

# Efficient Attention

Karl Stratos

## 1 FlashAttention

FlashAttention [2] is a GPU-friendly implementation of the transformer attention layer. It follows existing works on memory-efficient attention, but explicitly focuses on reducing the number of reads from the GPU’s high bandwidth memory (HBM). This is because data movement has been shown to be the performance bottleneck in transformers, not FLOP count [8, 3]. As a result, FlashAttention reduces the memory overhead *and* increases the wall-clock speed.

### 1.1 Forward Function

Numerically stable attention on  $v_1 \dots v_T \in \mathbb{R}^D$  with scores  $\alpha_1 \dots \alpha_T \in \mathbb{R}$  is defined as

$$o = \sum_{t=1}^T \frac{\exp(\alpha_t - \max_{s=1}^T \alpha_s)}{\sum_{l=1}^T \exp(\alpha_l - \max_{s=1}^T \alpha_s)} v_t \in \mathbb{R}^D$$

Naive calculation requires three passes over the sequences (for the max, sum, and output). Several works instead consider the streaming version [6, 4, 7]

$$\begin{aligned} c_0 &= -\infty & c_t &= \max(c_{t-1}, \alpha_t) \\ \pi_0 &= 0 & \pi_t &= \exp(c_{t-1} - c_t) \pi_{t-1} + \exp(\alpha_t - c_t) \\ o_0 &= 0^D & o_t &= \frac{\pi_{t-1} \exp(c_{t-1} - c_t)}{\pi_t} o_{t-1} + \frac{\exp(\alpha_t - c_t)}{\pi_t} v_t \end{aligned}$$

We can easily show that  $c_t = \max_{l \leq t} \alpha_l$ ,  $\pi_t = \sum_{l \leq t} \exp(\alpha_l - c_t)$ , and  $o_t = \sum_{l \leq t} \frac{\exp(\alpha_l - c_t)}{\pi_t} v_l$  for  $t = 1 \dots T$  (thus  $o = o_T$ ). In practice, we (i) overwrite the variables, (ii) stream in chunks, and (iii) batch queries.

The FlashAttention forward function is given below. We use the superscript  $[i]$  to denote the  $i$ -th batch and  $[j]$  to denote the  $j$ -th chunk. See Appendix A for the broadcasting notation.

**Input:**  $Q \in \mathbb{R}^{L \times d}$ ,  $K \in \mathbb{R}^{T \times d}$ ,  $V \in \mathbb{R}^{T \times D}$ , batch size  $\Gamma$ , chunk size  $\Lambda$   
**Output:**  $c \in \mathbb{R}^{L \times 1}$ ,  $\pi \in \mathbb{R}^{L \times 1}$ ,  $O \in \mathbb{R}^{L \times D}$

- $(c, \pi, O) \leftarrow (-\infty^{L \times 1}, 0^{L \times 1}, 0^{L \times D})$
- For  $j = 1 \dots \lceil \frac{T}{\Lambda} \rceil$ , load  $K^{[j]} \in \mathbb{R}^{\Lambda \times d}$ ,  $V^{[j]} \in \mathbb{R}^{\Lambda \times D}$ :
  - For  $i = 1 \dots \lceil \frac{L}{\Gamma} \rceil$ , load  $Q^{[i]} \in \mathbb{R}^{\Gamma \times d}$ :
 
$$A^{[i,j]} \leftarrow \frac{Q^{[i]}(K^{[j]})^\top}{\sqrt{d}} \in \mathbb{R}^{\Gamma \times \Lambda}$$

$$c_{\text{new}}^{[i]} \leftarrow \max(c^{[i]}, A^{[i,j]}. \mathbf{max}(2)) \in \mathbb{R}^{\Gamma \times 1}$$

$$\pi_{\text{new}}^{[i]} \leftarrow \exp(c^{[i]} - c_{\text{new}}^{[i]}) \otimes \pi^{[i]} + \exp(A^{[i,j]} \ominus c_{\text{new}}^{[i]}) . \mathbf{sum}(2) \in \mathbb{R}^{\Gamma \times 1}$$

$$O_{\text{new}}^{[i]} \leftarrow (\pi^{[i]} \odot \pi_{\text{new}}^{[i]}) \otimes \exp(c^{[i]} - c_{\text{new}}^{[i]}) \otimes O^{[i]} + (\exp(A^{[i,j]} \ominus c_{\text{new}}^{[i]}) \odot \pi_{\text{new}}^{[i]}) V^{[j]} \in \mathbb{R}^{\Gamma \times D}$$

$$(c^{[i]}, \pi^{[i]}, O^{[i]}) \leftarrow (c_{\text{new}}^{[i]}, \pi_{\text{new}}^{[i]}, O_{\text{new}}^{[i]})$$

Like previous memory-efficient attention works, the algorithm has a memory overhead of  $O(L)$ , rather than  $O(LT)$  in standard attention (A.1).

