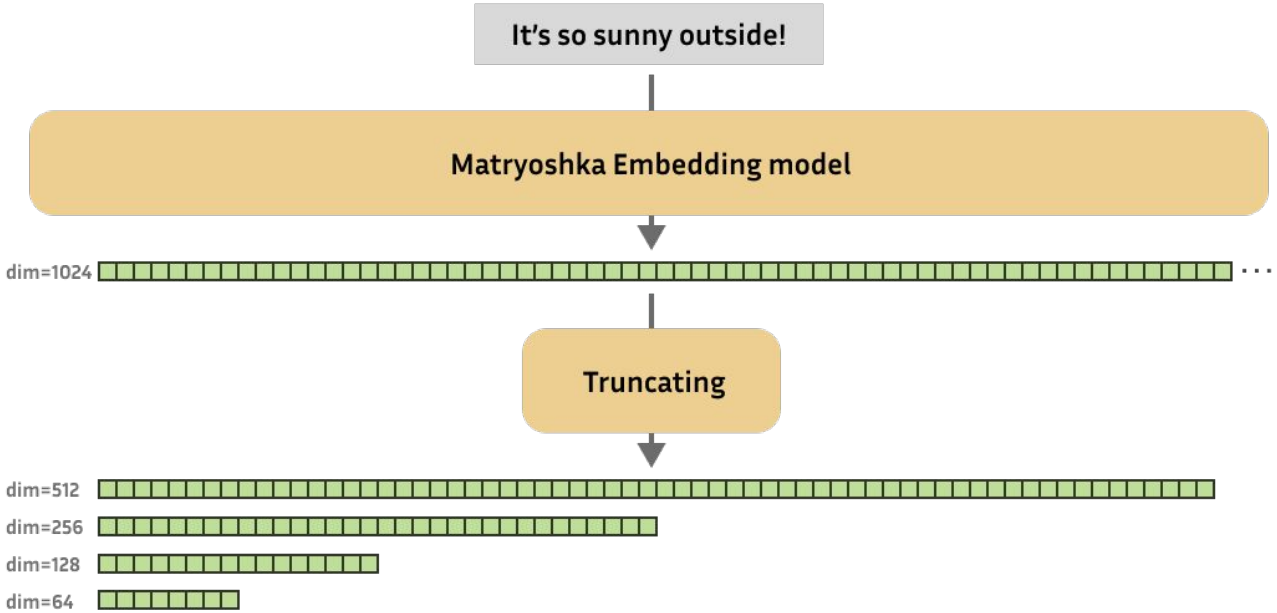
Compute the similarity of complex objects by computing the similarity of the respective embeddings!



Matryoshka Embeddings

New state-of-the-art (text) embedding models produce embeddings with increasingly higher output dimensions, i.e., every input text is represented using more values.

Although this improves performance, it comes at the cost of efficiency of downstream tasks such as search or classification.

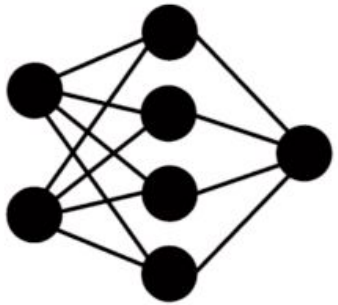


Consequently, Kusupati et al. (2022) were inspired to create embedding models whose embeddings could reasonably be shrunk without suffering too much on performance. These Matryoshka embedding models are trained such that these small truncated embeddings would still be useful. In short, Matryoshka embedding models can produce useful embeddings of various dimensions.

Matryoksha Dolls

"Matryoshka dolls", also known as "Russian nesting dolls", are a set of wooden dolls of decreasing size that are placed inside one another.

In a similar way, Matryoshka embedding models aim to store more important information in earlier dimensions, and less important information in later dimensions.



1. Compute Matryoshka Embedding

This characteristic of Matryoshka embedding models allows us to truncate the original (large) embedding produced by the model, while still retaining enough of the information to perform well on downstream tasks.

Why would you use 🪆 Matryoshka Embedding models?

