O'REILLY®

Hands-On Machine Learning with Scikit-Learn and PyTorch

Concepts, Tools, and Techniques to Build Intelligent Systems

Aurélien Géron

Online Bonus Content for Hands-On Machine Learning with Scikit-Learn and PyTorch

Aurélien Geron

O'REILLY®

Hands-On Machine Learning with Scikit-Learn and PyTorch
by Aurélien Geron

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nicole Butterfield
Develeopment Editor: Michele Cronin
Production Editor: Beth Kelly
Copyeditor: Sonia Saruba
Proofreader: Kim Cofer
Indexer: Potomac Indexing LLC
Cover Designer: Susan Brown
Cover Illustrator: José Marzan Jr.
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

October 2025: First Edition

Revision History for the First Edition
2025-10-22: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9798341607989* for release details.

979-8-341-60798-9

[LSI]

CHAPTER 17

Speeding Up Transformers

In Chapters 15 and 16, we built all kinds of transformers, from classifiers, translators and chatbots, to vision and multimodal transformers. While transformers are incredibly versatile and powerful, they are far from perfect. In particular, they can be very slow, especially when processing long input sequences.

Luckily, many techniques have been developed to speed up transformers of any size:

- To speed up decoding in generative transformers, we will use key/value caching and speculative decoding, then we will take of a quick look at several approaches to parallelize text generation.
- To accelerate multi-head attention (MHA), which is one of the most computationally expensive components of transformers, we will look at sparse attention, approximate attention, sharing projections, and FlashAttention.
- To speed up gigantic transformers of up to trillions of parameters, we will discuss mixture of experts (MoE).
- To train large transformers efficiently, we will discuss parameter-efficient fine-tuning (PEFT) using adapters such as Low-Rank Adaptation (LoRA), activation checkpointing, sequence packing, gradient accumulation, and parallelism.

Another way to speed up a transformer is to make it smaller. This can be done using reduced precision and quantization, which are discussed in Appendix B.

We have a lot on our plate, so let's get started: we will start by focusing on accelerating the decoding process in generative transformers.

Faster Decoding at Inference Time

In decoder-only and encoder-decoder models, the decoder generates one token at a time at inference time: to generate 1,000 tokens, it must be called 1,000 times. This is not only computationally expensive, it's also tricky to parallelize because each new token depends on the previous ones: it's a sequential generation process. That said, there are ways to speed up the decoding process, starting with caching.

Key/Value Caching

As we saw in Chapter 15, each attention head in a multi-head attention (MHA) layer computes the scaled dot-product attention equation (Equation 15-1): $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{Q}\mathbf{K}^{\intercal} / \sqrt{d_k})\mathbf{V}$, where **Q**, **K**, and **V** are the query, key, and value matrices, respectively, and d_k is the key's dimensionality. In a self-attention layer, the query, key, and value are all different projections of the MHA layer's input sequence.

The i^{th} row in each matrix is a projection of the i^{th} input token representation. Since decoders use *masked* self-attention layers, the i^{th} input token representation is only influenced by tokens 1 to i: it's a causal model (i.e., the future has no influence on the past). As a result, appending a new token $i + 1$ to the end of the input sequence will not affect rows 1 to i in matrices **Q**, **K**, and **V**, it will only append a new row to each of them. Great! We can cache these matrices, computing and appending only a single new row at each generation step.

In fact, we only need to cache **K** and **V**, no need to cache **Q**. Indeed, if you look closely at the scaled dot-product equation, you will see that the last row of the output depends on the full matrices **K** and **V**, but only on the last row of **Q**. Since we're only interested in the output for the last token, we only need to compute the last row of the output matrix at each generation step, which means we only need the last row of **Q**: no cache needed.

So how does key/value caching (KV caching) work exactly? Well, the first time we call the decoder, feeding it the initial prompt sequence containing tokens 1 to i, the decoder works normally, except the **K** and **V** matrices in all attention heads across all MHA layers are saved to a cache in memory (see Figure 17-1). This is called the *prefill stage*. Each of these **K** and **V** matrices has i rows (one per token). The decoder outputs a sequence containing its next-token predictions for tokens 1 to i. We only care about the last predicted token, since that's the new token $i + 1$.

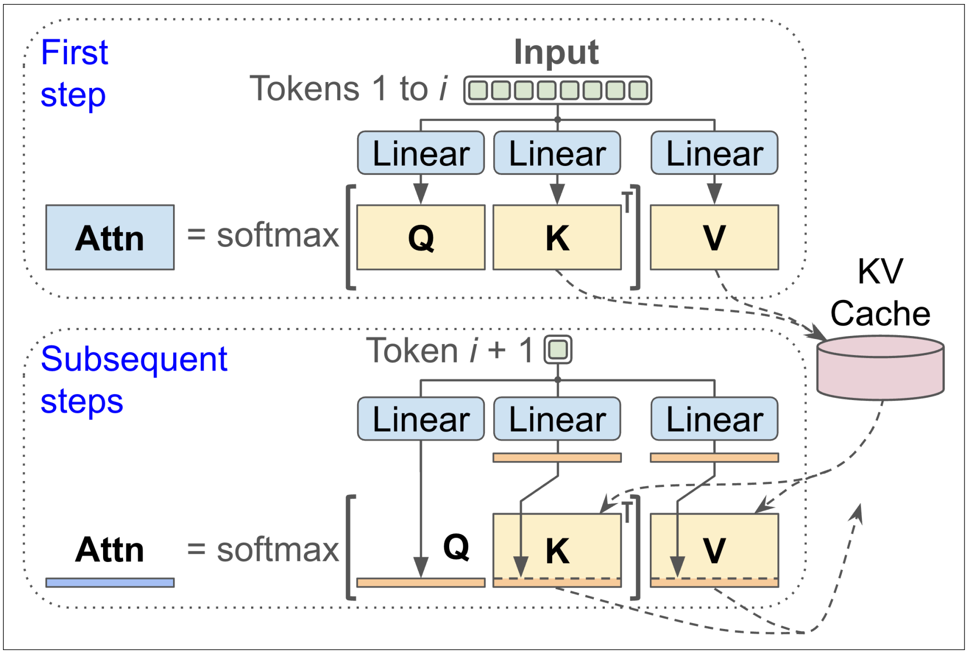


Figure 17-1. Key/value caching: the first step is normal but caches all key and value matrices; subsequent steps process only the last generated token, reusing the cached matrices and appending one row for the new token

Now when we call the decoder again to generate the next token—this is called the *decoding stage*—we only feed it the new token $i + 1$: there's no need to feed it tokens 1 to i since the cached **K** and **V** matrices already contain the rows for tokens 1 to i. In each MHA layer, the input—a single-token—is projected three times, which produces three single-row matrices: a single-row matrix **Q**, a second single-row matrix which is appended to the cached matrix **K**, and a third single-row matrix which is appended to the cached matrix **V**. The **Q** matrix contains a single row, but the updated matrices **K** and **V** now contain $i + 1$ rows each. We are finally ready to compute the scaled dot-product equation, and the output is once again a single-row matrix. After going through all layers, the decoder predicts a single token: the new token $i + 2$. This same process can be repeated for each new token after that.

KV caching allows us to generate new tokens with much less compute than the naive approach we implemented in Chapter 15 where we repeatedly fed the whole growing sentence back to the decoder. Even with KV caching, each new token takes longer and longer to generate because the **K** and **V** matrices keep growing, but at least it's a linear increase rather than a quadratic one. Note that generating a sequence of length L_o with KV caching still takes $O(L_o^2)$ time, since the sum of a linearly increasing quantity is quadratic: $1 + 2 + 3 + \ldots + L_o = L_o(L_o + 1)/2$. That said, quadratic is better than cubic!

Storing the KV cache in GPU RAM can be a challenge, not only because of its size, but also because sequences have variable lengths, leading to memory fragmentation and frequent costly reallocations. *PagedAttention*, introduced in the vLLM toolkit, addresses this by storing the KV cache in fixed-size memory pages, inspired by virtual memory systems. This makes batching more efficient and avoids wasting memory.

To implement KV caching using the Hugging Face Transformers library, you generally don't need to do anything at all: most decoder-only and encoder-decoder models support KV caching out of the box for the decoder, and the `generate()` method uses KV caching by default. Under the hood, the `generate()` method sets `use_cache=True` when calling the model, and it also sets the `past_key_values` argument to a cache object. Upon the first call, the cache is empty so the decoder works normally and stores the key and value matrices in the cache. Upon subsequent calls, the decoder only processes the last input token, using and updating the cache as we discussed earlier.

The default cache class used by the `generate()` method is `transformers.DynamicCache` which is well suited for the general case, but there are other cache classes for special use cases (e.g., when using sparse attention): see *https://homl.info/kvcache* for more details.

Now let's move on to another common technique to speed up decoding: speculative decoding.

Speculative Decoding

Speculative decoding, also known as *assisted generation*, was proposed by Google researchers in 2022.[1] They observed that most tokens are fairly easy to predict: for example, in the sentence "Once upon a time there lived a pony", most words can easily be predicted given the previous words, except for "Once" and "pony". So why not use a smaller and faster model to generate the easy tokens, and a larger, slower but smarter one for the hard ones? Unfortunately we don't know ahead of time which

1 Yaniv Leviathan et al., "Fast Inference from Transformers via Speculative Decoding", arXiv preprint arXiv:2211.17192 (2022).

tokens will be easy or hard, so the authors proposed the following technique (see Figure 17-2):

- At each generation step, we pass the context containing tokens c_1 to c_t to the small model—called the *draft model* (or the *assistant model*)—and we make it generate a sequence of n additional tokens d_{t+1} to d_{t+n}, one token at a time. This additional sequence is called the *draft*. Generating the draft is fast since the draft model is small. Note that we generally use KV caching alongside speculative decoding, so we really only pass one token at a time to the small model, but to simplify the explanations I'll pretend we're not using KV caching.
- Then we give the whole sequence (context plus draft) to the large model, called the *target model*. This includes tokens c_1, …, c_t, d_{t+1}, …, d_{t+n} (when using KV caching we only need to pass c_t, d_{t+1}, …, d_{t+n}). This step is also fast since the target model can process all the tokens in parallel, rather than sequentially.
- For each input token, the target model outputs tokens o_1 to o_{t+n} where the output token o_i is the predicted next token for position $i + 1$, assuming that all the previous tokens are correct. So we must now verify that the draft tokens are indeed correct: to do so, we check whether d_i matches o_{i-1}, for every position i from $t + 1$ to $t + n$. If they are all correct, that's great, we can safely use all the draft tokens d_{t+1} to d_{t+n}, since they match what the target model predicted, and we can even append the output token o_{t+n} since it's the target model's prediction for position $t + n + 1$. We have now generated $n + 1$ tokens in total, using only n steps with the draft model and a single step with the target model. Most importantly, we have the same result as if we had run the target model to generate the $n + 1$ new tokens.
- However, what if some of the draft tokens are incorrect, meaning they don't match the target model's predictions? Well, if d_k is the first incorrect draft token (e.g., token "I" in the figure), then the corresponding output token and all output tokens located after it are useless: indeed, the assumption that all previous tokens are correct is invalid anywhere after the bad draft token d_k. However, we can still safely use the draft tokens up to this point (e.g., "DEF"), as well as the last valid output token o_{k-1} (e.g., "G"). The worst case is when the very first draft token d_{t+1} is incorrect, but even in this case we still get at least one new token: o_t (i.e., the prediction for position $t + 1$).

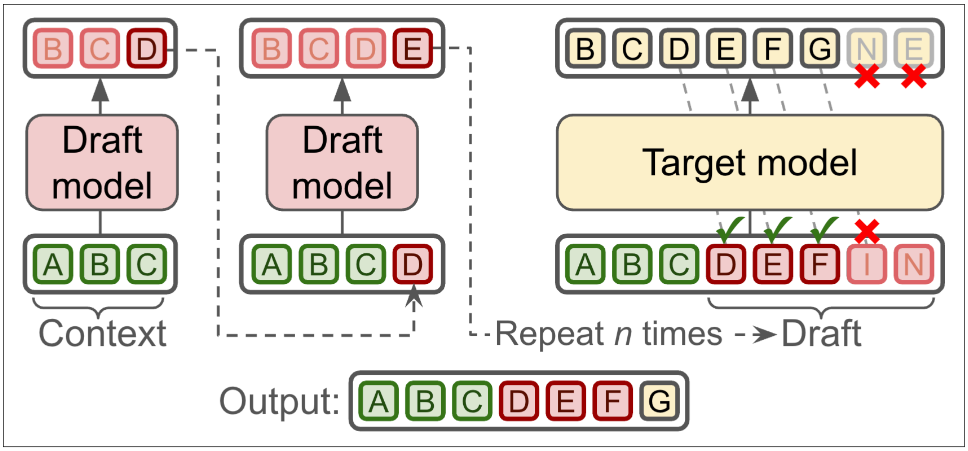


Figure 17-2. Speculative decoding: the draft model generates a draft sequence and the target model verifies it

But wait! This process only works if we are using greedy decoding, always picking the token with the largest logit. However, as we saw in Chapter 15, it's usually preferable to sample tokens randomly (typically using the probability distribution computed using the softmax of the logits divided by the temperature), as this generally produces nicer text, with less risk of weird repetitions. This means that the target model should sometimes accept tokens that are not its first choice, as long as they are not too bad. Ideally, we would like the token distribution using speculative decoding to be identical to the token distribution using only the target model. Luckily, it turns out to be possible! The authors proposed the following sampling strategy, named *speculative sampling*, and proved mathematically that it yields the same token distribution as the target model's distribution (see the paper's appendix A.1. for the proof). Here's how it works:

- When generating the draft token at position *i*, we run the draft model to compute the probability distribution $q_i(x)$ over the possible next tokens (x), and we sample one of them randomly, denoted d_i.
- Once we have generated a draft sequence, we can run the whole sequence (prompt plus draft) through the target model to compute the probability distribution $p_i(x)$ over the possible next tokens x, for each position i. However, this time we don't sample the output tokens just yet.
- Next, for each draft token d_i, we accept the token if $q_i(d_i) \le p_i(d_i)$. However, if $q_i(d_i) > p_i(d_i)$ then we only accept the token with probability $p_i(d_i) / q_i(d_i)$, otherwise we reject it.
- Anytime a draft token is accepted, we simply move on to the next draft token. If all draft tokens are accepted, then we can sample one more token for free from

the last probability distribution output by the target model. We now have $n + 1$ new tokens.

- However, if a draft token d_k is rejected, then we immediately stop iterating and reject all the draft tokens at positions $i > k$ as well. We can still sample one last token for free from the target model, but this time using a slightly adjusted probability distribution $\text{p'}_i(x) = \max(0, \text{p}_i(x) - \text{q}_i(x)) / s_i$, where s_i is a normalization constant, chosen so the probabilities add up to one. This adjustement ensures that we don't oversample the tokens that the draft model can sample directly.

You might be wondering how long the draft sequences should be? If they're too short then they won't provide much speed up, if any, but if they're too long, most draft tokens will be dropped, wasting compute both when generating the draft tokens and when verifying them. The Hugging Face Transformers library supports several strategies to choose the draft length (also called the *speculative lookahead* or SL):

- Use a constant draft length: this is a hyperparameter you can tune.
- Use a basic heuristic: if all draft tokens are accepted during one generation step, then increase the length of the draft in the next generation step, otherwise decrease it.
- Use *dynamic speculative decoding*:[2] keep generating draft tokens until the draft model's confidence drops below a given threshold (i.e., the draft token's estimated probability is too low). This is the default strategy.

To implement speculative decoding using the Hugging Face Transformers library, you just have to pass the draft model to the `generate()` method using the `assistant_model` argument. You can optionally set the draft model's confidence threshold for dynamic speculative decoding:

```python
from transformers import AutoModelForCausalLM, AutoTokenizer

target_model = AutoModelForCausalLM.from_pretrained("facebook/opt-350m",
                                                    device_map="auto")
draft_model = AutoModelForCausalLM.from_pretrained("facebook/opt-125m",
                                                   device_map="auto")
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-350m")
prompt = "Once upon a time there lived"
inputs = tokenizer(prompt, return_tensors="pt").to(target_model.device)
outputs = target_model.generate(**inputs, max_new_tokens=100, do_sample=True,
                                temperature=1, assistant_model=draft_model)
result = tokenizer.decode(outputs[0], skip_special_tokens=True)
```

2 "Dynamic Speculation Lookahead Accelerates Speculative Decoding of Large Language Models", arXiv preprint arXiv:2405.04304 (2024).

The target and draft models must be based on the same tokenizer or else you will get an error. If you really know what you are doing, you can set the `tokenizer` and `assistant_tokenizer` arguments when calling the `generate()` method, but beware: if the vocabularies are different, the assistant's suggestions will likely be terrible, slowing down text generation.

How much will speculative decoding help? Well, first it depends on the pair of models: if the draft model is not much faster than the target model, everything could actually be much slower! So the draft model should be small and fast. But not too small, or else it will be too approximate, and the target model will reject most of the draft tokens, so there will be no speed up at all—quite the contrary. In short, you have to experiment with various draft models until you find the right one for your target model. As a rule of thumb, the best draft model will typically be 10 to 20 times smaller than the target model.

The speed up also depends on the task: easier tasks will get more boost since the draft model will produce better drafts. For example, Hugging Face researchers got a 1.8× speed up on a summarization task using OPT-6.7B by Meta as the target model and OPT-125M as the draft model, but the speed up was only 1.2× on an open-ended generation task.

The draft-and-verify approach is not the only way to parallelize decoding; there are many more. Let's discuss the main approaches.