## 1.2 Backward Function

To maintain the memory efficiency in the backward function, we must recompute the  $L \times T$  attention matrix in blocks (i.e., gradient checkpointing). We write  $z_X = \frac{\partial L}{\partial X} \in \mathbb{R}^{m \times n}$  to denote the gradient of the loss  $L \in \mathbb{R}$  with respect to tensor  $X \in \mathbb{R}^{m \times n}$  (see Appendix C if you need a review of backpropagation). Forget about the streaming attention and consider the correct implementation:

$$\begin{aligned}
 A &= \frac{QK^\top}{\sqrt{d}} \in \mathbb{R}^{L \times T} & P &= \exp(A \odot c) \odot \pi \in \mathbb{R}^{L \times T} & O &= PV \in \mathbb{R}^{L \times D} \\
 z_A &= P \circledast (z_P \ominus (P \circledast z_P).\mathbf{sum}(2)) \in \mathbb{R}^{L \times T} & z_P &= z_O V^\top \in \mathbb{R}^{L \times T} & z_O &\in \mathbb{R}^{L \times D} \\
 z_Q &\leftarrow z_Q + \frac{z_A K}{\sqrt{d}} \in \mathbb{R}^{L \times d} & z_K &\leftarrow z_K + \frac{z_A^\top Q}{\sqrt{d}} \in \mathbb{R}^{T \times d} & z_V &\leftarrow z_V + P^\top z_O \in \mathbb{R}^{T \times D}
 \end{aligned}$$

(We assume that the intermediate variables  $A, P$  are not used outside this operation.) We have  $c, \pi \in \mathbb{R}^{L \times 1}$  from forward and can easily recompute the attention matrix  $P$  in blocks. The bad news is that  $z_A$  requires reducing over  $T$  elements in  $u = (P \circledast z_P).\mathbf{sum}(2) \in \mathbb{R}^{L \times 1}$ . We can avoid this issue by reparameterizing  $u = (O \circledast z_O).\mathbf{sum}(2)$  where the sum is now over  $D$  dimensions [2]. The identity is not standard linear algebra but can be directly verified: for any query  $i \in [L]$ ,

$$u_i = \sum_{t=1}^T P_{i,t} z_{P,i,t} = \sum_{t=1}^T \underbrace{P_{i,t}}_{\mathbb{R}} \underbrace{z_{O,i}}_{1 \times D} \underbrace{v_t^\top}_{D \times 1} = \underbrace{z_{O,i}}_{1 \times D} \underbrace{\left( \sum_{t=1}^T P_{i,t} v_t^\top \right)}_{D \times 1} = \underbrace{z_{O,i}}_{1 \times D} \underbrace{O_i}_{D \times 1}$$

Putting together, we give the FlashAttention backward function below:

**Input:**  $Q \in \mathbb{R}^{L \times d}, K \in \mathbb{R}^{T \times d}, V \in \mathbb{R}^{T \times D}$ , batch size  $\Gamma$ , chunk size  $\Lambda$   
**From forward:**  $c \in \mathbb{R}^{L \times 1}, \pi \in \mathbb{R}^{L \times 1}, O \in \mathbb{R}^{L \times D}$   
**Gradients slots:**  $z_O \in \mathbb{R}^{L \times D}$  (read);  $z_Q \in \mathbb{R}^{L \times d}, z_K \in \mathbb{R}^{T \times d}, z_V \in \mathbb{R}^{T \times D}$  (read/write)

- For  $j = 1 \dots \lceil \frac{T}{\Lambda} \rceil$ , load  $K^{[j]} \in \mathbb{R}^{\Lambda \times d}, V^{[j]} \in \mathbb{R}^{\Lambda \times D}, z_K^{[j]} \in \mathbb{R}^{\Lambda \times d}, z_V^{[j]} \in \mathbb{R}^{\Lambda \times D}$ :
  - For  $i = 1 \dots \lceil \frac{L}{\Gamma} \rceil$ , load  $Q^{[i]} \in \mathbb{R}^{\Gamma \times d}, c^{[i]}, \pi^{[i]} \in \mathbb{R}^{\Gamma \times 1}, O^{[i]} \in \mathbb{R}^{\Gamma \times D}, z_Q^{[i]} \in \mathbb{R}^{\Gamma \times d}, z_O^{[i]} \in \mathbb{R}^{\Gamma \times D}$ :
 
$$\begin{aligned}
 P^{[i,j]} &\leftarrow \exp\left(\frac{Q^{[i]}(K^{[j]})^\top}{\sqrt{d}} \ominus c^{[i]}\right) \odot \pi^{[i]} \in \mathbb{R}^{\Gamma \times \Lambda} & z_V^{[j]} &\leftarrow z_V^{[j]} + (P^{[i,j]})^\top z_O^{[i]} \in \mathbb{R}^{\Lambda \times D} \\
 z_A^{[i,j]} &\leftarrow P^{[i,j]} \circledast (z_O^{[i]}(V^{[j]})^\top \ominus (O^{[i]} \circledast z_O^{[i]}).\mathbf{sum}(2)) \in \mathbb{R}^{\Gamma \times \Lambda} & z_K^{[j]} &\leftarrow z_K^{[j]} + \frac{(z_A^{[i,j]})^\top Q^{[i]}}{\sqrt{d}} \in \mathbb{R}^{\Lambda \times d} \\
 z_Q^{[i]} &\leftarrow z_Q^{[i]} + \frac{z_A^{[i,j]} K^{[j]}}{\sqrt{d}} \in \mathbb{R}^{\Gamma \times d} & &
 \end{aligned}$$

