Quantization

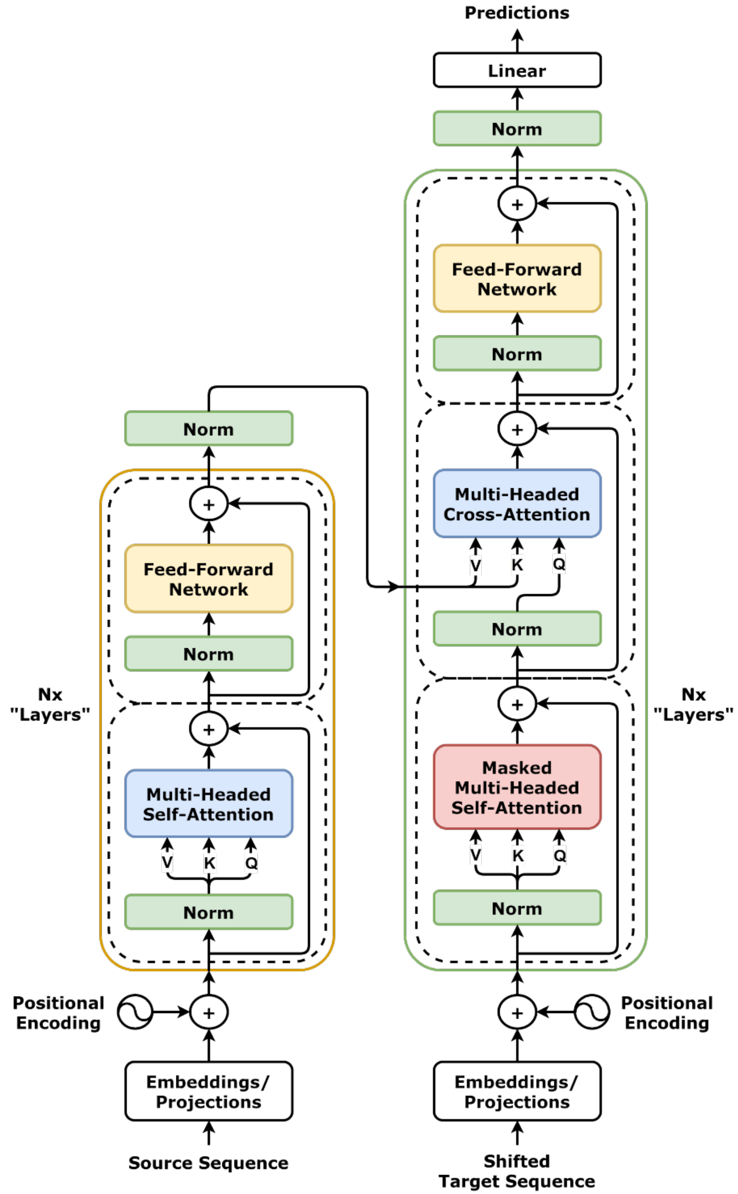
by Daniel Voigt Godoy

Data Science Retreat – October 2024

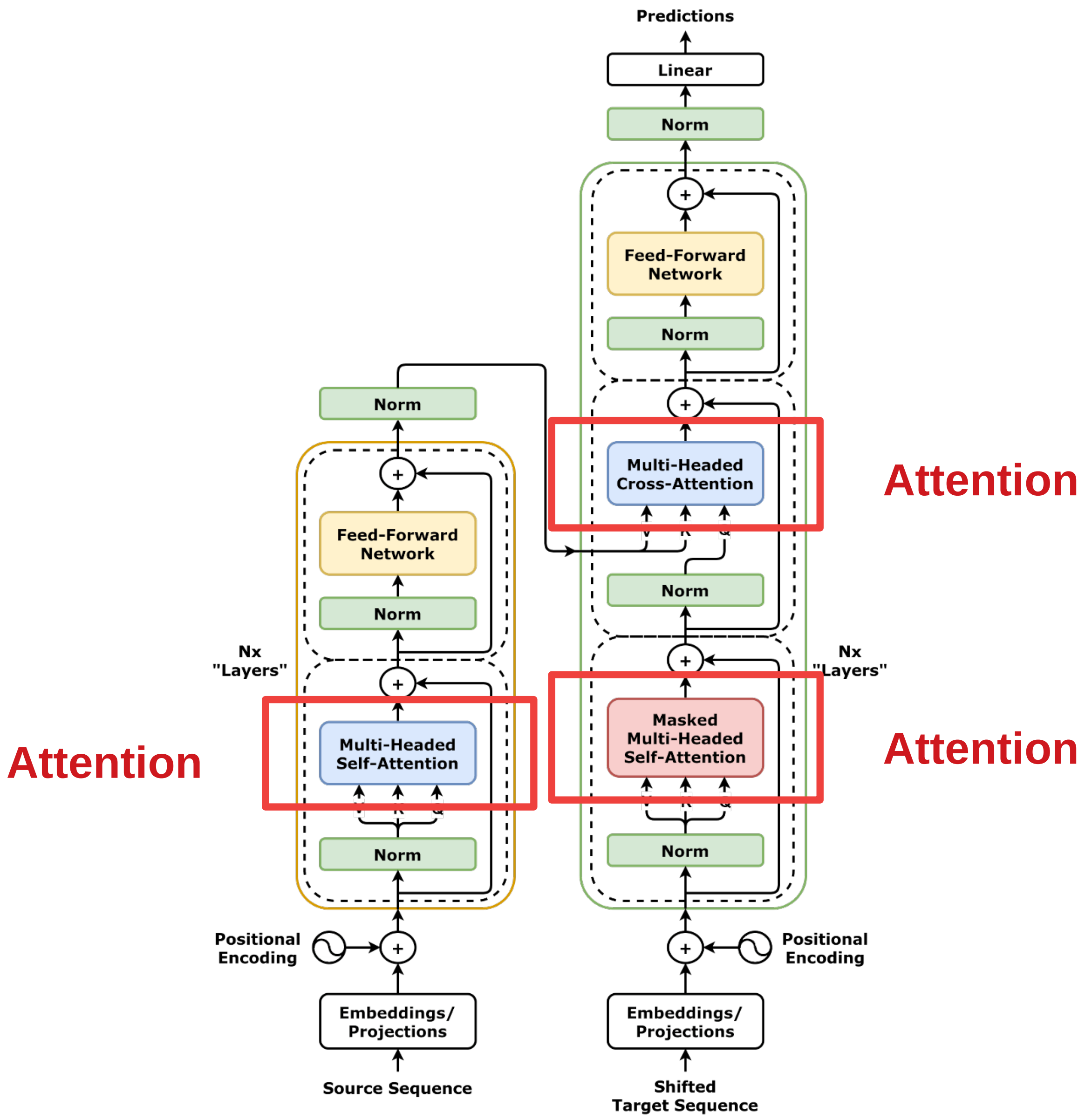
Why Quantize?

- To lower memory usage
  - Transformers are HUGE
  - Models do not fit into GPU RAM
  - FP32 is wasteful
- For faster inference
  - Loads weights faster
  - Integer arithmetics runs faster