Main Approaches to Parallelize Decoding

A survey published by Lingzhe Zhang et al. in August 2025[3] lists over 130 papers that propose various ways to parallelize text generation. The authors grouped these papers into six categories:

Draft and verify
: This includes speculative decoding and multiple variants.

Decompose and fill
: The idea is to break the problem into independent tasks, then tackle these tasks in parallel. For example, given the prompt "Tell me 10 chicken jokes", the LLM can be run in parallel to generate each joke. Similarly, given the prompt "Which of these words come from French: mayor, state, cuisine, …", each word can be processed in parallel. Lastly, given the prompt "Tell me a bedtime story", the LLM

3 Lingzhe Zhang et al., "A Survey on Parallel Text Generation: From Parallel Decoding to Diffusion Language Models", arXiv preprint arXiv:2508.08712 (2025).

can first generate the storyline, then write all chapters in parallel based on the storyline.

Multiple token prediction (MTP)
: In this approach, the LLM is trained or fine-tuned to generate multiple tokens at a time, producing a draft that it can then verify just like with speculative decoding: no need for a draft model. One way to do this is to append special lookahead tokens at the end of the prompt: for example, if the prompt is "Once upon", then we might feed the model the sequence "Once upon <lookahead> <lookahead> <lookahead>". If the LLM is fine-tuned to predict the next tokens whenever it sees the lookahead token, it might predict "upon a time there was", which it can then verify. The more accurate its predictions, the greater the speed up.

One-shot generation
: One-shot models produce the output in a single parallel step. This is extremely fast, but since all generated tokens are independent, it often leads to repetitions, missing words, and incoherence. Various techniques can be used to improve the quality of the output, for example by first producing a template, then feeding it to the model when producing the final output.

Masked generation
: The model starts with a fully masked sequence, and gradually predicts some of the words. At each generation step, it predicts multiple masked tokens and refines some of its previous predictions, gradually improving coherence and fixing errors.

Edit-based refinement
: An initial quick draft is gradually improved by an *editor model*, trained to edit the text using insertions, deletions, or substitutions in parallel across the whole sequence.

Dynamic Batching and In-Flight Batching

If your service is popular enough to receive many requests per second, you will want to look into *dynamic batching*: instead of processing requests immediately as they arrive, you queue them and process them by batch at frequent intervals. Increasing the batch size will better utilize the GPU and result in a higher throughput, but it will also increase the average wait time in the queue, increasing latency. So ideally, you want to adjust the batch size and frequency dynamically, depending on the incoming traffic. This technique works for all neural networks, not just LLMs.

With LLMs, you can also use a *request scheduler* that groups requests of similar lengths, reducing the need for mask tokens to pad shorter sequences. Or you can go even further and implement *in-flight batching* (also called *continuous batching*): at each generation step, new requests join the batch, and completed requests leave it. As a result, each generation step produces tokens for different positions depending on the instance (e.g., it might generate the first token for a request and simultaneously generate the third token a couple others, and so on). This removes the need for masking entirely, improves GPU utilization, and reduces the risk of long requests blocking short ones. Moreover, this approach gives you more granularity with scheduling: for example, if a request is taking too long to generate, the scheduler can temporarily place it in the queue while other requests are being processed.

In-flight batching is implemented by several libraries and toolkits, including Hugging Face Text Generation Inference (TGI), vLLM, TensorRT-LLM, SGLang, and others.

So far we have focused on speeding up decoding, but there are many other ways to speed up a transformer, including encoder-only transformers. Let's start by speeding up the multi-head attention layers.

Speeding Up Multi-Head Attention

The MHA layers are a key component of transformers. If we can accelerate them, then we can speed up transformers considerably. Looking back at the scaled dot-product attention equation, $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{QK}^\intercal / \sqrt{d_k})\mathbf{V}$, the most computationally expensive part is $\mathbf{QK}^\intercal$: this matrix multiplication computes one score for every query/key token pair. If L_q and L_k are the lengths of the query and key, then the result is an $L_q \times L_k$ matrix. In a self-attention layer, the query and the key have the same length ($L_q = L_k$) since they are two projections of the same sequence, so $\mathbf{QK}^\intercal$ is a square matrix containing L_k^2 cells: the number of cells grows with the square of the key length, which is why compute scales quadratically with the sequence length, making it impractical to feed a very long document to a transformer.

Luckily, there are various ways to speed up MHA:

- Drastically reducing the number of key tokens that each query token pays attention to: this is called *sparse attention*.
- Tweaking the scaled dot-product attention equation to get a quick approximation rather than the exact result: this is *approximate attention*.
- Simplifying the MHA architecture by sharing projections such as **K** and **V** across attention heads.
- Taking full advantage of the hardware architecture, using FlashAttention.

Let's go through each of these, starting with sparse attention.

Sparse Attention

In a standard masked MHA layer, each query token attends to every single key token up to its position (see the lefthand side of Figure 17-3). That's a lot of tokens to attend to: why not pay attention to only a subset? We will look at some of the most influential architectures based on this idea: Sparse Transformer, Longformer, BigBird, and Routing Transformer.

Sparse Transformer

The *Sparse Transformer* architecture was introduced by OpenAI researchers in a 2019 paper:[4] it's a decoder-only architecture where half of the attention heads can only attend to n local tokens (e.g., the last 256 tokens), while the other half can only attend to every n tokens globally. This number n is called the *stride*, and the authors recommended choosing a value close to the square root of the maximum sequence length, for maximum efficiency. The authors proposed two variants:

- Strided sparse attention (see the center of Figure 17-3), where both the local and global masks slide smoothly along with the query token. The authors found that this variant works best with images and data with a clear periodic structure that aligns well with the stride.
- Fixed sparse attention (see the righthand side of Figure 17-3), where the query is split into fixed blocks of n tokens each, and the local mask is limited to the query's block, while the global mask includes the last key token of each block. This variant is preferred for text and other data without a clear periodicity.

4 Rewon Child et al., "Generating Long Sequences with Sparse Transformers", arXiv preprint arXiv:1904.10509 (2019).

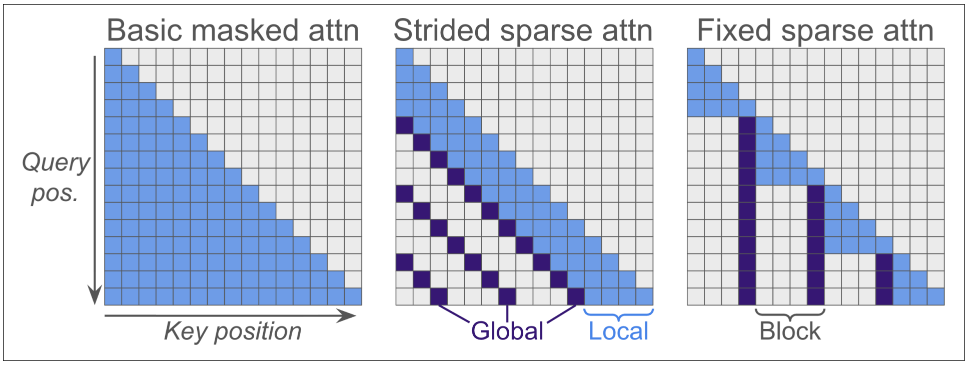


Figure 17-3. Regular causal attention mask (left), and two variants for the Sparse Transformer (center: strided, right: fixed). A colored cell indicates that the query token can attend the key token. In the Sparse Transformer, half of the attention heads use the blue mask while half use the dark purple mask.

With this approach, the computational complexity is drastically reduced: each query token can only attend to up to n local tokens, plus up to i / n global tokens, where i is the query token's position. For example, when using a window size of $n = 256$, a sequence of 10,000 tokens only requires computing about 5 million attention scores in total, whereas regular attention would require computing 100 million: that's a 20× speed up! More generally, this architecture scales with $O(L_q\sqrt{L_{max}})$, which is much better than $O(L_qL_k)$.

However, this speed up does come at a cost: information has to travel through global tokens to reach nonlocal tokens, which can cause *information bottlenecks*. In practice, this means that some critical nonlocal information may sometimes be missed. For example, if the conclusion of an essay must refer to a particular detail in the text, the reference may be inaccurate. The Sparse Transformer architecture is best for tasks such as summarizing or classifying long documents, but it can struggle on tasks such as textual entailment or reasoning, where attention to detail is often critical.

We can do better! Let's now turn to the Longformer architecture.

Longformer

In April 2020, researchers from the Allen Institute for AI proposed the *Longformer* (or long-document transformer), another transformer architecture based on sparse attention. As opposed to the Sparse Transformer, Longformer has an encoder-only architecture (based on RoBERTa), so its attention masks allow each query token to attend to key tokens located after it, making these masks symmetrical about the main diagonal (see Figure 17-4). The paper also introduced the *Longformer Encoder-Decoder* (LED), with an architecture similar to the original Transformer, but using sparse attention in the encoder.

The Longformer is based on a simple sliding window for local attention, similar to local attention in strided sparse attention (except it is centered on the query token's position). The width of the window is quite large (e.g., 512 tokens), so the Longformer does not provide any benefits for small documents.

Most importantly, the Longformer uses a fixed number of global tokens, chosen with care for each task. For example, for classification tasks, you may use the class token [CLS] (introduced in Chapter 15) as the only global token (see the lefthand side of Figure 17-4): it can "see" every token, and every token can "see" it, so it serves as a hub for information to propagate between distant tokens. For multiple choice question answering (MCQA), you may get better results by making all the question tokens global, since it's often important for query tokens to be able to refer to the question details. You can experiment with different global tokens for your task and see what works best.

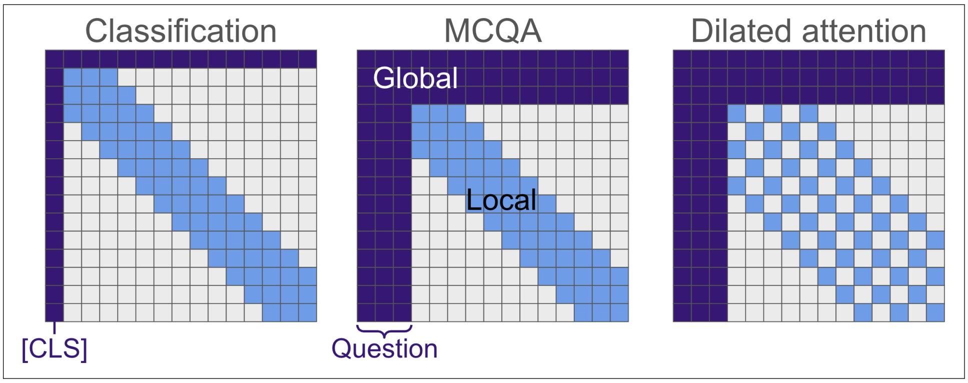


Figure 17-4. Longformer attention mask: the global tokens are task-motivated (left and center); the Longformer also uses dilated attention (right)

The Longformer paper also proposed using shorter windows in lower layers and longer windows in higher layers: since lower layers often focus on small-scale patterns, this technique saves compute without significantly reducing the model's accuracy.

Moreover, the authors proposed using dilated windows for local attention (see the righthand side of Figure 17-4), inspired by the WaveNet architecture (introduced in Chapter 12). By increasing the dilation head in upper layers, it's possible to expand the receptive field of each query token without exploding the computational budget.

It's possible to use the same sparse attention mechanism in a decoder: you just need to apply a causal mask on top of the local and global masks.

Thanks to the constant number of global tokens and the dilated attention windows, this architecture scales linearly to very long documents, and it generally performs better than the Sparse Transformer. Can we do even better?

BigBird

In July 2020, Google researchers introduced BigBird,[5] which you can think of as Longformer minus dilation plus random attention: just like in Longformer, each query token attends to a window of local tokens (but not dilated), as well as a fixed number of global tokens, plus a small number of random tokens. For example, in Figure 17-5, each query token can attend local tokens, global tokens, and two random tokens.

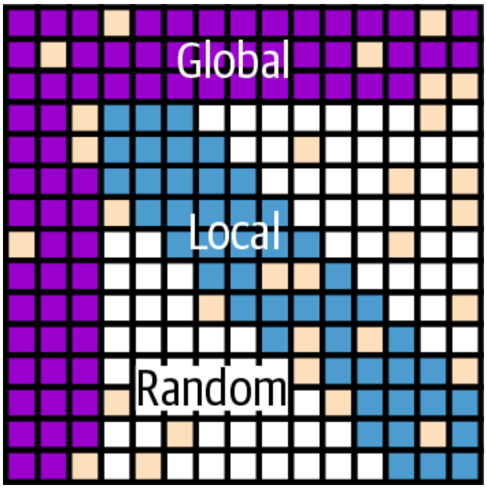


Figure 17-5. BigBird attention mask: similar to Longformer minus dilation plus random attention

The authors proposed two variants: BigBird-ITC (which stands for internal transformer construction), and BigBird-ETC (extended transformer construction). In the former, the global tokens are chosen among existing tokens, while in the latter we add a few extra global tokens at the start, such as [CLS]. In practice, these extra tokens trade a little bit of speed for a higher accuracy.

Importantly, the paper showed that the introduction of random attention allows BigBird to offer strong theoretical guarantees: despite scaling to long sequences linearly, BigBird is just as expressive as the original Transformer architecture. In particular, it can theoretically model any continuous function. In fact, it can even execute any

5 Manzil Zaheer et al., "Big Bird: Transformers for Longer Sequences", arXiv preprint arXiv:2007.14062 (2020).

algorithm, assuming we call the model repeatedly, feeding its outputs back to itself, and assuming the float precision is sufficient: under these conditions, BigBird is *Turing complete*. That said, BigBird does require more layers than the original Transformer architecture to reach the same accuracy: this is because information must go through multiple layers to sufficiently propagate across tokens. Overall, BigBird is great on tasks involving long documents, and it generally outperforms Longformer.

Each attention head uses a different random attention mask; this reduces the number of layers needed for information to propagate between distant tokens. Moreover, each random attention mask is sampled when the model is created, then it remains constant; this makes it easier for the model to learn how to exploit these connections to propagate information across distant tokens.

As you might expect, several BigBird variants are available on the Hugging Face Hub. For example:

```
from transformers import pipeline

model_id = "google/bigbird-roberta-base"
pipeline = pipeline(task="fill-mask", model=model_id)
pipeline("She took some [MASK] medicine as she was feeling ill.")
```

Let's now look at one last transformer architecture based on a sparse attention: the Routing Transformer.

Routing Transformer

The *Routing Transformer* was proposed by Google researchers in March 2020,[6] a few weeks before the Longformer.

The previous sparse attention mechanisms all decided which query tokens should attend to which key tokens strictly based on their positions. In contrast, each attention head in the Routing Transformer starts by grouping both the query tokens and the key tokens (together) into k clusters during each forward pass, then it computes the scaled dot-product attention separately for each cluster. This means that each query token can only attend to the key tokens located within the same cluster (see Figure 17-6). This restriction doesn't degrade quality too much since a query token mostly pays attention to its most similar key tokens anyway, and similar tokens are very likely to belong to the same cluster.

6 Aurko Roy et al., "Efficient Content-Based Sparse Attention with Routing Transformers", arXiv preprint arXiv:2003.05997 (2020).

Clustering is based on the similarity between tokens, not their positions. However, since each token carries positional information thanks to the positional encodings, clustering is influenced both by semantics and position: it has a bias toward local tokens. This is usually a good thing, since local tokens are generally more relevant than distant ones.

To perform the clustering step, the Routing Transformer uses a variant of mini-batch *k*-means (introduced in Chapter 8): each attention head has *k* trainable centroids (i.e., each centroid is a vector of the same dimensionality as the tokens), and during the forward pass, for each centroid, the set of *s* tokens closest to that centroid forms a cluster; *s* is roughly equal to the number of tokens divided by *k*. This process produces *k* clusters of equal sizes, which can partly overlap. During training, each *k*-means centroid is updated at each iteration using an exponential moving average of the closest *s* tokens. At inference time, the centroids are fixed.

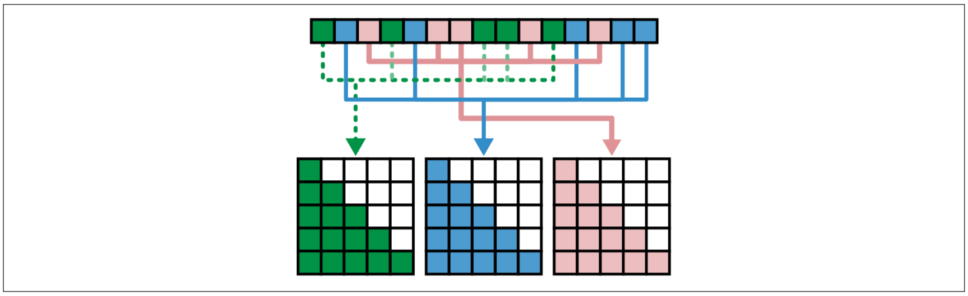


Figure 17-6. Routing attention: query tokens can only attend to key tokens within the same cluster

Before the clustering step, each attention head normalizes the query and key tokens to unit vectors. This makes it possible to use the dot product as a measure of similarity between the tokens and the centroids, which speeds up *k*-means. The normalized tokens are also used when computing the scaled dot-product attention.

This technique gives excellent results, even on short sequences. The authors recommend using a number of clusters *k* close to the square root of the maximum sequence length, which ensures that the transformer scales with $O(L_q\sqrt{L_{max}})$, just like the Sparse Transformer. However, the Routing Transformer is fairly complex to implement, so it hasn't gained a lot of traction.