- **Shortlisting and reranking**: Rather than performing your downstream task (e.g., nearest neighbor search) on the full embeddings, you can shrink the embeddings to a smaller size and very efficiently "shortlist" your embeddings. Afterwards, you can process the remaining embeddings using their full dimensionality.
- **Efficiency and Speed:** Matryoshka models will allow you to scale your embedding solutions to your desired storage cost, processing speed, and performance.

How are 🪆 Matryoshka Embedding models trained?

The Matryoshka Representation Learning (MRL) approach can be adopted for almost all embedding model training frameworks.

Normally, a training step for an embedding model involves producing embeddings for your training batch (of texts, for example) and then using some loss function to create a loss value that represents the quality of the produced embeddings. The optimizer will adjust the model weights throughout training to reduce the loss value.

For Matryoshka Embedding models, a training step also involves producing embeddings for your training batch, but then you use some loss function to determine not just the quality of your full-size embeddings, but also the quality of your embeddings at various different dimensionalities. For example, output dimensionalities are 768, 512, 256, 128, and 64. The loss values for each dimensionality are added together, resulting in a final loss value. The optimizer will then try and adjust the model weights to lower this loss value.

In practice, this incentivizes the model to frontload the most important information at the start of an embedding, such that it will be retained if the embedding is truncated.

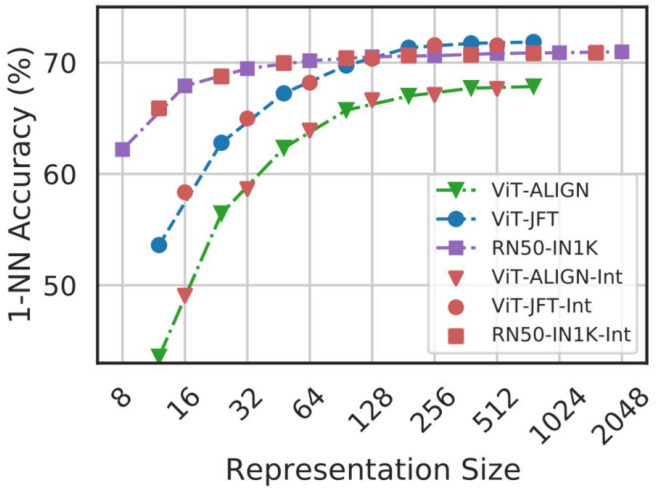
At inference, can we use any doll?

We only trained MRL at specific doll sizes d_8, d_{32}, ..., d_{2048}
Can we just use any doll of size that doesn't lie in these specific values we used to train the MRL model?

It turns that yes, you can - MRL model accuracies seamlessly interpolate at all doll sizes between the fixed doll sizes it was trained!

See the figure on the right, where the X-Axis is the doll size or representation size, and all the red points are evaluations at the interpolated sizes.

This means we can freely remove some numbers from the end of the sequence of any representation, and use that embedding directly.



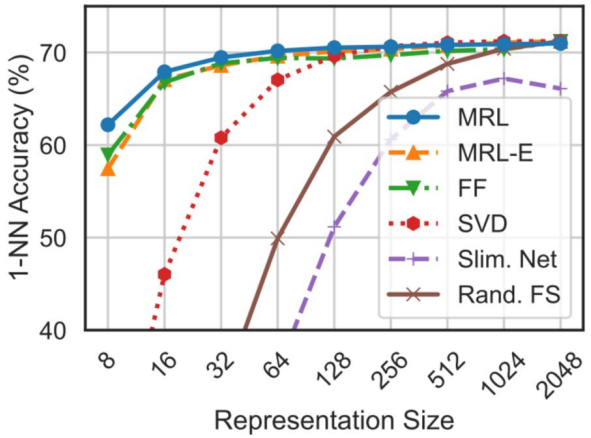
How good is MRL?