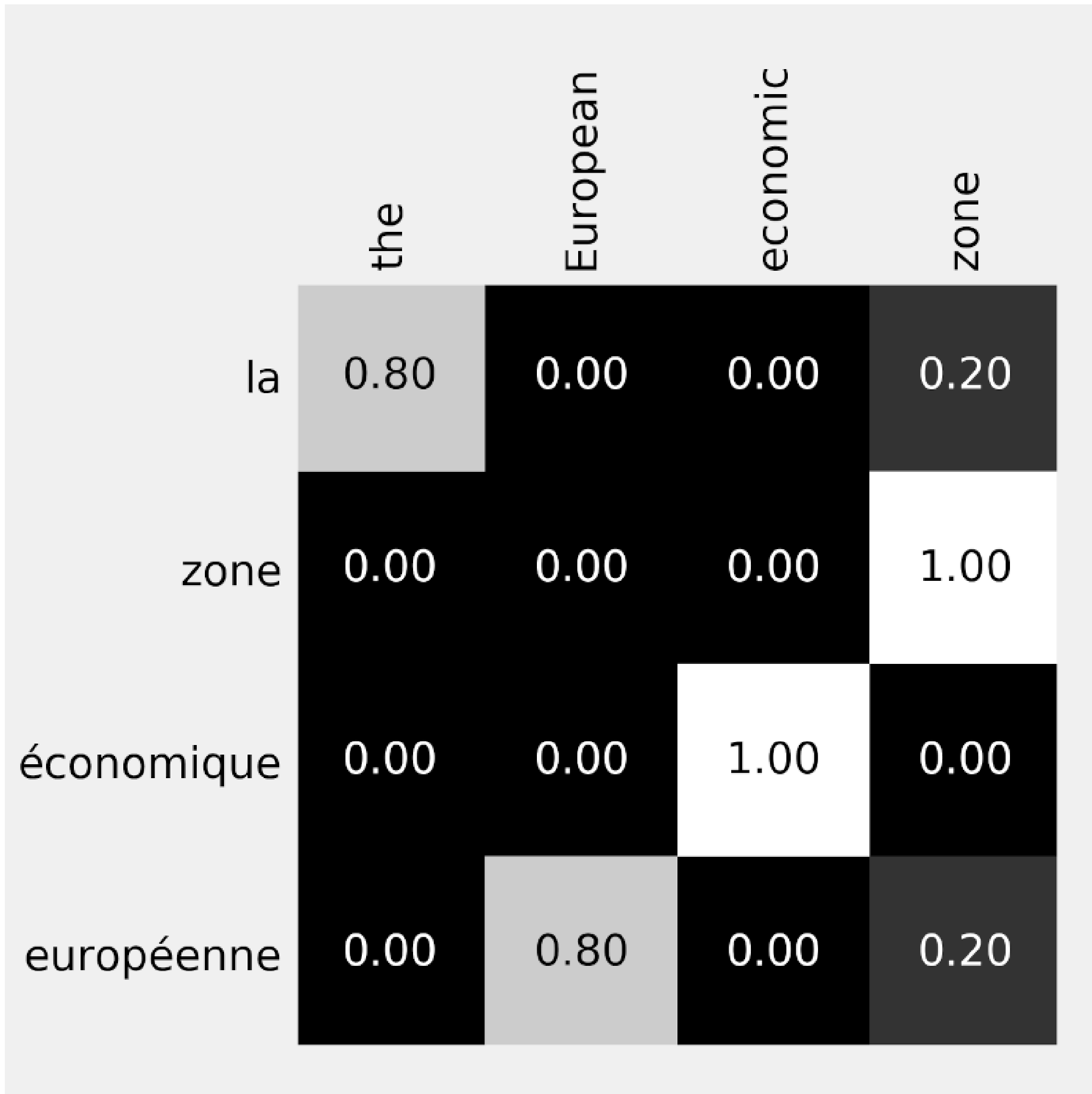
Attention is All You Need



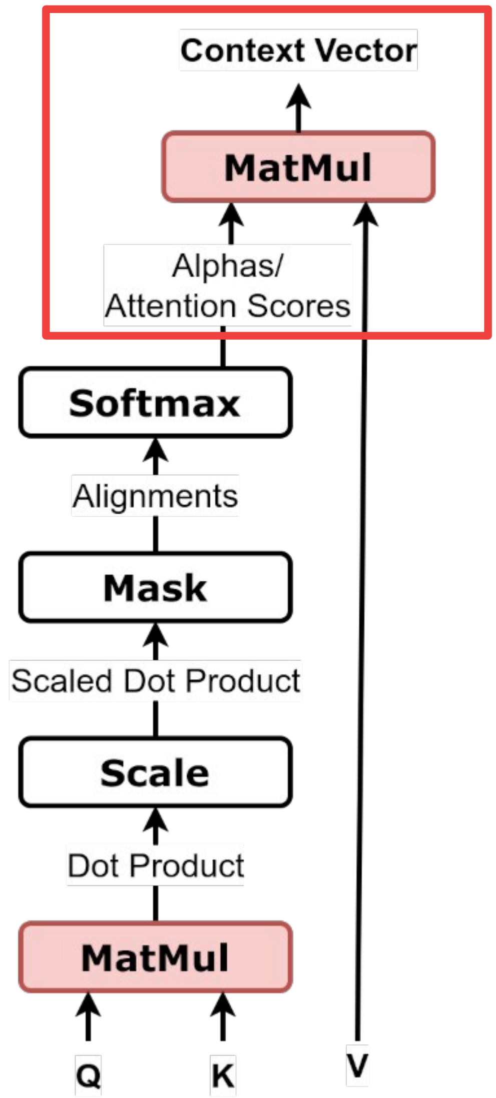
Attention is All You Need



Attention is All You Need

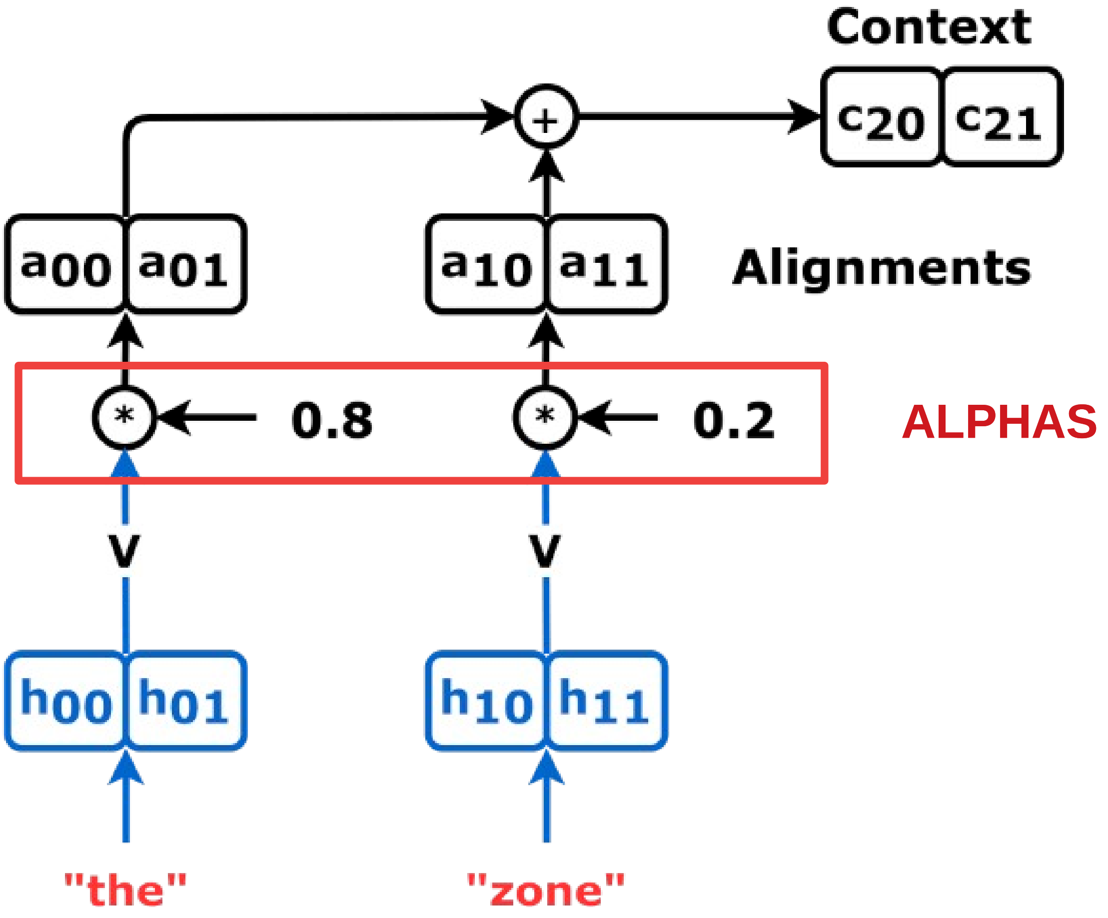


Attention is All You Need

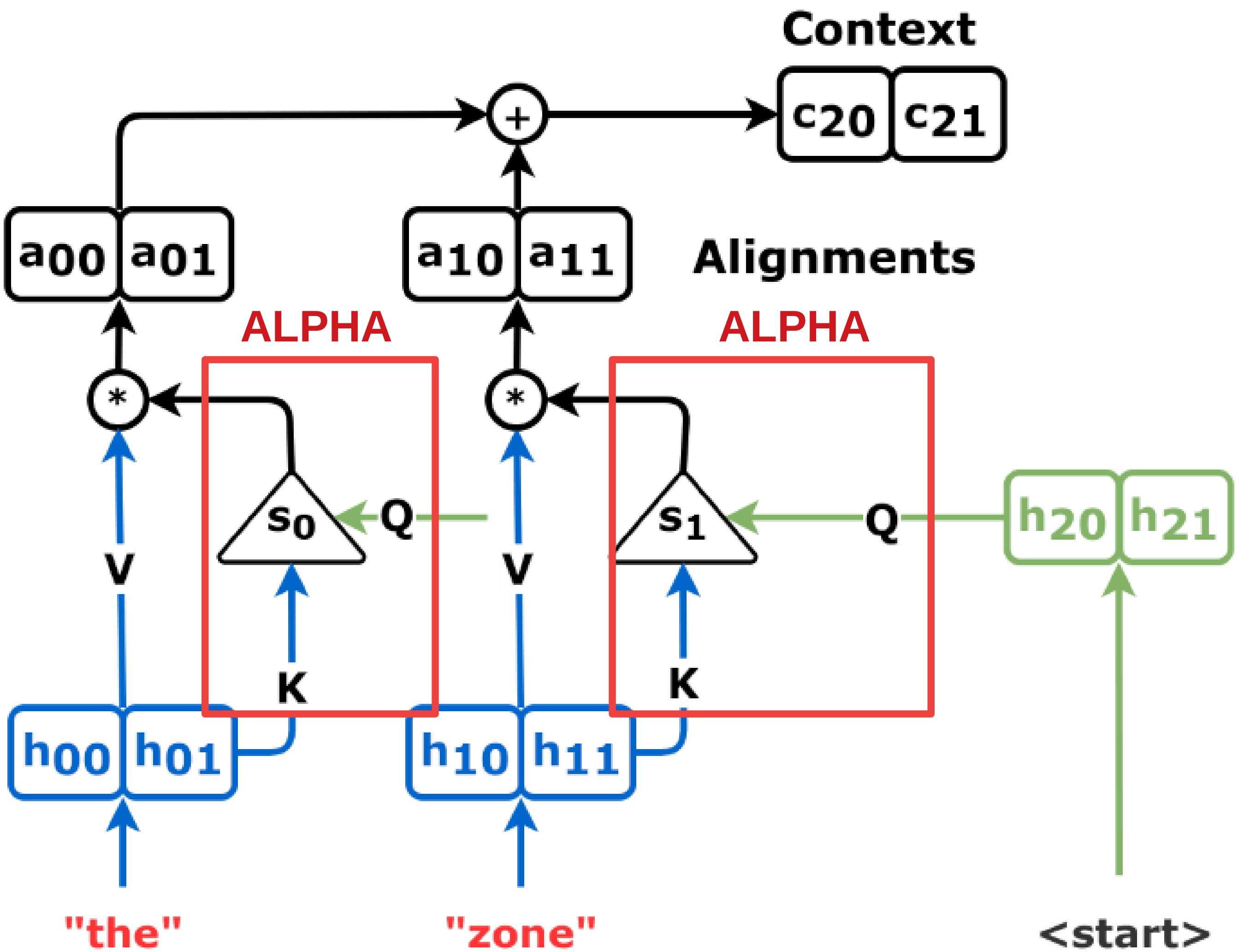


Scaled Dot-Product
Attention

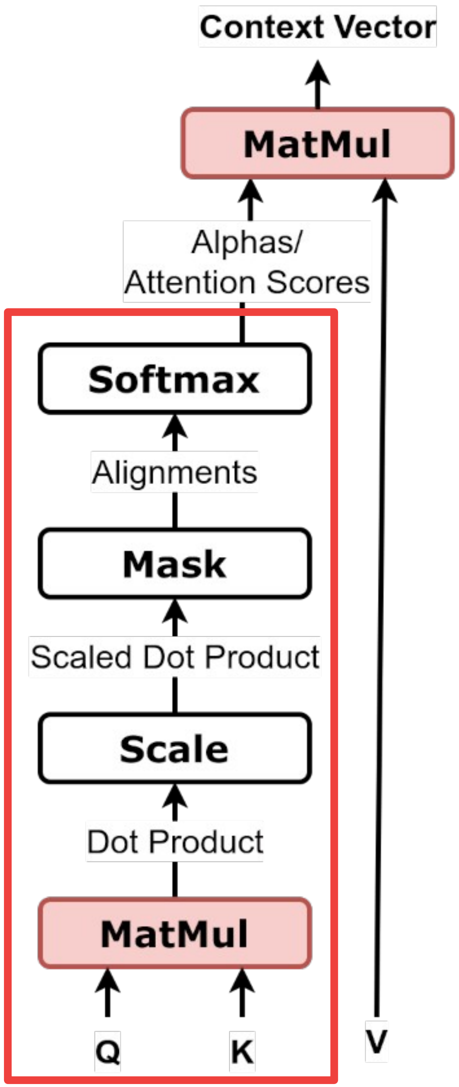
Attention is All You Need



Attention is All You Need



Attention is All You Need



Scaled Dot-Product
Attention

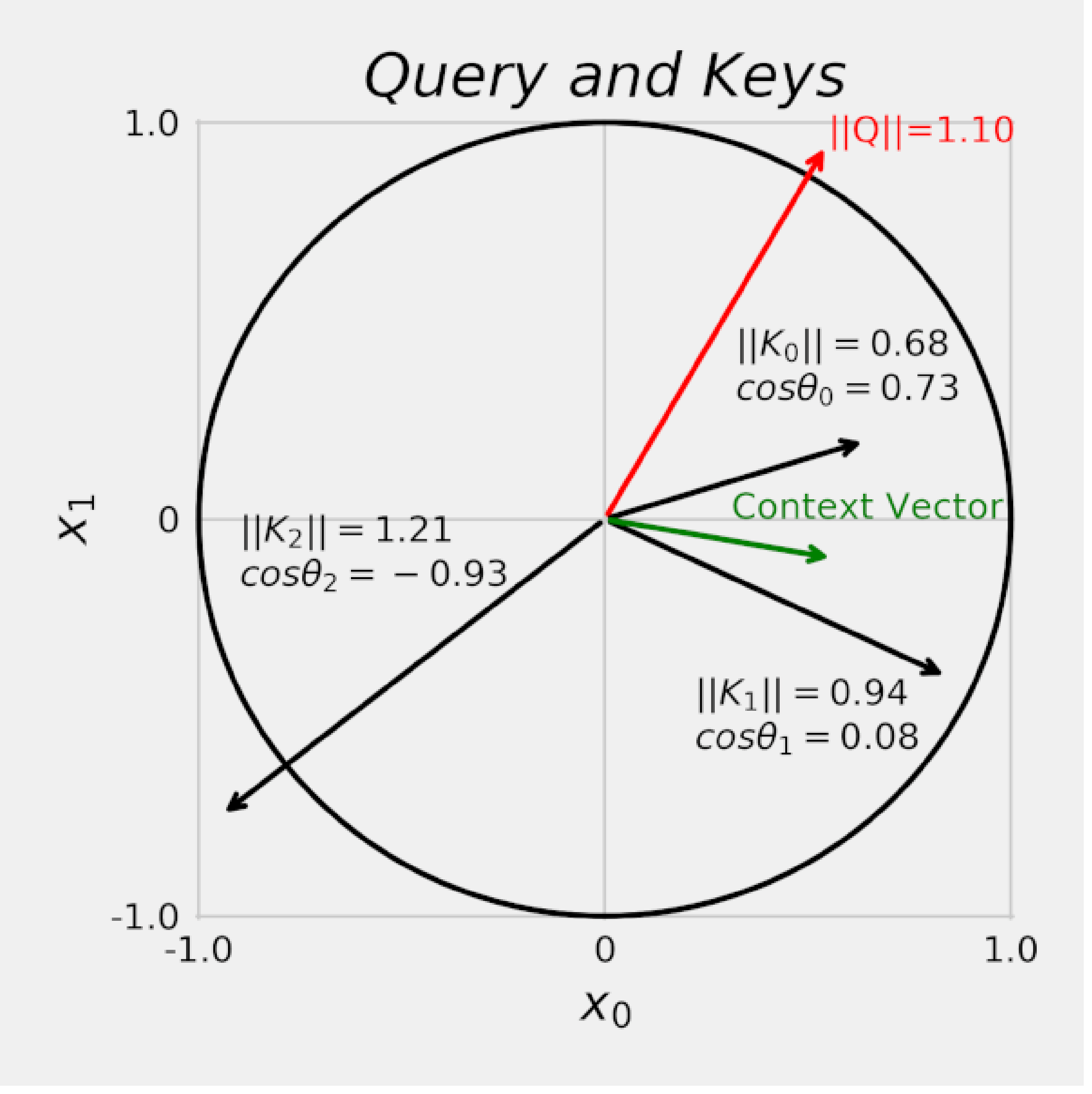
Attention is All You Need

$$\text{scaled dot product} = \frac{Q \cdot K}{\sqrt{d_k}}$$

```
n_dims = 10
vector1 = torch.randn(10000, 1, n_dims)
vector2 = torch.randn(10000, 1, n_dims).permute(0, 2, 1)
torch.bmm(vector1, vector2).squeeze().var()
```

$$\cos\,\theta\ ||Q||\ ||K|| = Q \cdot K$$

Attention is All You Need



Attention is All You Need

$$\alpha_{00}, \alpha_{01} = \mathrm{softmax}(\frac{Q_0 \cdot K_0}{\sqrt{2}}, \frac{Q_0 \cdot K_1}{\sqrt{2}})$$

$$\text{context vector}_0 = \alpha_{00} V_0 + \alpha_{01} V_1$$

$$\alpha_{10}, \alpha_{11} = \mathrm{softmax}(\frac{Q_1 \cdot K_0}{\sqrt{2}}, \frac{Q_1 \cdot K_1}{\sqrt{2}})$$

$$\text{context vector}_1 = \alpha_{10} V_0 + \alpha_{11} V_1$$

Attention is All You Need

```python
class Attention(nn.Module):
    def __init__(self, hidden_dim, input_dim=None,
                 proj_values=False):
        super().__init__()
        self.d_k = hidden_dim
        self.input_dim = hidden_dim if input_dim is None \
                         else input_dim
        self.proj_values = proj_values
        # Affine transformations for Q, K, and V
        self.linear_query = nn.Linear(self.input_dim, hidden_dim)
        self.linear_key = nn.Linear(self.input_dim, hidden_dim)
        self.linear_value = nn.Linear(self.input_dim, hidden_dim)
        self.alphas = None

    def init_keys(self, keys):
        self.keys = keys
        self.proj_keys = self.linear_key(self.keys)
        self.values = self.linear_value(self.keys) \
                      if self.proj_values else self.keys
```