## 1.3 Memory Read Analysis

Assume  $L = T, D = d$ , and  $T > d$ . Standard attention makes  $O(T^2)$  reads since it explicitly computes the full attention probability matrix  $P \in \mathbb{R}^{T \times T}$  to read from. FlashAttention makes  $O(Td + \frac{T^2 d}{\Lambda})$  reads: loading  $K, V \in \mathbb{R}^{T \times d}$  in the outer loop and  $Q, O \in \mathbb{R}^{T \times d}$  for  $\frac{T}{\Lambda}$  times in the inner loop. Assuming an SRAM size of  $M < Td$ , FlashAttention sets  $\Lambda = \frac{M}{4d}$  as the largest chunk size that can fit the query/key/value vectors and other intermediate results. This yields  $O(\frac{T^2 d^2}{M})$  reads, which is fewer than  $O(T^2)$  if  $d < \sqrt{M}$ .

## 1.4 FlashAttention-2

FlashAttention-2 (FA-2) [1] further optimizes FlashAttention (FA-1) especially for longer sequences (i.e., large  $T$ ) with additional engineering insights.

- Parallelization over chunks.** FA-1 only parallelizes over batches/heads (by putting them on different thread blocks).<sup>1</sup> FA-2 additionally parallelizes over chunks.<sup>2</sup> This is trivial in the forward function since the chunks are independent (i.e., the steps in the outer loop do not depend on each other). They are not

<sup>1</sup>We conflate heads with batches since heads can be viewed as additional queries (Appendix B.3.1).

<sup>2</sup>Confusingly, parallelization over chunks was first branded as [Flash-Decoding](#).

independent in the backward function because of the shared  $z_Q^{[i]}$ ; updating this is handled separately with an additional block.

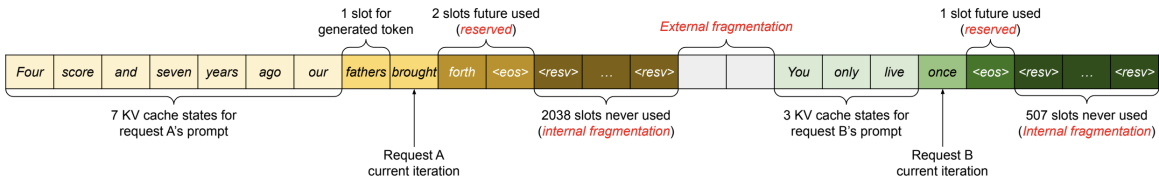
- **Parallelization over KV cache loading.** At decoding time, FA-2 splits the KV cache across different blocks (i.e., multiple blocks load the cache at the same time), as done in PagedAttention [5]. This speeds up loading the KV cache which is the primary bottleneck in inference. The blocks communicate intermediate results through HBM.
- **Work partitioning between warps.** Even within a thread block, we can optimize how work is partitioned between different warps (groups of 32 threads). FA-2 splits  $Q$  across warps while keeping  $K, V$  accessible to all warps to remove communication between warps.
- **Reducing non-matmul operation.** We can avoid repeating some non-matmul operations in FA-1: pulling out  $\mathcal{O}\pi_{\text{new}}^{[i]}$  when updating  $O_{\text{new}}^{[i]}$  in forward and combining  $c$  and  $\pi$  through logsumexp in backward.
- **Exploiting masking.** Causal attention makes it unnecessary to compute attention for a half of the blocks, so they are skipped. For the rest, only one block at the end of each row needs masking.

