INDIA-AI IMPACT SUMMIT 2026

MINISTRY OF ELECTRONICS & INFORMATION TECHNOLOGY GOVERNMENT OF INDIA

INDIAai

Workshop on Systems for LLMs

Systems for Large Language Models (LLMs) Workshop with Hands-On
Exploring the Infrastructure Behind Intelligent Systems

Sessions Overview:
LLM building blocks
GPUs for LLM
Inference serving with vLLM
Distributed Training

Tools: Hugging Face, PyTorch

Details
Date: 6th,7th September
Time: 9:30 am-5pm
Register @
https://forms.gle/a8rQQ835jJG42i3eA

Supported by: IBM-IITB Academic Research Partnership

IBM

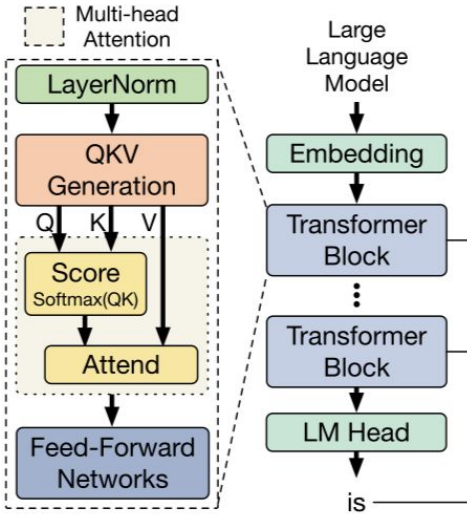
CSE IIT Bombay

Inference Systems

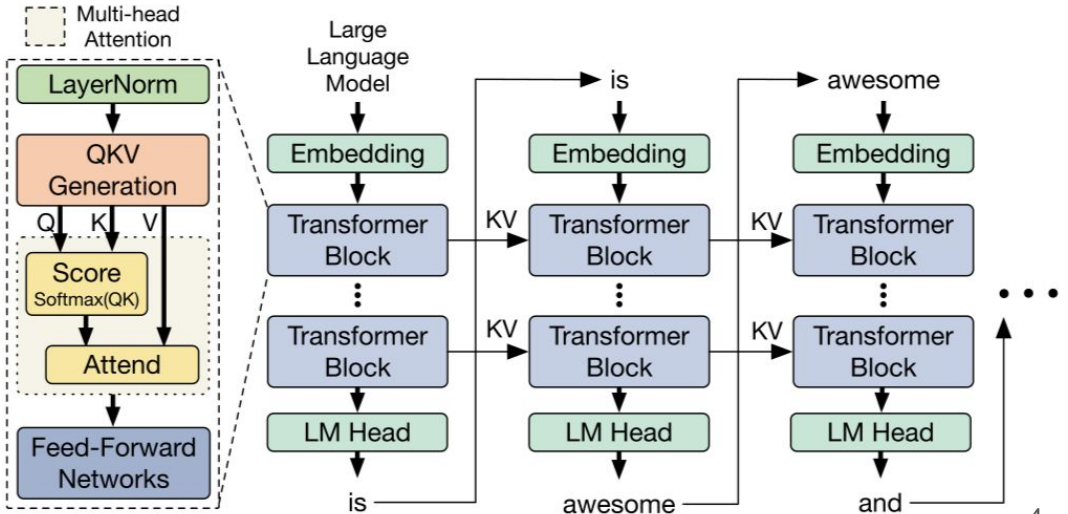
What is inference?

forward vs generate

- Forward: single forward pass through LLM

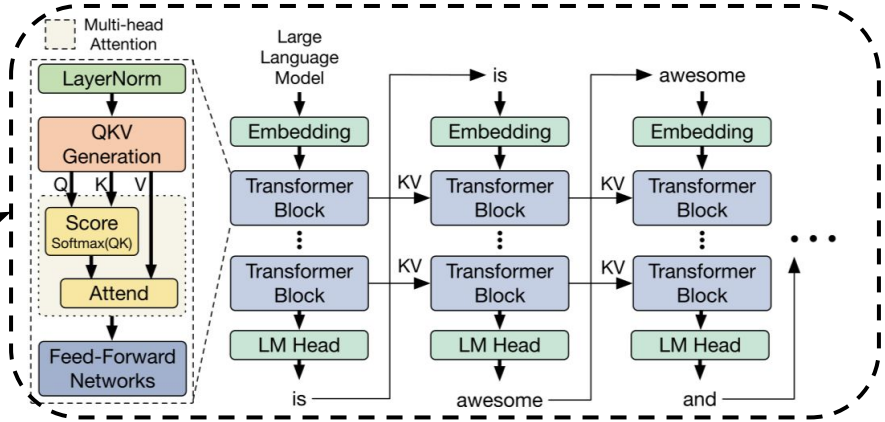


- Generate:
  - Generates multiple tokens, until EOS or max tokens reached

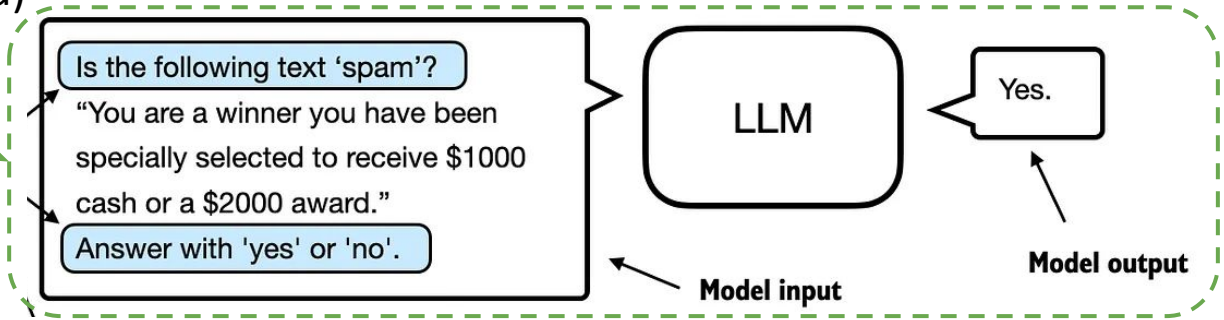


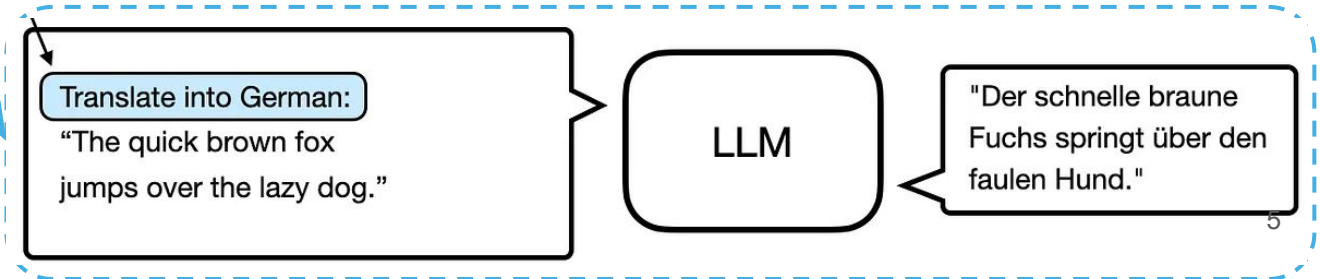
source : https://arxiv.org/pdf/2408.05499

What is inference?

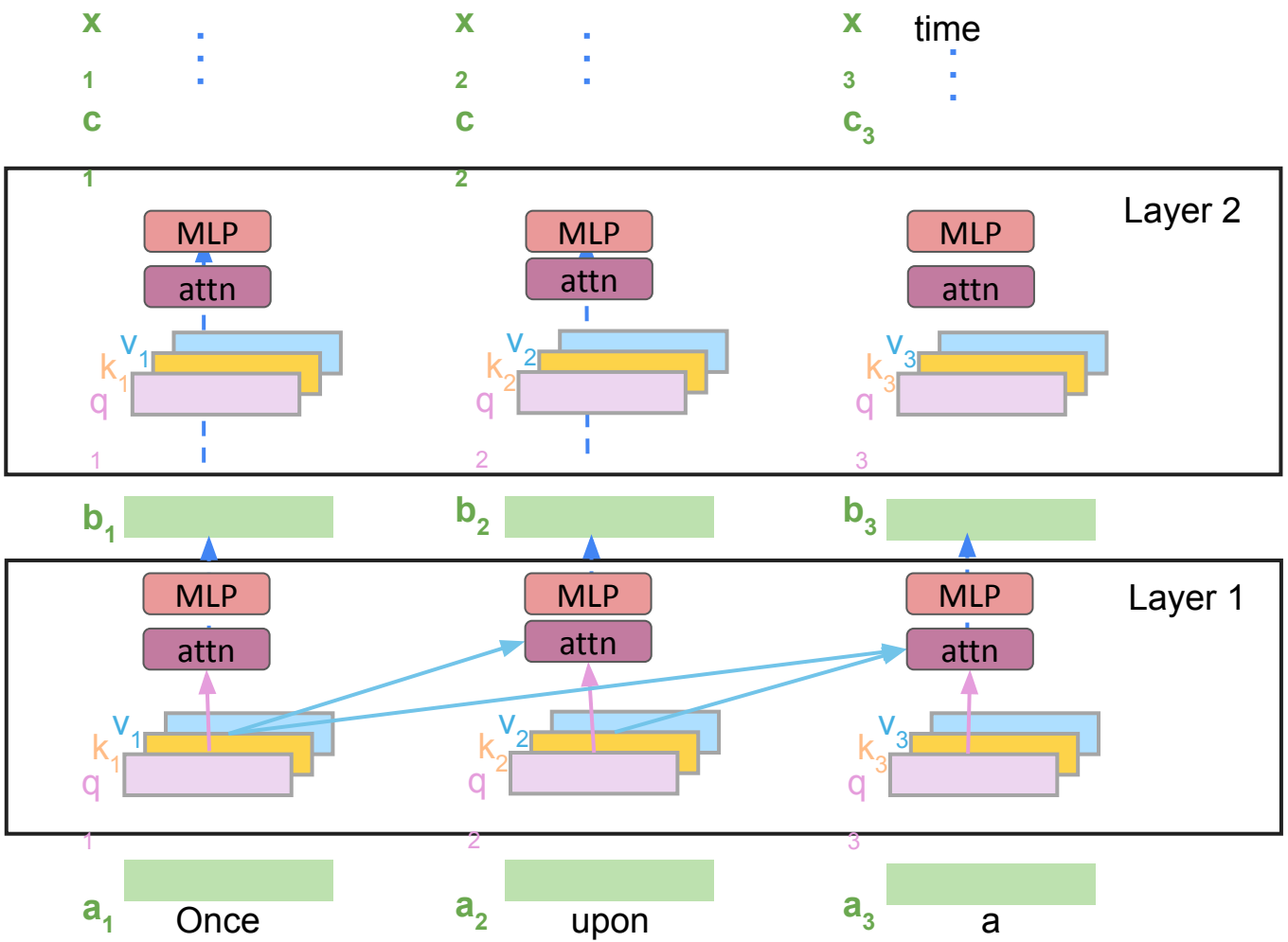


- Text generation (LM Head)
- Classification
- Translation





https://magazine.sebastianraschka.com/p/building-a-gpt-style-llm-classifier


x
1
c
1
x
2
c
2
x
3
c3
time
Layer 2
MLP
attn
v1
k1
q
1
MLP
attn
v2
k2
q
2
MLP
attn
v3
k3
q
3
b1
b2
b3
Layer 1
MLP
attn
v1
k1
q
1
MLP
attn
v2
k2
q
2
MLP
attn
v3
k3
q
3
a1
Once
a2
upon
a3
a