PROJECTION LAYERS

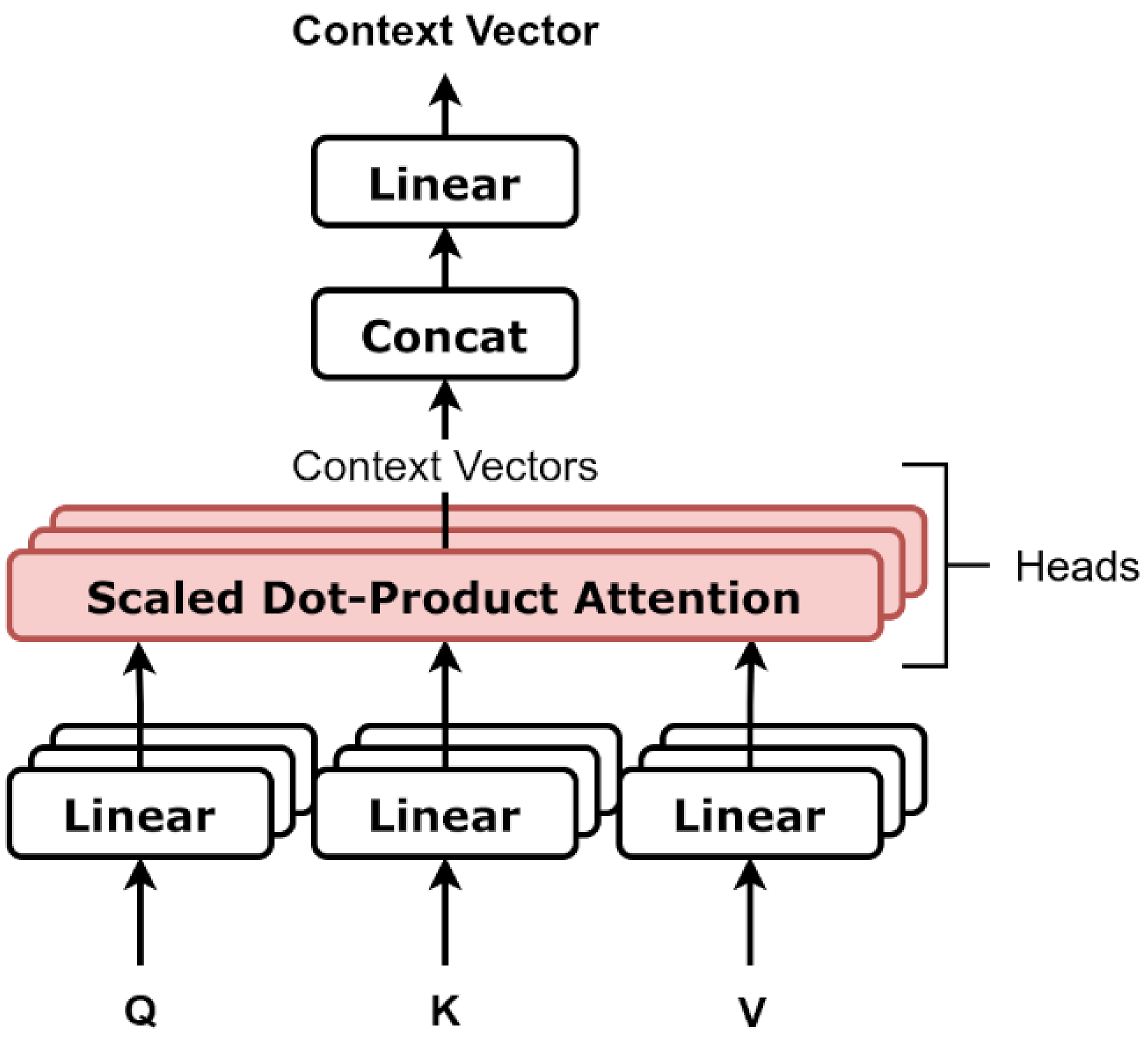
Projections of Ks and Vs or simply Ks and Vs

Attention is All You Need

```python

    def score_function(self, query):
        proj_query = self.linear_query(query)
        # scaled dot product
        # N, 1, H x N, H, L -> N, 1, L
        dot_products = torch.bmm(proj_query,
                                 self.proj_keys.permute(0, 2, 1))
        scores =  dot_products / np.sqrt(self.d_k)
        return scores

    def forward(self, query, mask=None):
        # Query is batch-first N, 1, H
        scores = self.score_function(query) # N, 1, L    ①
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        alphas = F.softmax(scores, dim=-1) # N, 1, L     ②
        self.alphas = alphas.detach()

        # N, 1, L x N, L, H -> N, 1, H
        context = torch.bmm(alphas, self.values)         ③
        return context
```

Projections of Qs

Multi-Head Attention (MHA)

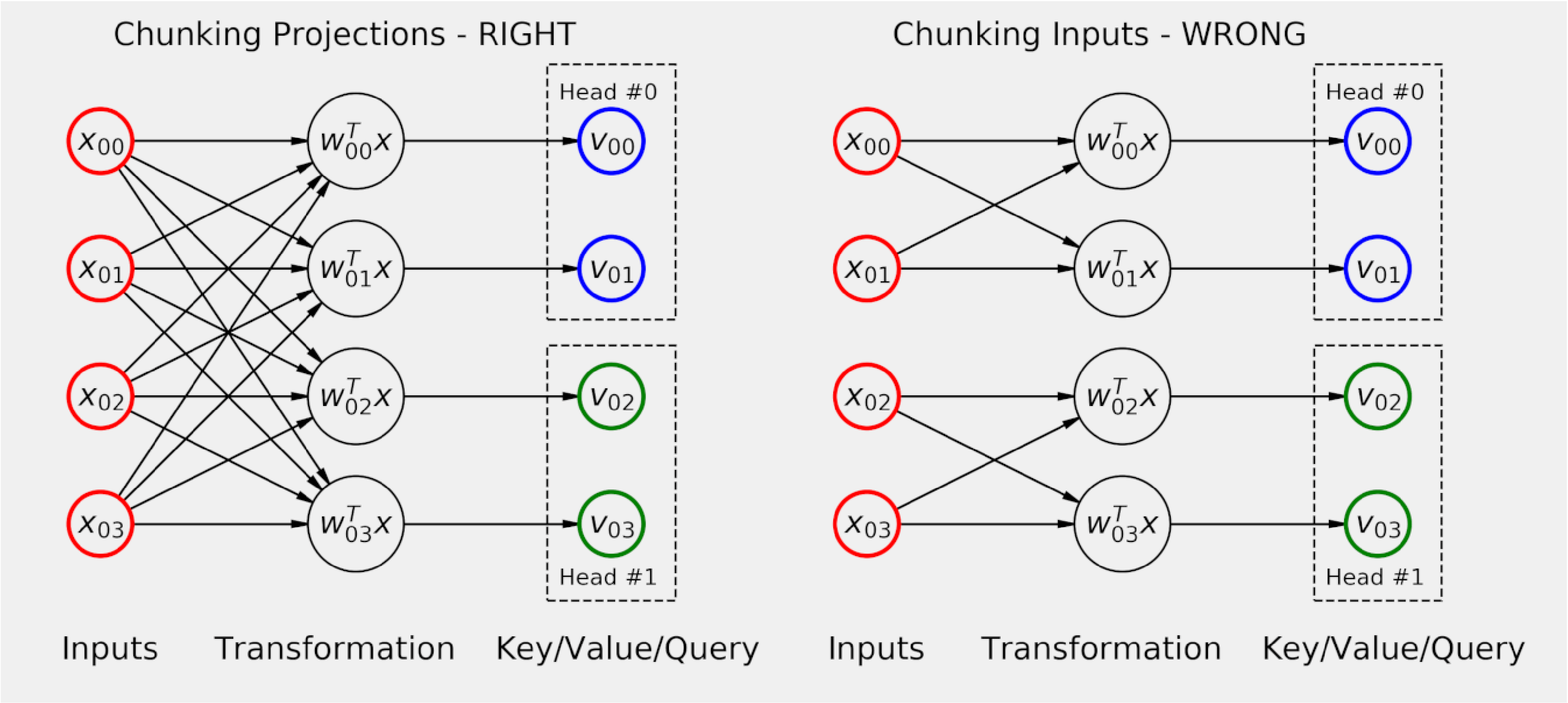


Multi-Head Attention (MHA)

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, n_heads, d_model,
                 input_dim=None, proj_values=True):
        super().__init__()
        self.linear_out = nn.Linear(n_heads * d_model, d_model)
        self.attn_heads = nn.ModuleList(
            [Attention(d_model,
                       input_dim=input_dim,
                       proj_values=proj_values)
             for _ in range(n_heads)]
        )

    def init_keys(self, key):
        for attn in self.attn_heads:
            attn.init_keys(key)

    @property
    def alphas(self):
        # Shape: n_heads, N, 1, L (source)
        return torch.stack(
            [attn.alphas for attn in self.attn_heads], dim=0
        )

    def output_function(self, contexts):
        # N, 1, n_heads * D
        concatenated = torch.cat(contexts, axis=-1)
        # Linear transf. to go back to original dimension
        out = self.linear_out(concatenated) # N, 1, D
        return out

    def forward(self, query, mask=None):
        contexts = [attn(query, mask=mask)
                    for attn in self.attn_heads]
        out = self.output_function(contexts)
        return out
```

Multiple heads!

Multi-Head Attention (MHA)



Multi-Head Attention (MHA)

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, n_heads, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.d_k = int(d_model / n_heads)                          ①
        self.linear_query = nn.Linear(d_model, d_model)
        self.linear_key = nn.Linear(d_model, d_model)
        self.linear_value = nn.Linear(d_model, d_model)
        self.linear_out = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(p=dropout)                       ④
        self.alphas = None

    def make_chunks(self, x):                                      ①
        batch_size, seq_len = x.size(0), x.size(1)
        # N, L, D -> N, L, n_heads * d_k
        x = x.view(batch_size, seq_len, self.n_heads, self.d_k)
        # N, n_heads, L, d_k
        x = x.transpose(1, 2)
        return x

    def init_keys(self, key):
        # N, n_heads, L, d_k
        self.proj_key = self.make_chunks(self.linear_key(key)) ①
        self.proj_value = \
            self.make_chunks(self.linear_value(key))               ①
```

PROJECTION LAYERS

CHUNKING

CHUNKING

Multi-Head Attention (MHA)

```

    def score_function(self, query):
        # Scaled dot product
        proj_query = self.make_chunks(self.linear_query(query))①
        # N, n_heads, L, d_k x N, n_heads, d_k, L ->
        # N, n_heads, L, L
        dot_products = torch.matmul(                              ②
            proj_query, self.proj_key.transpose(-2, -1)
        )
        scores =  dot_products / np.sqrt(self.d_k)
        return scores

    def attn(self, query, mask=None):                             ③
        # Query is batch-first: N, L, D
        # Score function will generate scores for each head
        scores = self.score_function(query) # N, n_heads, L, L
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        alphas = F.softmax(scores, dim=-1) # N, n_heads, L, L

        alphas = self.dropout(alphas)                             ④
        self.alphas = alphas.detach()

        # N, n_heads, L, L x N, n_heads, L, d_k ->
        # N, n_heads, L, d_k
        context = torch.matmul(alphas, self.proj_value)           ②
        return context

```

CHUNKING

Multi-Head Attention (MHA)

```python
    def output_function(self, contexts):
        # N, L, D
        out = self.linear_out(contexts) # N, L, D
        return out

    def forward(self, query, mask=None):
        if mask is not None:
            # N, 1, L, L - every head uses the same mask
            mask = mask.unsqueeze(1)

        # N, n_heads, L, d_k
        context = self.attn(query, mask=mask)
        # N, L, n_heads, d_k
        context = context.transpose(1, 2).contiguous()          ⑤
        # N, L, n_heads * d_k = N, L, d_model
        context = context.view(query.size(0), -1, self.d_model)⑤
        # N, L, d_model
        out = self.output_function(context)
        return out
```

CONCAT
CHUNKS

Multi-Head Attention (MHA)

HEAD #0

FOUR PROJECTIONS FOR Q, K, V

q							
0.23	0.03	0.1	0.3	0.87	0.84	0.3	0.3
0.27	0.61	0.7	0.02	0.83	0.94	0.12	0.21
0.79	0.23	0.03	0.28	0.02	0.47	0.97	0.61
0.11	0.1	0.3	1.0	0.08	0.88	0.83	0.69
0.07	0.01	0.16	0.05	0.51	0.54	0.23	0.47