## 2 PagedAttention

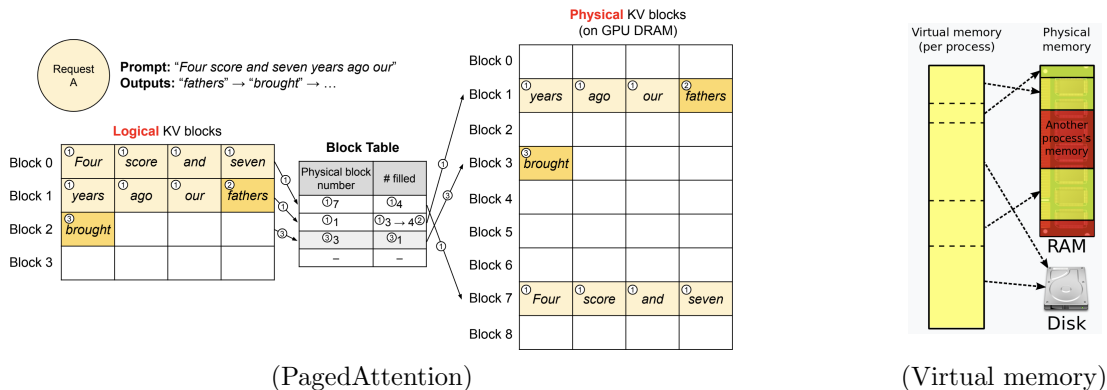
Transformer inference throughput is memory-bound: it’s low not because we don’t have enough compute power but because we don’t have enough memory to serve more requests concurrently. PagedAttention [5], implemented in the popular vLLM system, is a method to reduce memory fragmentation in managing KV caches. At inference time, a decoder-only transformer generates a response to a prompt  $x_1 \dots x_m \in \mathcal{X}$  in two stages.

1. Step  $t = 1$  (matrix-matrix): Compute  $x_{m+1} \sim p(\cdot | x_1 \dots x_m)$  with  $m + 1$  queries  $x_1 \dots x_m$  by causal attention.
2. Step  $t \geq 2$  (matrix-vector): Compute  $x_{m+t} \sim p(\cdot | x_1 \dots x_{m+t-1})$  with 1 new query  $x_{m+t-1}$ .

In the second stage, we compute multi-head attention from 1 query to all previously computed key-value embeddings: the **KV cache**. At the  $t$ -th step, the cache consists of  $(k_i^{(l)}, v_i^{(l)}) \in \mathbb{R}^d \times \mathbb{R}^d$  for all layers  $l = 1 \dots L$  and positions  $1 \leq i < t$ . Since they are typically packed into tensors, which are stored as contiguous sequences of bytes in memory (Appendix B), a naive memory management scheme such as reserving a maximum-length number of slots for each request results in memory fragmentation as illustrated by Kwon *et al.* [5]:



PagedAttention solves this problem by non-contiguously storing the KV cache as “physical” blocks of size  $B \stackrel{\text{default}}{=} 16$  on GPU memory. These blocks are dynamically managed by representing a request’s cache as “logical” blocks. This is analogous to the concept of virtual memory that allows each program to act as if it has access to a large, contiguous block of memory (image credit: [Wikipedia](#)).




PagedAttention allows for (i) simultaneous handling of multiple requests’ caches, (ii) sharing the same cache for different requests until not possible (e.g., parallel sampling, in-context learning, more complicated cases like beam search), (iii) scheduling which requests to handle first given limited GPU memory (e.g., “swap” the cache of an evicted sequence to CPU RAM). It also supports model parallel inference by managing multiple shards (which share the same cache). Since it needs additional computation for cache management, it results in a minor increase in latency. But it eliminates much of memory fragmentation, dramatically increasing the throughput when serving a large number of streaming requests.

## References

- [1] Dao, T. (2024). Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*.
- [2] Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. (2022). Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, **35**, 16344–16359.
- [3] Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., and Hoefler, T. (2021). Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, **3**, 711–732.
- [4] Jang, H., Kim, J., Jo, J.-E., Lee, J., and Kim, J. (2019). Mnnfast: A fast and scalable system architecture for memory-augmented neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 250–263.
- [5] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. (2023). Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP ’23*, page 611–626, New York, NY, USA. Association for Computing Machinery.
- [6] Milakov, M. and Gimelshein, N. (2018). Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*.
- [7] Rabe, M. N. and Staats, C. (2021). Self-attention does not need  $o(n^2)$  memory. *arXiv preprint arXiv:2112.05682*.
- [8] Shazeer, N. (2019). Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*.

## A Broadcasting Notation

We use the following binary operators for broadcasting: elementwise addition  $\oplus$ , elementwise subtraction  $\ominus$ , elementwise division  $\oslash$ , elementwise multiplication  $\otimes$ , elementwise division  $\oslash$ , pairwise maximum **max**, matrix multiplication **matmul**. If broadcasting is not needed, we use the standard mathematical notation (e.g., a matrix  $A \in \mathbb{R}^{m \times n}$  divided by a constant  $c \in \mathbb{R}$  is written as  $\frac{A}{c} \in \mathbb{R}^{m \times n}$ , standard matrix multiplication between  $A \in \mathbb{R}^{m \times n}$  and  $B \in \mathbb{R}^{n \times p}$  is written as  $AB \in \mathbb{R}^{m \times p}$ ). Remember the rules of broadcasting (poet: [Sasha Rush](#)):

There is a rule you should learn at long last, combination of tensors the task.	
Dims right-aligned,	$9 \times 1 \times 3$
extra left 1s assigned,	$1 \times 8 \times 1$
match paired dimensions: Broadcast!	<hr style="width: 50%; margin: 0 auto;"/>
	$9 \times 8 \times 3$