We have discussed four transformer architectures based on sparse attention: the Sparse Transformer, Longformer, BigBird, and the Routing Transformer. In fact, Swin transformers would also belong in this section if we hadn't already discussed them in Chapter 16. All of these models speed up MHA considerably, use much less memory, and scale linearly (or at least sub-quadratically) to large inputs, however they can struggle with some tasks that are sensitive to long-range details.

Now let's move on to a very different approach to speeding up attention.

Approximate Attention

Approximate attention methods reduce the computational complexity and memory usage of MHA by computing an approximation of the scaled dot-product attention equation: this is done either by tweaking the equation itself to compute a fast approximation, or by compressing the inputs. We already saw two examples of the latter approach in Chapter 16:

- The Pyramid Vision Transformer (PVT) uses spatial reduction attention (SRA), which compresses the key and value by shrinking them along the spatial dimensions: this makes its computational complexity and memory usage $O(L_q L_k / r^2)$ where r is the reduction factor, instead of $O(L_q L_k)$ for regular attention.
- The Perceiver uses cross-attention to compress the input sequence into a smaller latent sequence of constant length ℓ. This allows it so scale linearly with the sequence length: it's $O(\ell L_k)$.

In this section, we will discuss four more influential transformer architectures based on approximate attention: Reformer, Linformer, and Performer. Let's start with the Reformer architecture.

Reformer: LSH-based approximate attention

The Reformer architecture, proposed in January 2020 by a team of Google and U.C. Berkeley researchers,[7] uses a combination of several techniques to speed up transformers, including *locality-sensitive hashing (LSH) attention*, which has a computational complexity of $O(L_q \log(L_k))$ instead of $O(L_q L_k)$, while only slightly degrading accuracy. Let's see how it works.

In the scaled dot-product attention equation, Attention($\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$) = softmax($\mathbf{QK}^\intercal / \sqrt{d_k}$)$\mathbf{V}$, the computationally nasty $\mathbf{QK}^\intercal$ term is inside the softmax function. Recall that the softmax function's output is mostly determined by the largest terms in its inputs, which in this case correspond to the query/key pairs with the largest dot-product: in other words, the output is mostly influenced by the most similar pairs. So if we can find an algorithm that can quickly identify the most similar query/key pairs, we won't need to compute $\mathbf{QK}^\intercal$ for all possible pairs: just for the most similar ones. The result won't be perfectly accurate since we will be neglecting all the contributions from less aligned query/key pairs, but it should be pretty close.

7 Nikita Kitaev et al., "Reformer: The Efficient Transformer", arXiv preprint arXiv:2001.04451 (2020).

One such algorithm was introduced in 1998 by Piotr Indyk and Rajeev Motwani:[8] *locality-sensitive hashing* (LSH). Given a set of high-dimensional vectors, LSH can quickly group them into clusters of nearby vectors. To do this, each vector is passed through a hash function that directly outputs the cluster ID. Which hash function should we choose? Well, if we used a hash function such as MD5 or SHA256 (modulo the number of clusters) then LSH would just group vectors pseudorandomly, which wouldn't be useful at all. Therefore, we must instead use a hash function that tends to produce similar hashes for similar input vectors: this is called a locality-sensitive hash function.

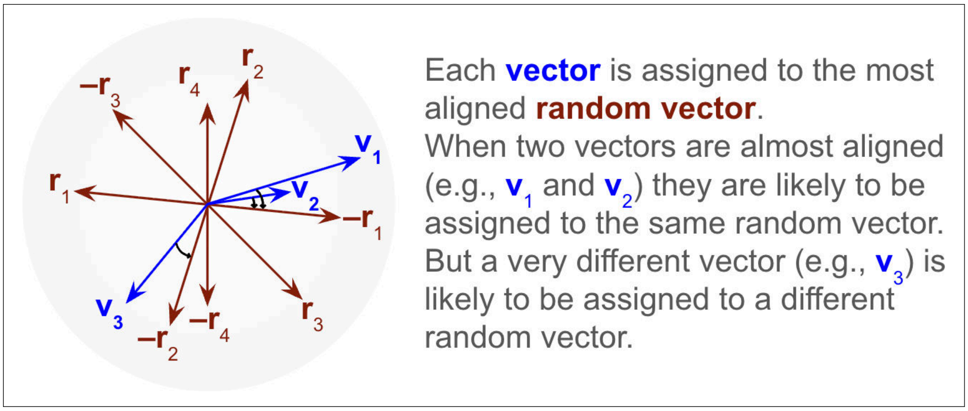
Various LSH functions have been invented over time. The *cross-polytope LSH*, more commonly called *angular LSH*, is a particularly fast and accurate one. It was proposed in 2015 by Alexandr Andoni et al.[9] Here is how it works:

- If k is the desired number of clusters, we start by creating a list $\mathbf{L}$ of random vectors by sampling $\mathbf{r}_1$ to $\mathbf{r}_{k/2}$ independently from a Gaussian distribution.
- We then append the opposite vectors $-\mathbf{r}_1$ to $-\mathbf{r}_{k/2}$ to the list. It now contains k vectors.
- For each vector $\mathbf{v}$ that we want to hash, we find the vector in $\mathbf{L}$ that is most aligned with it, and we use its index in $\mathbf{L}$ as the hash for vector $\mathbf{v}$. For example, if $\mathbf{v}$ is most aligned with the vector at index 42 in $\mathbf{L}$, then its hash is 42.

This algorithm can be used as a locality-sensitive hash function because if two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ are very similar (i.e., almost aligned), then it's highly likely that they will both be most aligned with the same vector in $\mathbf{L}$, so they will have the same hash (see Figure 17-7). Conversely, dissimilar vectors are likely to have a different hash.

8 Piotr Indyk and Rajeev Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality", *Proceedings of the 30th Annual ACM Symposium on Theory of Computing* (1998).

9 Alexandr Andoni et al., "Practical and Optimal LSH for Angular Distance", arXiv preprint arXiv:1509.02897 (2015).



$-\mathbf{r}_3$
$\mathbf{r}_4$
$\mathbf{r}_2$
$\mathbf{v}_1$
$\mathbf{r}_1$
$\mathbf{v}_2$
$-\mathbf{r}_1$
$\mathbf{v}_3$
$-\mathbf{r}_4$
$-\mathbf{r}_2$
$\mathbf{r}_3$
Each vector is assigned to the most aligned random vector.
When two vectors are almost aligned (e.g., $\mathbf{v}_1$ and $\mathbf{v}_2$) they are likely to be assigned to the same random vector.
But a very different vector (e.g., $\mathbf{v}_3$) is likely to be assigned to a different random vector.

Figure 17-7. Angular LSH: two similar vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ are likely to share the same most similar random vector

To implement angular LSH efficiently using PyTorch, we can use the following function:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

def angular_lsh(vectors, k):
    R = torch.randn(vectors.size(-1), k // 2, device=vectors.device)
    normalized_vectors = F.normalize(vectors, p=2.0, dim=1)
    V_proj = normalized_vectors @ R
    V_concat = torch.cat([V_proj, -V_proj], dim=1)
    return torch.argmax(V_concat, dim=1)
```

Let's go through this code:

- We start by generating a random matrix where the column vectors are $\mathbf{r}_1$ to $\mathbf{r}_{k/2}$.
- Next, we normalize the input matrix so that all row vectors have length 1.
- Then we multiply both matrices: since the input vectors are normalized and the random vectors have similar lengths,[10] the resulting matrix roughly contains the cosine similarity score for each pair of input vector and random vector (times a constant).
- We then concatenate the opposite of this matrix along the horizontal dimension: this gives us the similarity scores between each vector and the opposite of vectors $\mathbf{r}_1$ to $\mathbf{r}_{k/2}$, for free.

10 As the number of dimensions d increases, the length of a random vector sampled from a Gaussian distribution approaches the square root of d.

- We now have all the similarity scores we need, so we are ready to return the index of the highest similarity score in each row.

Let's try using this function to cluster 16 random 512-dimensional vectors into $k = 4$ clusters:

```
>>> torch.manual_seed(42)
>>> vectors = torch.rand(16, 512)
>>> angular_lsh(vectors, k=4)
tensor([2, 2, 0, 3, 0, 2, 2, 2, 2, 1, 1, 3, 3, 1, 2, 1])
```

Now we could just cluster all query tokens and key tokens together using angular LSH, then make each query token attend to only the key tokens within the same cluster. This would be fast, but sadly too approximate: indeed, angular LSH can sometimes assign similar vectors to different clusters, and if a query token ends up in a cluster that does not contain some of its most similar key tokens, then the softmax function will be missing its most important inputs; the result will be too imprecise.

To solve this issue, there's a simple solution: we can just run several independent rounds of angular LSH (e.g., 4 rounds) and make each query token attend to all key tokens that it was clustered with in at least one round. With that, we can be reasonably confident that each query token will attend to all similar key tokens.

The Reformer architecture uses several other advanced techniques:

- Reversible residual layers, introduced in Chapter 12.
- *Chunked feedforward layers*, where the sequence is processed chunk by chunk in the feedforward layers to reduce peak memory usage.
- *Axial positional encodings* where the large positional embedding matrix is replaced with two much smaller ones, like the x and y positional embeddings in some vision models.
- Shared Query/Key (Shared-QK), where instead of using two different projections of the same input sequence to produce the query and key for each attention head, we use the same projection, so the key and query sequences are the same: $\mathbf{Q} = \mathbf{K}$. This works well in combination with LSH attention as it makes it easier to mix queries and keys. However, it does reduce the model's flexibility, and ultimately its accuracy.

The Reformer proved that approximating attention was possible, and it pushed the community to find more compute- and memory-saving tricks. A few months later, another approximate-attention model was proposed: the Linformer. It's surprisingly simple, especially in comparison with the Reformer, as we will see now.

Linformer: Low-rank approximate attention

The Linformer architecture was introduced in June 2020 by Meta researchers.[11] To understand how it works, let's start by rewriting the scaled dot-product equation as Attention(**Q**, **K**, **V**) = **PV**, where **P** = softmax($\mathbf{QK}^\intercal / \sqrt{d_k}$). The matrix **P** is called the *context mapping matrix*, and its shape is $L_q \times L_k$. Importantly, the authors analyzed various pretrained transformers, and they observed that this matrix is always close to a very low-rank matrix. In self-attention layers (where $L_q = L_k$), **P** is generally close to a matrix of rank $r \approx \log(L_q)$. This means that it should be possible to compress **P** considerably without losing too much information.

But how? We could compress **P** using PCA, but we would just be wasting compute and accuracy for nothing, since the cost of computing **P** is precisely what we want to reduce. So the authors proposed shortening the keys and values using two linear projections **E** and **F**, learned during training. To be clear: this reduces the *length* of the key and value sequences (i.e., the number of tokens), *not* the dimensionality of each token. This *low-rank attention* mechanism is based on Equation 17-1, which is represented in Figure 17-8.

11 Sinong Wang et al., "Linformer: Self-Attention with Linear Complexity", arXiv preprint arXiv:2006.04768 (2020).

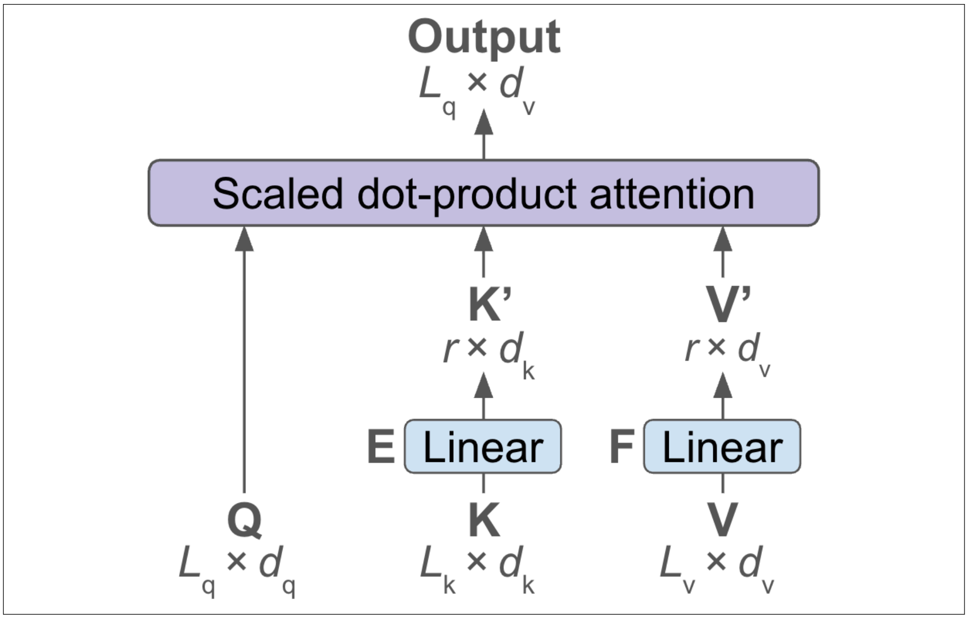


Figure 17-8. Low-rank attention: the key and value are shortened using linear projections

Equation 17-1. Linformer’s low-rank approximate attention

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) \approx \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}'^{\top}}{\sqrt{d_k}}\right)\mathbf{V}'$$
$$\text{where } \mathbf{K}' = \mathbf{E}\mathbf{K} \text{ and } \mathbf{V}' = \mathbf{F}\mathbf{V}$$

As you can see, it is the normal equation for scaled dot-product attention, except we are using the compressed key **K′** and the compressed value **V′** instead of **K** and **V**. In this equation:

- **E** is a trainable matrix of shape $r \times L_k$.
- **F** is a trainable matrix of shape $r \times L_k$ (recall that $L_v = L_k$).
- r is the compressed sequence length, which is a hyperparameter you can tune (e.g., 64).

Since L_k varies depending on the input sequence, the authors proposed using trainable matrices **E′** and **F′** with enough columns for the longest possible sequences, then truncating them on the fly to the appropriate number of columns.

The authors also experimented with various parameter sharing techniques to further reduce memory usage:

- Sharing **E** and **F** across all attention heads within each layer.
- Using the same projection matrix **G** to compress both key and value (**K′** = **GK** and **V′** = **GV**), and using this matrix **G** across all attention heads within each layer.
- Using the same matrix **G** to compress both the key and value, across all attention heads and all layers.

Since the key and value sequences are much shorter, with just *r* tokens each, low-rank attention is significantly faster and less memory-hungry than regular attention. Most importantly, since *r* is fixed (it is chosen before training and never changes), low-rank attention scales linearly to long sequences, hence the name of the architecture—Linformer—which stands for *linear transformer*.

Although a given Linformer model has a fixed *r* and scales linearly to long sequences, the optimal value for *r* increases roughly with the logarithm of the input sequence length, so you will have to increase *r* if you want quality to remain acceptable for longer sequences. Therefore, one could argue that Linformer actually scales with $O(L_q \log(L_k))$.

In short, the Linformer architecture scales well, it's relatively easy to implement, and it's accuracy is close to the Reformer's (as long as *r* is not too small). It also inspired later architectures based on other low-rank approximation methods, such Nyströmformer.[12] Now let's turn to the Performer, which approximates attention using a very different mathematical technique.

Performer: Kernel-based approximate attention

The Performer architecture was proposed in September 2020 by a team of researchers from Google, DeepMind, the University of Cambridge, and the Alan Turing Institute.[13] Just like Linformer, Performer scales linearly to long input sequences thanks to a quick approximation of the scaled dot-product attention equation. However, it does so in a very different way.

12 Yunyang Xiong et al., "Nyströmformer: A Nyström-Based Algorithm for Approximating Self-Attention", arXiv preprint arXiv:2102.03902 (2021).

13 Krzysztof Choromanski et al., "Rethinking Attention with Performers", arXiv preprint arXiv:2009.14794 (2020).

This section is quite math-heavy, so please feel free to skip it if you're not interested in the mathematical details. The short version is that Performer computes a quick approximation of dot-product attention using a kernel trick, allowing it to scale linearly with the input sequence length.

To understand how Performer works, we first have to remember that the softmax function computes the exponential of every term in its input matrix, then divides each row by its sum: this ensures that each row adds up to 1. This can be written as: softmax($\mathbf{M}$) = $\mathbf{D}^{-1}\mathbf{A}$ where $\mathbf{A}$ = exp($\mathbf{M}$) and $\mathbf{D}$ is a diagonal matrix where each value in the main diagonal corresponds to the sum of each row in $\mathbf{A}$, and $\mathbf{D}^{-1}$ is its inverse (i.e., each diagonal term is replaced by its inverse).

We can now rewrite the scaled dot-product attention equation as Attention($\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$) = softmax($\mathbf{QK}^\intercal$)$\mathbf{V}$ = $\mathbf{D}^{-1}\mathbf{AV}$, where $\mathbf{A}$ = exp($\mathbf{QK}^\intercal$). Note that I left out the scaling factor $\frac{1}{\sqrt{d_k}}$ to simplify the equation: assume that $\mathbf{Q}$ and $\mathbf{K}$ have each been scaled by the square root of this factor (i.e., $\sqrt{\frac{1}{\sqrt{d_k}}} = d_k^{-\frac{1}{4}}$) so after the matrix multiplication $\mathbf{QK}^\intercal$, we get the desired factor ($d_k^{-\frac{1}{4}} \times d_k^{-\frac{1}{4}} = \frac{1}{\sqrt{d_k}}$).

Now here comes the crucial part. The authors proposed a way to quickly approximate $\mathbf{A}$: they defined a function ϕ (which I will present shortly) such that $\mathbf{A}$ = exp($\mathbf{QK}^\intercal$) ≈ $\phi(\mathbf{Q})\phi(\mathbf{K})^\intercal$. This allowed them to propose the approximate attention equation shown in Equation 17-2.

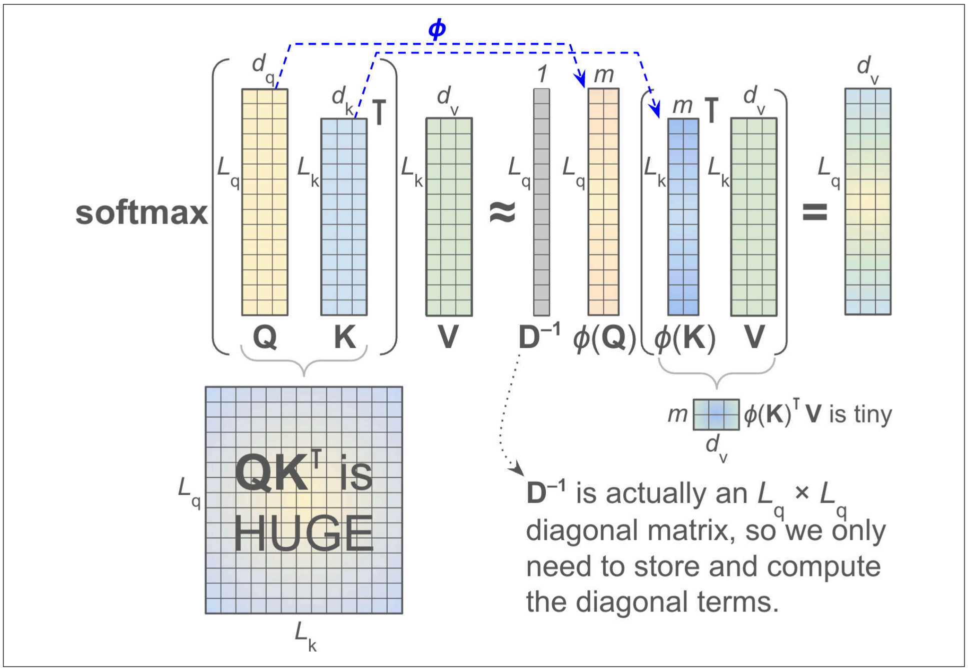
Equation 17-2. Performer's kernel-based approximate attention

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) \approx \mathbf{D}^{-1}\phi(\mathbf{Q})\phi(\mathbf{K})^\intercal\mathbf{V}$$

$$\text{where } \mathbf{Q} \text{ and } \mathbf{K} \text{ are already scaled by } d_k^{\frac{1}{4}}$$

Notice that this equation is just the product of several matrices, and since matrix multiplication is associative—meaning that ABC = (AB)C = A(BC)—we can start by computing $\mathbf{G} = \phi(\mathbf{K})^\intercal\ \mathbf{V}$, then compute $\mathbf{H} = \phi(\mathbf{Q})\mathbf{G}$, and finally compute Attention ≈ $\mathbf{D}^{-1}\mathbf{H}$.

The important point to note is that all of these steps involve fairly small matrices (see Figure 17-9): we never need to compute $\mathbf{QK}^\intercal$, which is a huge $L_q \times L_k$ matrix. Indeed, $\phi(\mathbf{K})^\intercal$ is a small $m \times L_k$ matrix (where L_k may be large, but m is a small hyperparameter you can tune, e.g., 256), and $\mathbf{V}$ is an $L_k \times d_v$ matrix (small since d_v is small, it's the value's dimensionality, e.g., 64), so $\mathbf{G}$ is a tiny $m \times d_v$ matrix. Similarly, $\phi(\mathbf{Q})$ is a small $L_q \times m$ matrix, so $\mathbf{H}$ is a small $L_q \times d_q$ matrix (where d_q is the query's dimensionality, e.g., 64). $\mathbf{D}^{-1}$ is a large $L_q \times L_q$ matrix, but since it's a diagonal matrix, it doesn't require much computations or memory.[14] Overall, this approximate attention equation requires $O(Lmd)$ computations, and $O(Lm + Ld + md)$ memory space, where $L = L_k + L_q$ and d is the model's dimensionality. As you can see, this offers linear scaling with regards to L.



ϕ
softmax
Q
K
T
V
≈
D⁻¹
ϕ(Q)
ϕ(K)
T
V
=
QKᵀ is HUGE
m ϕ(K)ᵀ V is tiny
D⁻¹ is actually an Lq × Lq diagonal matrix, so we only need to store and compute the diagonal terms.

Figure 17-9. Performer's kernel-based attention

If you have read the online Appendix C on support vector machines (SVMs), you might have noticed that Equation 17-2 is a *kernel-based* equation. A kernel is a function $K(\mathbf{x}, \mathbf{y})$ that computes the dot product $f(\mathbf{x})\cdot f(\mathbf{y})$ for some function f, where f maps its input vector to another space. In an SVM, we never actually compute $f(\mathbf{x})$ or $f(\mathbf{y})$ since the kernels we use can also be expressed using simple equations based directly on $\mathbf{x}$ and $\mathbf{y}$. In fact, we don't even have to know what f is: it's often just an implicit function that maps input vectors to some higher-dimensional space (even infinite-dimensional in some cases). The whole point is to avoid having to explicitly map $\mathbf{x}$ and $\mathbf{y}$ to a different space, while still obtaining the same result: $f(\mathbf{x})\cdot f(\mathbf{y})$. This is called the *kernel trick*.

14 In fact, $\mathbf{D}$ is typically computed by appending a column of 1s to $\mathbf{V}$, then evaluating $\mathbf{A} = \phi(\mathbf{Q})\phi(\mathbf{K})^\top\mathbf{V}$. The last column of $\mathbf{A}$ contains the diagonal terms of $\mathbf{D}$. So if we divide each row of $\mathbf{A}$ by its last element, then remove the last column, we get the same result as Equation 17-2.

However, in the Performer, we do the exact opposite: we explicitly compute $f(\mathbf{x})\cdot f(\mathbf{y})$ to avoid having to evaluate $K(\mathbf{x}, \mathbf{y})$. Specifically, we compute $\phi(\mathbf{Q})\phi(\mathbf{K})^\intercal$ so we don't have to evaluate the *exponential kernel* $K(\mathbf{X}, \mathbf{Y}) = \exp(\mathbf{Q}\mathbf{K}^\intercal)$. Isn't math just beautiful?

Now the difficulty is to find a function ϕ such that $\phi(\mathbf{Q})\phi(\mathbf{K})^\intercal \approx \exp(\mathbf{Q}\mathbf{K}^\intercal)$. That's exactly what the authors did. Their method, named *Fast Attention Via positive Orthogonal Random features* (FAVOR+), is actually even more general: it can provide such a function ϕ for various kernel functions, not just the exponential function. However, we will focus on the exponential kernel.

The exponential kernel is sometimes (imprecisely) called the *softmax kernel* (e.g., in the Performer paper).

To find ϕ, the authors explored the vast family of functions of the form shown in Equation 17-3 (this is a vectorized form of equation (5) in the Performer paper). I've also represented it in Figure 17-10.

Equation 17-3. The family of functions explored by the Performer authors for the feature map ϕ (vectorized)

$$\phi(\mathbf{X}) = \frac{h(\mathbf{X})}{\sqrt{m}}\left[f_1(\mathbf{XW}); f_2(\mathbf{XW}); \cdots; f_\ell(\mathbf{XW})\right]$$

In this equation:

- $\phi(\mathbf{X})$ maps $\mathbf{X}$ from a d-dimensional space to an ℓm-dimensional space (that's ℓ times m), where m is a hyperparameter (e.g., 256), and ℓ is the number of functions f_1 to f_ℓ.
- $h(\mathbf{X})$ is any function that returns a scaling factor for each row of $\mathbf{X}$.
- f_1 to f_ℓ are any itemwise real functions (e.g., exp).
- $\mathbf{W}$ is any nontrainable $d \times m$ matrix. It is used to map $\mathbf{X}$ from d dimensions to m dimensions. $\mathbf{W}$ can be deterministic or random. If it's random, then each column vector is sampled independently from the same distribution; typically a standard Gaussian distribution.
- The notation $[\cdot;\cdot;\ldots;\cdot]$ means concatenation along the horizontal axis.

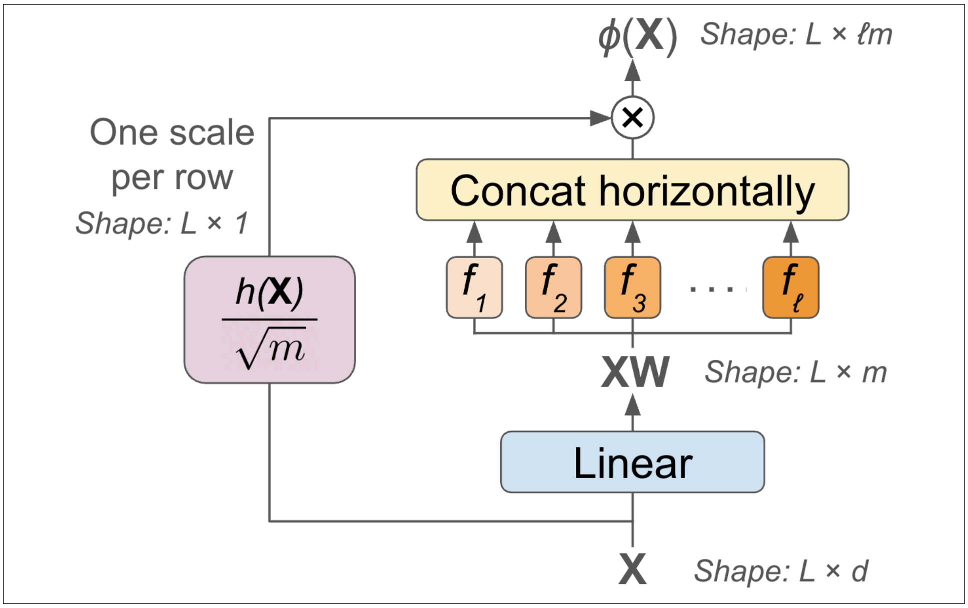


Figure 17-10. A visual representation of Equation 17-3

The authors mathematically proved that the following function from this family can be used to approximate the exponential kernel (see appendix F in the Performer paper for the proof):

- $h(\mathbf{X})$ computes $\exp\left(-\frac{1}{2}\parallel\mathbf{x}_i\parallel^2\right)$ for each row $\mathbf{x}_i$ in $\mathbf{X}$, where $\|\mathbf{x}_i\|$ is the length of the vector $\mathbf{x}_i$.
- $\ell = 1$
- $f_1(x) = \exp(x)$
- $\mathbf{W}$ is random, with each element sampled independently from the standard Gaussian distribution: `W = torch.randn(d, m)`.
- m is a hyperparameter you can tune (e.g., 256): increasing m makes the approximation more accurate, but it also linearly increases the memory and computational costs.

The authors also proposed a second function, less efficient but more precise:

- $h(\mathbf{X})$ computes $\frac{1}{\sqrt{2}}\exp\left(-\frac{1}{2}\parallel\mathbf{x}_i\parallel^2\right)$ for each row $\mathbf{x}_i$.
- $\ell = 2$
- $f_1(x) = \exp(x)$ and $f_2(x) = \exp(-x)$

- **W** and *m* are just like in the first function.

Let's implement the first function using PyTorch:

```
def phi(X, W):
    squared_norms = X.square().sum(dim=-1, keepdim=True)
    return torch.exp(X @ W - squared_norms / 2) / W.size(-1) ** 0.5
```

This implementation supports inputs with more than two dimensions, which is useful since **Q** and **K** typically have a shape of [*batch size*, *heads*, *length*, *dim*], and **W** has a shape of [*heads*, *length*, *dim*]: this makes it possible to process all sequences in the batch and all attention heads at the same time. Moreover, instead of computing `exp(X @ W)` and `exp(-squared_norms / 2)` separately, we merge them into the same exponential, which is more efficient.

Sadly, this implementation is not very stable, numerically: indeed, if any value in **XW** is very large, the exponential may overflow to infinity. To avoid this, we can subtract the maximum value v of **XW** from all values (independently for each instance in the batch and for each attention head) before computing the exponential: this means that the output will be off by a constant factor $\exp(-v)$ (independently for all instances and heads). Luckily, this doesn't matter, since we will normalize $\phi(\mathbf{Q})\phi(\mathbf{K})^\intercal$ later. This technique is called *safe softmax*. So let's implement the stable version of `phi()`:

```
def phi(X, W, dim_subtract_max=(-2, -1)):
    squared_norms = X.square().sum(dim=-1, keepdim=True)
    X_proj = X @ W
    max_vals = X_proj.amax(dim=dim_subtract_max, keepdim=True)
    return torch.exp(X_proj - max_vals - squared_norms / 2) / W.size(-1) ** 0.5
```

By default, we subtract the maximum value of the full matrix **XW** from all of its values (independently for each instance and head). However, if we set `dim_subtract_max=-1`, we can subtract the maximum value independently for each row, thereby preserving some precision: we can afford to do this when computing $\phi(\mathbf{Q})$, since all of the rows of $\phi(\mathbf{Q})\phi(\mathbf{K})^\intercal$ will be normalized independently.

Now we're ready to put everything together into the `FavorAttention` module:

```
class FavorAttention(nn.Module):
    def __init__(self, d_model, n_heads, n_features):
        super().__init__()
        self.d_head = d_model // n_heads
        W = torch.randn(n_heads, self.d_head, n_features)  # h, d, m
        self.register_buffer("W", W)

    def forward(self, Q, K, V):
        scale = self.d_head ** -0.25
        Qp = phi(Q * scale, self.W, dim_subtract_max=-1)
        Kp = phi(K * scale, self.W)
        D = Qp @ Kp.sum(dim=-2).unsqueeze(-1)  # B, h, Lq, 1
        Kp_T_V = Kp.transpose(-2, -1) @ V  # B, h, m, d
        return (Qp @ Kp_T_V) / (D + 1e-6)
```

Let's go through this code:

- The constructor generates the random matrix $\mathbf{W}$ and registers it as a buffer, which is useful when saving the model.
- The `forward()` method starts by computing the standard scale for the dot-product attention: $\frac{1}{\sqrt{d_k}}$. It then applies ϕ using the `phi()` function to $\mathbf{Q}$ and $\mathbf{K}$ (after scaling them, as explained earlier). Note that we set `dim_subtract_max=-1` for the query, as we just saw.
- Next, we compute $\mathbf{D}$, which contains the sum of each row in `Qp @ Kp.T`. We do this efficiently, without actually having to compute `Qp @ Kp.T` (calculating `Qp @ Kp.T` would defeat the whole purpose of FAVOR+, since this is a huge $L_q \times L_k$ matrix).
- Lastly, we compute the rest of Equation 17-2, and we divide by `D` for normalization. Note that we add a tiny value `1e-6` to avoid division by zero.

FAVOR+ was inspired by a 2007 technique named *random Fourier features*,[15] which can approximate various kernel functions using $\phi(\mathbf{X}) = \frac{2}{\sqrt{m}} \cos(\mathbf{XW} + \mathbf{b})$, where the column vectors of $\mathbf{W}$ and the vector $\mathbf{b}$ are sampled from a standard Gaussian distribution. However, this technique is limited to shift-invariant kernel functions, where $\mathrm{K}(\mathbf{x}, \mathbf{y})$ only depends on the difference $\mathbf{x} - \mathbf{y}$. However, the exponential kernel is *not* shift-invariant.

So far we've seen how to obtain fast attention via random feature maps: this covers the FAV and R parts of the FAVOR+ acronym. But what about O and +? The + just highlights the fact that $\mathbf{XW}$ is mapped to positive values (i.e., $\exp(\mathbf{XW})$). This helps improve the precision of the approximation, since the exponential kernel is always positive. The O stands for "orthogonal": the authors showed that by tweaking the random column vectors in $\mathbf{W}$ to make them orthogonal to each other, the approximation becomes significantly more accurate (especially when m is small), so we can afford to reduce m if we want to save even more compute and memory.

For this, the authors proposed using a standard technique named *Gram-Schmidt orthogonalization*: one way to implement it is to first factorize the matrix $\mathbf{W}$ using

15 Ali Rahimi and Benjamin Recht, "Random Features for Large-Scale Kernel Machines", *Advances in Neural Information Processing Systems* 20 (2007): 1177–1184.

QR decomposition, so $\mathbf{W} = \mathbf{QR}$, where $\mathbf{Q}$ contains orthogonal unit vectors and $\mathbf{R}$ is upper triangular. We then drop $\mathbf{R}$, keeping only $\mathbf{Q}$. Finally, we rescale $\mathbf{Q}$ by a factor of $\sqrt{d}$, thereby restoring the original average scale of the random vectors. Since a d-dimensional space cannot have more than d orthogonal vectors, we process $\mathbf{Q}$ by chunks of d columns at a time, so each chunk contains d orthogonal vectors. This whole process can be implemented in PyTorch in just a few lines of code:

```python
def orthogonalize(W):
    d_head = W.size(-2)
    W_orth = torch.cat([torch.linalg.qr(W_chunk)[0]
                        for W_chunk in W.split(d_head, dim=-1)], dim=-1)
    return W_orth * d_head ** 0.5
```

To use this function, simply add `W = orthogonalize(W)` in the constructor, right after generating `W`.

As you can see, while this technique is quite mathematically challenging (it took me some time to fully grasp it), it is not too difficult to implement, and it gives excellent results, especially when dealing with very long sequences.

Now that we've discussed several methods for sparse attention and approximate attention, let's look at another approach to accelerate multi-head attention: sharing projections across attention heads.

Sharing Projections Across Attention Heads

When an MHA layer receives a query, key, and value, it starts by applying a linear transformation to each of these, then it splits the results across all attention heads (see Figure 15-4). For example, if there are $h = 8$ attention heads and the input has a dimensionality $d = 512$, then each of the three linear transformations produces a 512-dimensional output, and after splitting the three outputs we get 8 projected queries, 8 projected keys, and 8 projected values, all 64-dimensional.

When we say *query*, *key*, or *value*, it can be unclear whether we are referring to the MHA layer's inputs, or the projections fed to any individual attention head. Luckily, the context usually makes it clear which one we are talking about, but whenever needed I will specify *input* query/key/value for the MHA's inputs, and note them $\mathbf{X}_\mathrm{Q}$, $\mathbf{X}_\mathrm{K}$, and $\mathbf{X}_\mathrm{V}$, and *projected* query/key/value for the projections fed to a given attention head, which I will denote as $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ (or $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$ to specifically refer to the inputs of the ith attention head).

This process relies on three $d \times d$ trainable projection matrices, which introduces a large number of model parameters and a significant computational cost for the projections. More importantly, the projections consume a vast amount of memory space. This is especially problematic at inference time when generating text, as the KV cache expands with each new token, potentially exhausting the available VRAM. This can crash the program, or at least cause painful slowdowns due to memory page swapping. Furthermore, reading the entire KV cache to generate each new token can easily saturate the memory bandwidth, drastically slowing down the model.

Do we really need all of these projections? Couldn't we share some of them across attention heads? We have already seen that the Reformer architecture shares the same projection for the query and key (shared-QK), and the Linformer authors proposed sharing the extra projection matrices **E** and **F** in various ways. In this section we will discuss three more ways to share projections: multi-query attention, grouped-query attention, and multi-head latent attention.

Multi-query attention (MQA)

The first paper to propose the idea of sharing projections across attention heads was published in 2019[16] by Noam Shazeer, one of the authors of the original Transformer architecture. He proposed to share the same projected key and value across all attention heads, highlighting the benefits for text generation. Since there is still a different projected query for each attention head, he called this technique *multi-query attention* (MQA).

Although MQA uses the same projected key and the same projected value across all attention heads, these are two different projections: $\mathbf{K} \neq \mathbf{V}$.

MQA reduces the KV cache size by a factor equal to the number of attention heads. However, it also reduces the model's flexibility considerably, which has a negative impact on quality. If you want a middle ground between MQA and MHA, you can use grouped-query attention instead.

Grouped-query attention (GQA)

GQA was proposed in 2023 by Google researchers.[17] The idea is quite simple: we split the attention heads into multiple groups, and each group shares the same projected key and value. In fact, you can think of MQA as GQA with a single group. MHA, MQA, and GQA are represented in Figure 17-11.

16 Noam Shazeer, "Fast Transformer Decoding: One Write-Head is All You Need", arXiv preprint arXiv:1911.02150 (2019).

17 Joshua Ainslie et al., "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints", arXiv preprint arXiv:2305.13245 (2023).

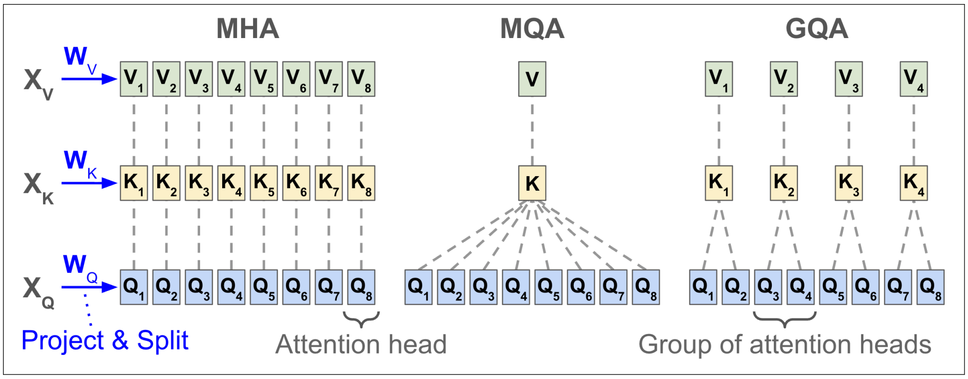

MHA
MQA
GQA
X_V
W_V
X_K
W_K
X_Q
W_Q
V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8
K_1 K_2 K_3 K_4 K_5 K_6 K_7 K_8
Q_1 Q_2 Q_3 Q_4 Q_5 Q_6 Q_7 Q_8
V
K
V_1 V_2 V_3 V_4
K_1 K_2 K_3 K_4
Project & Split
Attention head
Group of attention heads

Figure 17-11. MHA, MQA, and GQA

In PyTorch, you can implement MQA and GQA efficiently by setting `enable_gqa=True` when calling the `F.scaled_dot_product_attention()` function.[18] The query tensor contains the projected query for each attention head, with a shape of [*batch size*, *h*, L_q, *d*] (where *h* is the number of heads), but the key and value tensors contain only one projection per head, so they have a shape of [*batch size*, *g*, L_k, *d*] (where *g* is the number of groups). For example, the following code runs GQA with 8 attention heads, split into 2 groups which share the same projected key and value:

```
batch_size, Lq, Lk, d_head = 32, 100, 90, 64
n_heads = 8
n_groups = 2
query = torch.randn(batch_size, n_heads, Lq, d_head)
key = torch.randn(batch_size, n_groups, Lk, d_head)
value = torch.randn(batch_size, n_groups, Lk, d_head)
attn = F.scaled_dot_product_attention(query, key, value, enable_gqa=True)
```

Grouped-query attention is a great compromise between MHA and GQA, and it gives you the flexibility of choosing the number of groups, so you can balance efficiency and quality. That said, multi-head latent attention seems to outperform GQA consistently.

Multi-head latent attention (MLA)

In December 2023, the Chinese AI company DeepSeek began releasing impressive models, starting with a general-purpose LLM simply named DeepSeek-LLM. In January 2024, they released DeepSeek Coder, a model designed for code generation and completion, soon followed by DeepSeek-VL, a vision-language model. In May, they released the second version of their flagship general-purpose LLM, DeepSeek-V2, based on mixture of experts and MLA (which I will discuss shortly). They continued

18 At the time of writing, this is still an experimental feature.

to release new versions of these models, as well as Janus, a multimodal transformer, in October. In January 2025, they introduced DeepSeek-R1, a powerful reasoning model which outperformed OpenAI's o1 model on several benchmarks, despite having been trained for a fraction of the cost. DeepSeek-V3.1 was released in August 2025. That's an impressive track record!

The good news is that all the major DeepSeek models are open source and free for both research and commercial use, and DeepSeek researchers published many papers and technical reports, openly sharing the key innovations that allowed them to train such impressive models with a much lower budget than giants like Google or OpenAI. One of these techniques is *multi-head latent attention* (MLA), presented in the DeepSeek-V2 paper:[19] MLA reduces the KV cache roughly as much as MQA does, but without reducing the model's quality. In fact, it even improves it! Let's see how it works.

In both self-attention and cross-attention, the input key and value are the same input sequence denoted $\mathbf{X}_{\text{KV}}$. The key idea of MLA is to compress this d-dimensional sequence by projecting it to a smaller d_c-dimensional space—the latent space—using a trainable projection matrix $\mathbf{W}_{\text{KV}}$, then use the compressed sequence $\mathbf{L}_{\text{KV}}$ to create the key and value projections for all attention heads (see Figure 17-12).

At inference time, instead of storing the key and value projections in the KV cache, we can store $\mathbf{L}_{\text{KV}}$, and use it to recompute the projected keys and values when needed: this saves plenty of space since there's a single compressed sequence regardless of the number of attention heads.

19 DeepSeek-AI, "DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model", arXiv preprint arXiv:2405.04434 (2024).

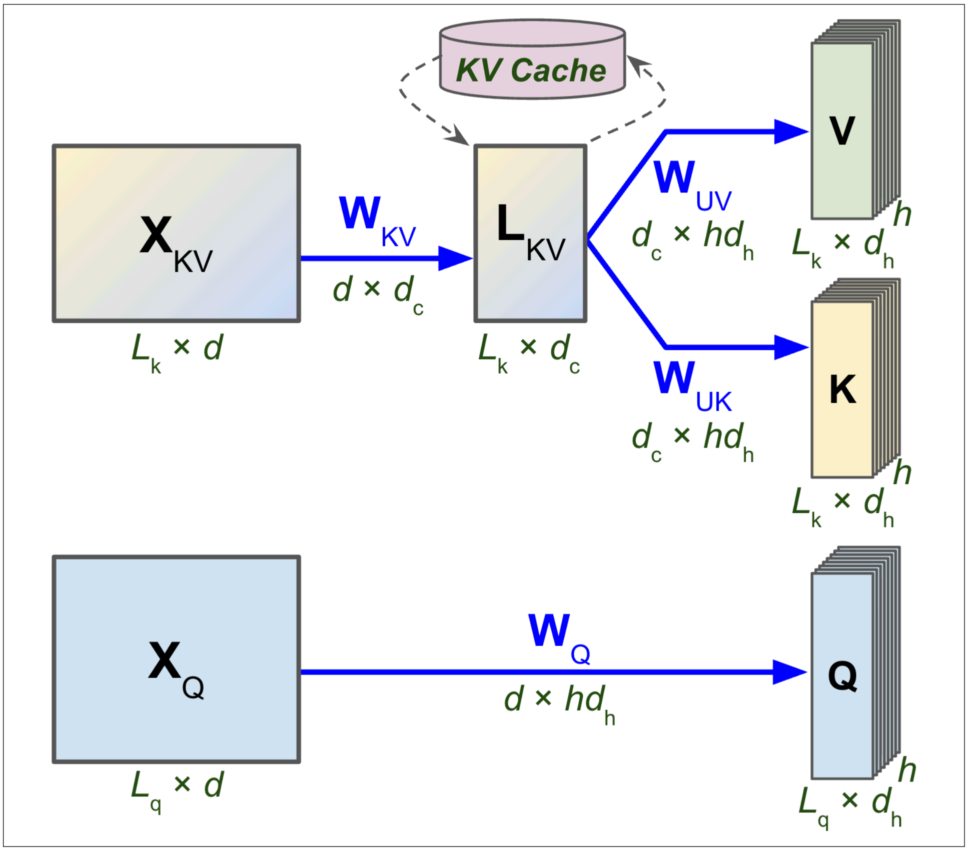


Figure 17-12. Multi-head latent attention (MLA)

This technique greatly reduces the size of the KV cache, as you can see in Table 17-1, which compares the KV cache size when using MHA, MQA, GQA, or MLA.

Table 17-1. KV cache size for MHA, MQA, GQA, and MLA

Architecture	KV cache size equation	Example
MHA	$p \times n_L \times n_t \times d_h \times h \times 2$	3,750 MB
MQA	$p \times n_L \times n_t \times d_h \times 2$	29 MB
GQA	$p \times n_L \times n_t \times d_h \times g \times 2$	469 MB
MLA	$p \times n_L \times n_t \times d_c$	59 MB

In this table:

- p is the floating point precision, in bytes (e.g., 2).
- n_L is the number of MHA layers in the model (e.g., 60).

- n_t is the total number of tokens stored in the cache (e.g., 1,000).
- d_h is the attention head dimensionality (e.g., 128).
- d_c is the dimensionality of the latent space (e.g., 512).
- h is the number of attention heads (e.g., 128).
- g is the number of groups in GQA (e.g., 16).

As you can see, in this particular example (which is based on the DeepSeek-V2 hyperparameters), MLA shrinks the KV cache by a factor of 64! That's not quite as much as MQA, but the MLA model has a much higher quality. There are two reasons for this:

- Firstly, the MLA model has projected keys and values for each attention head, so it has much more flexibility than MQA and GQA.
- Secondly, since we use much smaller matrices to project the keys and values, the model is both smaller and faster. Thanks to this speed up, MLA actually reaches a better performance than MHA in the same amount of training time!

To implement MLA, the simplest option is to reuse an open source implementation, such as the one used in DeepSeek-V3. Alternatively, you can write your own custom module, and the good news is that it can use the optimized `F.scaled_dot_product_attention()` function under the hood for each attention head.

With that, we've covered some of the main techniques used to speed up MHA, using sparse attention, approximate attention, and sharing projections across attention heads. Now let's turn to the last attention optimization technique in this chapter: FlashAttention.

FlashAttention: A Fast and Accurate Attention Implementation

FlashAttention is an optimized implementation of the now-familiar scaled dot-product attention: Attention($\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$) = softmax($\mathbf{QK}^\top / \sqrt{d_k}$)$\mathbf{V}$. It takes full advantage of the hardware architecture of modern GPUs, computing attention up to 2 to 4 times as fast as previous optimized implementations, without approximation, and is particularly efficient with long input sequences.

It was proposed in May 2022 by a team of researchers from Stanford University and the University at Buffalo.[20]

20 Tri Dao et al., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness", arXiv preprint arXiv:2205.14135 (2022).

To understand FlashAttention, it's important to know that a GPU's memory is split into different memory types, from the fastest to the slowest:

Registers
: A GPU contains multiple clusters of processing cores running parallel threads. Each cluster is called a *streaming multiprocessor* (SM). For example, Nvidia's A100 GPU contains 108 SMs. Each SM has direct access to a number of 32-bit registers, for example 2^{16} = 65,536 registers (that's only a total of 256 kB). Each individual thread is restricted to its own private set of registers (e.g., 255), so the data is fragmented across threads. On the other hand, registers can be accessed in a single clock cycle, allowing the highest possible bandwidth (e.g., 100+ TB/s across all SMs). Registers are used to hold intermediate results, counters, and accumulators such as partial sums during matrix multiplications.

Shared memory / L1 cache
: Each SM also has a small amount memory (e.g., 192 kB) shared across all threads. This memory lives on-chip and has a very high bandwidth (e.g., 20 TB/s). It can be used to store chunks of data during computations. For example, matrices are typically chopped into little chunks, which can be stored in this shared memory, a few at a time. Shared memory can also be used for communication between threads. Lastly, part of this memory can be allocated for the level 1 (L1) cache: the GPU automatically caches frequently accessed data in this space.

Level 2 (L2) cache
: This memory also lives on-chip. It is shared across all SMs, and is managed automatically by the GPU to cache frequently accessed data, such as some model weights. It has a larger capacity (e.g., 40 MB) but it is slower (e.g., 7 TB/s).

Global memory
: The previous memory types all live in SRAM (Static RAM), which is fast but expensive. However, global memory (e.g., *High Bandwidth Memory*, or HBM) lives off-chip in a cheaper type of RAM called DRAM (Dynamic RAM), making it possible to have a much larger capacity (e.g., 80 GB) at the cost of a lower bandwidth (e.g., 2 TB/s). This global memory is what we generally mean when we talk about GPU memory: this is where the models live, and where new batches of data go before they are processed.

To make our algorithms run as fast as possible, we have to make sure that the GPU spends as little time as possible transferring data from global memory to SRAM. This is precisely what FlashAttention does: it's an *I/O-aware* implementation. Recall that the scaled dot-product attention equation performs one matrix multiplication $\mathbf{QK}^\intercal$, then it computes the softmax of the output, and finally it performs another matrix multiplication. How can we speed up these operations?

Speeding up matrix multiplication is done through tiling: we first chop the two matrices that we want to multiply into tiles, which are small matrices that can easily fit in SRAM, then we multiply these tiles and add up the results to compute the output tiles gradually, as shown in Figure 17-13: in this example, we multiply two 8 × 8 matrices by first chopping them into 4 × 4 matrices, then we use these tiles to compute the output tiles as shown (you can verify that this gives the correct result). This process keeps the processing cores busy with data that they can fetch quickly from SRAM, while minimizing the number of fetches from global memory.

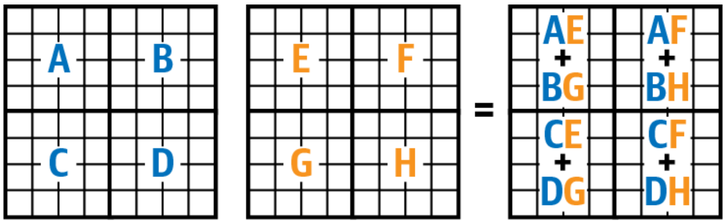


Figure 17-13. Fast matrix multiplication using tiling

The softmax operation can also be sped up significantly. A naive implementation of softmax(**M**) would first evaluate **E** = exp(**M**), then it would compute the sum of each row **D** = `E.sum(dim=-1)`,[21] and finally **O** = **E** ⊘ **D** (where ⊘ represents itemwise division). However, this approach will involve fetching **M** from global memory to compute **E**, and writing the result to global memory (assuming **M** is large), then fetching **E** to compute **D**, storing **D**, fetching **E** and **D** to compute **S**, and finally writing **S** to global memory. That's a lot of fetching and writing!

A much more efficient way to compute the softmax is to fuse the computation of **E** and **D**: for this, we can just iterate over each row of **M** and keep track of a running total of exp($M_{i,j}$), for each row. This gives us **D**. Then we iterate a second time over **M** and this time we compute exp($M_{i,j}$) / D_i. Since we're avoiding reading and writing **E** altogether, the result is much faster. This is called *online softmax*.

FlashAttention actually goes several steps further: firstly, it also fuses the computation of $\mathbf{QK}^\intercal$ using tiled matrix multiplication, as well as the computation of **SV**. Moreover, it avoids numerical overflows by subtracting the max row value before exponentiating (this is safe softmax, as we saw earlier). This involves computing a running estimate of the max value, and adjusting the running total whenever we find a larger max

21 The mathematical notations for per-column operations, per-row operations, and broadcasting, can be misleading. This is why I preferred using a bit of code here. In papers, you might see something like $\mathbf{D} = \mathbf{E}\,\mathbf{1}_d$, where $\mathbf{1}_d$ is a column vector of dimensionality d (equal to the number of columns in **M** in this case). However, we never actually create this vector or perform a matrix multiplication, we just compute the sum of each row.

value. Please see the notebook for a toy Python implementation, and see page 5 in the paper for the full algorithm.

Python is fantastic for high-level coding but it's not designed for low-level coding such as writing custom kernels (i.e., GPU programs), so we have to switch to a systems language such as C++. Languages like Julia and Mojo attempt to solve this *two-languages problem*: they support simple, expressive, and dynamic high-level coding, as well as typed, optimized, low-level coding giving you full flexibility. Mojo even aims to be a superset of Python.

Tri Dao, the first author of FlashAttention, published a second version of FlashAttention in July 2023: FlashAttention-2[22] speeds up the original algorithm by optimizing work partitioning and parallelism across GPU threads. While FlashAttention typically reaches 25% to 40% utilization of the GPU, FlashAttention-2 can reach 50% to 75%.

FlashAttention-2 doesn't ignore padding tokens, and it only supports 16-bit float precision (`torch.float16` or `torch.bfloat16`).

One year later, Jay Shah et al. published FlashAttention-3[23], which further optimizes the algorithm by exploiting the asynchronous execution and low-precision capabilities of newer GPU architectures, such as the NVIDIA H100.

PyTorch's `F.scaled_dot_product_attention()` function automatically uses FlashAttention-2 if it detects that your hardware is compatible: it supports Nvidia CUDA GPUs (with a compatible CUDA version), and AMD GPUs through the ROCm toolkit. For more control, you can use the flash-attn library directly: it can be installed using `pip install flash-attn --no-build-isolation`.

When using the Hugging Face Transformers library, most models support FlashAttention-2 out of the box: you just have to specify `attn_implementation="flash_attention_2"` when loading the model with the `from_pretrained()` method.

22 Tri Dao, "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning", arXiv preprint arXiv:2307.08691 (2023).

23 Jay Shah et al., "FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision", arXiv preprint arXiv:2407.08608 (2024).

FlashAttention (1, 2, and 3) completely changed the game when it comes to accelerating attention: many recent architectures dropped sparse and approximate attention methods in favor of FlashAttention, as it provides a huge efficiency boost without compromising quality. Note that you can combine FlashAttention and MLA: MLA uses regular scaled dot-product attention in each attention head, so it also benefits from FlashAttention; they make an excellent pair.

Let's step back for a second: we first looked at techniques to accelerate decoding, including KV caching, speculative decoding, and a few others, then we discussed many methods to accelerate multi-head attention, including sparse attention, approximate attention, sharing projections across attention heads, and lastly FlashAttention. Now let's leave multi-head attention behind and focus on the other major component of the Transformer architecture, the feedforward neural networks, which we will optimize using a technique named *mixture of experts*.

Scaling Up with Mixture of Experts

When we have a medical question, we ask a doctor. When we have a legal question, we ask a lawyer. This idea of routing different requests to different experts was first proposed in machine learning by Robert A. Jacobs, Michael I. Jordan, et al. in a 1991 paper.[24] They dubbed this technique *mixture of experts* (MoE). MoE remained niche for over two decades, but it gained popularity when it was successfully applied to transformers in a 2017 paper by Noam Shazeer et al.,[25] which introduced the *sparsely-gated MoE layer* (I will simply call it the *MoE layer*).

In practice, as we will see, the "experts" don't specialize across high-level domains like humans do; rather, they focus on different regions of the input space at a fairly low level, and there's actually quite a lot of overlap between them. Moreover, an MoE transformer contains several MoE layers, each with its own local set of experts, and different tokens may be processed by different experts in each MoE layer. So it's not as though a medical question is handled by a medical transformer; instead, tokens are split across several experts in each MoE layer. Therefore, the name "mixture of experts" is misleading—it might be better to call them "sort-of specialized subnetworks splitting subtasks" (SSSSS), but it's not as catchy.

In any case, MoE transformers have been incredibly successful, so let's see how the MoE layer works.

24 Robert A. Jacobs, Michael I. Jordan, et al., "Adaptive mixtures of local experts." *Neural computation 3.1: 79–87* (1991).

25 Noam Shazeer et al., "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer", arXiv preprint arXiv:1701.06538 (2017).

The Mixture of Experts Layer

The core idea is to replace the FFN in some transformer blocks (typically every 2 to 4 blocks) with an MoE layer composed of multiple FFNs, called *experts*, plus a small router network that decides which expert(s) should process each input token (see Figure 17-14).

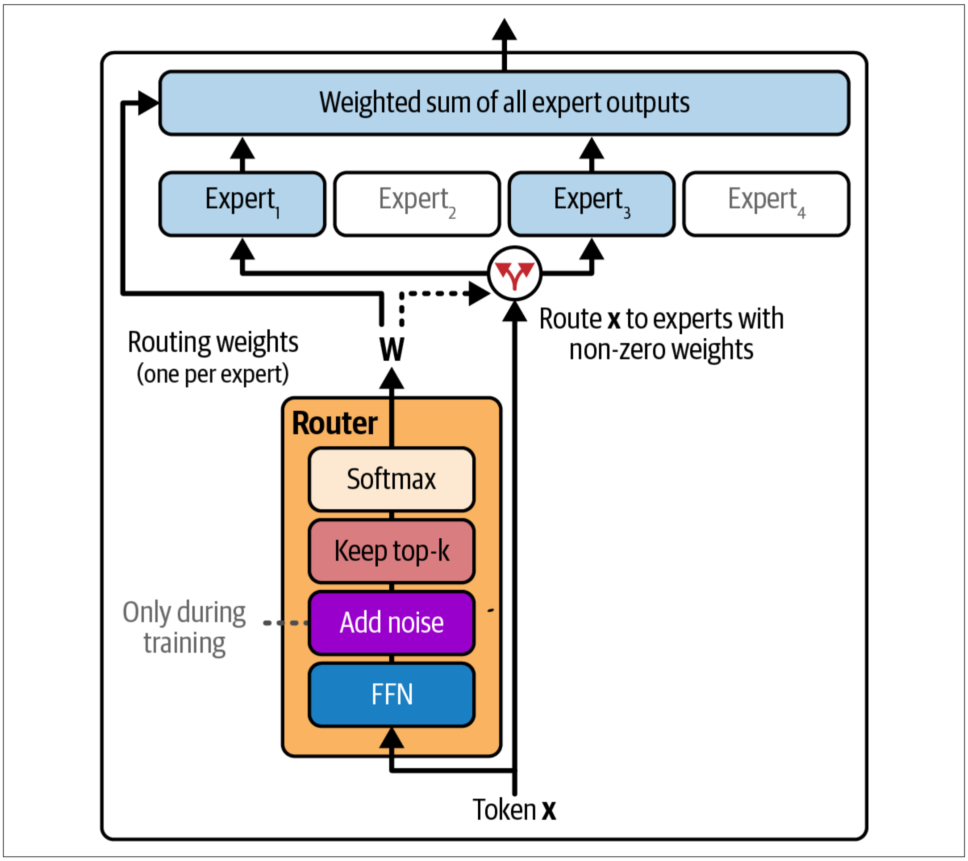

Weighted sum of all expert outputs
Expert₁
Expert₂
Expert₃
Expert₄
Route **x** to experts with non-zero weights
Routing weights (one per expert)
w
Router
Softmax
Keep top-k
Add noise
FFN
Only during training
Token **x**

Figure 17-14. A modern MoE layer: this layer replaces the FFN module in each transformer block

- Each token begins its journey by going through the router network: it's just a small FFN (usually a single linear layer) with a softmax activation function. This router outputs a vector **w** containing one router weight per expert. All the weights add up to 1.
- Next, we find the top-*k* weights (e.g., *k* = 2 in this example) and we route the input token **x** to the corresponding *k* experts. All the other experts can take a nap.

- Lastly, we compute a weighted sum of the top-k expert outputs: to be precise, each output is multiplied by its corresponding routing weight, and all the weighted outputs are summed up. In some architectures with $k > 1$, the top-k weights are normalized just before computing the final weighted sum.

The MoE architecture was gradually refined in later architectures, most notably in GShard,[26] Switch Transformers,[27] Vision-MoE,[28], GLaM,[29] and ST-MoE,[30] all published by Google researchers between 2020 and 2022 (but of course research continued after that, and not just at Google).

Each expert can itself be an MoE layer, with its own router network and sub-experts. The resulting MoE is called a *hierarchical MoE*, and it can scale up to thousands of experts.

OK, that's how MoE works, but what are its benefits?

Benefits of MoE

By routing each token to just a handful of experts out of dozens (or potentially hundreds) of experts, we can keep computations low while vastly increasing the model's power. An MoE transformer is typically huge, often requiring multiple GPUs to run, but it is much faster than its size would suggest, since each token only uses a small subset of all model weights.

The benefits are particularly clear at inference time when generating text one token at a time, since the MoE uses only the top-k experts for that token: it's extremely fast, and uses relatively little memory bandwidth since few model weights are needed.

However, during training and during the prefill stage at inference time, the MoE must process potentially long sequences of tokens, and each token may be routed to a different expert. As a result, most experts are likely to be needed simultaneously,

26 Dmitry Lepikhin et al., "GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding", arXiv preprint arXiv:2006.16668 (2020).

27 William Fedus et al., "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity", arXiv preprint arXiv:2101.03961 (2021).

28 Carlos Riquelme et al., "Scaling Vision with Sparse Mixture of Experts", arXiv preprint arXiv:2106.05974 (2021).

29 Loic Themyr et al., "GLaM: Efficient Scaling of Language Models with Mixture-of-Experts", arXiv preprint arXiv:2112.06905 (2021).

30 "ST-MoE: Designing Stable and Transferable Sparse Expert Models", arXiv preprint arXiv:2202.08906 (2022).

even though each one will only process a few tokens: we lose most of the memory bandwidth savings, but we still get a large computational benefit.

Sadly, there's no free lunch: building, training, running, and even maintaining a large MoE transformer can be quite difficult and costly. Let's see why.

The Challenges of MoEs

There are three major difficulties with MoEs: balancing the workload across experts, reducing the overlap between experts, and finally engineering challenges. Let's start with load balancing.

Load balancing

The first major difficulty with MoEs is ensuring that the router distributes the workload evenly across experts, at least on average, especially during training. Indeed, if an expert is worse than the others at any point during training, it will not be selected by the router network, and as a result it will not be trained and will not improve. It's a classic rich-gets-richer feedback loop, which in this case leads to *expert collapse*, meaning that the router network ends up completely ignoring most experts, favoring a handful of lucky winners. This destroys the model's accuracy and makes training unstable.

To solve this issue, one common solution is to add an auxiliary *load-balancing loss* that encourages the router network to evenly distribute the workload across experts (the load-balancing loss of each MoE layer is just added to the overall loss). In the original sparsely-gated MoE, this loss involved adding Gaussian noise with a trainable per-expert scale just before computing the softmax, and the loss pushed the model to give an equal *expected* load to each expert. However, in Switch Transformers and most recent MoE architectures, the load-balancing loss is simpler and doesn't require adding noise at all (see Equation 17-4).

Equation 17-4. Auxiliary load-balancing loss, used in many recent MoE transformers

$$L_{\text{LB}} = \alpha N \sum_{i=1}^{N} f_i Q_i$$

In this equation:

- α is a hyperparameter you can tune: it determines how strongly the model will be pushed toward fair load-balancing (e.g., 0.01).
- N is the total number of experts (not just the top k).

- f_i is the proportion of tokens in the current batch that were assigned to the i^{th} expert.
- Q_i is the mean routing weight assigned to the i^{th} expert in the batch.

This loss pushes every f_i and Q_i toward $1 / N$: an equal share of the load for each expert.

A couple additional techniques, introduced by the GShard authors, are often used to better balance the load across experts:

Random routing
: When $k = 2$, the top expert is always chosen, but the second is picked randomly, with a probability proportional to its routing weight.

Expert capacity
: Each expert can only process up to C tokens per batch, where C is a hyperparameter called the *expert capacity*, typically a bit greater than the number of tokens per batch divided by the total number of experts. When an expert has reached its maximum capacity, its assigned tokens are typically re-routed to the other experts. If all candidate experts are full then the token typically just skips the MoE layer: for this, the MoE layer outputs zero, but the skip connection around the MoE layer allows the token to continue to the next transformer block. In others architectures, such as some vision MoEs, the token may be dropped entirely, so the output sequence is shorter.

Now let's turn to the second major difficulty with MoEs: reducing overlap between experts.

Reducing overlap between experts

You might expect the router network to send mathematical queries to a math expert, and medical queries to medical experts, but as I mentioned earlier, the router seems to focus on much lower-level features: it might send punctuation tokens and emojis to one expert, adjectives and numbers to another, and so on. It's better than nothing, but you can see how this can lead to a large overlap across experts, since they all end up having to learn about every possible topic. If all your experts are generalists, then they're not really experts at all! This redundancy effectively reduces the model's power, leading to lower accuracy.

One way to increase expert specialization is to use a more powerful router network, but this comes at a cost. Another option, introduced in DeepSeek-V2, is to add *shared experts* which are used for every token: these shared experts tend to learn the most common cases, which encourages the routed experts to specialize on more specific tasks. For example, DeepSeek-V3 uses 1 shared expert plus 256 routed experts (8 active) in each MoE layer. Improving expert specialization is still an area of active research.

Let's now move on to the third major difficulty of MoEs: engineering.

Engineering challenges with MoEs

MoEs can be a major engineering challenge, both for training and for inference, especially for the larger models. Here are just a few engineering challenges you may face:

- MoE transformers can be so large that the experts must be sharded across multiple GPUs, and even across multiple servers. This means that the inputs and outputs of each MoE layer must be sent across devices or even across servers, typically using *all-to-all communication*. This I/O can easily become a major bottleneck. Moreover, it requires careful synchronization: for example, we must wait for all expert outputs before we can compute the weighted sum.
- If the model is sharded across servers, the infrastructure must detect and handle node failures. Monitoring is key.
- Tokens must be routed dynamically to the right experts, and batched to increase GPU utilization. Batching is not as simple as in dense models.
- A few *hot experts* (overloaded) may slow down the whole model: you may want to implement fine-grained monitoring to detect this and dynamically solve the issue, for example by rerouting tokens, replicating experts, or autoscaling your infrastructure.

In short, it takes quite a bit of complex and costly engineering to make large MoEs production-ready at scale. This is why many early MoE models such as GLaM or Switch were mostly research-only.

Now the good news is that there are now several open source libraries to train and run MoEs efficiently, such as DeepSeek-MoE, so you don't need to reinvent the wheel. There are also many pretrained MoE models available for download (see Table 17-2).

Table 17-2. Popular open source MoEs

Model name	Release date	Active/total params	Active/total (+ shared) experts
Mixtral 8x7B	Dec 2023	13B / 47B	2 / 8
DBRX Instruct	Mar 2024	36B / 132B	4 / 16
Mixtral 8x22B	Apr 2024	39B / 141B	2 / 8
DeepSeek-V2	May 2024	21B / 236B	6 / 160 (+2)
OpenMoE-8B	Nov 2023	2B / 8B	? / 32
Grok-1	Mar 2024	86B / 314B	2 / 8

Model name	Release date	Active/total params	Active/total (+ shared) experts
Jamba-1.5-L	Aug 2024	94B / 398B	2 / 16
OLMoE-7B	Sep 2024	1B / 7B	8 / 64
Aria (vision & language)	Oct 2024	3.5B to 3.9B / 25B	6 / 64 (+2)
DeepSeek-V3	Dec 2024	37B / 671B	8 / 256 (+1)

If you'd like to further explore the MoE literature, a great entry point is the 2025 MoE survey by Siyuan Mu and Sen Lin.[31]

That's all for MoEs. Now let's look at techniques that will help you train transformers faster!

Faster Training

Transformers are often very large and slow to train. There are two main reasons for this: first, a larger model typically requires more computations than a smaller one; second, a lot of data will need to be moved around in GPU VRAM, so the memory bandwidth will become a bottleneck. Let's take a 7B parameter model, for example:

Model weights
: At 16-bit float precision, the weights take 14 GB of memory. At inference time, that's most of the memory you'll need, but at training time, the list continues. Firstly, when using mixed-precision training (see Appendix B), we also need a 32-bit master copy of the trainable weights: that's another 28 GB.

Activations
: During training, we must save all of the activations during the forward pass since they are needed for the backward pass. The exact amount depends on the model architecture, the batch size, the sequence length, and the float precision, but for a 7B parameter model, it's might be around 40 GB.

Gradients
: We need one gradient per trainable parameter. If we train all 7B parameters, at 16-bit float precision, that's 14 GB.

Optimizer states
: Optimizers often require additional data for each trainable parameter. For example, Adam requires two running averages for each trainable parameter: at 16-bit

31 Siyuan Mu and Sen Lin, "A Comprehensive Survey of Mixture-of-Experts: Algorithms, Theory, and Applications", arXiv preprint arXiv:2503.07137 (2025).

precision, this would be another 28 GB. However, for precision and training stability, these are usually stored in 32-bit precision, so we're talking about 56 GB!

That's a total of 152 GB of memory for a 7B parameter model, even though we're using mixed-precision training! Your hardware might not have that much VRAM, and even if it does, memory bandwidth saturation will be a nightmare.

One solution is to shrink the model: it will reduce both the computational cost and the memory usage. This is discussed in Appendix B. You can also use the techniques discussed earlier to accelerate MHA, or use MoE. In this section, we will cover a few more ways to train transformers faster:

- Parameter-efficient fine-tuning (PEFT) reduces the number of trained parameters, which reduces the size of the gradients and optimizer states, as well as the size of the 32-bit master copy of the trainable parameters used for mixed-precision training.
- Activation checkpointing reduces the size of the activations that we must save for the backward pass: we already discussed it in Chapter 12, so in this chapter I'll focus on how to apply it to transformers.
- Sequence packing and bucketing, to eliminate wasteful padding tokens.
- Gradient accumulation, where we reduce the memory load by processing multiple batches per optimizer step.
- Parallelism, where we split the computations and memory load across multiple GPUs.

Let's start with PEFT.

Parameter-Efficient Fine-Tuning (PEFT)

You may occasionally need to train a transformer from scratch, but more often than not you will instead download a pretrained model and fine-tune it for your task. We saw how to do this in Chapter 15. To speed up fine-tuning, one approach is to reduce the number of trainable parameters, thereby reducing the computational and memory costs of computing and storing the trainable weights, gradients, and optimizer states: this is called *parameter-efficient fine-tuning* (PEFT).

There are several PEFT techniques, but the most popular approach is to use *adapters*: you freeze the weights of the large pretrained model, and insert several small trainable adapters inside it, then fine-tune the resulting *adapter model*. Since the adapters have few trainable parameters, the efficiency gain is substantial.

There are various ways to insert adapters; for example, *prefix adapters* use learnable key/value prefixes (this is closely related to prompt tuning, which we briefly discussed in Chapter 15). However, the most popular adapter technique is *Low-Rank Adapta-*

tion (LoRA), which was proposed in 2021 by Microsoft researchers.[32] Let's see how it works.

First, you freeze the pretrained model and choose some weight matrices that you want to fine-tune using LoRA adapters: in general, it's a good idea to adapt at least the query and value projection matrices in every MHA layer. If you need even more precision, at the cost of adding more adapters, you may also want to adapt the value and output projection matrices in every MHA layer, and possibly even the weight matrices in the FFN blocks. It's a trade-off between efficiency and accuracy.

Second, each matrix multiplication $\mathbf{XW}_o$ involving one of the chosen matrices $\mathbf{W}_o$ is replaced with $\mathbf{XW}_o + \mathbf{XBA}$, where $\mathbf{X}$ is the input matrix, $\mathbf{W}_o$ is the frozen pretrained weight matrix, and $\mathbf{A}$ and $\mathbf{B}$ are the new adapter matrices—small and trainable. If $\mathbf{W}_o$ has a shape of $d \times k$, then $\mathbf{B}$ is a $d \times r$ matrix and $\mathbf{A}$ is an $r \times k$ matrix, where r is a hyperparameter (e.g., $r = 8$). In practice, this is usually implemented by attaching an adapter module (containing two linear layers) to the layer we want to adapt, as shown in Figure 17-15.

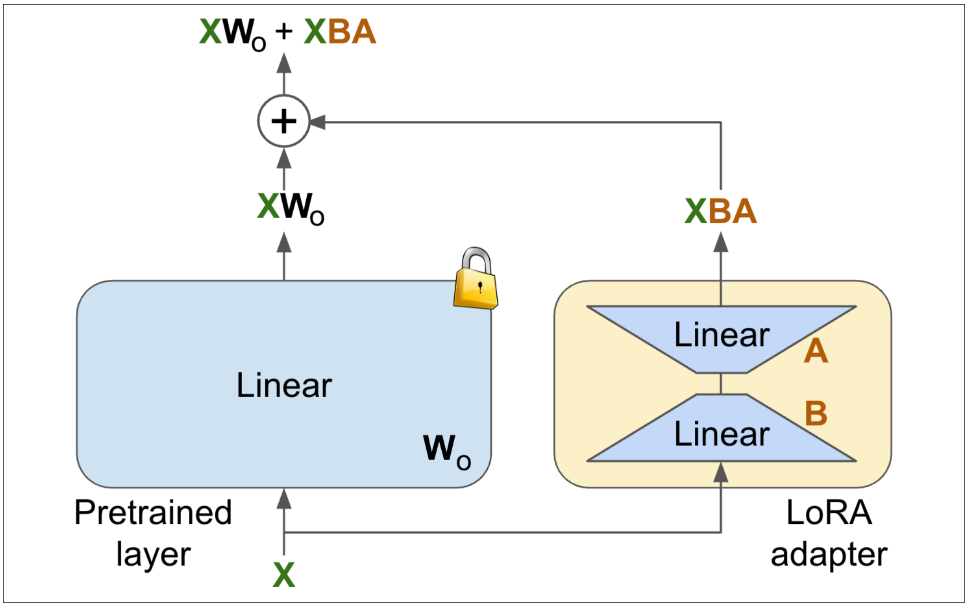


Figure 17-15. Low-Rank Adaptation

32 "LoRA: Low-Rank adaptation of large language models", arXiv preprint arXiv:2106.09685 (2021).

If the pretrained layer has a bias parameter, you can choose to make it trainable or not (it's the usual efficiency/quality trade-off). However, the LoRA adapter does not have a bias parameter, as it would be redundant.

You can usually start with $r = 8$, but you can reduce it (sometimes even down to 1) and still get reasonably good results, or increase it to 16, 32, 64, or more if you want to trade some speed and memory for higher quality.

And that's it, you're ready to fine-tune the adapted model. Enjoy the speed boost and the quality of the fine-tuned model!

But why? How can this technique offer both speed and quality? Well, the speed boost is easy to explain: instead of fine-tuning $d \times k$ parameters, we fine-tune $r \times (d + k)$ parameters. Since r is small relative to d and k, we have far fewer parameters to train, meaning less computations and less risk of memory bandwidth saturation.

Moreover, even though **BA** is a fairly large $d \times k$ matrix, we never actually need to evaluate it: indeed, when we compute **XBA**, we first calculate $\mathbf{P} = \mathbf{XB}$, which is a $b \times r$ matrix, where b is the batch size, then we compute **PA**, which is a $b \times k$ matrix.

Now why does LoRA lead to high-quality models? The authors reasoned that directly fine-tuning the original pretrained model (without using LoRA) would turn $\mathbf{W}_o$ into $\mathbf{W}_{\text{tuned}} = \mathbf{W}_o + \mathbf{W}_\Delta$, where $\mathbf{W}_\Delta$ is the difference between the original matrix $\mathbf{W}_o$ and the fine-tuned matrix $\mathbf{W}_{\text{tuned}}$. They hypothesized that $\mathbf{W}_\Delta$ has a low "intrinsic" rank, which would imply that it can be well approximated by the product of two low-rank matrices (hence the name of the technique): in other words, $\mathbf{W}_\Delta \approx \mathbf{BA}$, where **A** and **B** are both low-rank matrices. And this turns out to be true in practice: by keeping $\mathbf{W}_o$ frozen and adding **BA** to it, learning only matrices **A** and **B**, the resulting model performs almost as well as full fine-tuning (while training far fewer parameters and using much less memory).

To fine-tune a model using LoRA, you can use the Transformers and peft libraries by Hugging Face. Both are preinstalled on Colab. Let's load a model using the Transformers library, then create a LoRA model based on it using the `peft.get_peft_model()`:

```
import peft
from transformers import AutoModelForCausalLM

model_id = "EleutherAI/gpt-neo-125M"  # a small model for this example
model = AutoModelForCausalLM.from_pretrained(model_id, device_map="auto",
                                             dtype=torch.float16)
lora_config = peft.LoraConfig(r=8, target_modules=["q_proj", "v_proj"])
peft_model = peft.get_peft_model(model, lora_config)
peft_model.print_trainable_parameters()
```

Let's go through this code:

- After the imports, we load a pretrained model using 16-bit precision.
- Next, we create a `LoraConfig` object, specifying the rank *r*, and the name of the layers we want to adapt.
- Then we call the `peft.get_peft_model()` function, passing it the model and the config, and in return we get a parameter-efficient model, adapted using LoRA. This API is nice and easy (there are many other options, so please see the documentation for more details).

You can now fine-tune the model normally, just like we saw in Chapter 15.

After fine-tuning, you can optionally get rid of the adapters and replace each original weight matrix $\mathbf{W}_o$ with the matrix $\mathbf{W}_{\mathrm{LORA}} = \mathbf{W}_o + \mathbf{BA}$: this means that LoRA will have zero computational or memory cost at inference time. To do this, you can simply call the PEFT model's `merge_and_unload()` method:

```
merged_model = peft_model.merge_and_unload()
```

However, if the original model is quantized (see Appendix B), you may be better off without merging, keeping the adapters. Indeed, merging would require dequantizing the weights first, then adding **BA**, then requantizing the result. This would lose some precision, and since adapters are very lightweight, the efficiency gain would be minimal.

In summary, PEFT reduces the number of trainable parameters, thereby reducing computations and the amount of memory needed during fine-tuning, and resulting in a large speed boost. Now let's turn to activation checkpointing, which drastically reduces memory usage for the activations during training, in exchange for a slightly higher computational cost.

Activation Checkpointing

We already discussed activation checkpointing in Chapter 12 and we saw how to implement it in PyTorch, but not with the Transformers library, so this short section will focus on that. As a quick reminder, activation checkpointing is a technique to reduce memory usage for the activations during training: instead of saving all activations during the forward pass, we only save some of them, called checkpoints (e.g., just the main layer outputs), then during the backward pass we recompute the missing activations whenever needed, using the checkpoints. The result is less memory usage, but more computations.

With the Transformers library, implementing activation checkpointing is as simple as setting `gradient_checkpointing=True` in the `TrainingArguments` object that you pass to your `Trainer` (it's often used in conjunction with mixed-precision training, by setting `fp16=True` or `bf16=True`):

```python
from transformers import TrainingArguments

args = TrainingArguments(output_dir="./my_great_model", num_train_epochs=3,
                         per_device_train_batch_size=8,
                         gradient_checkpointing=True, fp16=True)
```

Another option is to enable activation checkpointing directly on the model itself by calling its `gradient_checkpointing_enable()` method (alternatively, you can do this on individual modules, for fine-grained control).

Now let's move on to sequence packing, which can accelerate training when the input sequences have variable lengths.

Sequence Packing and Bucketing

When training with variable-length sequences, we usually pad them to the same length when creating a batch. But this wastes compute and memory on many padding tokens, and results in batches of variable size, which are not hardware-friendly.

Another option is sequence packing: we concatenate multiple input sequences to form a much longer sequence, of length *L*. If the packed sequence is not exactly *L* tokens long, we can either crop the excess tokens or use padding tokens to reach *L*. Alternatively, we can pack all the sequences in the batch into a single sequence: no need for cropping or padding. This is called *flattening*.

Sequence packing gets rid of all padding tokens (when cropping or flattening) or most of them (when padding), and it also ensures higher GPU utilization, which accelerates training. However, it requires a lot of care:

- The attention masks must be adjusted to ensure that tokens from one sentence cannot attend to tokens in other sentences.
- Positional encodings should also be adjusted to ensure that the positions are relative to the start of each sentence.
- The loss must be computed on the right outputs. This is mostly a challenge for encoder-only models, since they often have a fixed number of outputs (e.g., for the [CLS] token), so we must keep track of their indices to properly compute the loss.
- Lastly, the implementation of MHA must be smart enough to handle sparse attention masks: indeed, a naive implementation of MHA would run into the quadratic attention problem, since the packed sequence can be very long: this would entirely negate the advantages of sequence packing. However, a smart

implementation can skip computations for the parts of the attention mask that are full of zeros.

Sadly, PyTorch doesn't provide a simple way to implement sequence packing: you have to manually take care of everything yourself. However, the Transformers library provides a convenient solution: just set the `data_collator` argument in your `Trainer` to an instance of the `DataCollatorWithFlattening` class. This will automatically take care of sequence packing for you, using the flattening strategy. Importantly, it uses position IDs and FlashAttention-2 to efficiently handle sparse attention masks.

A much simpler—albeit less efficient—technique, often used for encoder-only models, is to create batches of sequences of similar lengths: this is called *sequence bucketing*. This drastically reduces the number of padding tokens required, and it's much simpler to implement in PyTorch. One approach is to split your dataset into buckets depending on the sequence length (e.g., [1-10], [11-20], [21-30], and so on), then when creating a batch, pick a bucket randomly and sample sequences from it. Another approach is to sort the dataset by length, then when creating a batch, pick a random index in the sorted dataset, and sample nearby sequences.

Figure 17-16 compares regular batching with sequence packing and sequence bucketing.

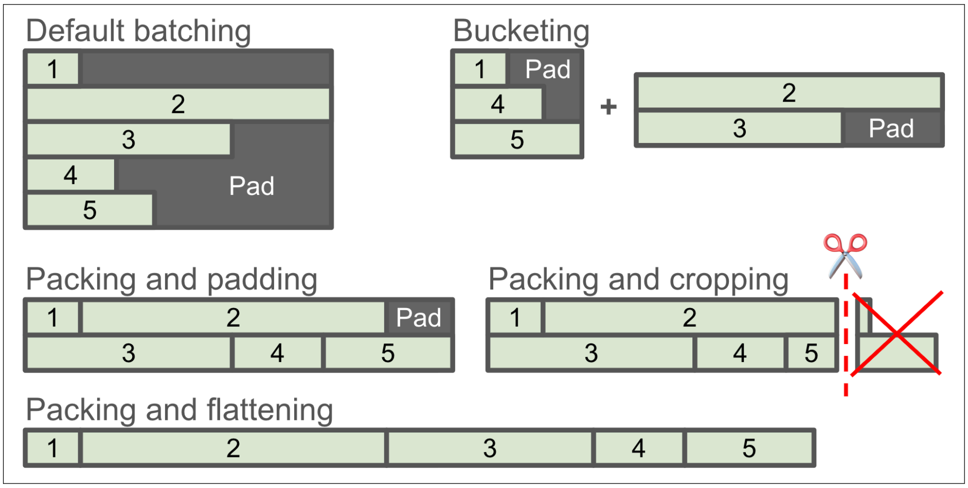


Figure 17-16. Sequence packing and sequence bucketing versus regular batching

Next up is gradient accumulation, which lets you increase the batch size without using more memory.

Gradient Accumulation

Instead of performing an optimizer step for each input batch, we can accumulate gradients across multiple batches, and only perform one optimizer step every *n* batches using the accumulated gradients. The procedure is as follows:

- Set the gradients to zero.
- Compute the gradients for batch 1, but do *not* perform any optimizer step.
- Repeat for batches 2 to *n*, accumulating the gradients.
- After *n* batches, perform an optimizer step using the accumulated gradients (optionally divided by *n* to maintain the gradient scale).
- Rinse and repeat for the next set of *n* batches.

This approach is mathematically equivalent to increasing the batch size by a factor of *n*, but it uses much less memory. This makes it possible to train the model with a large effective batch size without running into out-of-memory errors or slowdowns caused by memory bandwidth saturation. However, gradient accumulation is not quite as computationally efficient as increasing the batch size (assuming you have enough memory), since the GPU doesn't process as much data in parallel.

Implementing gradient accumulation in PyTorch is straightforward:

```
[...]  # create the model, optimizer, criterion, and data_loader
accumulation_steps = 4
optimizer.zero_grad()  # reset gradients before starting
for batch_index, (X_batch, y_batch) in enumerate(data_loader):
    X_batch, y_batch = X_batch.to(device), y_batch.to(device)
    y_pred = model(X_batch)
    loss = criterion(y_pred, y_batch)
    loss = loss / accumulation_steps
    loss.backward()
    if (batch_index + 1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

The code is mostly self-explanatory, but it's important to note that the gradients must be set to zero before starting the training loop. Dividing the loss by the number of accumulation steps also divides the gradients by the same number, which ensures that we get the same total gradients as if we had used a larger batch size (alternatively, you could divide the learning rate by the number of accumulation steps).

If the model uses batch-normalization, or if you use a batch-wide loss (e.g., the MoE load-balancing loss), then gradient accumulation will *not* be equivalent to increasing the batch size. This is partly why transformers use layer-norm instead of batch-norm. As for batch-wide losses, there's sadly no easy fix, but training might still work fine; you just have to test it.

Implementing gradient accumulation with the Transformers library is as simple as can be: just set `gradient_accumulation_steps` to the desired number in the `TrainingArguments`. That's it!

Last but not least, let's discuss various ways you can speed up training by using multiple GPUs, possibly across multiple servers.

Speeding Up Training Using Parallelism

If we can manage to efficiently use multiple GPUs in parallel during training, we can potentially achieve a linear speed up, proportional to the number of GPUs. There are many ways to parallelize training: across the data dimension (data parallelism), across the features dimension (tensor parallelism), across model layers (pipeline parallelism), across the sequence dimension in the case of transformers (context parallelism), across experts in the case of MoE models (expert parallelism), and more. These techniques can also be combined. Let's start with data parallelism, which is arguably the simplest approach.

Data Parallelism

The core idea behind data parallelism is to split each input batch into multiple chunks, and process each chunk on a different device, all in parallel. The most common strategy is to replicate the model across all devices and keep the replicas synchronized so they always have the same model parameters. This is called the *mirrored strategy*.