→

q/0		q/1		q/2		q/3	
0.23	0.03	0.1	0.3	0.87	0.84	0.3	0.3
0.27	0.61	0.7	0.02	0.83	0.94	0.12	0.21
0.79	0.23	0.03	0.28	0.02	0.47	0.97	0.61
0.11	0.1	0.3	1.0	0.08	0.88	0.83	0.69
0.07	0.01	0.16	0.05	0.51	0.54	0.23	0.47

k							
0.08	0.41	0.36	0.1	0.15	0.03	0.95	0.16
0.7	0.77	0.57	0.9	0.65	0.36	0.58	0.32
0.77	0.29	0.42	0.58	0.16	0.49	0.17	0.73
0.94	0.36	0.16	0.03	0.31	0.67	0.81	0.94
0.76	1.0	0.45	0.94	0.6	0.49	0.68	0.54

→

k/0		k/1		k/2		k/3	
0.08	0.41	0.36	0.1	0.15	0.03	0.95	0.16
0.7	0.77	0.57	0.9	0.65	0.36	0.58	0.32
0.77	0.29	0.42	0.58	0.16	0.49	0.17	0.73
0.94	0.36	0.16	0.03	0.31	0.67	0.81	0.94
0.76	1.0	0.45	0.94	0.6	0.49	0.68	0.54

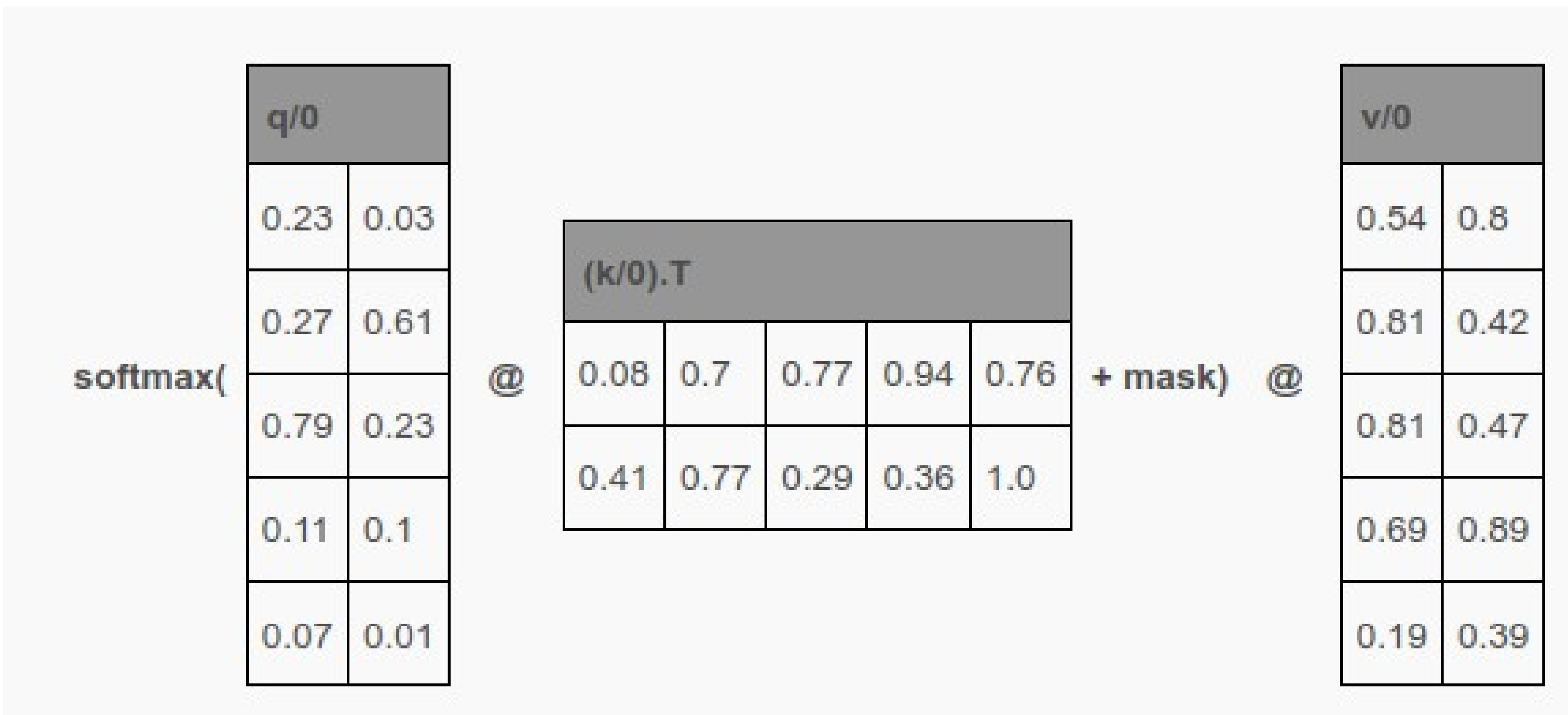
v							
0.54	0.8	0.73	0.35	0.97	0.05	0.07	0.45
0.81	0.42	0.13	0.33	0.6	0.75	0.41	0.36
0.81	0.47	0.2	0.05	0.63	0.75	0.58	0.66
0.69	0.89	0.09	0.49	0.49	0.63	0.91	0.88
0.19	0.39	0.22	0.36	1.0	0.17	0.66	0.02

→

v/0		v/1		v/2		v/3	
0.54	0.8	0.73	0.35	0.97	0.05	0.07	0.45
0.81	0.42	0.13	0.33	0.6	0.75	0.41	0.36
0.81	0.47	0.2	0.05	0.63	0.75	0.58	0.66
0.69	0.89	0.09	0.49	0.49	0.63	0.91	0.88
0.19	0.39	0.22	0.36	1.0	0.17	0.66	0.02

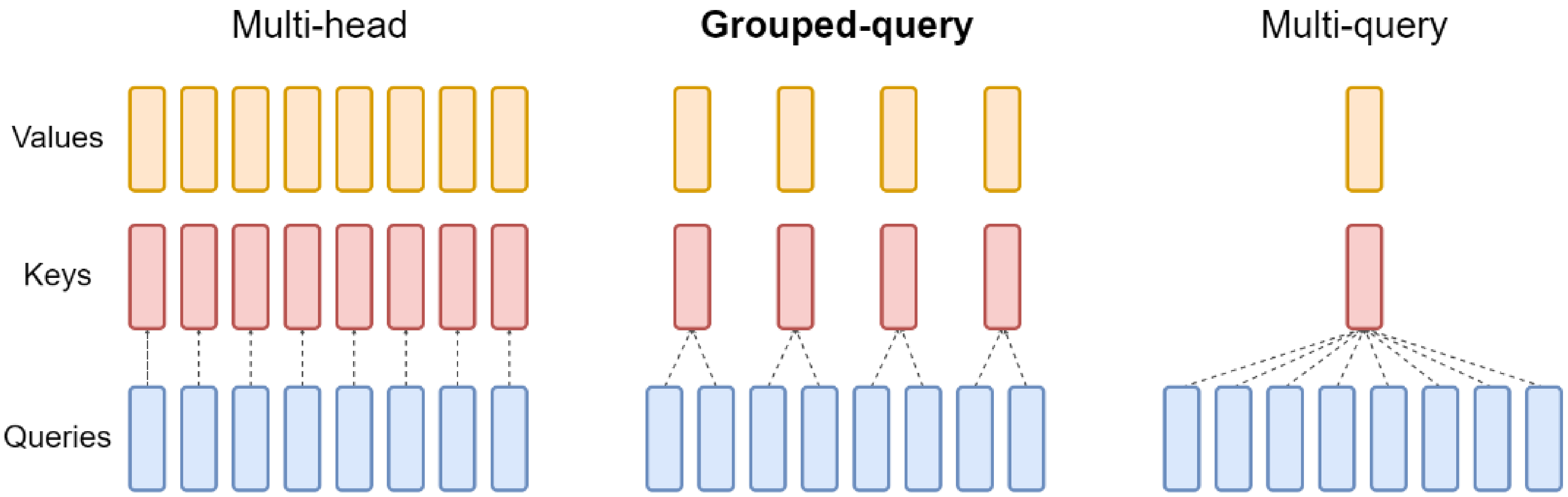
Source: https://vgel.me/posts/faster-inference

Multi-Head Attention (MHA)



Source: https://vgel.me/posts/faster-inference

MHA, GQA, MQA



Multi-Query Attention(MQA)

FOUR PROJECTIONS FOR Q

q							
0.23	0.03	0.1	0.3	0.87	0.84	0.3	0.3
0.27	0.61	0.7	0.02	0.83	0.94	0.12	0.21
0.79	0.23	0.03	0.28	0.02	0.47	0.97	0.61
0.11	0.1	0.3	1.0	0.08	0.88	0.83	0.69
0.07	0.01	0.16	0.05	0.51	0.54	0.23	0.47

→

q/0	
0.23	0.03
0.27	0.61
0.79	0.23
0.11	0.1
0.07	0.01

q/1	
0.1	0.3
0.7	0.02
0.03	0.28
0.3	1.0
0.16	0.05

q/2	
0.87	0.84
0.83	0.94
0.02	0.47
0.08	0.88
0.51	0.54

q/3	
0.3	0.3
0.12	0.21
0.97	0.61
0.83	0.69
0.23	0.47

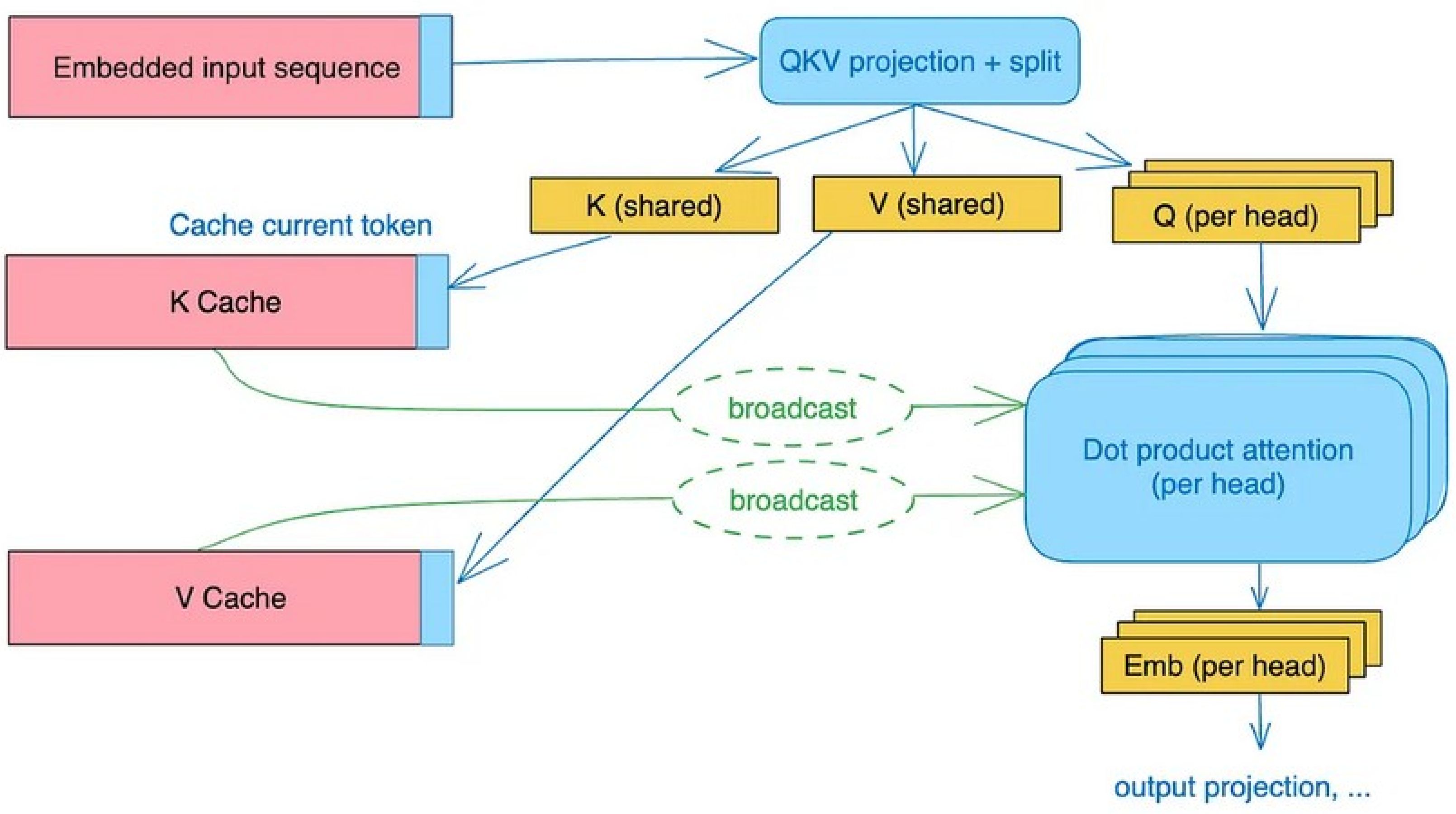
SINGLE PROJECTIONS FOR K AND V (SHARED WITH EACH PROJECTION OF Q)