For example, if  $A \in \mathbb{R}^{9 \times 1 \times 3}$  and  $B \in \mathbb{R}^{8 \times 1}$ , the result  $C \leftarrow A \oplus B \in \mathbb{R}^{9 \times 8 \times 3}$  contains  $C_{i,j,k} = A_{i,1,k} + B_{j,1}$ ;  $C \leftarrow \mathbf{max}(A, B) \in \mathbb{R}^{9 \times 2 \times 3}$  contains  $C_{i,j,k} = \max(A_{i,1,k}, B_{j,1})$ . Matrix multiplication broadcasts over dimensions excluding the last two: if  $A \in \mathbb{R}^{9 \times 1 \times 3 \times 2}$  and  $B \in \mathbb{R}^{8 \times 2 \times 4}$ , the result  $C \leftarrow \mathbf{matmul}(A, B) \in \mathbb{R}^{9 \times 8 \times 3 \times 4}$  contains matrices  $C_{i,j} = A_{i,1} B_j \in \mathbb{R}^{3 \times 4}$  where tensors are sliced left to right. We write boldfaced  $A.\mathbf{max}(i)$  and  $A.\mathbf{sum}(i)$  to denote the maximum and summation along the  $i$ -th axis of  $A$ . When no axis is provided, all axes are used. We keep the collapsed dimension to preserve the number of dimensions. For example, if  $A \in \mathbb{R}^{3 \times 5 \times 2}$ , then  $A.\mathbf{max}(3) \in \mathbb{R}^{3 \times 5 \times 1}$ . See Appendix B for an overview of how tensors are stored in memory.

## A.1 Standard Attention

An implementation of the forward/backward function of standard attention, in our broadcasting notation, is given below:

**Input:**  $Q \in \mathbb{R}^{L \times d}$ ,  $K \in \mathbb{R}^{T \times d}$ ,  $V \in \mathbb{R}^{T \times D}$   
**Memory overhead (in length):**  $O(LT)$   
**Output:**  $O \in \mathbb{R}^{L \times D}$

1.  $A \leftarrow \frac{QK^\top}{\sqrt{d}} \in \mathbb{R}^{L \times T}$
2.  $c \leftarrow A.\mathbf{max}(2) \in \mathbb{R}^{L \times 1}$
3.  $S \leftarrow \exp(A \odot c) \in \mathbb{R}^{L \times T}$
4.  $\pi \leftarrow S.\mathbf{sum}(2) \in \mathbb{R}^{L \times 1}$
5.  $P \leftarrow S \oslash \pi \in \mathbb{R}^{L \times T}$
6.  $O \leftarrow PV \in \mathbb{R}^{L \times D}$

**Input:**  $Q \in \mathbb{R}^{L \times d}$ ,  $K \in \mathbb{R}^{T \times d}$ ,  $V \in \mathbb{R}^{T \times D}$ ;  $z_O \in \mathbb{R}^{L \times D}$ ,  
 $P \in \mathbb{R}^{L \times T}$  from forward  
**Memory overhead (in length):**  $O(LT)$   
**Output:**  $z_Q \in \mathbb{R}^{L \times d}$ ,  $z_K \in \mathbb{R}^{T \times d}$ ,  $z_V \in \mathbb{R}^{T \times D}$

1.  $z_V \leftarrow P^\top z_O \in \mathbb{R}^{T \times D}$
2.  $z_P \leftarrow z_O V^\top \in \mathbb{R}^{L \times T}$
3.  $z_A \leftarrow P \otimes z_P - (P \otimes z_P).\mathbf{sum}(2) \otimes P \in \mathbb{R}^{L \times T}$
4.  $z_Q \leftarrow \frac{z_A K}{\sqrt{d}} \in \mathbb{R}^{L \times d}$
5.  $z_K \leftarrow \frac{z_A^\top Q}{\sqrt{d}} \in \mathbb{R}^{T \times d}$

## B How Tensors Are Stored in Memory

### B.1 Memory Tape

Conceptually, memory (either CPU or GPU) can be viewed as a long sequence (“tape”) of bytes. The CPU memory (RAM) is managed by the operating system; the GPU memory is managed by the GPU itself. Modern addressing systems are 64-bit, yielding  $2^{64}$  bytes (16 billion GBs) of theoretically possible memory. A **pointer** is the index on the memory tape, representing the unique address of the referred byte (e.g., the pointer value `0xffe2ac6c` refers to the 4,293,045,356-th byte in hexadecimal notation).