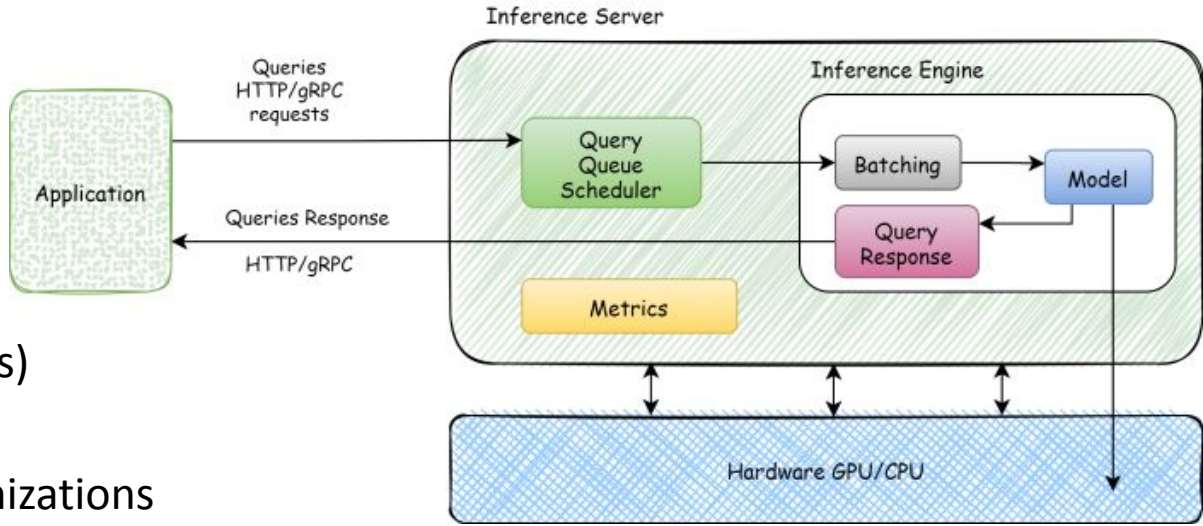
Simplest generate example

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

tokenizer = AutoTokenizer.from_pretrained("gpt2")
model = AutoModelForCausalLM.from_pretrained("gpt2")

prompt = "My trip to Yosemite was"
inputs = tokenizer(prompt, return_tensors="pt")

output = model.generate(inputs.input_ids, max_length=100)