k	
0.08	0.41
0.7	0.77
0.77	0.29
0.94	0.36
0.76	1.0

v	
0.54	0.8
0.81	0.42
0.81	0.47
0.69	0.89
0.19	0.39

Source: https://vgel.me/posts/faster-inference

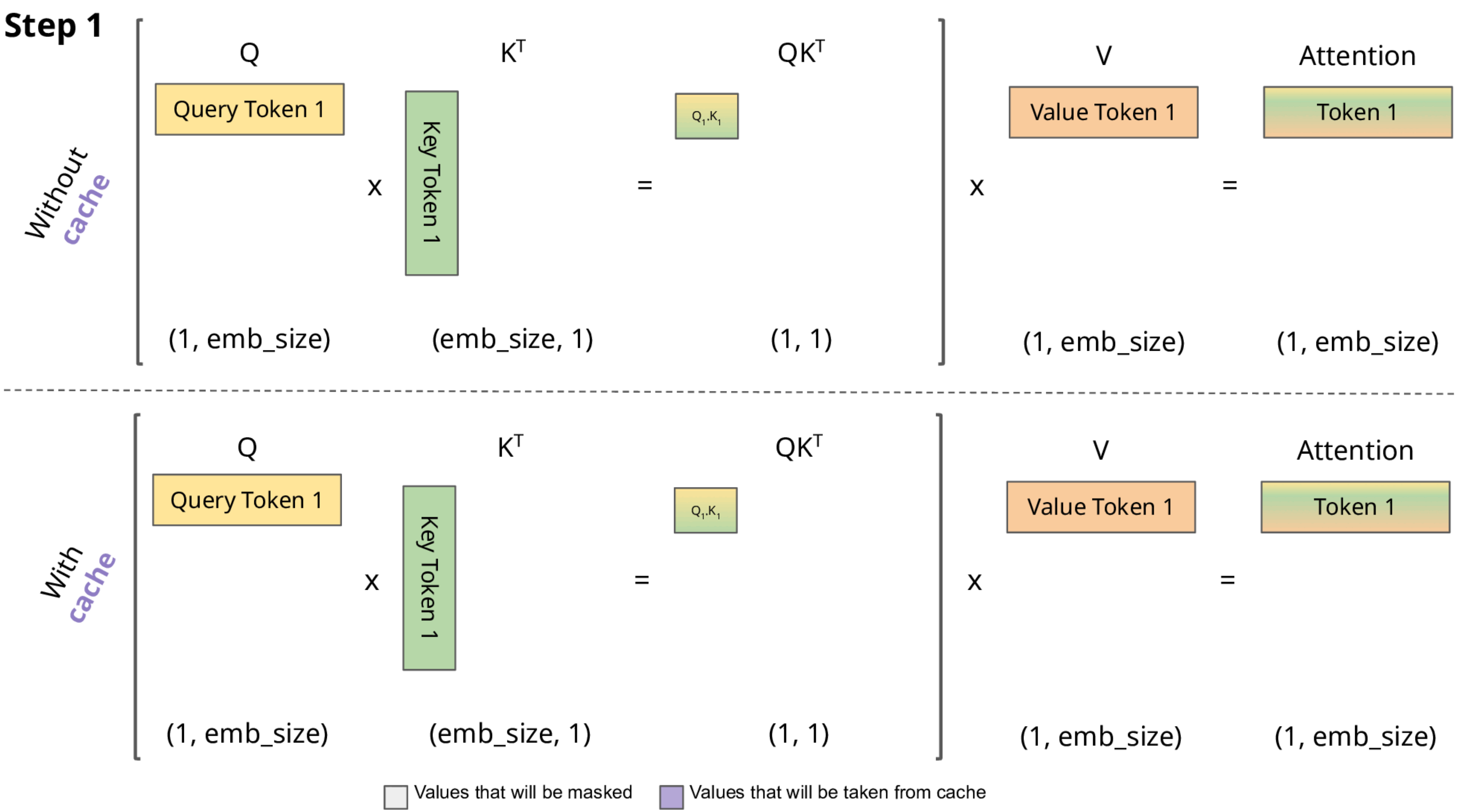
Multi-Query Attention(MQA)



Multi-Query Attention

Source: https://blog.fireworks.ai/multi-query-attention-is-all-you-need-db072e758055

KV Caching



Source: https://medium.com/@joaolages/kv-caching-explained-276520203249

Hands-On

Notebook 1 - MHA vs MQA

Sparse Attention

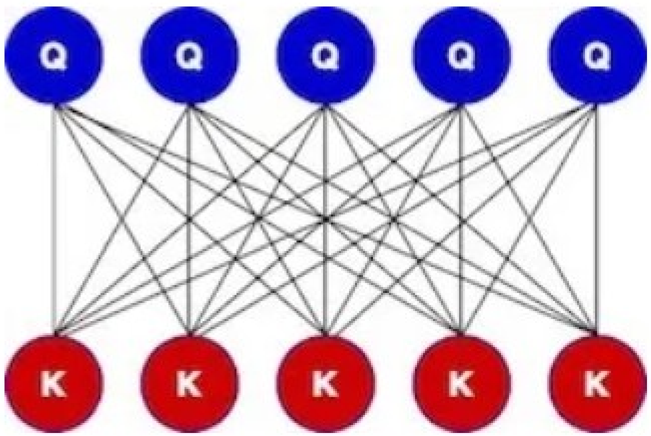


Figure 1. Attention Layer Representation

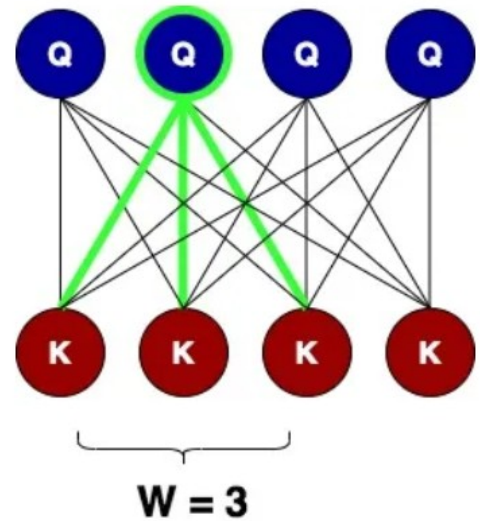


Figure 4. Sliding Attention Window of Size 3.

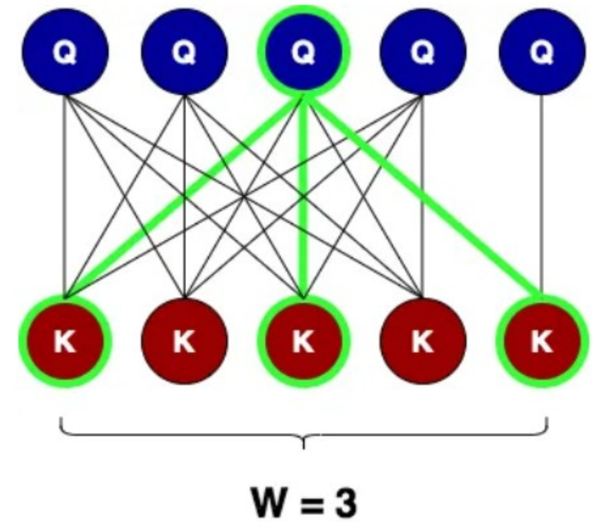
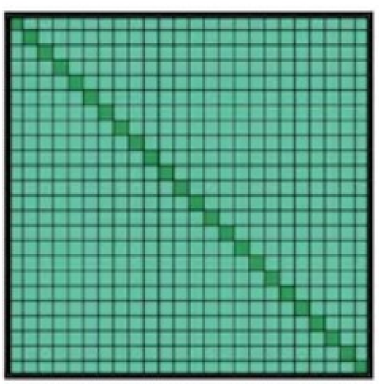
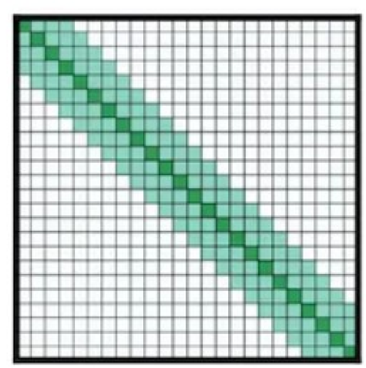
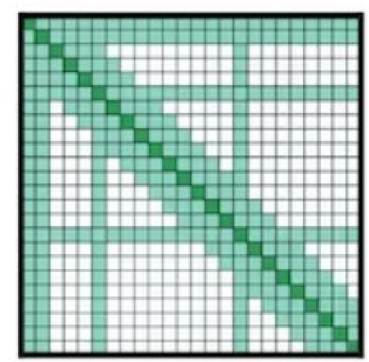


Figure 7. Dilated Sliding Window Attention Layer



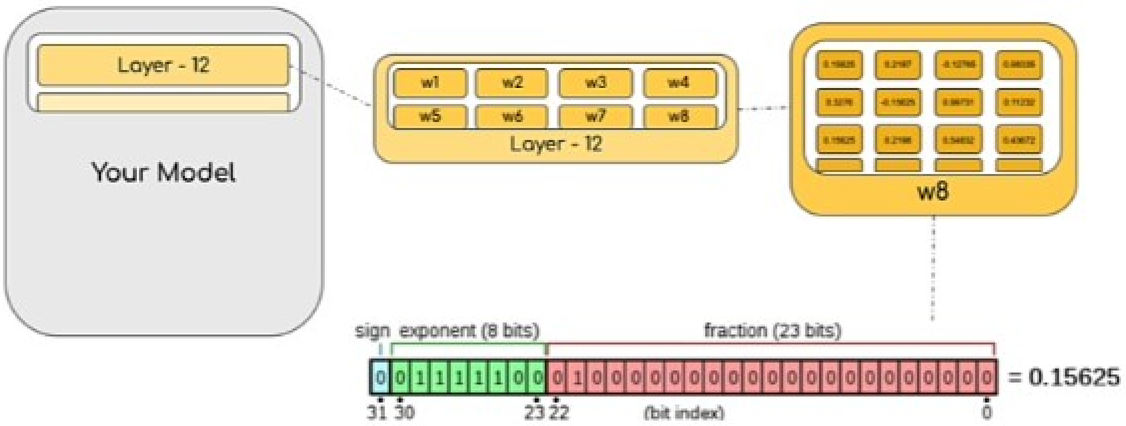




Source: https://ahelhady.medium.com/understanding-longformers-sliding-window-attention-mechanism-f5d61048a907

Quantization

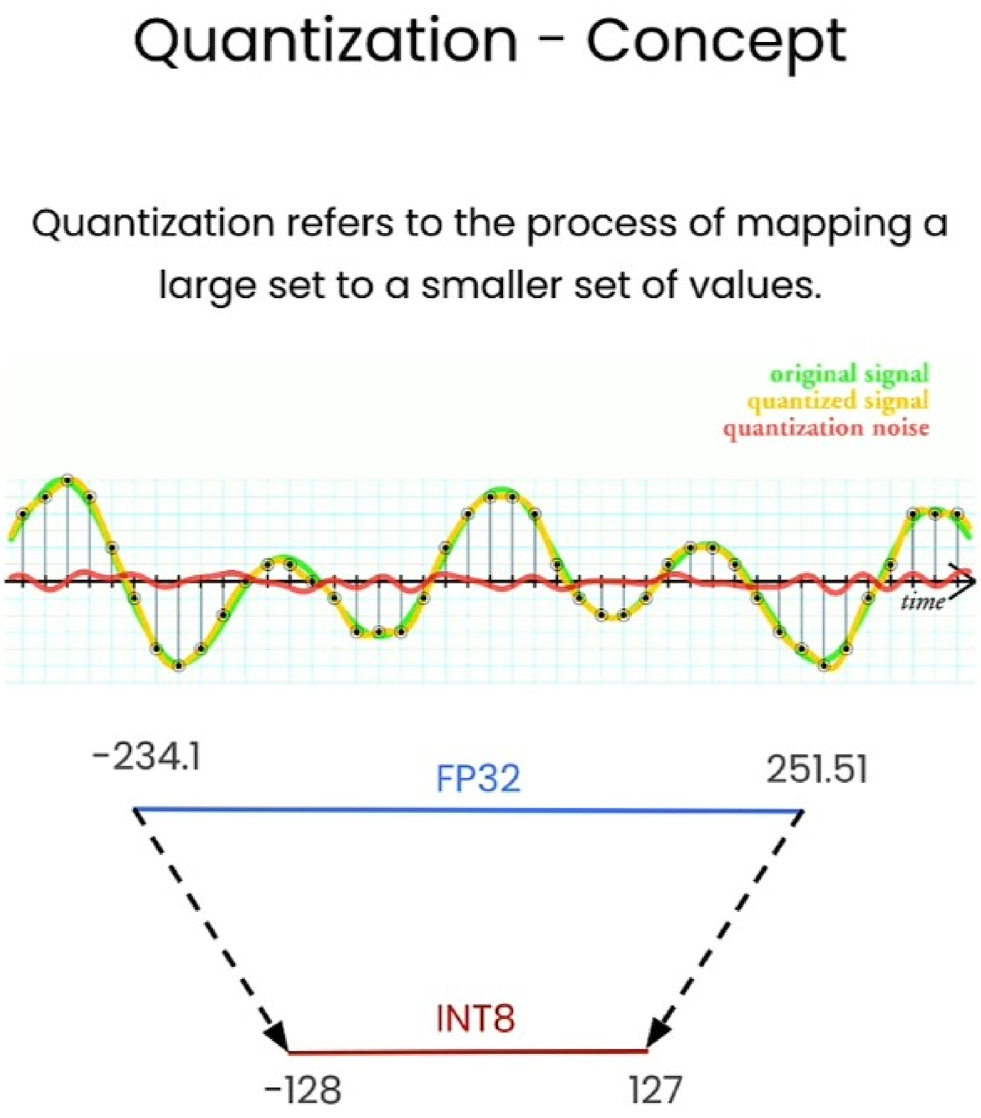
What is a model parameter?