In this section only, we assume indices start from 0. We can stride by the number of bytes needed for the considered data type to address memory at the level of the data type. For instance, if  $u = (u_0, u_1, \dots, u_7)$  is a vector of 8 `float32` numbers, and  $u_{\text{ptr}}$  is a pointer to  $u$ , the address of the  $i$ -th element  $u_i$  is

$$\text{address}(u_i) = u_{\text{ptr}} + \text{sizeof}(\text{float32})i$$

where `sizeof(float32) = 4`. In high-level languages like Triton, the data loading functions automatically take care of index scaling for the considered data type (e.g.,  $u_i = \text{load}(u_{\text{ptr}} + i)$  performs scaling under the hood). Thus for our purposes, we will view memory as a tape of floats.

### B.2 Flattening

To store multi-dimensional tensors in memory, they must be flattened. Two popular schemes are (i) row-major (NumPy, PyTorch) and (ii) column-major (Matlab, Fortran). For instance, the matrix  $A = [[a_{1,1}, a_{1,2}]; [a_{2,1}, a_{2,2}]] \in \mathbb{R}^{2 \times 2}$  is flattened as  $[a_{1,1}, a_{1,2}, a_{2,1}, a_{2,2}] \in \mathbb{R}^4$  under row-major ordering and  $[a_{1,1}, a_{2,1}, a_{1,2}, a_{2,2}] \in \mathbb{R}^4$  under column-major ordering. They have different implications in efficiency, but we will assume row-major. If we have a 3-dimensional tensor like  $A \in \mathbb{R}^{m \times n \times p}$  at  $A_{\text{ptr}}$ , the value  $A_{i,j,k}$  can be accessed as

$$A_{i,j,k} = \text{load}(A_{\text{ptr}} + (np)i + (p)j + k)$$

We must use the correct stride for each dimension to account for row-major flattening. For instance, the value in the 11-th column of the 5-th row of the 4-th matrix has the address  $A_{\text{ptr}} + (np)3 + (p)4 + 10$  because each row has  $p$  floats and each matrix has  $np$  floats. Using the range notation (e.g., `range(3) = [0, 1, 2, 3]`) and broadcasting, we can load the entire tensor as

$$A = \text{load}(A_{\text{ptr}} \oplus (n \times p \times \text{range}(m))).\mathbf{view}(m, 1, 1) \oplus (p \times \text{range}(n)).\mathbf{view}(1, n, 1) \oplus (\text{range}(p)).\mathbf{view}(1, 1, p)$$

### B.3 Memory-Free Operations

**Transpose.** Switching the axes  $i$  and  $j$  of a tensor  $A$  can be achieved without making a copy of  $A$ . For instance, the elements of  $B \leftarrow A.\mathbf{transpose}(1, 2) \in \mathbb{R}^{m \times p \times n}$  where  $A \in \mathbb{R}^{m \times n \times p}$  is pointed to by  $A_{\text{ptr}}$  can be accessed as

$$B_{i,j,k} = \text{load}(A_{\text{ptr}} + (np)i + (n)k + j)$$

**View.** Viewing a tensor as one with a new shape by either merging or splitting certain consecutive dimensions can be achieved without making a copy. For instance, the elements of  $B \leftarrow A.\mathbf{view}(m, np) \in \mathbb{R}^{m \times np}$  where  $A \in \mathbb{R}^{m \times n \times p}$  is pointed to by  $A_{\text{ptr}}$  can be accessed as

$$B_{i,j} = \text{load}(A_{\text{ptr}} + (np)i + (n)k + l) \quad j = pk + l$$

Note that any  $j \in [np - 1]$  can be uniquely decomposed as  $j = pk + l$  where  $k \in [n - 1]$  and  $l \in [p - 1]$  (the formula is  $l = j \bmod p$  and  $k = \frac{j-l}{p}$ ). Similarly, the elements of  $C \leftarrow A.\mathbf{view}(m_1, m_2, n, p) \in \mathbb{R}^{m_1 \times m_2 \times n \times p}$  where  $m = m_1 m_2$  can be accessed as

$$C_{i,j,k,l} = \text{load}(A_{\text{ptr}} + (np)((m_2)i + j) + (p)k + l)$$

The viewing trick only works when the dimensions in the new shape are in the same order as in the old shape (after merging or splitting). For instance, calling **transpose** to switch dimensions, then merging those dimensions (in the resulting transposed tensor) by **view** will not work. In this case, we make an explicit copy of the tensor by calling **reshape**.

### B.3.1 Application

We can convert standard attention to multi-head (almost) for free of additional memory as follows (FLOP count remains the same).