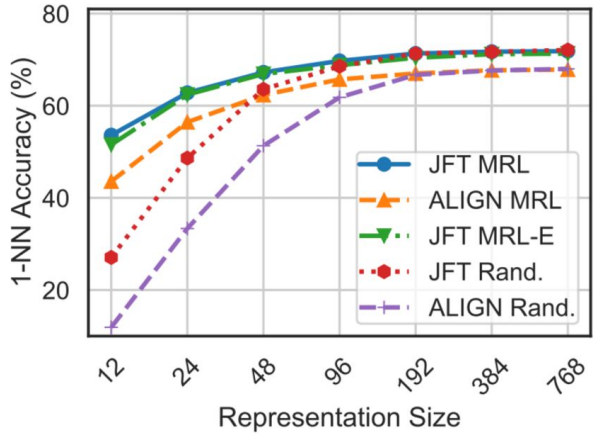
How does learning Matryoshka dolls compare to training a new doll from scratch at different dimensionality d every time?

While training all dolls at once with MRL is much more efficient, surely each MRL doll's performance will be worse than than its corresponding independently trained doll?

MRL dolls outperform independently trained dolls at each dimensionality, as seen in the figures below from the MRL paper

1. Million scale on ImageNet-1K with a ResNet-50 Neural Encoder at d ϵ (8, 16, .., 2048)
2. Billion scale on JFT-300M with ViT B/16 and ALIGN Neural Encoders with d ϵ (12, 24, .., 768)





Summarizing

MRL provides little to no accuracy drop for large efficiency gains across:

- Data Scale - million to billion
- Data Modality - vision, language, vision + language
- Neural Encoder Architecture - ResNet-50, ConvNeXt, ViT, BERT, ALIGN

Google DeepMind

Practical: Matryoksha Representation Learning

Colab Notebook

tme: 5 mins

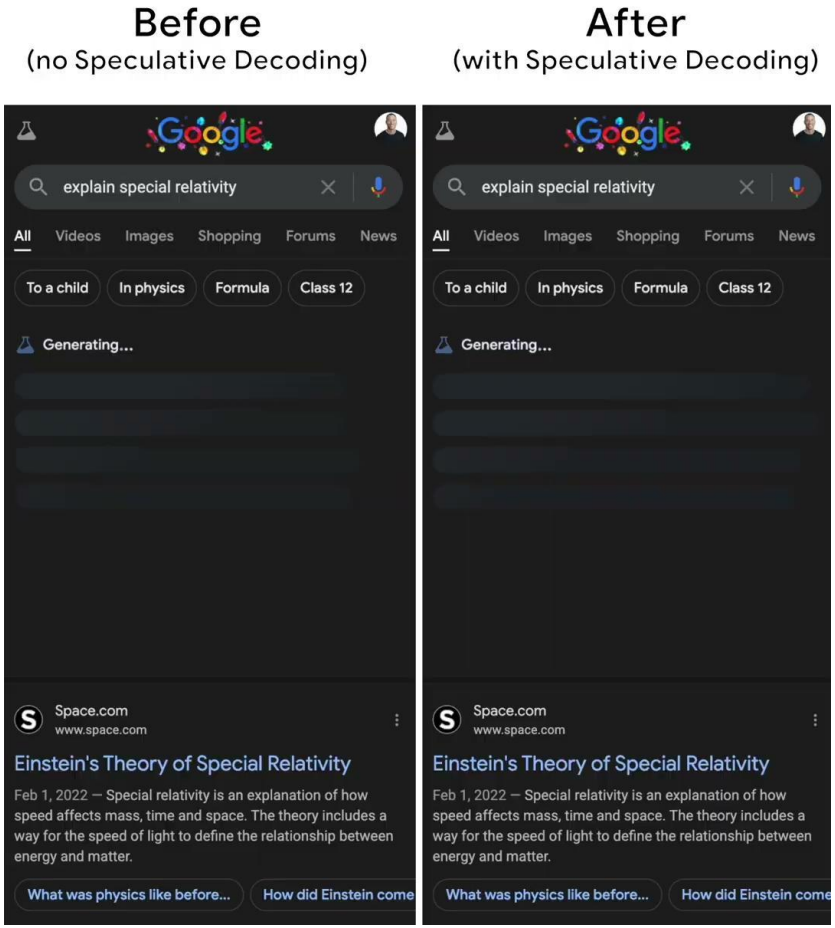
02

Speculative Decoding

Faster Inference

Speculative decoding has proven to be an effective technique for faster and cheaper inference from LLMs without compromising quality. It has also proven to be an effective paradigm for a range of optimization techniques.