generated_text = tokenizer.batch_decode(output, skip_special_tokens=True)
print(generated_text)
```

Inference engines

What are inference engines?

- Wrap systems concepts around the process of inference
  - Batching
  - Streaming
  - Fault management
  - Error handling
  - Parallelism (multi-GPUs)
- Optimizations:
  - Memory storage optimizations
  - Offloading
- Most popular open-source is vllm. Others include SGLang, NVIDIA's Tensor-RT LLM



https://ranjankumar.in/large-language-models-llms-inference-and-serving/

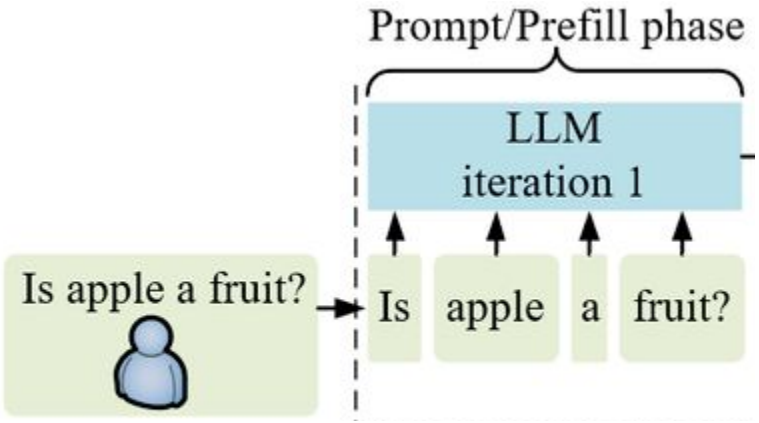
Engines	Batch Optimization				Parallelism				Compression			Fine-tuning	Caching			Attention Optimization						Sampling Optimization	Output Optimization
	Dynamic Batching	Continuous Batching	Nano Batching	Chunked-prefills	Data Parallelism	Fully Sharded Data Parallel	Tensor Parallelism	Pipeline Parallelism	Quantization[†] / Support Quantized Model	Pruning	Sparsity[‡]	LoRA	Prompt Caching	Prefix Caching	KV Caching	PagedAttention	vAttention	FlashAttention	RadixAttention	FlexAttention	FireAttention	Speculative Decoding	Structured Outputs[*]
Ollama [194]	✘	✘	✘	✘	✘	✘	✔	✔	✔	✔	✔ (MoE, SL)	✔	✔	✘	✔	✘	✘	✔	✘	✘	✘	✔	✔ (GG)
llama.cpp [82]	✘	✔	✘	✘	✘	✘	✔	✔	✔	✘	✔ (MoE, SL)	✔	✔	✘	✔	✘	✘	✔	✘	✘	✘	✔	✔ (GG)
vLLM [125]	✘	✔	✘	✔	✔	✔	✔	✔	✔ (A, A*, B, D, G, L, M)	✔	✔ (BS, MoE, N:M)	✔	✘	✔	✔	✔	✘	✔ (v3)	✘	✘	✘	✔	✔ (LM, OA, OL, XG)
DeepSpeed-FastGen [102]	✘	✔	✘	✔	✔	✔	✔	✔	✔	✔ (S, U)	✔ (N:M, NxM, MoE, SA)	✔	✘	✘	✔	✔	✘	✔ (v2)	✘	✘	✘	✘	✘
Unsloth [244]	✘	✘	✘	✘	✘	✘	✘	✘	✔ (B)	✘	✘	✔	✘	✘	✔	✘	✘	✔	✘	✔	✘	✘	✘
MAX [179]	✘	✔	✘	✔	✘	✘	✔	✘	✔	✘	✔ (MoE)	✔	✘	✔	✔	✔	✘	✔ (v3)	✘	✘	✘	✔	✔ (XG)
MLC-LLM [177]	✘	✔	✘	✔	✘	✘	✔	✔	✔ (A)	✘	✔ (MoE)	✘	✘	✔	✔	✔	✘	✘	✘	✘	✘	✔	✔ (XG)
llama2.c [21]	✘	✘	✘	✘	✘	✘	✘	✘	✔	✘	✘	✘	✘	✘	✔	✘	✘	✘	✘	✘	✘	✘	✘
bitnet.cpp [251]	✘	✘	✘	✘	✘	✘	✘	✘	✔ (A, G)	✘	✔ (MoE)	✘	✘	✘	✔	✘	✘	✘	✘	✘	✘	✘	✘
SGLang [295]	✘	✔	✘	✔	✔	✔	✔	✘	✔ (A, B, G, L, M)	✔	✔ (DSA, N:M, MoE)	✔	✘	✔	✔	✔	✘	✘	✔	✘	✘	✔	✔ (LL, OL, XG)
LitGPT [145]	✘	✔	✘	✘	✔	✔	✔	✘	✔ (B)	✘	✔ (MoE)	✔	✘	✘	✔	✘	✘	✔ (v2)	✘	✘	✘	✔	✘
OpenLLM [30]	✘	✔	✘	✘	✔	✘	✘	✘	✔ (B, G)	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘
TensorRT-LLM [191]	✔	✔	✘	✔	✔	✘	✔	✔	✔ (A, G, S, W)	✔	✔ (BSA, MoE)	✔	✔	✘	✔	✔	✘	✘	✘	✘	✘	✔	✔ (XG)
TGI [110]	✘	✔	✘	✘	✘	✘	✔	✘	✔ (A, E, E*, G, M)	✔	✔ (N:M, MoE)	✔	✘	✔	✔	✔	✘	✔ (v2)	✘	✘	✘	✔	✔ (OL)
PowerInfer [227, 270]	✘	✔	✘	✘	✔	✘	✘	✔	✔	✘	✔	✔	✔	✘	✔	✘	✘	✔	✘	✘	✘	✔	✔ (GG)
LMDeploy [162]	✘	✔	✘	✔	✘	✘	✔	✘	✔ (A, G, S)	✔	✔ (MoE)	✔	✘	✔	✔	✔	✘	✘	✘	✘	✘	✘	✔ (PT)
LightLLM [142]	✔	✘	✘	✔	✘	✘	✔	✘	✔	✘	✔ (MoE)	✘	✔	✘	✔	✘	✘	✔ (v1)	✘	✘	✘	✘	✔ (OL, XG)
NanoFlow [300]	✘	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✔	✘	✘	✘	✘	✘	✘	✘	✘
DistServe [297]	✔	✔	✘	✔	✘	✘	✔	✔	✘	✘	✘	✘	✘	✘	✔	✔	✘	✔ (v1)	✘	✘	✘	✘	✘
vAttention [206]	✔	✘	✘	✘	✔	✘	✔	✔	✔	✔	✔ (N:M)	✔	✘	✘	✔	✔	✔	✔ (v2)	✘	✘	✘	✘	✘
Sarathi-Serve [8]	✘	✘	✘	✔	✘	✘	✔	✔	✘	✘	✔ (MoE)	✘	✘	✘	✔	✘	✘	✔ (v2)	✘	✘	✘	✘	✘
Friendli Inference [71]	–	✔	–	–	–	–	✔	✔	✔	–	✔ (MoE)	✔	–	–	–	–	✘	–	–	✘	✘	✔	✔
Fireworks AI [67]	–	✔	–	–	–	–	–	–	✔	✔	✔ (MoE)	✔	✔	–	✔	–	✘	–	–	✘	✔	✔	✔ (OA)
Groq Cloud [89]	✘	–	–	–	✔	–	✔	✔	✔	✔	✔ (MoE)	–	–	–	–	–	–	–	–	✘	✘	✔	✔ (OA)
Together Inference [239]	–	–	–	–	–	✔	–	–	✔	–	✔ (MoE)	✔	✔	–	–	–	✘	✔ (v3)	–	✘	✘	✔	✔

A Survey on Inference Engines for Large Language Models: Perspectives on Optimization and Efficiency (https://arxiv.org/pdf/2505.01658)

Terminology

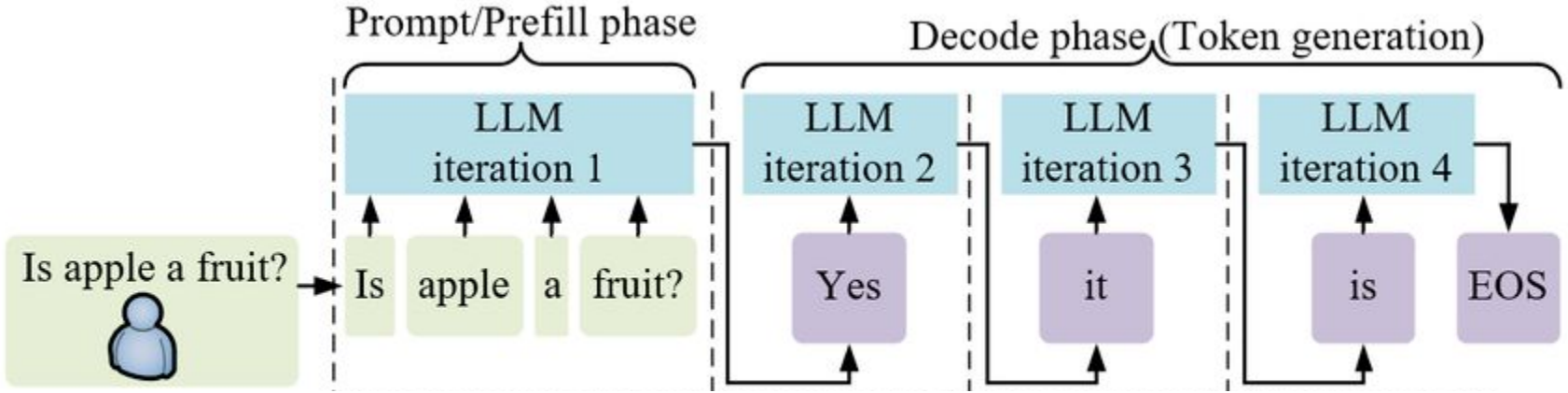
Prefill vs decode

- Prefill: input/prompt phase. Generation of the first output token (ingestion of the entire input in parallel)

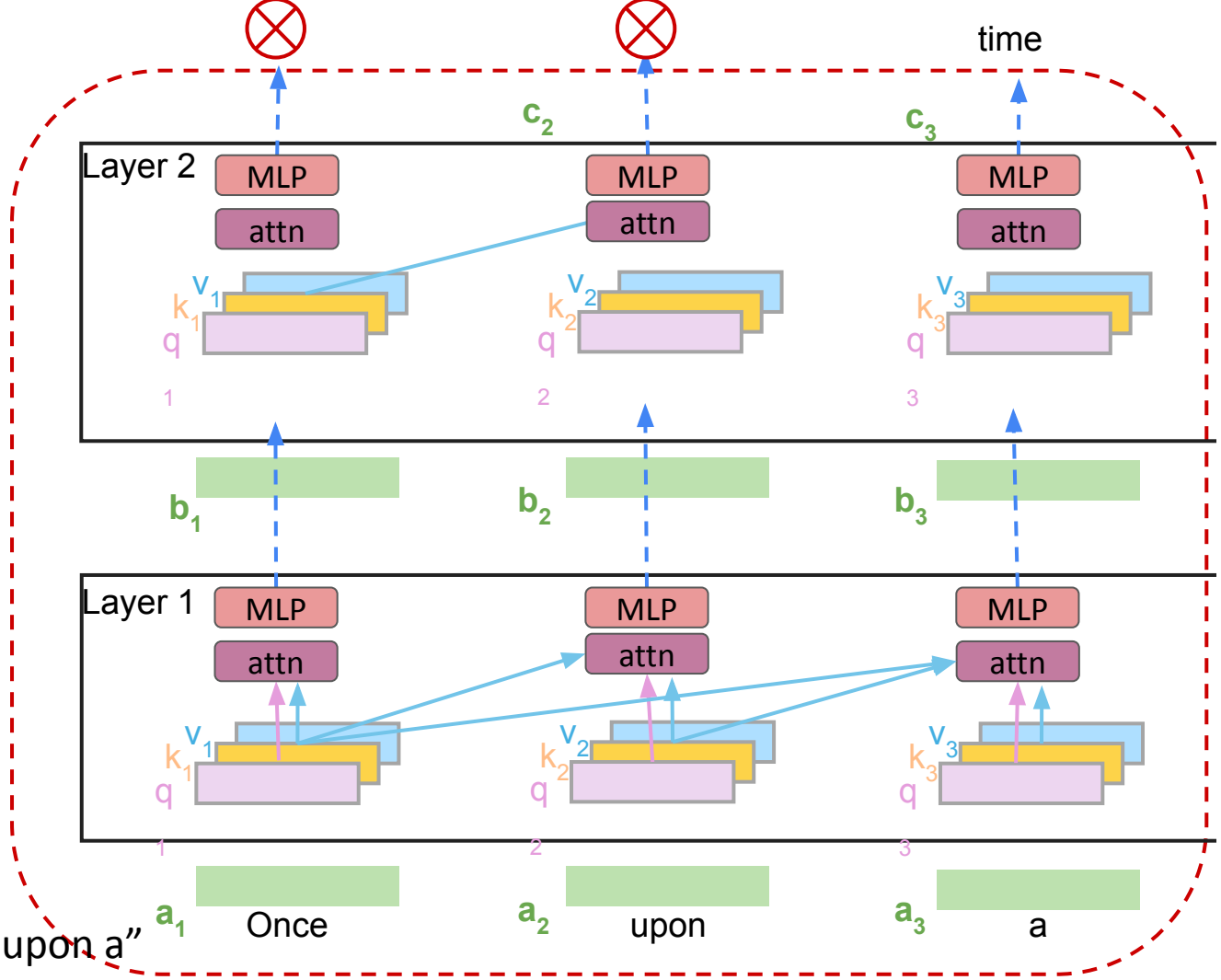


Prefill vs decode

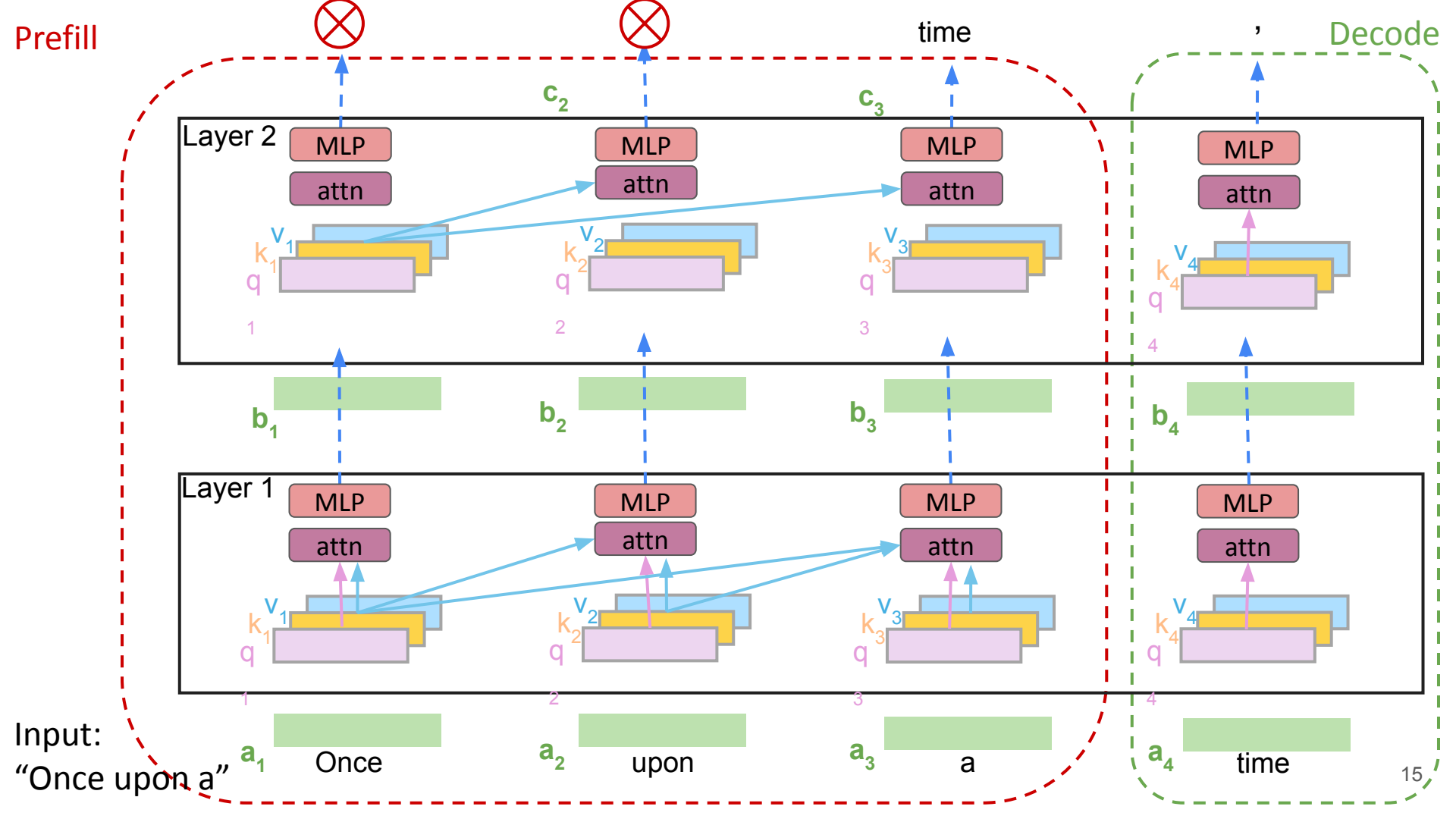
- Prefill: input/prompt phase. Generation of the first output token (ingestion of the entire input in parallel)
- Decode: output/generation phase. Decoding of subsequent output tokens (sequential)



Prefill

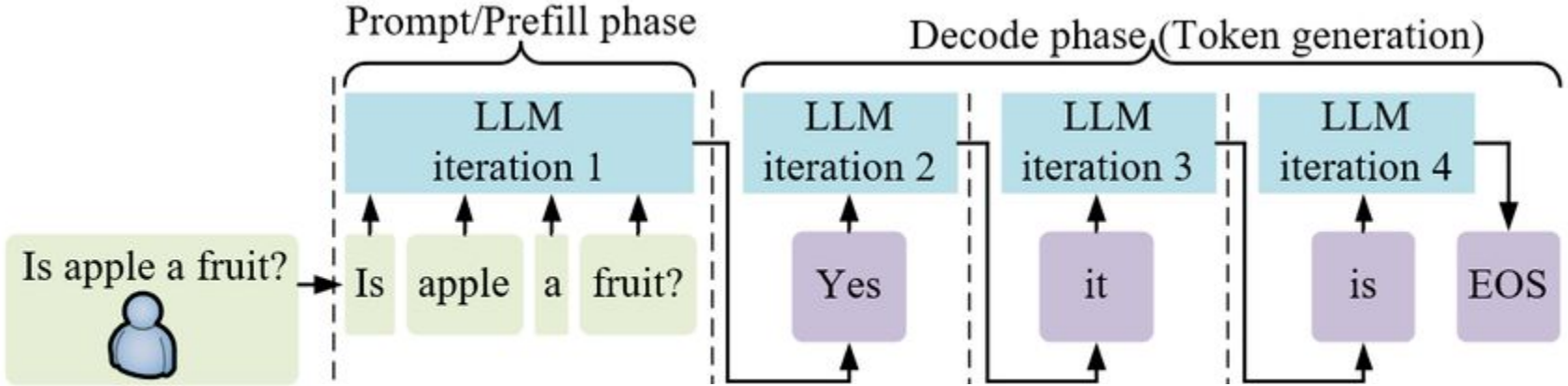


Input:
“Once upon a”


Prefill
time
,
Decode
Layer 2
Layer 1
MLP
attn
c_2
c_3
v_1 k_1 q 1
v_2 k_2 q 2
v_3 k_3 q 3
v_4 k_4 q 4
b_1
b_2
b_3
b_4
a_1 Once
a_2 upon
a_3 a
a_4 time
Input:
“Once upon a”
15

Metrics

- Throughput: Number of Tokens processed per second
- Latency:
  - Time to first token (prefill time)
  - Time between tokens (decode time)



Let’s write our own generate

Step-wise plan

Make sure pytorch_model.bin is in the exercises directory

```
wget https://huggingface.co/gpt2/resolve/main/pytorch_model.bin
  -   pytorch_model.bin
  -   simple
  -   kvcache
```

First let's take a look at gpt.py

1. Create position ids tensor (What is the position ids for this input?) `[1, 2, … n]`
2. Call forward(inputs, position_ids) (Make sure both are tensors)
3. What is output (logits) shape? We will learn about this later
4. Append output token to the inputs tensor
5. Wrap entire code in a "for" loop (What is the exit condition?)