It is possible to inspect each parameter's data type!

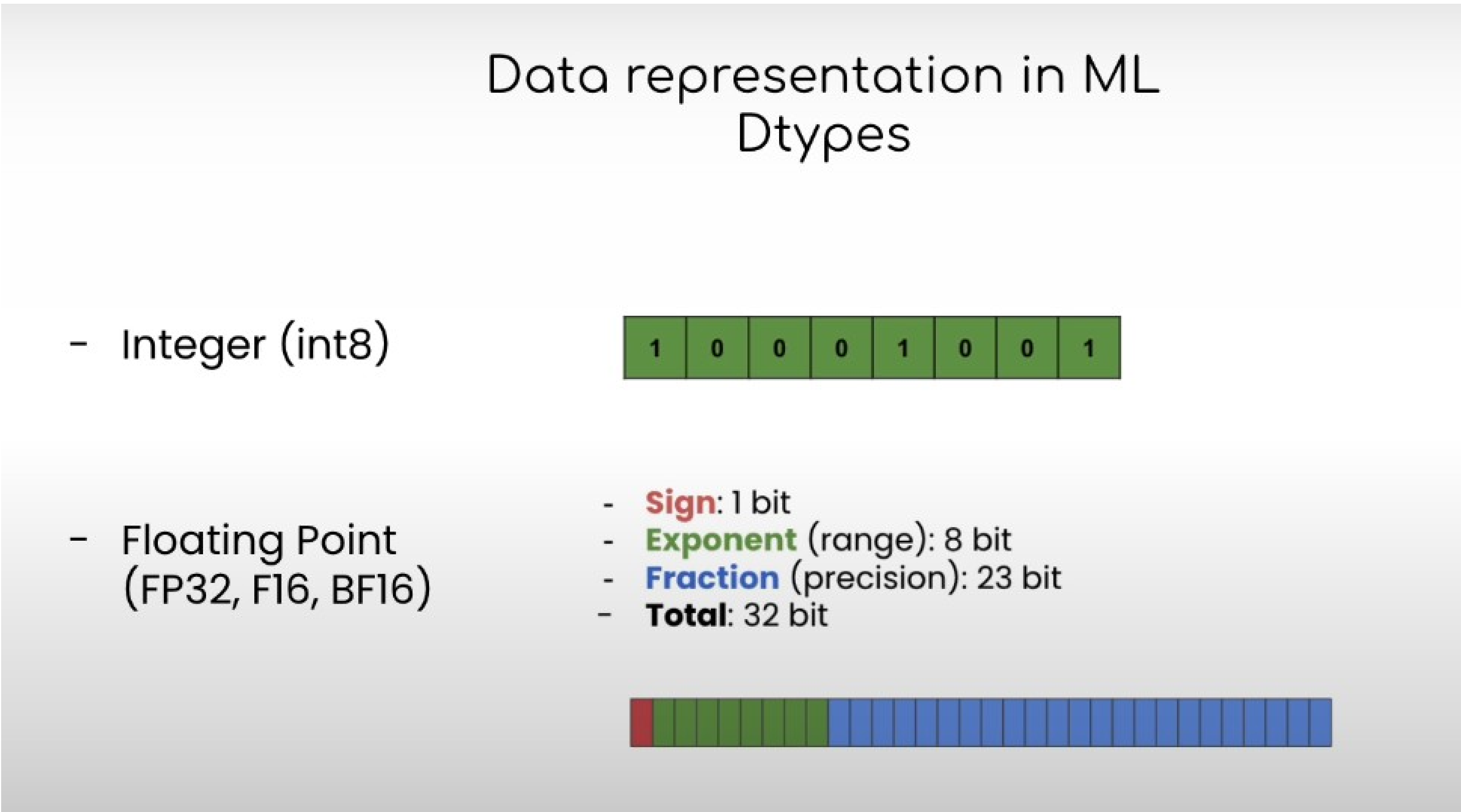
Source: https://www.deeplearning.ai/short-courses/quantization-fundamentals-with-hugging-face/

Quantization



Source: https://www.deeplearning.ai/short-courses/quantization-fundamentals-with-hugging-face/

Quantization



Source: https://www.deeplearning.ai/short-courses/quantization-fundamentals-with-hugging-face/

Quantization

Integer

- Unsigned integer:
  - Range for n-bits: $[0, 2^n - 1]$

Example with 8-bit (torch.uint8): $[0, 255]$

1	0	0	0	1	0	0	1

$$\underbrace{2^7}_{128} + 0 + 0 + 0 + \underbrace{2^3}_{8} + 0 + 0 + \underbrace{2^0}_{1} = 137$$

- Signed integer two's complement representation
  - Range for n-bits: $[-2^{n-1}, 2^{n-1} - 1]$

Example with 8-bit (torch.int8): $[-128, 127]$

1	0	0	0	1	0	0	1

$$\underbrace{-2^7}_{-128} + 0 + 0 + 0 + \underbrace{2^3}_{8} + 0 + 0 + \underbrace{2^0}_{1} = -119$$

Integer - PyTorch - Ranges

- Unsigned integer : $[0, 2^n - 1]$

```
# torch.uint8
torch.iinfo(torch.uint8)
```

two "i"s 👀

```
iinfo(min=0, max=255, dtype=uint8)
```

Source: https://www.deeplearning.ai/short-courses/quantization-fundamentals-with-hugging-face/

Quantization

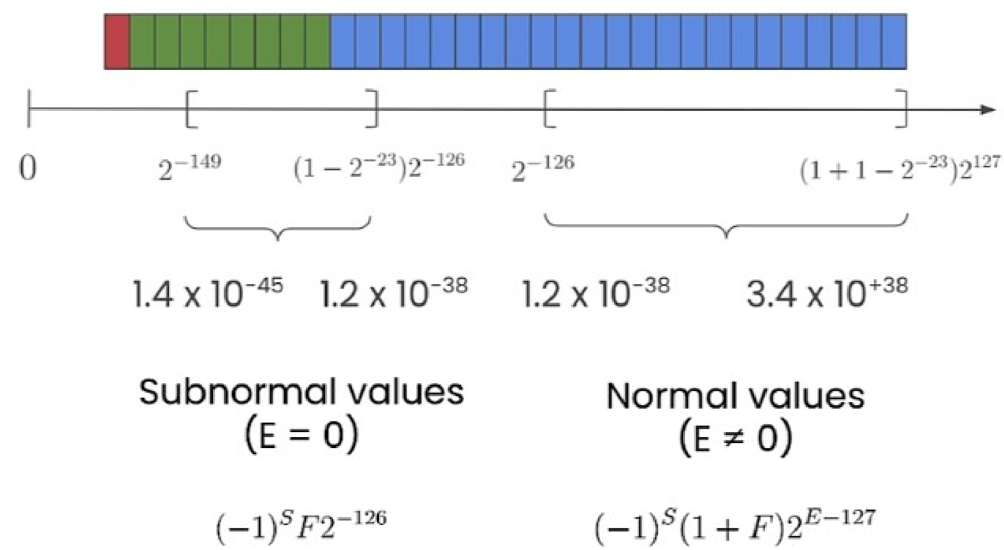
Floating Point

3 components in floating point:
- **Sign**: positive/negative (always 1 bit)
- **Exponent** (range): impact the representable range of the number
- **Fraction** (precision): impact on the precision of the number

FP32, BF16, FP16, FP8 are floating point format with a specific number of bits for **exponent** and the **fraction.**

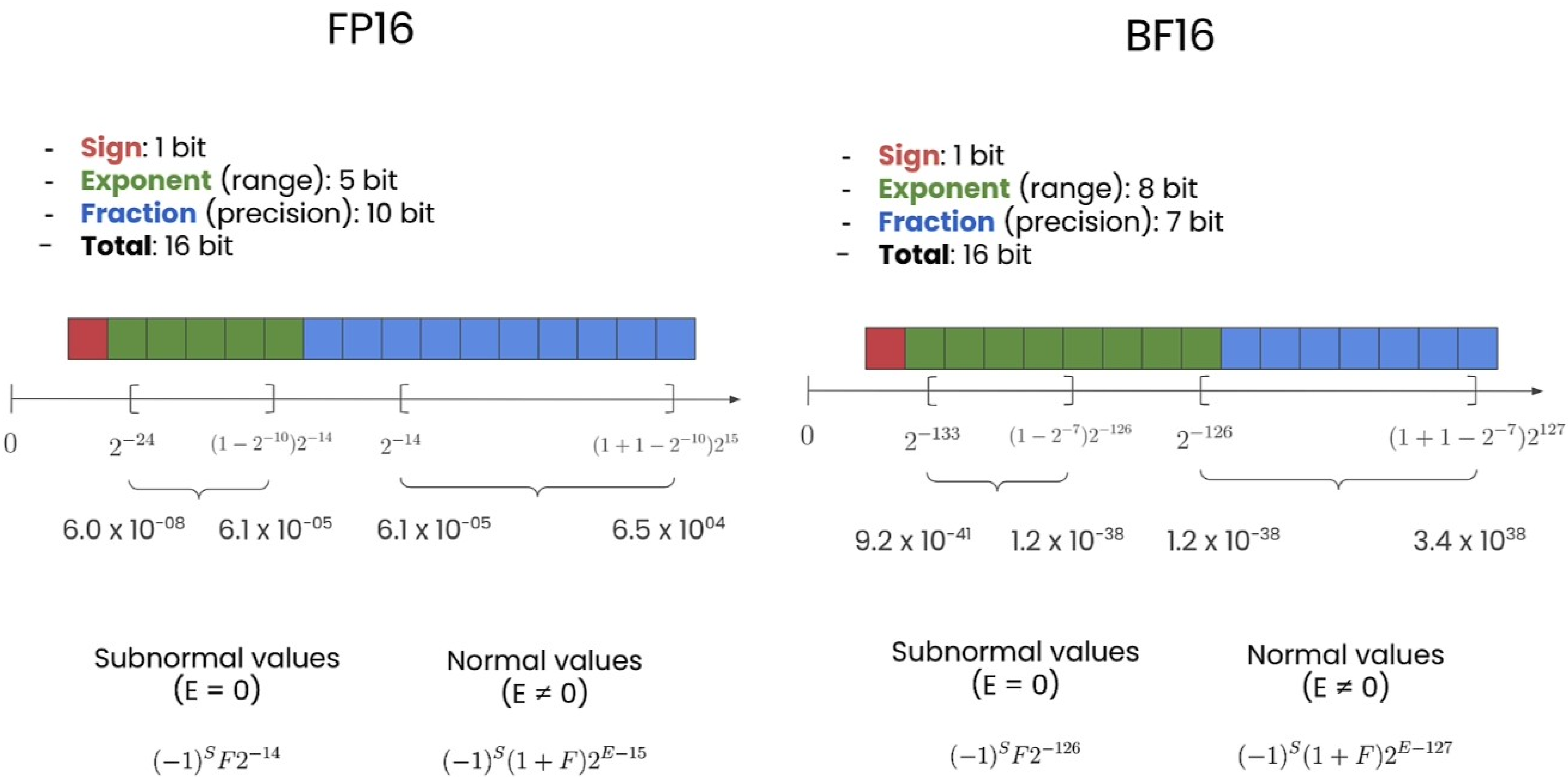
FP32

- **Sign**: 1 bit
- **Exponent** (range): 8 bit
- **Fraction** (precision): 23 bit
- **Total**: 32 bit



Source: https://www.deeplearning.ai/short-courses/quantization-fundamentals-with-hugging-face/

Quantization



Source: https://www.deeplearning.ai/short-courses/quantization-fundamentals-with-hugging-face/

Quantization

Comparison

Data Type	Precision	maximum
FP32	best	~10^{+38}
FP16	better	~10^{04}
BF16	good	~10^{38} 🤗

Floating Point - PyTorch Downcasting

Advantages:

- **Reduced memory footprint.**
  - More efficient use of GPU memory.
  - Enables the training of larger models
  - Enables larger batch sizes
- **Increased compute and speed**
  - Computation using low precision (fp16, bf16) can be faster than fp32 since it require less memory.
    - Depends on the hardware (e.g. Google TPU, NVIDIA A100)

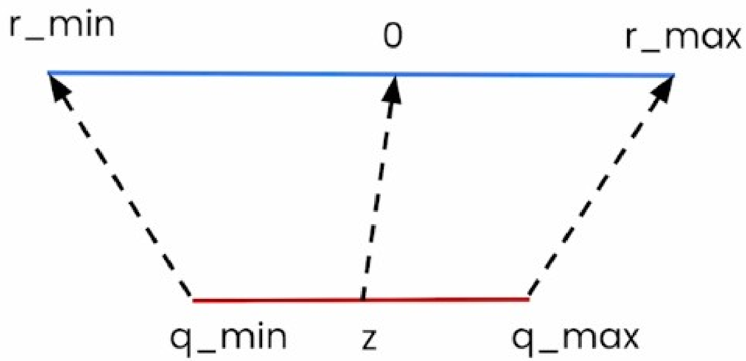
Disadvantages:

- **Less precise** : We are using less memory, hence the computation is less precise.