The algorithm speeds up generation from autoregressive models by computing several tokens in parallel, without affecting output quality; in fact, the method guarantees an identical output distribution. Producing results faster with the same hardware also means that fewer machines are needed for serving the same amount of traffic, which translates yet again to a reduction in the energy costs of serving the same model.



Speculative decoding enables AI Overviews in Google Search to produce results faster than before, while maintaining the same quality of responses.

Background

An LLM generates its output one token at a time, where a token is usually a word or a part of a word. For example, with a common tokenizer, the sentence “One small step for man, one giant leap for mankind” is composed of 12 tokens. That means that to generate this sentence the LLM must run 12 times. Each such run is called a decoding step.

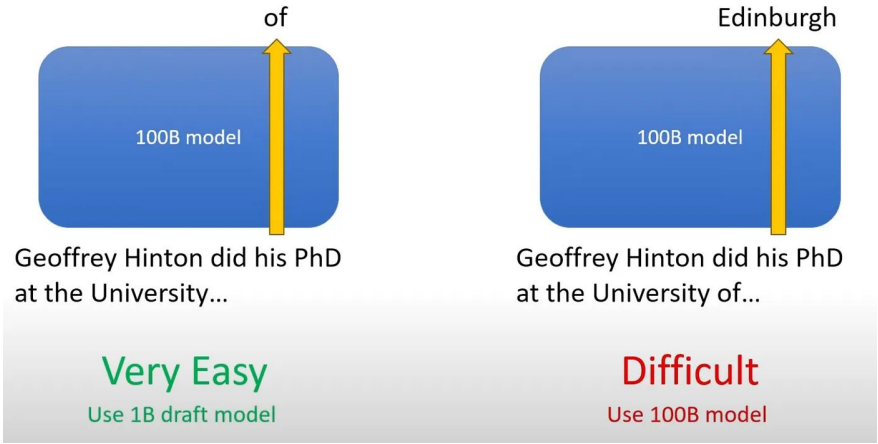
The larger the LLM, the more competent it can be. However, these larger models are also slower, because, for example, each decoding step needs to read the entirety of the model’s weights. This can mean that the model needs to read on the order of a tera-byte of data for each word it produces!

Typically, the goal is for LLMs to generate many words, such as a conversational response or a summary of a document, and since each token depends on the ones previously produced, they must be generated one by one, reading all of the model’s weights again and again. Two key observations motivate the speculative decoding method.

Observation 1: Some tokens are easier to generate than others

Not all tokens are alike: some are harder and some are easier to generate. Consider the example on the right:

This observation suggests that the large models are better due to better performance in difficult cases (e.g. "***of***"), but that in the numerous easy cases (e.g., "***Edinburgh***"), small models might provide reasonable approximations for the large models.



Observation 2: The bottleneck for LLM inference is usually memory

Machine learning hardware varieties, TPUs and GPUs, are highly parallel machines, usually capable of *hundreds of trillions* of operations per second, while their memory bandwidth is usually around just *trillions* of bytes per second — a couple of orders of magnitude lower. This means that when using modern hardware, we can usually perform hundreds of operations for every byte read from memory.

In contrast, the Transformer architecture that underlies modern LLMs usually performs only a few operations for every byte read during inference, meaning that there are ample spare computational resources available when generating outputs from LLMs on modern hardware.

Hardware can do	Transformers need
~100s–1000s operations/byte read	**~10** operations/byte read

Speculative Execution

Based on the expectation that additional parallel computational resources are available while tokens are computed serially, our method aims to increase concurrency by computing several tokens in parallel. The approach is inspired by speculative execution, an optimization technique whereby *a task is performed before or in parallel with the process of verifying whether it is actually needed*, resulting in increased concurrency.

For speculative execution to be effective, we need an efficient mechanism that can suggest tasks to execute that are likely to be needed. More generally, consider this abstract setting for speculative execution, with the assumption that $f(X)$ and $g(Y)$ are lengthy operations:

$Y = f(X)$

$Z = g(Y)$

The slow function $f(X)$ computes Y, which is the input to the slow function $g(Y)$. In the setting above, without speculative execution, we'd need to evaluate these serially.