```
MAX_SEQ_LEN = 10
config = GPT2Config.from_pretrained('gpt2')
model = GPTLMHead(config)
tokenizer = AutoTokenizer.from_pretrained("gpt2")

def generate(input):
    # Tokenize input
    tokenized = tokenizer.encode(input)
    inputs = torch.tensor(tokenized)

    while ???: # What is exit condition of the loop?
        positions = ??? # Create position ids tensor
        logits = model(???) # Call model.forward with inputs and position ids
        logits = logits[-1, :]
        next_token = torch.argmax(logits, dim=-1, keepdim=True)
        inputs = ??? # Concatenate next_token to inputs for next step here

    output = tokenizer.decode(inputs)
    return output

input = 'The quick brown fox jumped'
output = generate(input)
```

See the signature of GPTLMHead.forward()

```
MAX_SEQ_LEN = 10
config = GPT2Config.from_pretrained('openai-community/gpt2-large')
model = GPTLMHead(config)
tokenizer = AutoTokenizer.from_pretrained("gpt2")

def generate(input):
    # Tokenize input
    tokenized = tokenizer.encode(input)
    inputs = torch.tensor(tokenized,)

    while len(inputs[0]) < MAX_SEQ_LEN:
        positions = torch.arange(len(inputs))
        logits = model(inputs, positions)
        logits = logits[-1, :]
        next_token = torch.argmax(logits, dim=-1, keepdim=True)
        inputs = torch.cat((inputs, next_token), dim=0)