**Input:**  $Q \in \mathbb{R}^{L \times d}$ ,  $K \in \mathbb{R}^{T \times d}$ ,  $V \in \mathbb{R}^{T \times D}$   
**Parameters:**  $W_q, W_k \in \mathbb{R}^{d \times d}$  and  $W_v, W_f \in \mathbb{R}^{D \times D}$ ,  $H$  dividing both  $d$  and  $D$   
**Output:**  $O \in \mathbb{R}^{L \times D}$

- $\mathbf{Q} \leftarrow (QW_q).\mathbf{view}(L, H, \frac{d}{H}).\mathbf{transpose}(1, 2) \in \mathbb{R}^{H \times L \times \frac{d}{H}}$
- $\mathbf{K} \leftarrow (KW_k).\mathbf{view}(T, H, \frac{d}{H}).\mathbf{transpose}(1, 2) \in \mathbb{R}^{H \times T \times \frac{d}{H}}$
- $\mathbf{V} \leftarrow (VW_v).\mathbf{view}(T, H, \frac{D}{H}).\mathbf{transpose}(1, 2) \in \mathbb{R}^{H \times T \times \frac{D}{H}}$
- $\mathbf{A} \leftarrow \sqrt{\frac{H}{d}} \mathbf{matmul}(\mathbf{Q}, \mathbf{K}.\mathbf{transpose}(2, 3)) \in \mathbb{R}^{H \times L \times T}$
- $\mathbf{c} \leftarrow \mathbf{A}.\mathbf{max}(3) \in \mathbb{R}^{H \times L \times 1}$
- $\mathbf{S} \leftarrow \exp(\mathbf{A} \ominus \mathbf{c}) \in \mathbb{R}^{H \times L \times T}$
- $\pi \leftarrow \mathbf{S}.\mathbf{sum}(3) \in \mathbb{R}^{H \times L \times 1}$
- $\mathbf{P} \leftarrow \mathbf{S} \oslash \pi \in \mathbb{R}^{H \times L \times \frac{D}{H}}$
- $\mathbf{O} \leftarrow \mathbf{matmul}(\mathbf{P}, \mathbf{V}) \in \mathbb{R}^{H \times L \times \frac{D}{H}}$
- $O \leftarrow \mathbf{O}.\mathbf{transpose}(1, 2).\mathbf{reshape}(L, D)W_f \in \mathbb{R}^{L \times D}$  #  $O(LD)$  additional memory

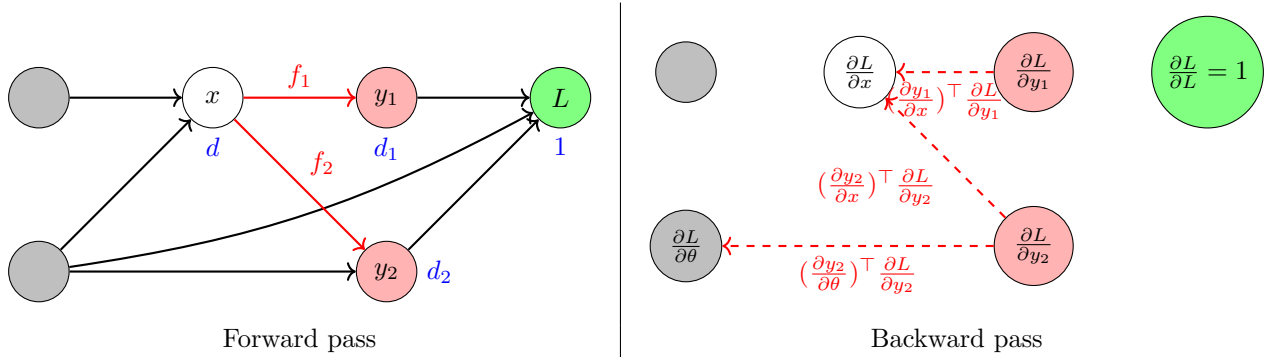
## C A Quick Review of Backprop

A loss  $L \in \mathbb{R}$  on a batch is the final node of a DAG whose root nodes are inputs and parameters; internal nodes are hidden states. The goal is to compute the gradient of  $L$  with respect to all nodes. A node  $x \in \mathbb{R}^d$  affects  $L$  only through its children  $y_k = f_k(x) \in \mathbb{R}^{d_k}$  where  $f_k : \mathbb{R}^d \rightarrow \mathbb{R}^{d_k}$  is some differentiable function. By the chain rule

$$\frac{\partial L}{\partial x} = \sum_{k=1}^K \left( \frac{\partial y_k}{\partial x} \right)^\top \frac{\partial L}{\partial y_k} \quad (1)$$

where  $\left( \frac{\partial y_k}{\partial x} \right)^\top \in \mathbb{R}^{d \times d_k}$  is the (transposed) Jacobian of  $f_k$  with respect to  $x$  (i.e.,  $\left( \frac{\partial y_k}{\partial x} \right)_{i,j}^\top = \frac{\partial y_{k,j}}{\partial x_i}$ ).<sup>3</sup> **Backprop** computes (1) for all nodes by traversing the DAG from  $L$  in a *reverse* topological order and at each node *accumulating* the Jacobian-gradient product to all its parents' gradient slots (initialized to zeros). This works because of the DAG structure. In the example, before reaching the node  $x$ , we will have finished accumulating  $\frac{\partial L}{\partial x}$  (thus the value will be correct):

<sup>3</sup>The sum over children is consistent with the "normal" chain rule without the sum if we view  $x$  as affecting  $L$  through a single node  $y = (y_1 \dots y_K) \in \mathbb{R}^{d_1 + \dots + d_K}$  so that  $\left( \frac{\partial y}{\partial x} \right)^\top \frac{\partial L}{\partial y} = \sum_{k=1}^K \left( \frac{\partial y_k}{\partial x} \right)^\top \frac{\partial L}{\partial y_k}$ .