Source: https://www.deeplearning.ai/short-courses/quantization-fundamentals-with-hugging-face/

Quantization

(Optional) Linear Quantization



Simple idea: **linear mapping**

Formula: $r = s(q - z)$

original value (e.g. in FP32)

Scale (e.g. in FP32)

quantized value (e.g. in INT8)

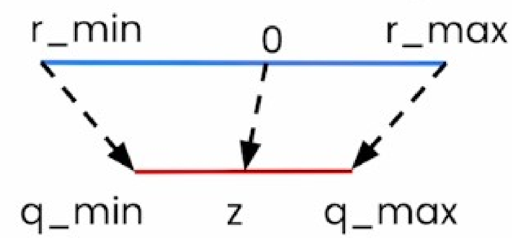
Zero point (e.g. INT8)

(Optional) scale and zero point

Linear quantization maps the floating point range $[r_{\min}, r_{\max}]$ to the quantized range $[q_{\min}, q_{\max}]$

If we look that the **extreme values**, we should get:

$$\begin{cases} r_{\min} = s(q_{\min} - z) \\ r_{\max} = s(q_{\max} - z) \end{cases}$$



If we subtract the first equation from the second one, we get the **scale s** :

$$s = (r_{\max} - r_{\min}) / (q_{\max} - q_{\min})$$

For the **zero point z**, we need to round the value since it is a n-bit integer:

$$z = \mathrm{int}(\mathrm{round}(q_{\min} - r_{\min} / s))$$

Source: https://www.deeplearning.ai/short-courses/quantization-fundamentals-with-hugging-face/

Hands-On

Notebook 2 - Quantization

BitsAndBytes

`bitsandbytes`

downloads 18M | downloads/month 2M | downloads/week 240k

The `bitsandbytes` library is a lightweight Python wrapper around CUDA custom functions, in particular 8-bit optimizers, matrix multiplication (LLM.int8()), and 8 & 4-bit quantization functions.

The library includes quantization primitives for 8-bit & 4-bit operations, through `bitsandbytes.nn.Linear8bitLt` and `bitsandbytes.nn.Linear4bit` and 8-bit optimizers through `bitsandbytes.optim` module.

There are ongoing efforts to support further hardware backends, i.e. Intel CPU + GPU, AMD GPU, Apple Silicon. Windows support is quite far along and is on its way as well.

class transformers.BitsAndBytesConfig <source>

```
( load_in_8bit = False, load_in_4bit = False, llm_int8_threshold = 6.0, llm_int8_skip_modules = None,
llm_int8_enable_fp32_cpu_offload = False, llm_int8_has_fp16_weight = False, bnb_4bit_compute_dtype =
None, bnb_4bit_quant_type = 'fp4', bnb_4bit_use_double_quant = False, bnb_4bit_quant_storage = None,
**kwargs )
```

Sources: https://github.com/TimDettmers/bitsandbytes
https://huggingface.co/docs/transformers/en/main_classes/quantization

BitsAndBytes

Quickstart

The basic way to load a model in 4bit is to pass the argument `load_in_4bit=True` when calling the `from_pretrained` method by providing a device map (pass `"auto"` to get a device map that will be automatically inferred).

```
from transformers import AutoModelForCausalLM

model = AutoModelForCausalLM.from_pretrained("facebook/opt-350m", load_in_4bit=True, device_map=
...
```

That's all you need!

As a general rule, we recommend users to not manually set a device once the model has been loaded with `device_map`. So any device assignment call to the model, or to any model's submodules should be avoided after that line - unless you know what you are doing.

Keep in mind that loading a quantized model will automatically cast other model's submodules into `float16` dtype. You can change this behavior, (if for example you want to have the layer norms in `float32`), by passing `torch_dtype=dtype` to the `from_pretrained` method.

```
import torch
from transformers import BitsAndBytesConfig

quantization_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.bfloat16
)
```


Sources: https://huggingface.co/blog/4bit-transformers-bitsandbytes

BitsAndBytes

Advanced usage

You can play with different variants of 4bit quantization such as NF4 (normalized float 4 (default)) or pure FP4 quantization. Based on theoretical considerations and empirical results from the paper, we recommend using NF4 quantization for better performance.

Other options include `bnb_4bit_use_double_quant` which uses a second quantization after the first one to save an additional 0.4 bits per parameter. And finally, the compute type. While 4-bit bitsandbytes stores weights in 4-bits, the computation still happens in 16 or 32-bit and here any combination can be chosen (float16, bfloat16, float32 etc).

The matrix multiplication and training will be faster if one uses a 16-bit compute dtype (default torch.float32). One should leverage the recent `BitsAndBytesConfig` from transformers to change these parameters. An example to load a model in 4bit using NF4 quantization below with double quantization with the compute dtype bfloat16 for faster training:

```
from transformers import BitsAndBytesConfig


nf4_config = BitsAndBytesConfig(
   load_in_4bit=True,
   bnb_4bit_quant_type="nf4",
   bnb_4bit_use_double_quant=True,
   bnb_4bit_compute_dtype=torch.bfloat16
)
```


Sources: https://huggingface.co/blog/4bit-transformers-bitsandbytes

BitsAndBytes

> bfloat16 is the ideal `compute_dtype` if your hardware supports it. While the default `compute_dtype`, float32, ensures backward compatibility (due to wide-ranging hardware support) and numerical stability, it is large and slows down computations. In contrast, float16 is smaller and faster but can lead to numerical instabilities. bfloat16 combines the best aspects of both; it offers the numerical stability of float32 and the reduced memory footprint and speed of a 16-bit data type. Check if your hardware supports bfloat16 and configure it using the `bnb_4bit_compute_dtype` parameter in BitsAndBytesConfig!

Sources: https://huggingface.co/docs/bitsandbytes/main/en/integrations

Hands-On

Notebook 3 - BitsAndBytes

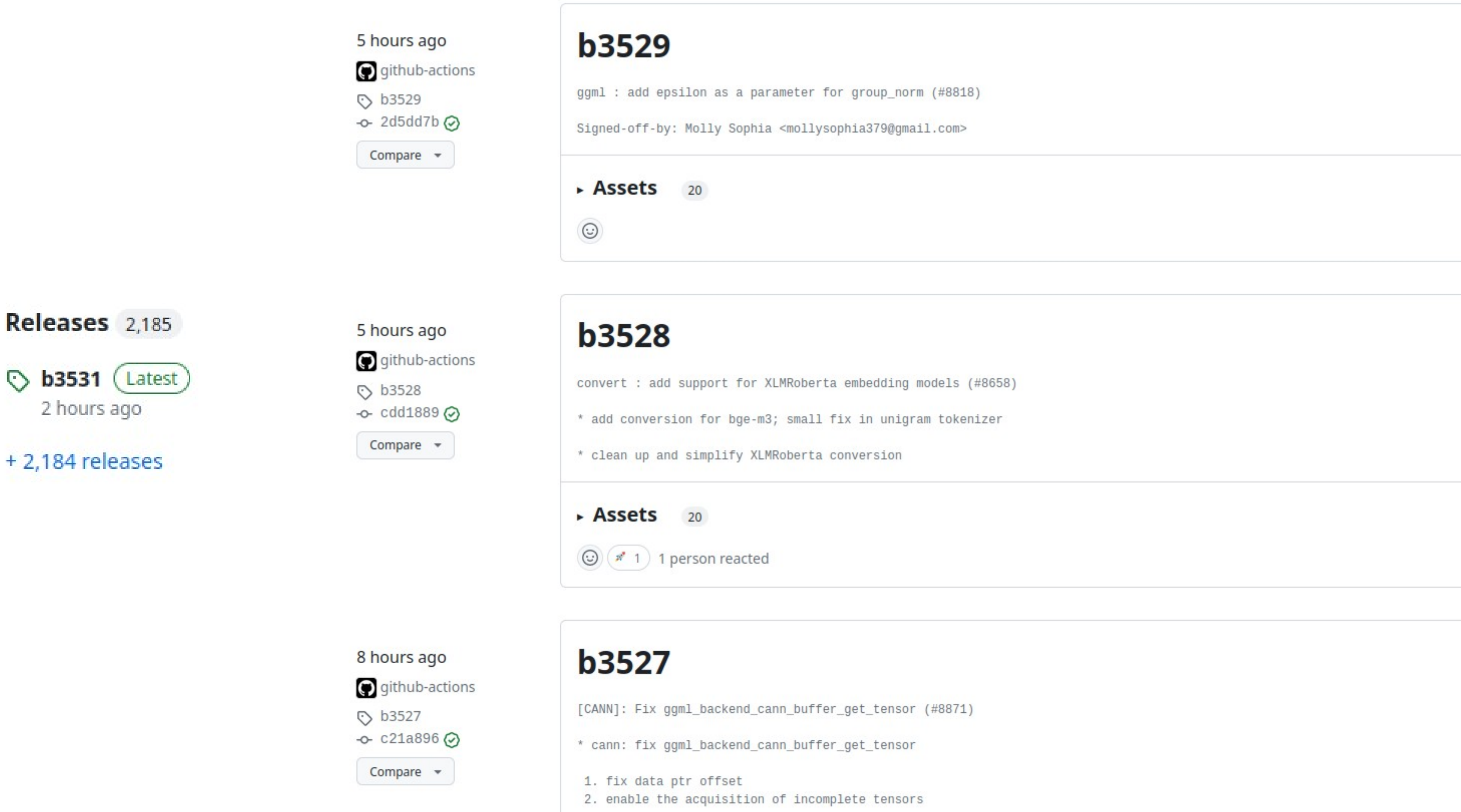
llama.cpp

Description

The main goal of `llama.cpp` is to enable LLM inference with minimal setup and state-of-the-art performance on a wide variety of hardware - locally and in the cloud.

Source: https://github.com/ggerganov/llama.cpp

Release Madness!



Sources: https://github.com/ggerganov/llama.cpp/releases

GGUF

GGUF

GGUF is a file format for storing models for inference with GGML and executors based on GGML. GGUF is a binary format that is designed for fast loading and saving of models, and for ease of reading. Models are traditionally developed using PyTorch or another framework, and then converted to GGUF for use in GGML.

It is a successor file format to GGML, GGMF and GGJT, and is designed to be unambiguous by containing all the information needed to load a model. It is also designed to be extensible, so that new information can be added to models without breaking compatibility.

For more information about the motivation behind GGUF, see Historical State of Affairs.

GGUF

Hugging Face Hub supports all file formats, but has built-in features for GGUF format, a binary format that is optimized for quick loading and saving of models, making it highly efficient for inference purposes. GGUF is designed for use with GGML and other executors. GGUF was developed by @ggerganov who is also the developer of llama.cpp, a popular C/C++ LLM inference framework. Models initially developed in frameworks like PyTorch can be converted to GGUF format for use with those engines.

Sources: https://github.com/ggerganov/ggml/blob/master/docs/gguf.md
https://huggingface.co/docs/hub/en/gguf

GGUF – File Format

Sources: https://huggingface.co/docs/hub/en/gguf

Hands-On

Notebook 4 - Converting to GGUF

Ollama

Get up and running with large language models.

Run Llama 3, Phi 3, Mistral, Gemma, and other models. Customize and create your own.

Download ↓

Available for macOS, Linux, and Windows (preview)

Ollama – Model Library

Model library

Ollama supports a list of models available on ollama.com/library

Here are some example models that can be downloaded:

Model	Parameters	Size	Download
Llama 3	8B	4.7GB	`ollama run llama3`
Llama 3	70B	40GB	`ollama run llama3:70b`
Phi-3	3.8B	2.3GB	`ollama run phi3`
Mistral	7B	4.1GB	`ollama run mistral`
Neural Chat	7B	4.1GB	`ollama run neural-chat`
Starling	7B	4.1GB	`ollama run starling-lm`
Code Llama	7B	3.8GB	`ollama run codellama`
Llama 2 Uncensored	7B	3.8GB	`ollama run llama2-uncensored`
LLaVA	7B	4.5GB	`ollama run llava`
Gemma	2B	1.4GB	`ollama run gemma:2b`
Gemma	7B	4.8GB	`ollama run gemma:7b`
Solar	10.7B	6.1GB	`ollama run solar`

> Note: You should have at least 8 GB of RAM available to run the 7B models, 16 GB to run the 13B models, and 32 GB to run the 33B models.

Source: https://github.com/ollama/ollama

Ollama – Model Details

phi

Phi-2: a 2.7B language model by Microsoft Research that demonstrates outstanding reasoning and language understanding capabilities.