    output = tokenizer.decode(inputs.tolist())
    return output

input = 'The quick brown fox jumped'
output = generate(input)
```

Core features

Core features of inference engines

Inference engines such as vllm etc. have certain features/optimizations

We will look at 3 such techniques:

1. KV Caching
2. Continuous Batching
3. Paged Attention

KV Caching

Why cache KVs?

What are K and V?

For causal self-attention, K/V of all past tokens is required to calculate attention of current token

Problem: It is recalculated again and again for every token

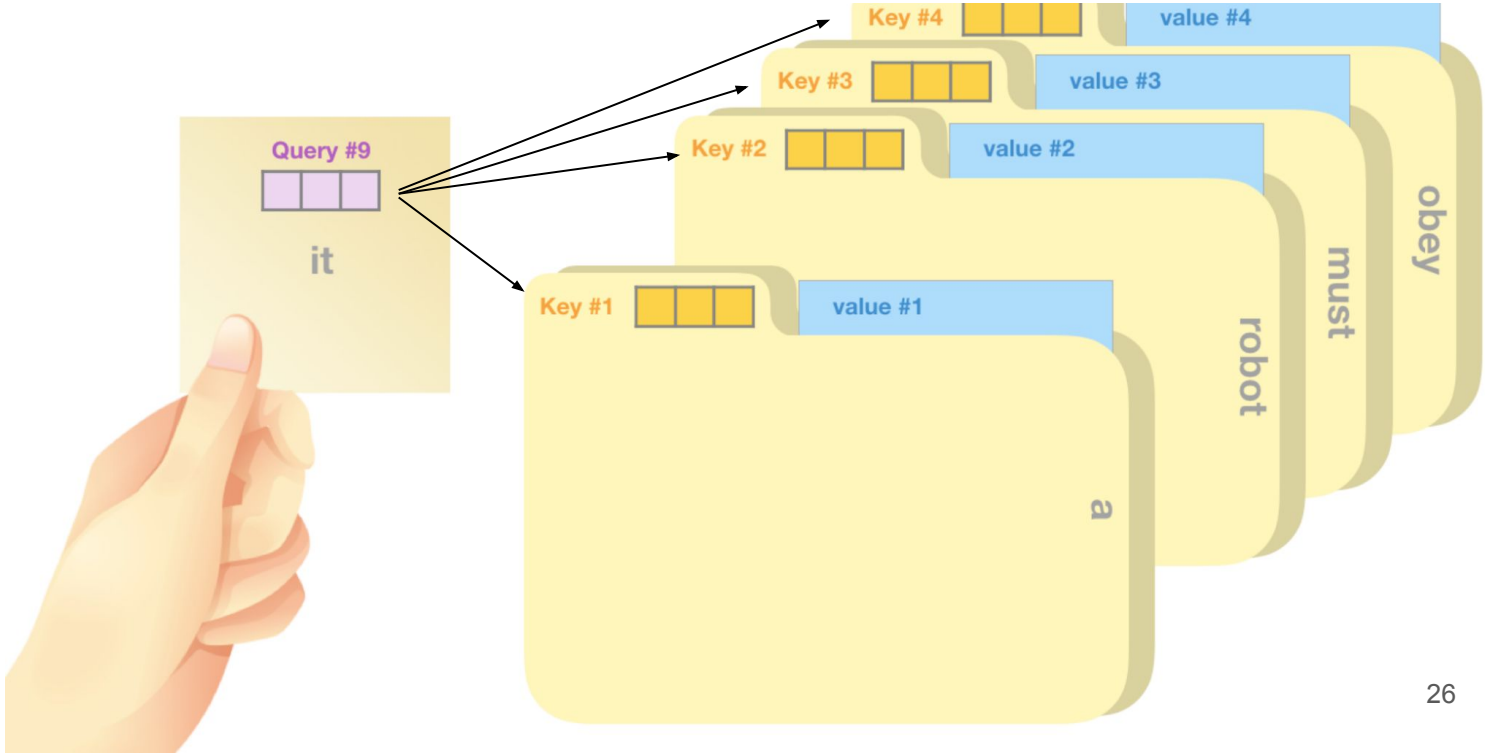
Idea: Generate and cache once, reuse for subsequent tokens

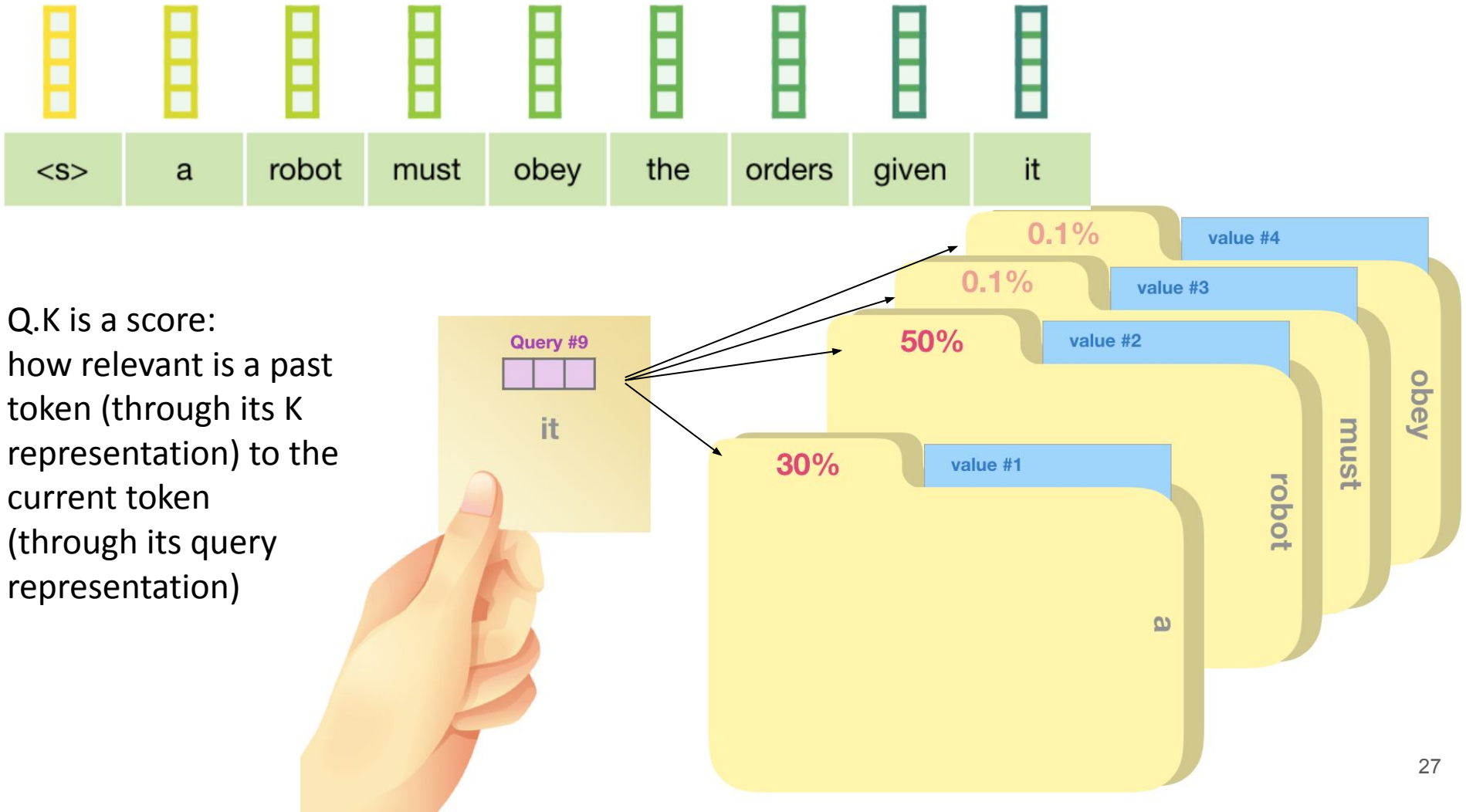
Query: The query is a representation of the current word used to score against all the other words (using their keys). We only care about the query of the token we're currently processing.

Key: Key vectors are like labels for all the words in the segment. They're what we match against in our search for relevant words.

Value: Value vectors are actual word representations, once we've scored how relevant each word is, these are the values we add up to represent the current word.

Recap of self attention

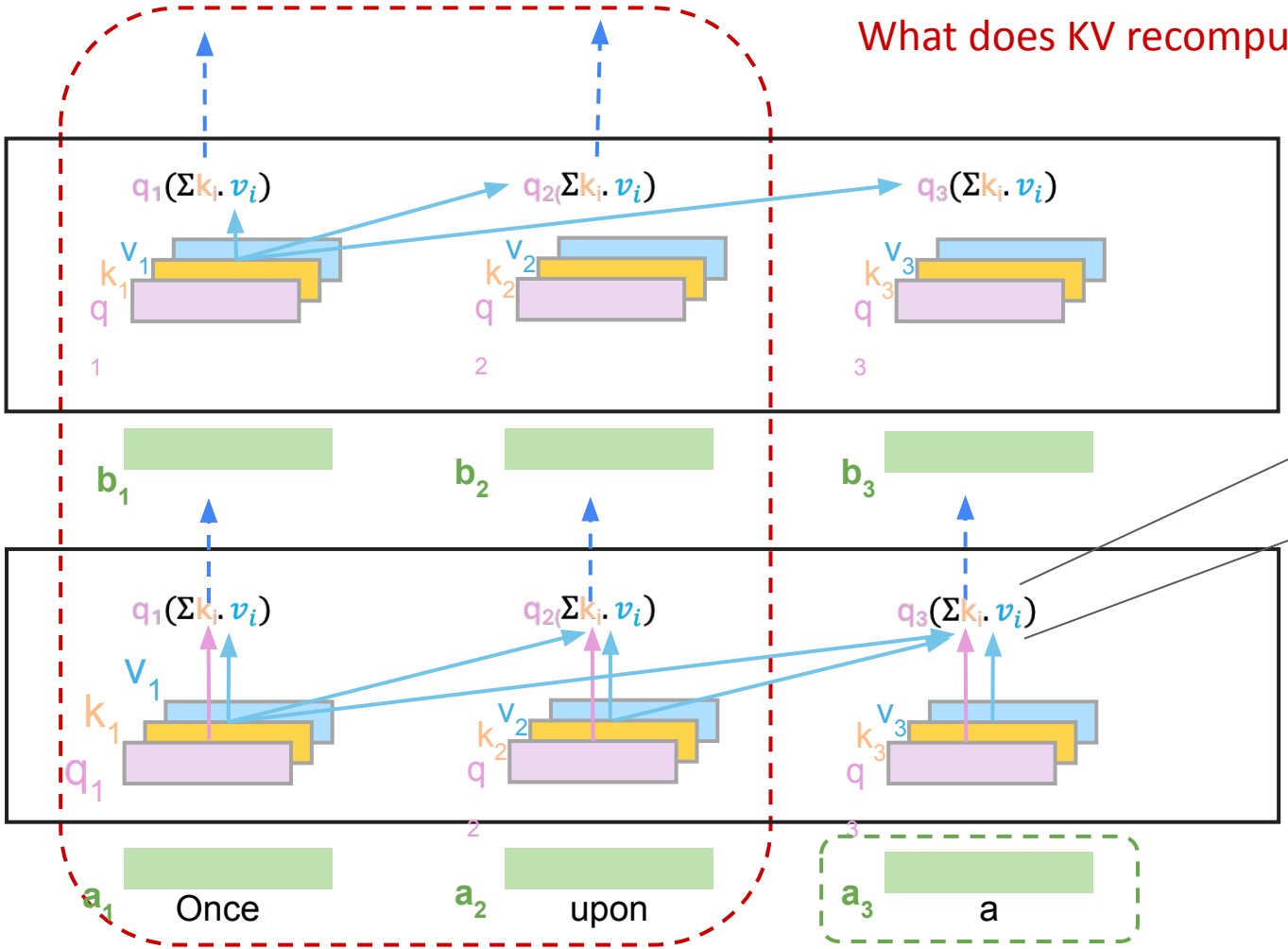



<s>
a
robot
must
obey
the
orders
given
it
Q.K is a score:
how relevant is a past token (through its K representation) to the current token (through its query representation)
Query #9
it
0.1%
value #4
0.1%
value #3
50%
value #2
30%
value #1
obey
must
robot
a

Word	Value vector	Score	Value X Score
<s>		0.001	
a		0.3	
robot		0.5	
must		0.002	
		0.001	
		0.0003	
		0.005	
given		0.002	
it		0.19	
		Sum:	

Finally, each past token (through its Value Representation) is multiplied with the score and summed up to obtain a new representation of the current token

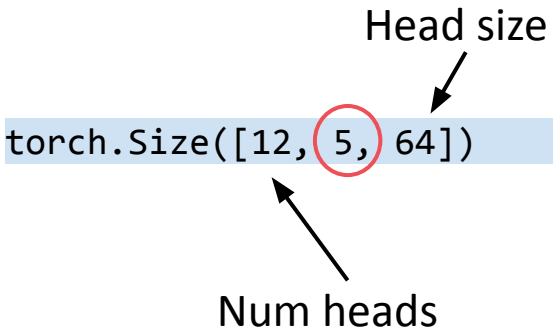
What does KV recomputation mean?



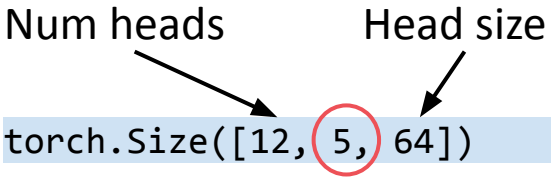
1. We need the K and V of all tokens in the past, at every layer
2. This requires calculation of b1, b2 etc. for all past tokens

Let’s see the repetition in action

1. GPTAttention has a field `self.idx` which refers to the layer
2. Now print the shape of the k or v tensor for a particular layer after the first 5 lines of the attention block before SDPA

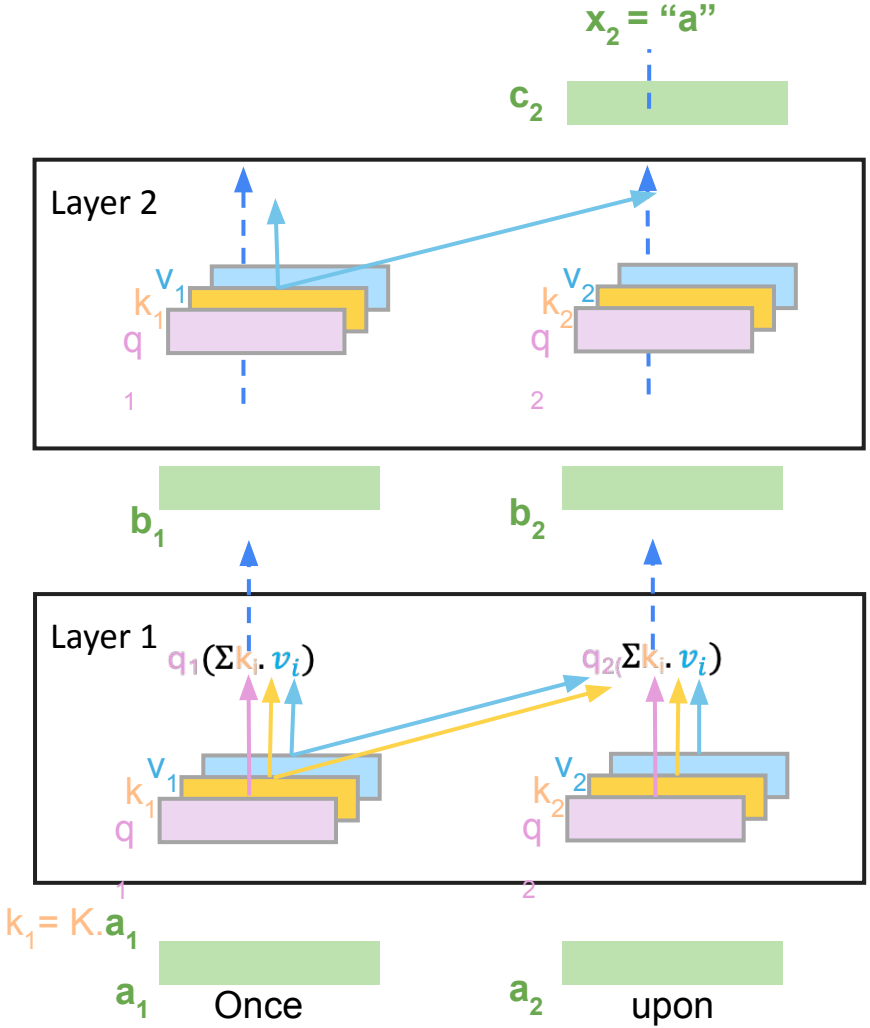


Let's see the repetition in action

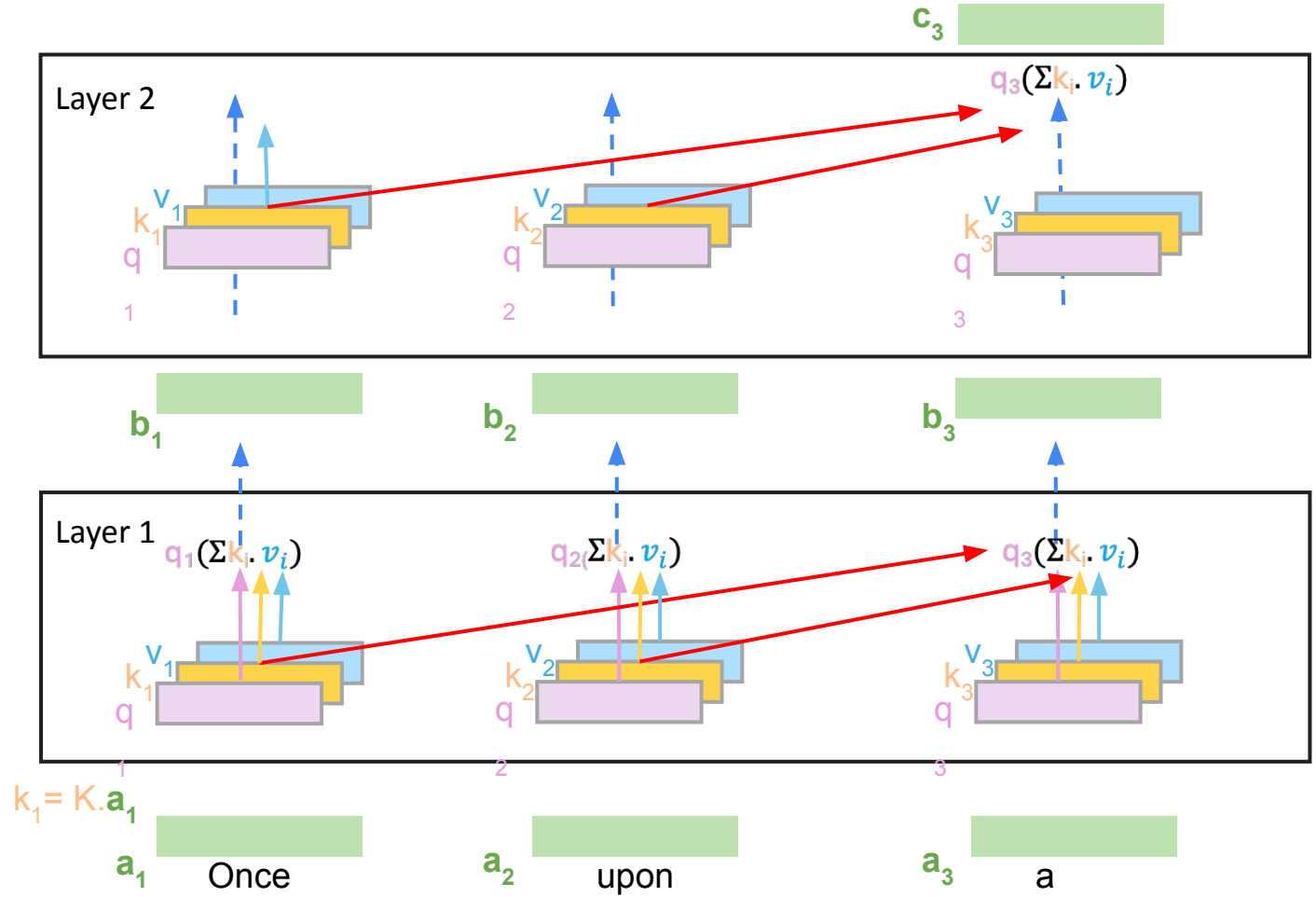


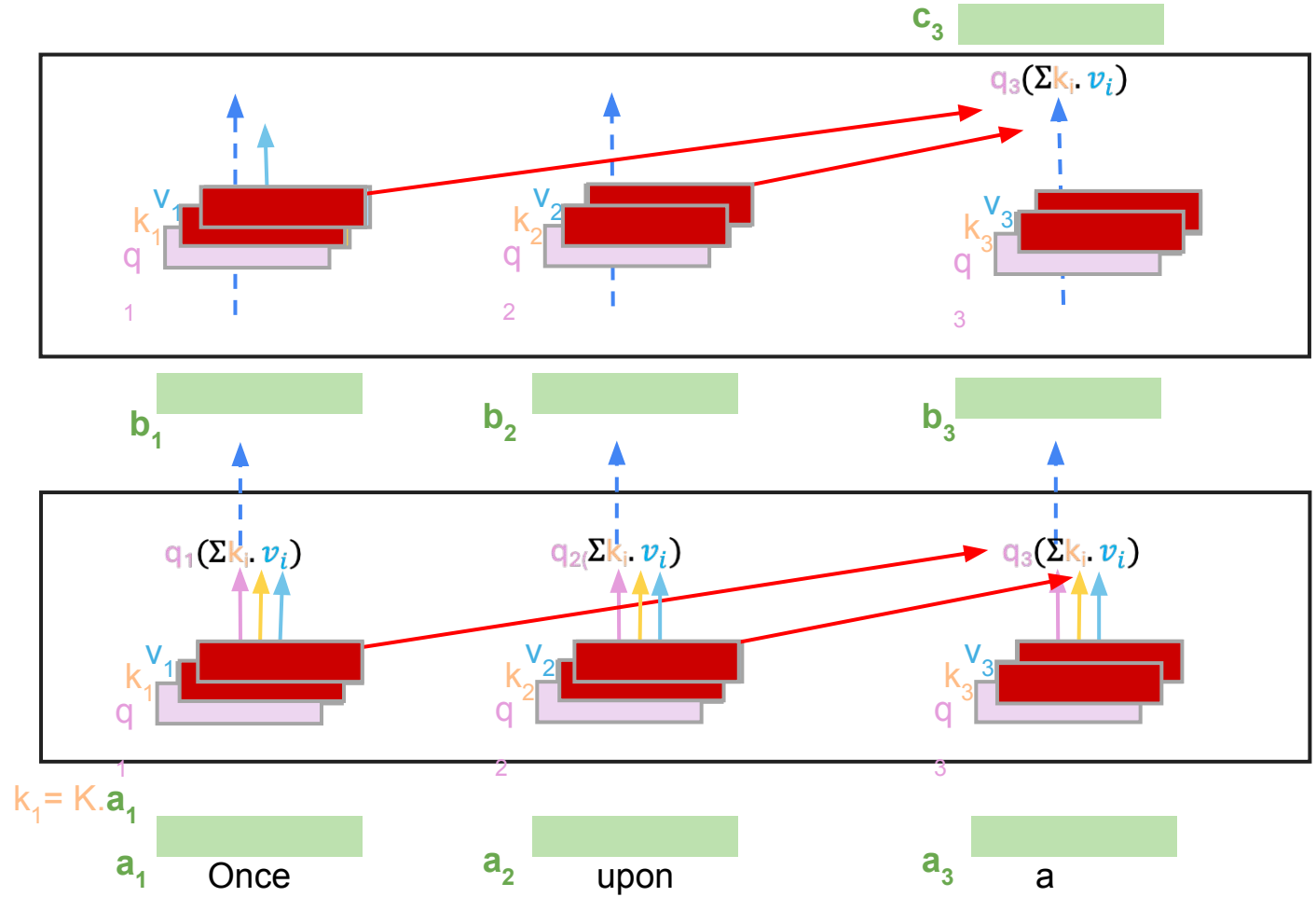
1. GPTAttention has a field `self.idx` which refers to the layer
2. Now print the shape of the k or v tensor for a particular layer
3. One dimension grows every iteration - that is the number of tokens
4. Pick a random idx and print keys[a][b][c] (Choose random valid values of a, b, c)

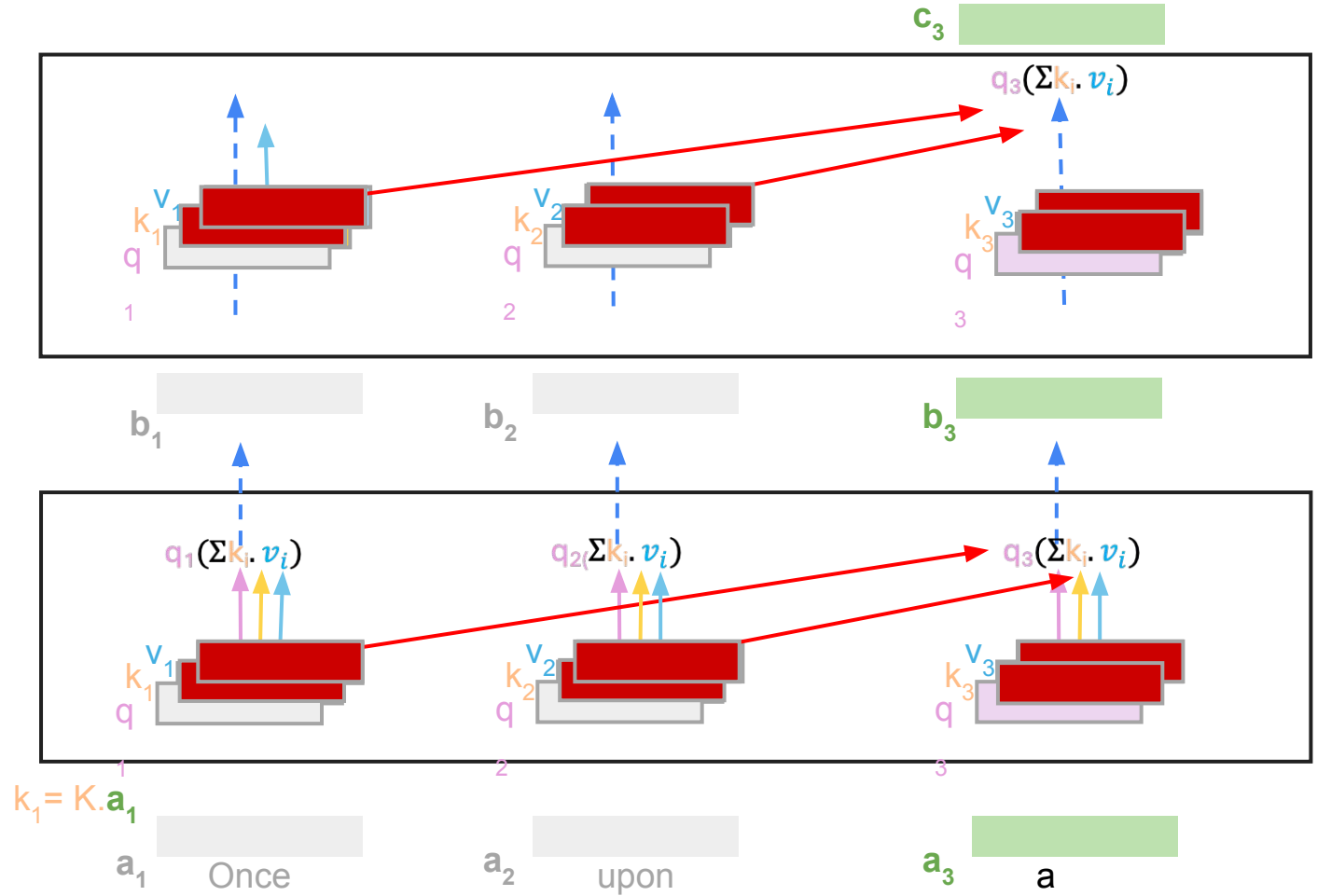
```
class GPTAttention(nn.Module):
    def forward(self, x):
        batch_size, seq_len, _ = x.shape
        q, k, v = ...
        queries = ...
        keys = ...
        values = ...