Speculative execution suggests that given any fast approximation function $f^*(X)$, we can evaluate the first slow operation $f(X)$ in parallel to evaluating $g(f^*(X))$.

Once $f(X)$ finishes and we obtain the correct value of Y, we can check the output of the fast approximation:

- Case 1: $f^*(X) == Y$ -> we managed to increase parallelization.
- Case 2: $f^*(X) != Y$ -> we can simply discard the computation of $g(f^*(X))$ and revert to calculating $g(Y)$ as in the serial case.

The more effective $f^*(X)$, i.e., the higher the likelihood that it outputs the same value as $f(X)$, the more likely it is to increase concurrency. We are guaranteed identical outputs either way.

Speculative Sampling

Speculative Sampling is a clever strategy to break out of this slow, one-by-one cycle.

The core idea is to use two models: a small, fast "draft" model and the large, slow "target" model you actually want to use.

Think of it like an Expert and an Intern.

The Target Model (The Expert): This is your big, powerful model (e.g., Llama-70B, GPT-4). It's very smart and accurate, but also very slow and expensive to consult.
The Draft Model (The Intern): This is a much smaller, faster model (e.g., Llama-7B). It's not as smart as the expert, but it can produce text very quickly.

The goal is to get the Expert's high-quality work, but at a speed closer to the Intern's.

Speculative Sampling Workflow

1. **Ask the Intern for a Draft:** The fast "Intern" model quickly generates a chunk of text, say 5-6 tokens ahead. This is the "speculation" or "draft."
2. **Give the Draft to the Expert for Review:** The slow "Expert" model now takes this entire 5-6 token draft and evaluates it all at once in a single forward pass. This is the key optimization—one big calculation instead of 5-6 small, sequential ones.
3. **The Expert Approves or Corrects:**
   The Expert checks the Intern's draft token by token.
   If the Expert agrees with the Intern's token, it's accepted. The moment the Expert disagrees with a token, it rejects that token and the rest of the draft. The Expert then provides its own, corrected token.
4. **Repeat:** The process starts over from the end of the accepted sequence.

The result?
In the best case, the Intern's entire draft is correct, and you just generated 5-6 tokens for the price of one single call to the slow Expert model—a 5-6x speedup for that step!

Speculative Decoding

Speculative Decoding is the most common and direct implementation of the Speculative Sampling idea. The terms are often used interchangeably, but "decoding" refers to the specific step-by-step algorithm.

Let's walk through a concrete example.

- **Target Model (Slow):** A giant, accurate model.
- **Draft Model (Fast):** A small, fast model.
- **Prompt:** "The quick brown fox"

Speculative Decoding

Step 1: Drafting (The Intern works)

The Draft Model gets the prompt and quickly generates a 4-token continuation: *"jumps over lazy kid"*

Step 2: Verification (The Expert reviews)

The Target Model now takes the full sequence *"jumps over the kid"* and evaluates it in one single forward pass. This pass calculates the probability the Target Model would have assigned to each of those tokens.

Speculative Decoding

Step 3: Acceptance/Rejection Loop

Now we compare the draft with the Target Model's probabilities, token by token.

- *Token 1: "jumps"*
  Was *jumps* a likely token for the Target Model to generate after *fox*? Yes.
  Result: ACCEPT *"jumps"*

- *Token 2: "over"*
  Was *over* a likely token for the Target Model to generate after *jumps*? Yes.
  Result: ACCEPT *"over"*.

- *Token 3: "lazy"*
  Was *lazy* a likely token for the Target Model to generate after *over*?
  Suppose No. The Target Model strongly preferred the token *the*.
  Result: REJECT *"lazy"* (and the rest of the draft "kid").

What Happens Inside the Expert Transformer in Step 3

When this combined sequence is fed into the large "expert" model for its one forward pass, here's the process:

Step 3.1: Input Embeddings
The entire sequence *“The quick brown fox jumps over lazy kid”* is turned into a series of numerical vectors (embeddings). Each token gets its own vector.