PyTorch provides a basic implementation of data parallelism using the mirrored strategy for a single-machine setup: just wrap your model in a `torch.nn.DataParallel` object, optionally specifying the IDs of the devices to use (by default it will use all available devices), then train the resulting model as usual. For example:

```
dp_model = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
[...] # train dp_model normally
```

That's it, all the complexity is taken care of for you by the `DataParallel` (DP) wrapper. During the forward pass, it splits the input batch across GPUs, broadcasts the model's latest parameters from the primary GPU (where the wrapped model lives) to the other GPUs, runs the forward pass in parallel on all replicas (each one processing its own chunk of data), and finally gathers and concatenates the outputs on the primary GPU. During training, your code then uses this output to compute the loss (usually on the same device). Then, during the backward pass, the gradients flow back through all the GPUs in parallel, and all the gradients are gathered and summed up on the primary GPU. Lastly, the gradient descent step only takes place on the primary GPU, and the model copies are discarded (until new copies are made at the next iteration).

This works and it's very easy to use, but it's really not the most efficient approach since the whole model gets copied to all GPUs at each iteration. Moreover, the primary GPU has more work and uses more memory than the other GPUs. Luckily, PyTorch provides a much better implementation named *Distributed Data Parallel* (DDP) which is far more efficient and also works with GPUs located on separate machines. Moreover, DDP uses multiprocessing rather than multithreading, which allows it to work around Python's global interpreter lock (GIL). Unless you want to run a quick test, I highly recommend you use DDP instead of `torch.nn.DataParallel`.

With DDP (see Figure 17-17), the model is only replicated once upon startup. Moreover, at each training step, all the replicas perform the entire forward pass and backward pass independently and in parallel (each on its own local chunk of data). The replicas then synchronize only to efficiently compute the mean of their gradients; then they all perform the exact same optimizer step using the same mean gradients, so they end up with the same updated model parameters without having to copy the whole model at each iteration.

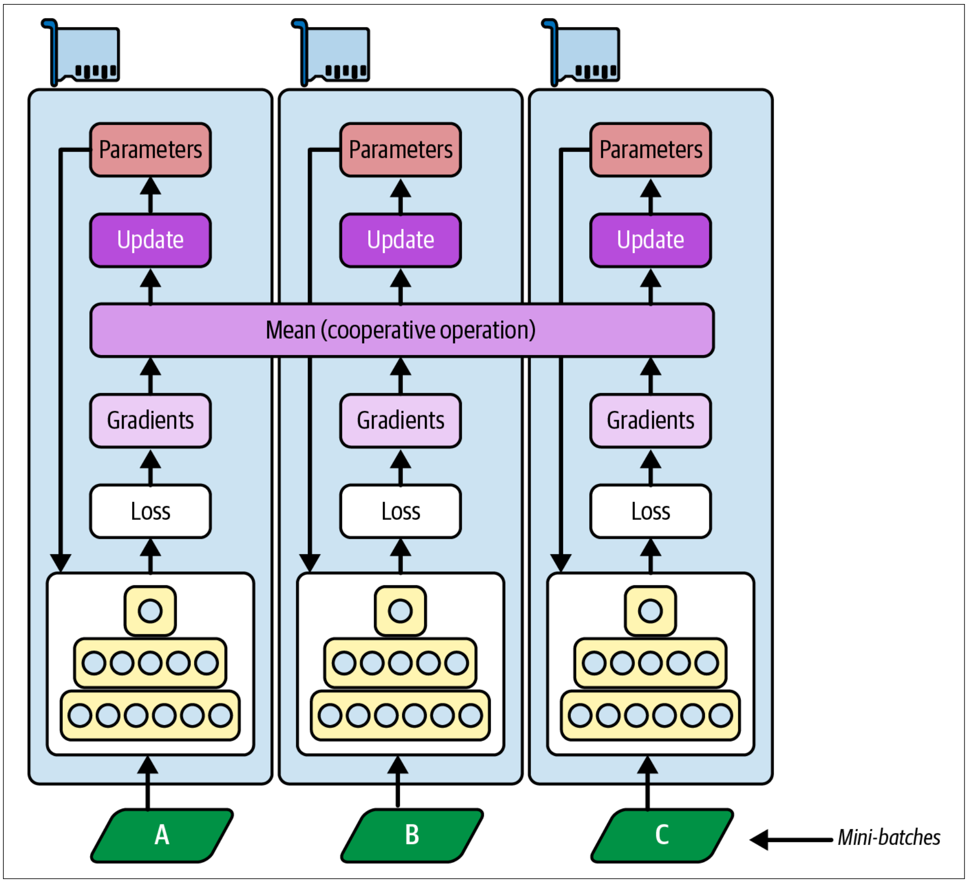


Figure 17-17. DDP implements data parallelism with the mirrored strategy; the replicas collaborate to compute the mean gradients using an all-reduce operation

You may be wondering how DDP efficiently computes the mean of all the gradients from all devices and distributes the result to all devices? This is done using an *all-reduce* algorithm, a class of algorithms where multiple nodes collaborate to efficiently perform a *reduce operation* (such as computing the mean, sum, and max), while ensuring that all nodes compute the same final result. Fortunately, there are off-the-shelf implementations of such algorithms, such as the Nvidia Collective Communications Library (NCCL) or PyTorch Gloo. DDP relies on these implementations under the hood, depending on your hardware. Please check out the DDP documentation for more details.

The mirrored strategy only works if all GPUs stay perfectly synchronized after each training step. Even tiny differences might compound over time and break synchronization. To avoid this, the all-reduce operation must produce the exact same result across all GPUs (NCCL and Gloo guarantee that), and the optimizer step must update the parameters in exactly the same way on each replica. In practice, this requires consistent hardware and software environments across all devices, and you must also ensure that the optimizer step only uses deterministic operations. Please see PyTorch's documentation for more details on how to make your code deterministic.

Data parallelism is great, but what if the whole model does not fit on a single GPU? For example, many large transformers are too big for even the largest GPUs. So let's try to split the neural net itself across GPUs, rather than entirely replicating it. This is called *model parallelism*.

Model Parallelism

There are many ways to split a neural net, each with different trade-offs. For example, Figure 17-18 shows how a fully connected network can be sliced horizontally—which is called *pipeline parallelism* (PP)—or vertically—which is called *tensor parallelism* (TP). In PP, the model is split into consecutive groups of layers called *stages* (e.g., one transformer block per GPU). In TP, each layer is split along the features dimension: for example, a `Linear` layer with 512 inputs and 512 outputs might be split into two `Linear` layers, each with 512 inputs and 256 outputs, and each living on its own GPU.

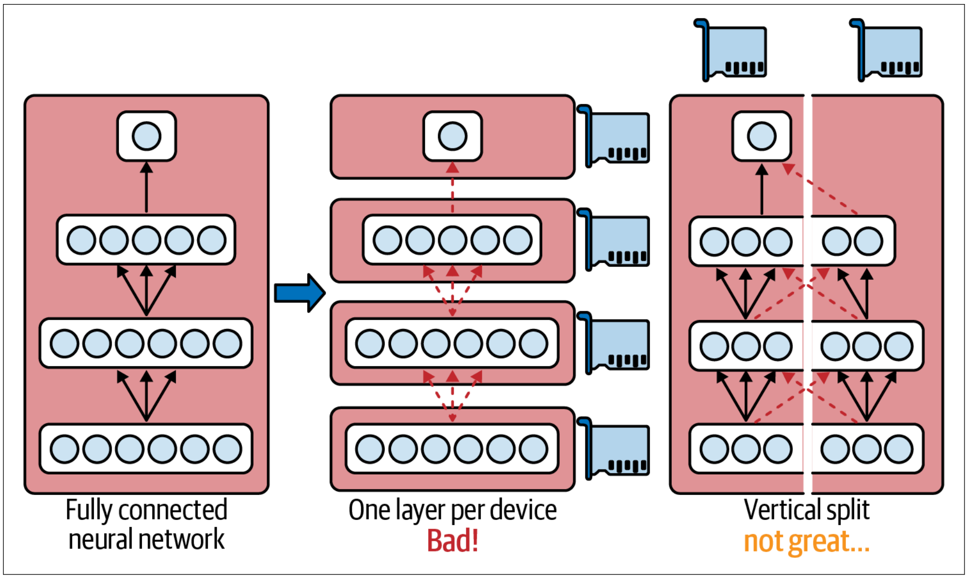


Figure 17-18. Pipeline parallelism versus tensor parallelism

The benefit of TP is that it allows you to train a very wide model since each GPU handles a different vertical slice. However, during the forward pass, each partial layer must send its outputs to every other device before the next layer can start, and the same is true during the backward pass. Luckily, modern interconnects (NVLink, NVSwitch, InfiniBand) have very high bandwidth and low latency, and we can use optimized *all-gather* operations that are designed to efficiently concatenate chunks of data spread across many nodes and ensure that each node ends up with a full copy. Nevertheless, I/O bottlenecks can be a major challenge.

The benefit of PP is that you can train a very deep model, since each stage lives on a separate GPU. In general, PP doesn't require as much I/O as TP, since we only need two communications per stage (one receive, one send) during the forward pass, and similarly two during the backward pass. However, a naive implementation would only use one GPU at a time, as the current batch progresses through each stage sequentially.

Fortunately, with some smart engineering, it's possible to make all GPUs work in parallel on different batches. That's a mix of data parallelism and pipeline parallelism. This idea was first proposed in a 2018 paper[33] by a team of researchers from Carnegie Mellon University, Stanford University, and Microsoft Research. In their system, named *PipeDream*, the model is split into successive stages, and each device alternates

33 Aaron Harlap et al., "PipeDream: Fast and Efficient Pipeline Parallel DNN Training", arXiv preprint arXiv:1806.03377 (2018).

a forward pass using a batch from its input queue, then a backward pass using a batch from its gradients queue, as shown in Figure 17-19. This results in an asynchronous pipeline in which all machines work in parallel with very little idle time.

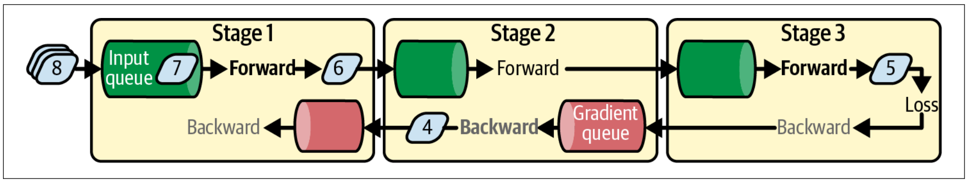


Figure 17-19. PipeDream's pipeline parallelism

However, as it's presented here, PipeDream would not work so well. To understand why, consider mini-batch #5 in Figure 17-19: when it went through stage 1 during the forward pass, the gradients from mini-batch #4 had not yet been backpropagated through that stage, but by the time #5's gradients flow back to stage 1, #4's gradients will have been used to update the model parameters, so #5's gradients will be a bit outdated. Such outdated gradients are called *stale gradients*; they can slow down convergence, introducing noise and wobble effects (the learning curve may contain temporary oscillations), or they can even make the training algorithm diverge. Indeed, if there are intermediate weight updates between the moment some gradients are computed and the moment they are applied, these gradients may not point in the right direction (see Figure 17-20).

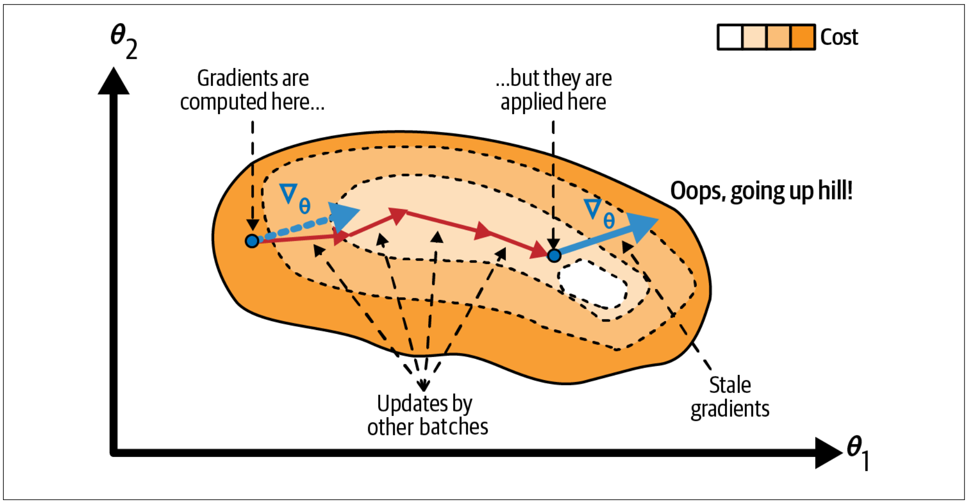


Figure 17-20. Stale gradients when using asynchronous updates

The more stages there are, the worse this problem becomes. The paper's authors proposed methods to mitigate this issue, though: for example, each stage saves weights during forward propagation and restores them during backpropagation, to ensure that the same weights are used for both the forward pass and the backward pass. This is called *weight stashing*. Thanks to this, PipeDream demonstrated impressive scaling capability.

If you don't have multiple GPUs to train or fine-tune large transformers, you can rent them using platforms such as Amazon Web Services, Google Cloud Platform, Microsoft Azure, RunPod, Lambda Labs, CoreWeave, Vast.ai, TensorDock, Paperspace, and many more.

When using transformers, there are two more ways to split the computations across GPUs:

Context parallelism
: Each GPU only stores and processes a subset of each input sequence, along the sequence dimension. For example, GPU #1 gets tokens for positions 1 to 100, GPU #2 gets tokens 101 to 200, and so on. In the MHA layers, each GPU computes matrices **Q**, **K**, and **V** only for its own tokens, but then all GPUs share their **K** and **V** matrices with every other GPU, so each GPU can then compute the attention scores and outputs for its own tokens. This reduces activation memory per GPU, since each GPU only needs to process a fraction of the token activations. It also reduces compute per GPU, since each GPU computes only a fraction of the attention matrix.

Expert parallelism
: In an MoE transformer, the experts can be distributed across different GPUs. The main difficulty is that this requires *all-to-all* communication (not just all-gather or all-reduce) to send the right tokens to the right experts. This is often done twice per MoE layer: once to send tokens in, once to return outputs. So once again, I/O bandwidth can quickly become the main bottleneck.

Another important technique to speed up training across multiple GPUs is the *Zero Redundancy Optimizer* (ZeRO), introduced in 2019 by Microsoft researchers.[34] It shards the optimizer states across GPUs, which reduces memory usage significantly.

This all sounds terribly difficult to implement, and it is. Luckily, some excellent open source libraries wrap all the complexity and allow you to efficiently train large transformers across multiple servers and GPU devices in just a few lines of code. Here are the main ones:

34 Samyam Rajbhandari et al., "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv preprint arXiv:1910.02054 (2019).

DeepSpeed, by Microsoft
: DeepSpeed is great for extreme scale multiserver training. It implements the ZeRO optimizer, supports offloading optimizer parameters, and many more features.

Megatron-LM, by Nvidia
: This library is more focused on single-server training using model parallelism, but it is making progress on multiserver as well. It offers great speed thanks to highly optimized kernels, tensor parallelism, pipeline parallelism, mixed precision, and activation checkpointing.

Fully Sharded Data Parallelism (FSDP), by the PyTorch core team
: As you might expect from the PyTorch team, FSDP (successor or FairScale) is fully integrated in PyTorch and offers a nice and simple API. FSDP shards model parameters, gradients, and optimizer states across devices. However, it is not quite as feature-rich as DeepSpeed.

Accelerate, by Hugging Face
: As always with Hugging Face libraries, Accelerate is very user friendly. It provides a unified abstraction layer on top of other libraries, including DeepSpeed and FSDP. It supports mixed-precision, gradient accumulation, and more. With Accelerate, you can write simple, device-agnostic PyTorch code, then use a configuration CLI to run it on any distributed setup. Under the hood, it may use DeepSpeed, FSDP, or even PyTorch's built-in DDP.

For more details on training large transformers on GPU clusters, I recommend The Ultra-Scale Playbook: Training LLMs on GPU Clusters by the Hugging Face team.

Wow, we've covered quite a few techniques in this chapter! We looked at KV caching, speculative decoding, and other techniques to accelerate decoding, then we explored many ways to accelerate multi-head attention, from sparse attention methods to approximate attention methods and sharing projection matrices across attention heads, and of course FlashAttention, then we saw how to scale transformers using mixture of experts, and finally we looked at various ways to accelerate training, including parameter-efficient fine-tuning techniques such as LoRA, activation checkpointing, sequence packing and bucketing, gradient accumulation, and finally distributing your data and computations across multiple GPUs. Phew!

In the next chapter (which you will find in the main book), we will switch to an entirely different topic: representation learning and generative ML using autoencoders, generative adversarial networks (GANs), and diffusion models.

Exercises

1. What is KV caching, and what is it used for? Is it best for training or inference? Is it best for encoder-only models, or decoder-only models?
2. When would you want to use speculative decoding? How does it work? Can you use any draft model or are there some constraints?
3. Can you name several techniques designed specifically to parallelize text generation?
4. What is the core idea of all sparse attention methods? Can you name some of the most important ones and summarize their main ideas and benefits?
5. Can you name a low-rank attention method? How about a kernel-based attention method? How do they work?
6. What are the trade-offs when tweaking the group size in GQA?
7. Does multi-head latent attention increase the KV cache size? Why?
8. What are the main types of memory in a GPU?
9. Why is FlashAttention so much faster than previous implementations?
10. What are the main challenges when training an MoE model? What are some of the most common solutions?
11. While training a large transformer, what are the four main causes of memory saturation? Can you name the main techniques that can reduce each of them?
12. Choose a large transformer model and try to fine-tune it on the dataset of your choice using LoRA.
13. If you have access to multiple GPUs, try using Hugging Face Accelerate to fine-tune a large transformer model on the dataset of your choice.