        if self.idx == 0:
            print(keys[7][4][29])
```



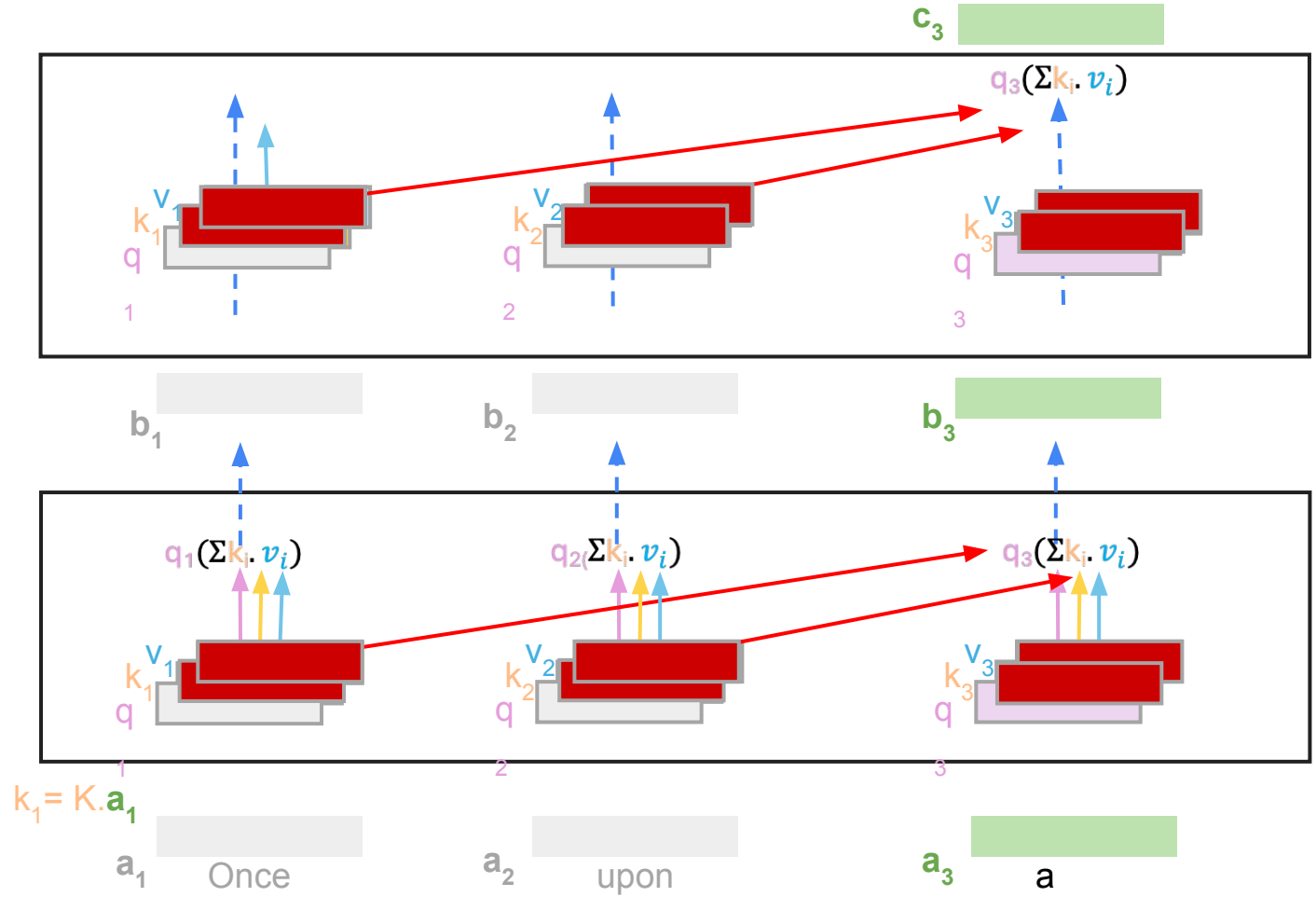
$q_1 = Q.a_1$
b_1 is output after layer 1
For every token, only its q matters
Causal attention: For every token, only its and past tokens k,v matter


c3
Layer 2
q3(Σki. vi)
v1
k1
q
1
v2
k2
q
2
v3
k3
q
3
b1
b2
b3
Layer 1
q1(Σki. vi)
q2(Σki. vi)
q3(Σki. vi)
v1
k1
q
1
v2
k2
q
2
v3
k3
q
3
k1= K.a1
a1
Once
a2
upon
a3
a


c3
q3(Σki. vi)
v1
k1
q
1
v2
k2
q
2
v3
k3
q
3
b1
b2
b3
q1(Σki. vi)
q2(Σki. vi)
q3(Σki. vi)
v1
k1
q
1
v2
k2
q
2
v3
k3
q
3
k1= K.a1
a1
Once
a2
upon
a3
a



We don't pass entire input for every decode run, only the last token



What is the size of the KV cache?

How many KVs are we storing?
Num layers x
Num tokens x
2

Size of K/V?:
Embedding size

Let’s implement KV Cache (gpt.py)

Only 12 lines to be added
Only 9 lines to be changed

1. Create an argument called `kv_cache` to the forward in all the modules calling Attention
   a. GPTLMHead, GPTModel, GPTBlock, GPTAttention

Let's implement KV Cache (gpt.py)

Only 12 lines to be added
Only 9 lines to be changed

1. Create an argument called kv_cache to the forward in all the modules calling Attention
   a. GPTLMHead, GPTModel, GPTBlock, GPTAttention