Step 3.2: The Transformer Layers and the Causal Mask
This causal attention mask enforces the rule that when the model is calculating the output for a specific token, it can only pay attention to the tokens that came before it. It is "causal" because it prevents information from the future from influencing the past.

What Happens Inside the Expert Transformer in Step 3

So, even though we are processing all tokens at once:

- To calculate the output for "*jumps*", the model can only see *The quick brown fox.*
- To calculate the output for "*over*", the model can only see *The quick brown fox jumps.*
- To calculate the output for "*lazy*", the model can only see *The quick brown fox jumps over.*

This mask is what allows the model to compute the "next-token prediction" for every position in the sequence simultaneously without cheating.

Step 3.3: The Output Logits

After the final transformer layer, the model produces a sequence of output vectors. Each of these vectors is then passed through a final linear layer to produce a logit vector. We are interested in the ones corresponding to the draft tokens.

What Happens Inside the Expert Transformer in Step 3

Let's look at the specific logits we get and what they mean:

- **Logit vector at the position of *fox***
  This vector was calculated using the context *"The quick brown fox"*. It represents the model's prediction for what should come next. We check this prediction against the first draft token, *jumps*.
  Verification: Does this logit vector predict jumps? (i.e., is the probability for the token jumps high enough to be accepted?). Let's say Yes. We accept "jumps".
- **Logit vector at the position of *jumps***
  This vector was calculated using the context *"The quick brown fox jumps"*. It represents the model's prediction for the next token. We check this against the second draft token, *over*.
  Verification: Does this logit vector predict over? Let's say Yes. We accept "over".
- **Logit vector at the position of *over***
  This vector was calculated using *"The quick brown fox jumps over"*. It's the model's prediction for the next token. We check this against the third draft token, *lazy*.
  Verification: Does this logit vector predict the? Let's say No. The highest probability is for the token "*the*". We reject "*lazy*".

What Happens Inside the Expert Transformer in Step 3

The Final Outcome of the "Evaluation"

- We stop the verification process at the first mismatch.
- **Accepted Tokens:** *jumps* and *over*.
- **Correction Token:** We use the prediction from the last valid step. The logit vector at the position of *over* predicted *the*. So, *the* is our new, corrected token.

The final output from this entire speculative cycle is the combination of the accepted tokens and the one correction: *jumps over the*.

The Payoff

We successfully generated 3 new tokens (*jumps*, *over*, *the*) but only had to run the expensive Target Model once. This is a 3x speedup for this step.
The process now repeats, with the Draft Model generating a new guess starting from ...*the*.

Speculative Sampling

We proposed a generalization of speculative execution to the stochastic setting, i.e., where a task *needs to be executed with some probability*. We call this generalization *speculative sampling*.

Consider the following setup, identical to that above, with the exception that $f(X)$ now outputs a probability distribution from which we *sample* the input to function g, Y:

$Y \sim f(X)$

$Z = g(Y)$

Similar to above, given any fast approximation $f^*(X)$, this time outputting a probability distribution, speculative sampling allows us to execute $f(X)$ in parallel to the execution of $g(sample(f^*(X))$. We could use standard speculative execution, and discard the calculation in case the samples don't match, but this would be inefficient.

Indeed, consider an example where $f(X)$ and $f^*(X)$ always output a uniform probability distribution from 1 to 100. Speculative execution would only accept f^*'s guess once every 100 times.

This is clearly inefficient — f and f^* are the same function and we can always accept the sample from f^*! Instead, speculative sampling offers a way to accept or discard f^*'s guesses probabilistically, depending on $f(X)$ and $f^*(X)$, guaranteeing optimality as well as an identical output distribution.

Speculative sampling could be useful in other settings, for example, in reinforcement learning or physics simulations (see paper for details).

Speculative Decoding

LLMs don’t produce a single next token, but rather a probability distribution from which we sample the next token.

For example, following the text “The most well known movie director is”, an LLM might produce the token “Steven” with 70% chance and the token “Quentin” with 30% chance).