A DAG is built by predefined operators that specify (1) **forward**: how to map parent tensors to an output tensor, and (2) **backward**: how to compute the Jacobian-gradient product for each parent. For example, if  $w = f(u, v)$  is a node created by the operator  $f(x, y) = \text{ReLU}(x \otimes y) \in \mathbb{R}^d$  with  $z = \frac{\partial L}{\partial w} \in \mathbb{R}^d$ , the backward function accumulates  $(\frac{\partial w}{\partial u})^\top z$  and to the gradient slot of  $u$  and  $(\frac{\partial w}{\partial v})^\top z$  and to the gradient slot of  $v$ . In PyTorch,

```

class Op(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, y):
        ctx.save_for_backward(x, y, x * y > 0)
        return (x * y > 0) * x * y

    @staticmethod
    def backward(ctx, z): # z is the grad wrt. the node
        x, y, inds = ctx.saved_tensors
        return inds * y * z, inds * x * z

u = torch.randn((8,), requires_grad=True)
L = (Op.apply(u, u) + 3 * u + u.exp()).sum() # Loss
L.backward() # Computes u.grad

class Layer(torch.nn.Module):
    """Computes ReLU(w * x) with a learnable w"""
    def __init__(self, d):
        super().__init__()
        self.w = torch.nn.Parameter(torch.empty(d))
        torch.nn.init.uniform_(self.w, -0.1, 0.1)

    def forward(self, x):
        return Op.apply(self.w, x)

layer = Layer(8)
u = torch.randn((8,), requires_grad=True)
L = layer(layer(u)).sum() # Loss
L.backward() # Computes u.grad, layer.w.grad

```

## C.1 Tips and Examples

**Matrix multiplication.** Treat a matrix of shape  $m \times n$  as a vector of length  $mn$ . We can then derive

$$\underbrace{C}_{m \times p} = \underbrace{A}_{m \times n} \underbrace{B}_{n \times p} : \quad \underbrace{z_A}_{m \times n} \leftarrow z_A + \underbrace{z_C}_{m \times p} \underbrace{B^\top}_{p \times n} \quad \underbrace{z_B}_{n \times p} \leftarrow z_B + \underbrace{A^\top}_{n \times m} \underbrace{z_C}_{m \times p}$$

**Independent dimensions.** If the dimensions along an axis are independent, the Jacobian is diagonal and the backward function can treat the dimensions as independent transformations. In the previous example, we had

$$t = \text{ReLU}(x \otimes y) \Leftrightarrow t_i = \max(0, x_i y_i) : \quad z_{x,i} \leftarrow z_{x,i} + \mathbb{1}(x_i y_i > 0) y_i z_{t,i} \quad z_{y,i} \leftarrow z_{y,i} + \mathbb{1}(x_i y_i > 0) x_i z_{t,i}$$

The independent transformations can be multi-dimensional, e.g., batched matrix multiplication

$$\underbrace{C}_{N \times m \times p} = \text{matmul}(\underbrace{A}_{N \times m \times n}, \underbrace{B}_{N \times n \times p}) : \quad \underbrace{z_A}_{N \times m \times n} \leftarrow z_A + \text{matmul}(\underbrace{z_C}_{N \times m \times p}, \underbrace{B.\text{transpose}(2, 3)}_{N \times p \times n})$$

**Softmax.** If  $p = \text{softmax}(x) \in \mathbb{R}^d$ , we can verify  $\frac{\partial p_i}{\partial x_i} = p_i(\mathbb{1}(i = j) - p_j)$  which implies

$$z_{x,i} \leftarrow z_{x,i} + p_i z_{p,i} - \left( \sum_{j=1}^d p_j z_{p,j} \right) p_i \quad \Leftrightarrow \quad z_x \leftarrow z_x + p \otimes z_p - (p \otimes z_p).\text{sum}() p \quad (2)$$

The shifted softmax  $p = \text{softmax}(x - x.\text{max}()) \in \mathbb{R}^d$  has the same backward function (2) (hint: the gradient wrt.  $x.\text{max}()$  vanishes by the property of softmax). The row-wise softmax  $P = X.\text{softmax}(2) \in \mathbb{R}^{N \times d}$  where we apply (shifted) softmax over the rows of  $X \in \mathbb{R}^{N \times d}$  independently is then

$$z_X \leftarrow z_X + P \otimes z_P - \underbrace{(P \otimes z_P).\text{sum}(2)}_{N \times 1} \otimes \underbrace{P}_{N \times d} = z_X + P \otimes (z_P \ominus (P \otimes z_P).\text{sum}(2))$$