```
class GPTModel(nn.Module):
    def forward(self, x, kv_cache=None):
        …
class GPTLMHead(nn.Module):
    def forward(self, inputs, position_ids, kv_cache=None):
        …
```

Let’s implement KV Cache (gpt.py)

Only 12 lines to be added
Only 9 lines to be changed

1. Create an argument called kv_cache to the forward in all the modules calling Attention
   b. In GPTModel, kv_cache should be initialized as [None] x n_layers for each block. Now, we’ll pass the kv_cache[i] to each block’s forward call

Let's implement KV Cache (gpt.py)

Only 12 lines to be added
Only 9 lines to be changed

1. Create an argument called kv_cache to the forward in all the modules calling Attention
   b. In GPTModel, kv_cache should be initialized as [None] x n_layers for each block. Now, we'll pass the kv_cache[i] to each block's forward call

```
class CausalModel(nn.Module):
    def forward(self, inputs, position_ids, kv_cache=None):
        if kv_cache is None:
            kv_cache = [None for _ in range(len(self.h))]
        …
          h(x, kv_cache[i])
```

Let’s implement KV Cache (gpt.py:Attention)

2. Init case (Prefill):
    i. kv_cache is passed as None
    ii. Populate it as tuple of (keys, values)

Let’s implement KV Cache (gpt.py:Attention)

2. Init case (Prefill):
    i. kv_cache is passed as None
    ii. Populate it as tuple of (keys, values)