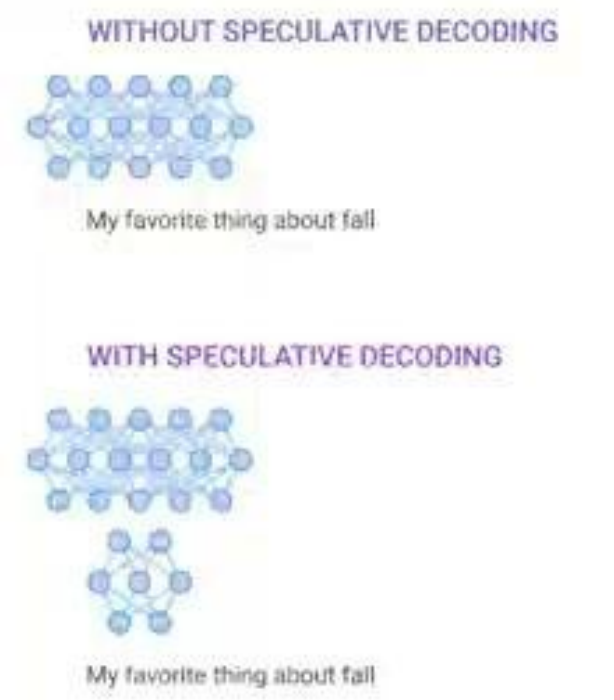
This means that a direct application of speculative execution to generate outputs from LLMs is very inefficient. Speculative decoding makes use of speculative sampling to overcome this issue.

With it, we are guaranteed that in spite of the lower cost, the generated samples come from exactly the same probability distribution as those produced by naïve decoding. Note that in the special case of greedy decoding, where we always sample the single most probable token, speculative execution can be applied effectively to LLM inference.

Speculative decoding is the application of speculative sampling to inference from autoregressive models, like transformers. In this case, both $f(X)$ and $g(Y)$ would be the same function, taking as input a sequence, and outputting a distribution for the sequence extended by one token.

Speculative decoding thus allows us to efficiently calculate a token and the tokens following it, in parallel, while maintaining an identical distribution (note that speculative decoding can parallelize the generation of more than two tokens, see the paper).

All that remains in order to apply speculative decoding is a fast approximation of the decoding function. Observation 1 above suggests that a small model might do well on many of the easier tokens. Indeed, in the paper they showed that using existing off-the-shelf smaller models or simple heuristics works well in practice. For example, when applying speculative decoding to accelerate an 11B parameter T5-XXL model for a translation task, and using a smaller 60M parameter T5-small as the guessing mechanism, we get ~3x improvement in speed.



An animation, demonstrating the speculative decoding algorithm in comparison to standard decoding. The text is generated by a large GPT-like transformer decoder. In the case of speculative decoding, a much smaller model is used as the guessing mechanism. The accepted guesses are shown in green and the rejected suggestions in red.

Google DeepMind

Practical: Speculative Decoding

Colab Notebook

tme: 5 mins

References

1. [2211.05102] Efficiently Scaling Transformer Inference
2. [2205.14135] FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness
3. [1805.02867] Online normalizer calculation for softmax
4. [1503.02531] Distilling the Knowledge in a Neural Network
5. [1412.6550] FitNets: Hints for Thin Deep Nets
6. [2209.15189] Learning by Distilling Context

References

1. Transformers KV Caching Explained | by João Lages | Medium
2. vLLM: Easy, Fast, and Cheap LLM Serving with PagedAttention
3. Optimizing GPU Memory for LLMs: A Deep Dive into Paged Attention
4. Understanding KV Cache and Paged Attention in LLMs: A Deep Dive into Efficient Inference | by Dewang Sultania | My musings with LLMs | Medium
5. FlashAttention & Paged Attention: GPU Sorcery for Blazing-Fast Transformers | by afoufa | Medium
6. ELI5: FlashAttention. Step by step explanation of how one of... | by Aleksa Gordić
7. From Online Softmax to FlashAttention
8. Everything You Need to Know about Knowledge Distillation
9. LLM distillation demystified: a complete guide | Snorkel AI
10. Knowledge Distillation: Principles, Algorithms, Applications
11. Model Distillation
12. Matryoshka Representation Learning (MRL) from the Ground Up | Aniket Rege
13. Looking back at speculative decoding