3B

101.1K Pulls   Updated 3 months ago

2.7b   18 Tags   `ollama run phi`

Updated 4 months ago		e2fd6321a5fe · 1.6GB
model	arch **phi2** · parameters **2.8B** · quantization **Q4_0**	1.6GB
params	{"stop":["User:","Assistant:","System:"]}	42B
template	{{ if .System }}System: {{ .System }}{{ end }} User: {{ .P…	77B
system	A chat between a curious user and an artificial intelligen…	132B
license	MIT License Permission is hereby granted, free of charge, …	1.0kB

Source: https://ollama.com/library/phi

Ollama – Custom Model

Customize a model

Import from GGUF

Ollama supports importing GGUF models in the Modelfile:

1. Create a file named `Modelfile`, with a `FROM` instruction with the local filepath to the model you want to import.

```
FROM ./vicuna-33b.Q4_0.gguf
```

2. Create the model in Ollama

```
ollama create example -f Modelfile
```

3. Run the model

```
ollama run example
```


Source: https://github.com/ollama/ollama

Ollama – Model File

```
> ollama show --modelfile llama3
# Modelfile generated by "ollama show"
# To build a new Modelfile based on this one, replace the FROM line with:
# FROM llama3:latest
FROM /Users/pdevine/.ollama/models/blobs/sha256-00e1317cbf74d901080d7100f57580ba8dd8de57203072dc6f668324ba545f29
TEMPLATE """{{ if .System }}<|start_header_id|>system<|end_header_id|>

{{ .System }}<|eot_id|>{{ end }}{{ if .Prompt }}<|start_header_id|>user<|end_header_id|>

{{ .Prompt }}<|eot_id|>{{ end }}<|start_header_id|>assistant<|end_header_id|>

{{ .Response }}<|eot_id|>"""
PARAMETER stop "<|start_header_id|>"
PARAMETER stop "<|end_header_id|>"
PARAMETER stop "<|eot_id|>"
PARAMETER stop "<|reserved_special_token"
```


Source: https://github.com/ollama/ollama/blob/main/docs/modelfile.md

Ollama

REST API

Ollama has a REST API for running and managing models.

Generate a response

```
curl http://localhost:11434/api/generate -d '{
  "model": "llama3",
  "prompt":"Why is the sky blue?"
}'
```

Chat with a model

```
curl http://localhost:11434/api/chat -d '{
  "model": "llama3",
  "messages": [
    { "role": "user", "content": "why is the sky blue?" }
  ]
}'
```


Source: https://github.com/ollama/ollama

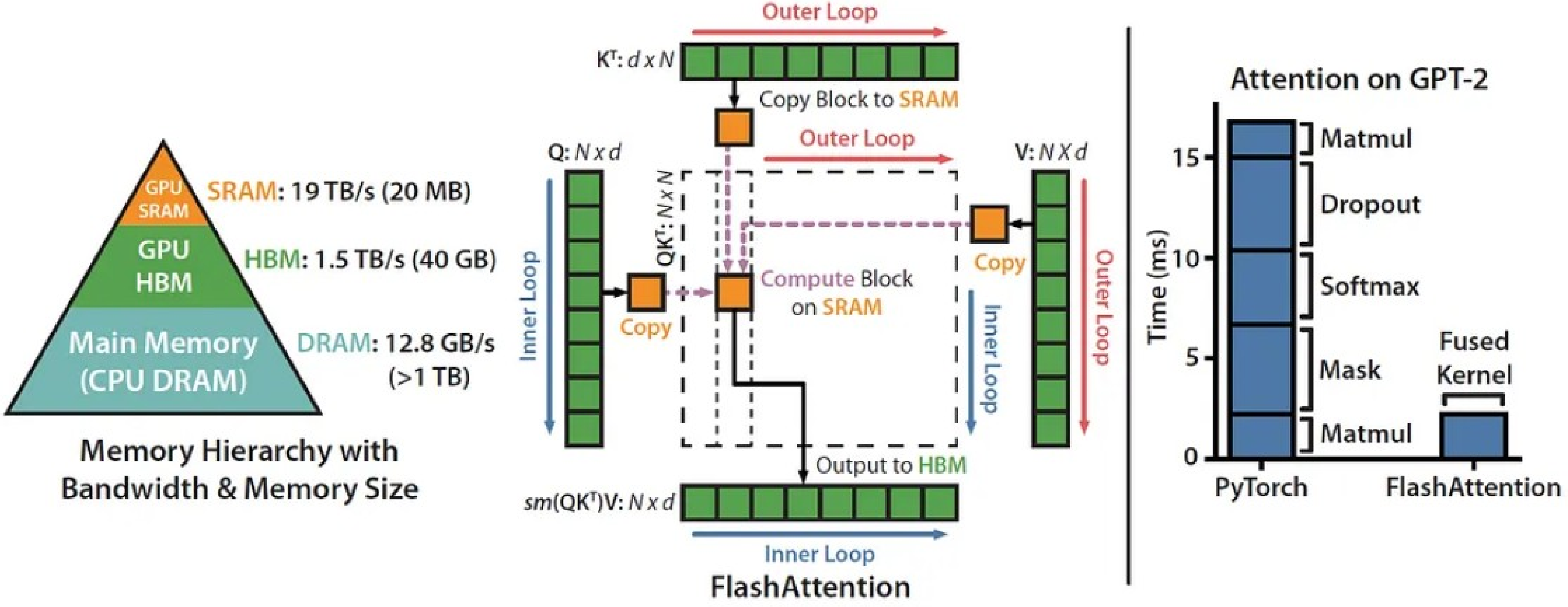
Hands-On

Notebook 5 - Running Ollama

Exercise

Exercise – Quantizing and Serving a Model

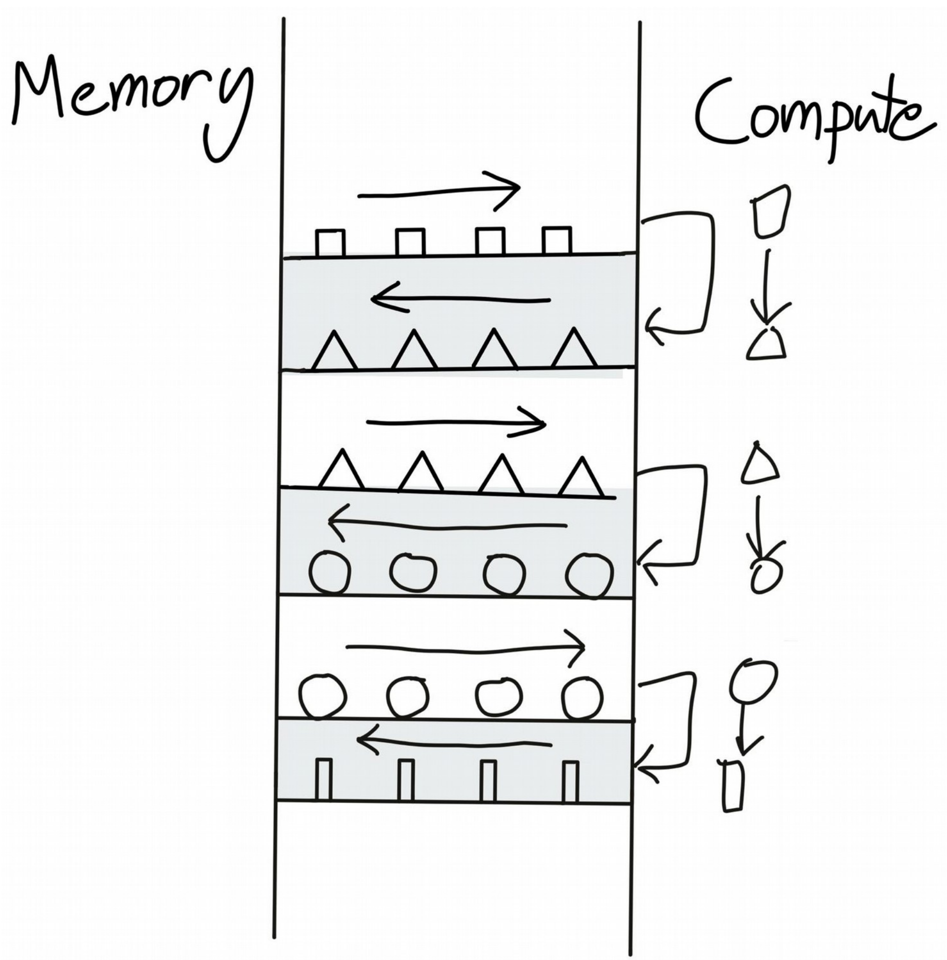
BONUS: Flash Attention



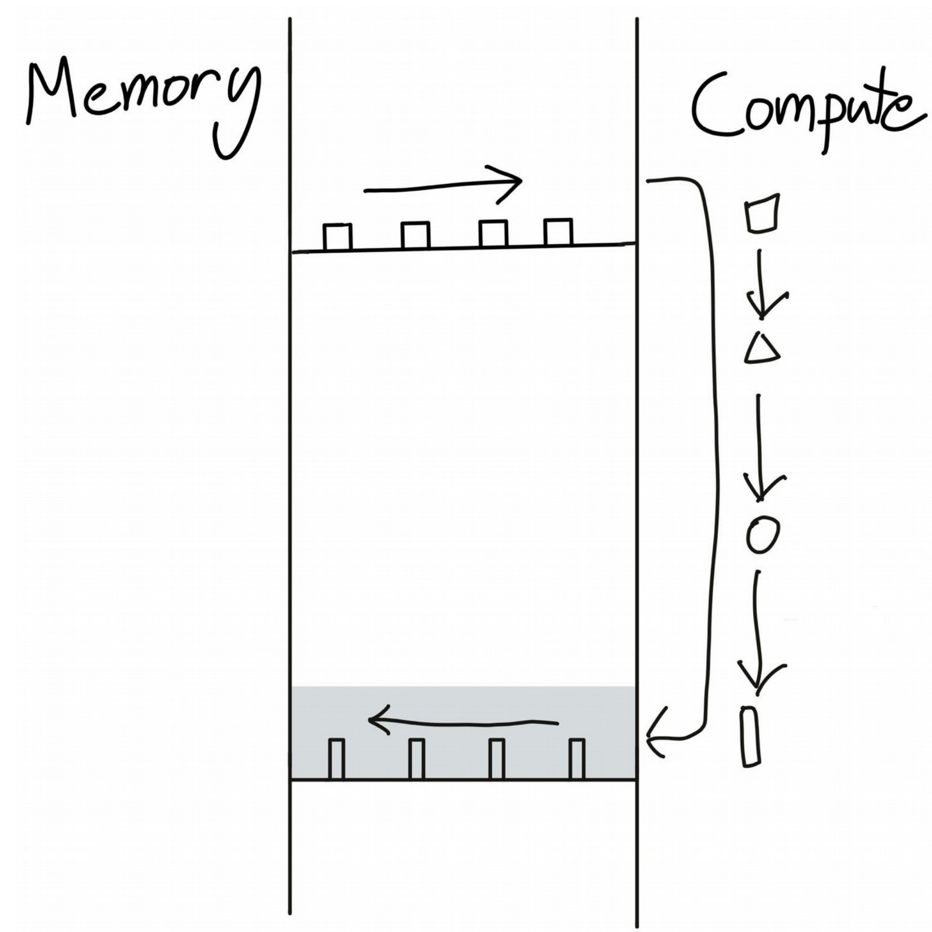
Source: https://arxiv.org/pdf/2205.14135

Flash Attention

BEFORE



AFTER



Source: https://horace.io/brrr_intro.html

Flash Attention

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
Return $\mathbf{O}$.

Algorithm 1 FlashAttention

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

Set block sizes $B_c = \left\lceil \frac{M}{4d} \right\rceil, B_r = \min\left(\left\lceil \frac{M}{4d} \right\rceil, d\right)$.
Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.
for $1 \le j \le T_c$ **do**
  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
  for $1 \le i \le T_r$ **do**
    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
    On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
    On chip, compute $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
    Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1}(\mathrm{diag}(\ell_i) e^{m_i - m_i^{\mathrm{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
    Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
  end for
end for
Return $\mathbf{O}$.

Source: https://arxiv.org/pdf/2205.14135

Tiling Softmax

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

$$m(x) := \max_i \quad x_i, \quad f(x) := \begin{bmatrix} e^{x_1 - m(x)} & \ldots & e^{x_B - m(x)} \end{bmatrix}$$

$$\ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}$$

For vectors $x^{(1)}, x^{(2)} \in \mathbb{R}^B$, we can decompose the softmax of the concatenated $x = \begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix} \in \mathbb{R}^{2B}$ as:

$$m(x) = m(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \begin{bmatrix} e^{m(x^{(1)})-m(x)} f(x^{(1)}) & e^{m(x^{(2)})-m(x)} f(x^{(2)}) \end{bmatrix},$$

$$\ell(x) = \ell(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = e^{m(x^{(1)})-m(x)} \ell(x^{(1)}) + e^{m(x^{(2)})-m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

Source: https://gordicaleksa.medium.com/eli5-flash-attention-5c44017022ad

Hands-On

BONUS - Flash Attention

The End

THANK YOU!