```
class GPTAttention(nn.Module):
    def forward(self, kv_cache=None):
        …
       if kv_cache is None:
          kv_cache = (keys, values)
```

Let’s implement KV Cache (gpt.py:Attention)

3. Decode case: kv_cache has the tuple (keys, values) of past values
    i. Read past_keys (values) from kv_cache. Print shape
    ii. Print shape of current keys
    iii. See which dimension to concatenate to create the merged keys and values `torch.cat((t1, t2), dim)`
    iv. Continue using the new keys and values tensor

Let’s implement KV Cache (gpt.py:Attention)

3. Decode case: kv_cache has the tuple (keys, values) of past values
    i. Read past_keys (values) from kv_cache. Print shape
    ii. Print shape of current keys
    iii. See which dimension to concatenate to create the merged keys and values `torch.cat((t1, t2), dim)`
    iv. Continue using the new keys and values tensor

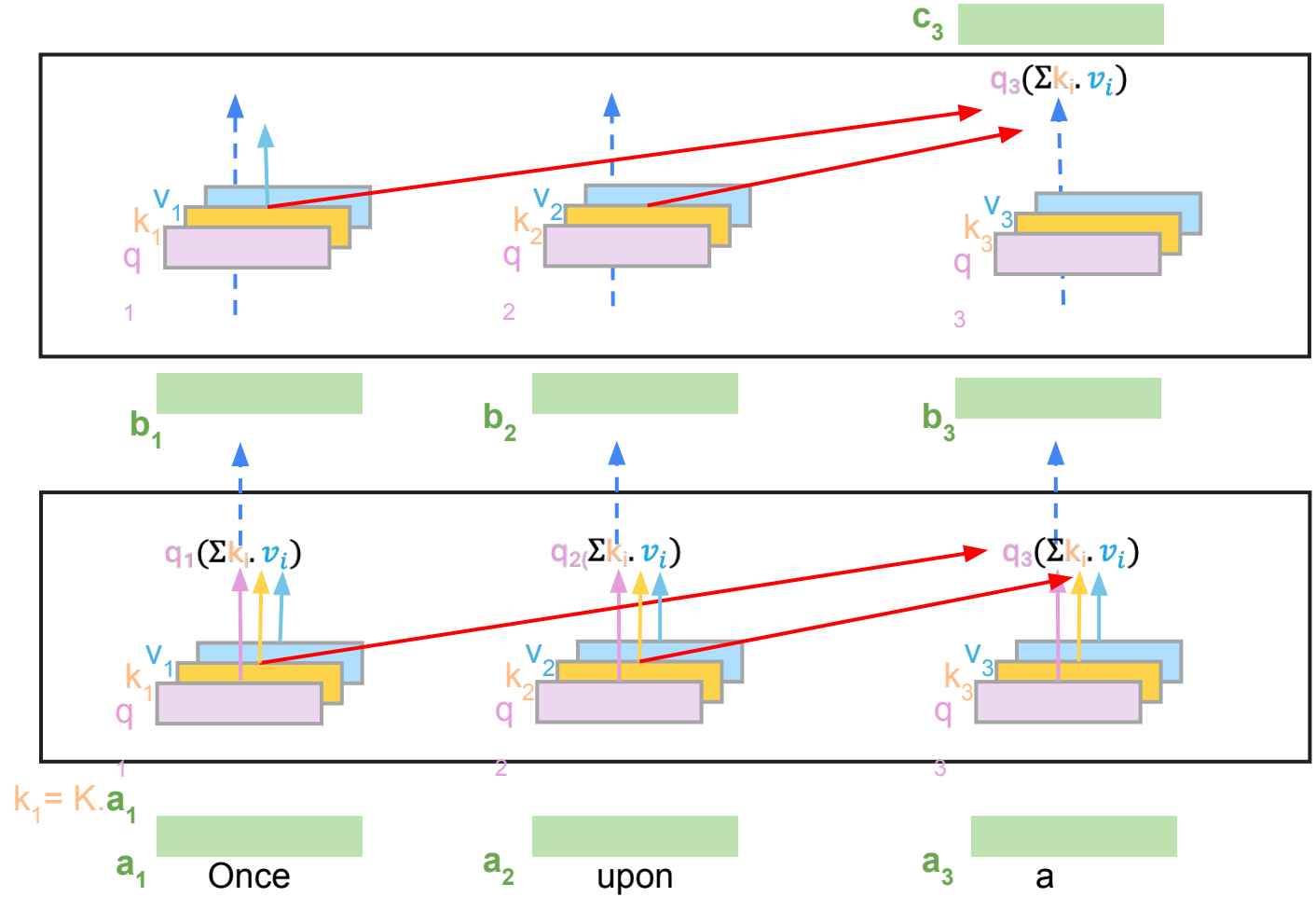
```
class GPTAttention(nn.Module):
    def forward(self, kv_cache=None):
        if kv_cache is not None:
            past_keys, past_values = kv_cache
            keys = torch.cat((past_keys, keys), dim=2)
```

Let’s implement KV Cache (gpt.py)

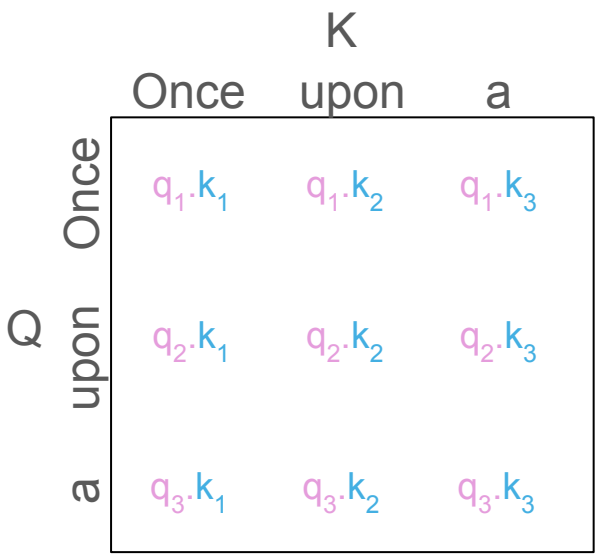
4. Return the populated KV cache at all modules:
    a. Return kv_cache at GPTAttention
    b. Return at GPTBlock
    c. Return at GPTModel (remember each layer will return its own kv_cache, so this has to be a list)
    d. Return at CausalModel back to user

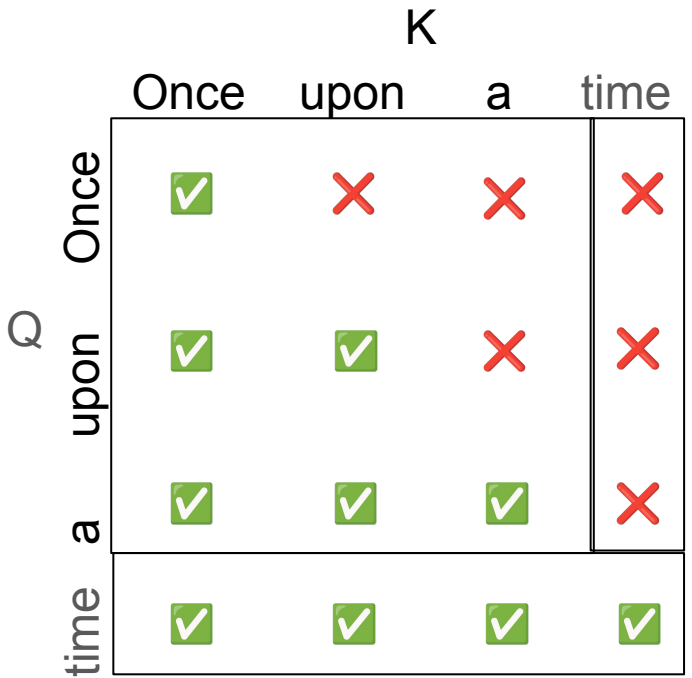
Let’s implement KV Cache

5. Changes at generate:
    a. Separate the prefill and decode calls. Prefill is once, decode is in a loop.
    b. Input: We will not pass the entire input to decode, just the output token of the last forward.
    c. kv_cache output of one call is passed as input to next call
    d. Position ids: What was the old position ids? What should the new one be?

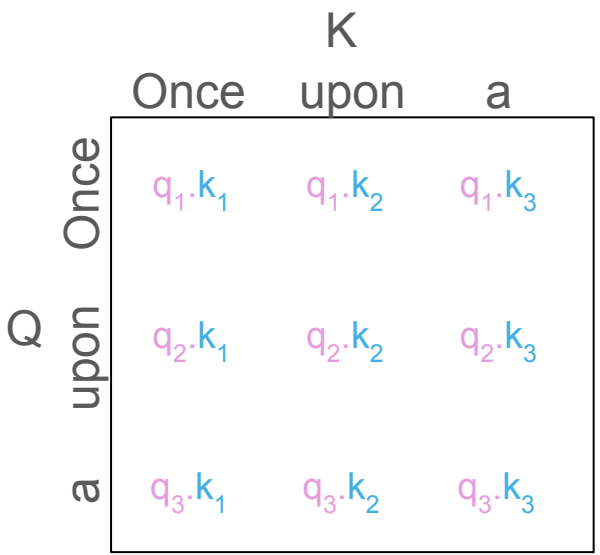

c3
q3(Σki. vi)
v1
k1
q1
v2
k2
q2
v3
k3
q3
b1
b2
b3
q1(Σki. vi)
q2(Σki. vi)
q3(Σki. vi)
k1= K.a1
a1
Once
a2
upon
a3
a

Attention mask

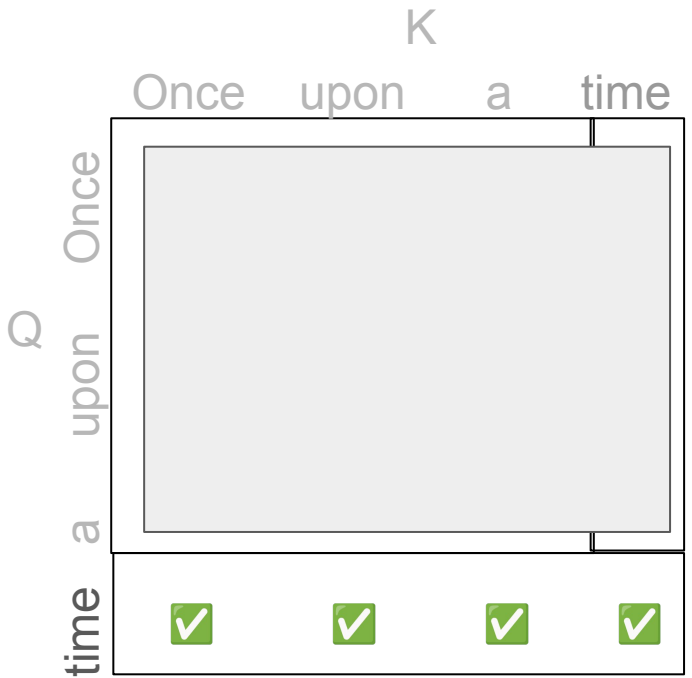




Attention mask



With KV Cache



Attention mask with KV Caching

6. In attn():
    a. “is_causal=True” fills the triangular attention mask by default. We want to turn it off for decode
    b. How to know whether prefill or decode? Use Q.shape and K.shape
    c. If prefill: Use is_causal=True
    d. If decode: Use is_causal=False

If everything goes right, your output should match the previous output

Savings

Run the timed_generate.py in the solutions file (Copy pytorch_model.bin)

You should see benefits of KV Cache for larger number of tokens

Let’s summarize KV Cache

1. Why do we cache KV?
2. What do we save? Compute or memory?
3. How are prefill and decode phases different?
4. What can we do if we run out of memory?

Batching

Batching

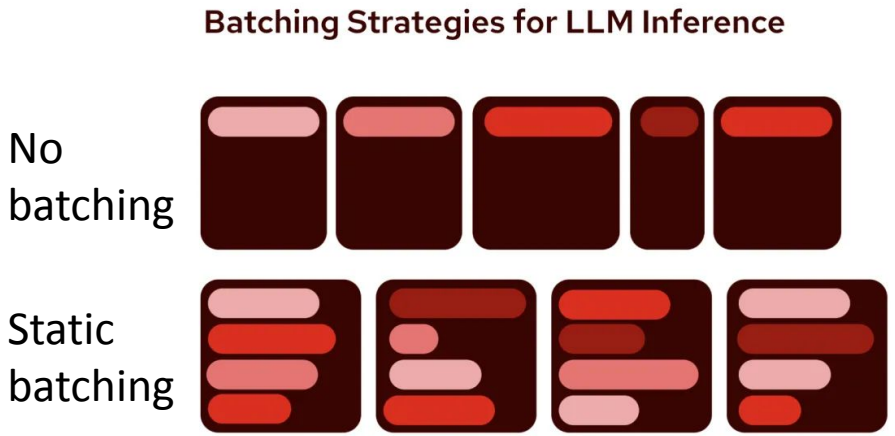
No batching:

- Generate runs 1 request at a time
- GPU not utilized fully

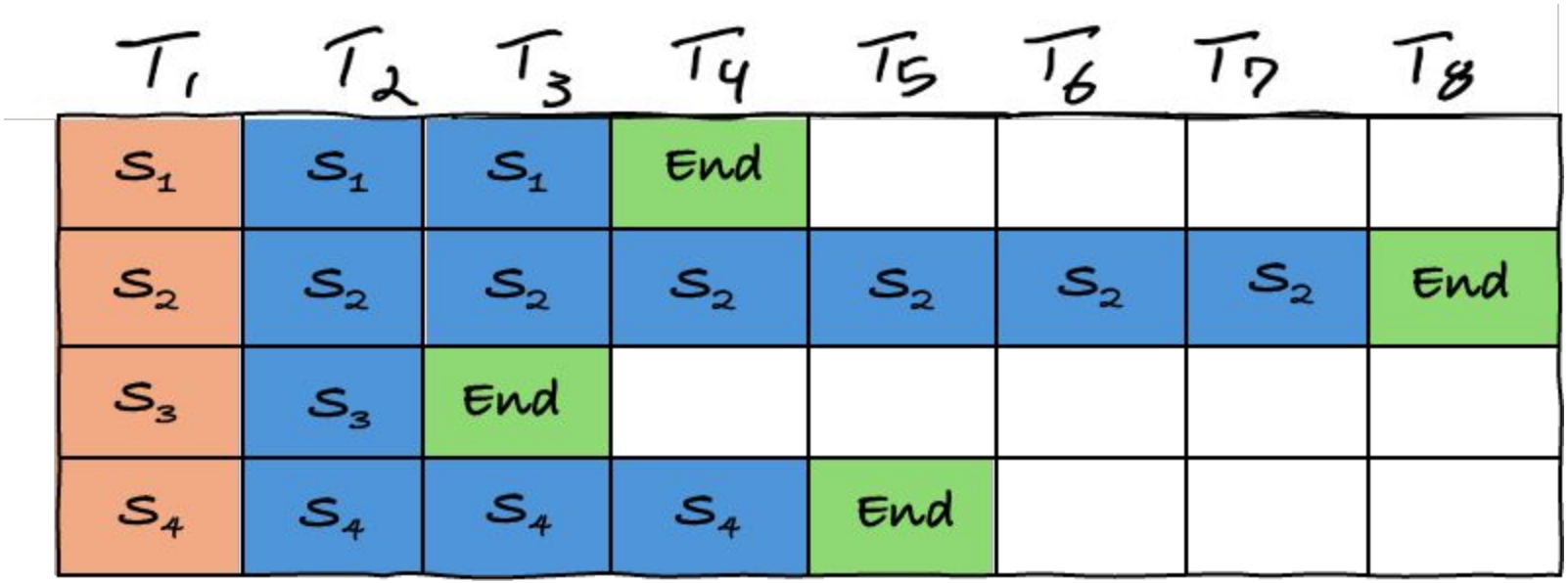
Batching

No batching:
- Generate runs 1 request at a time
- GPU not utilized fully

Static batching:
- Request level batching:
- Batch a set of requests
- More parallelization



Static batching

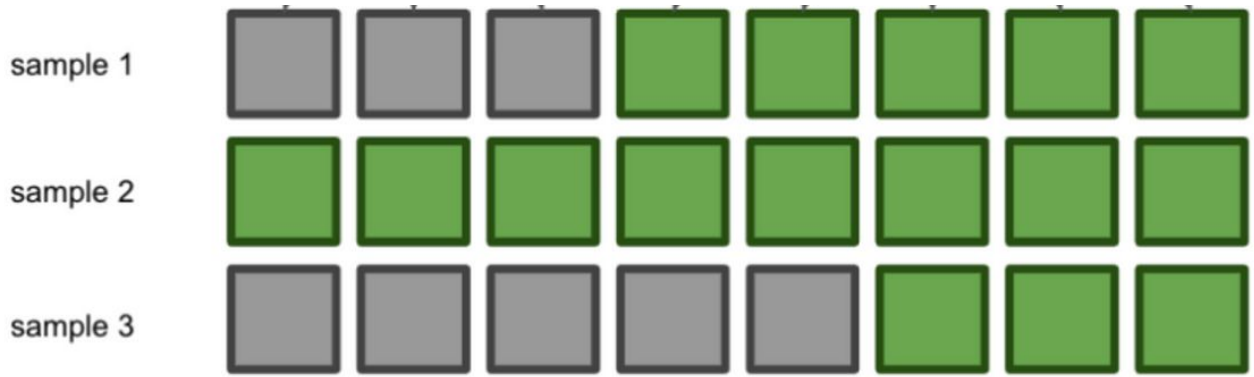


Static Batching

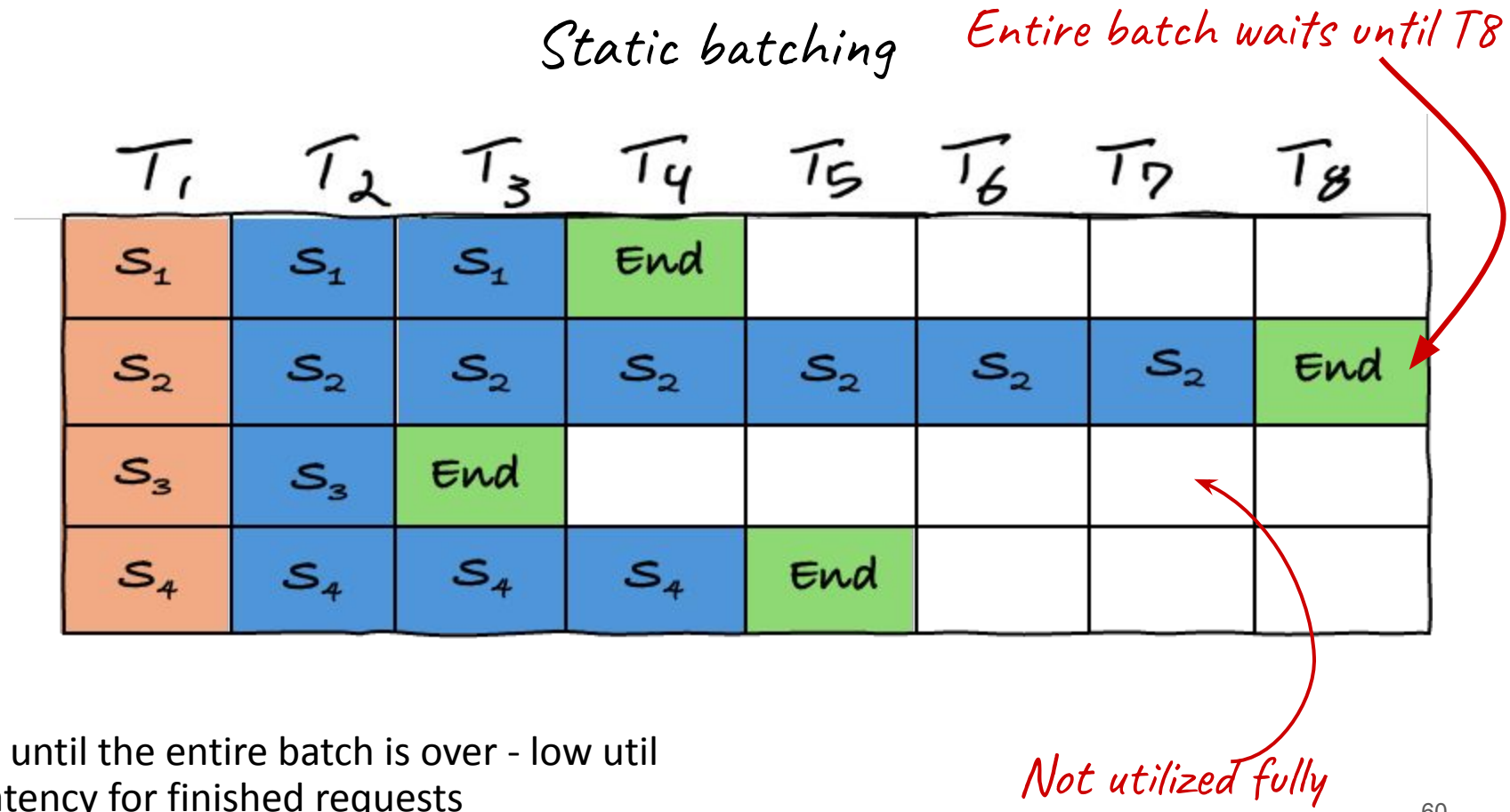
First dimension in inputs is batch size. Previously, we set it to 1.

How to achieve static batching:

1. Batch multiple requests together
2. How to handle different size requests? Padding



What are the problems in this approach?



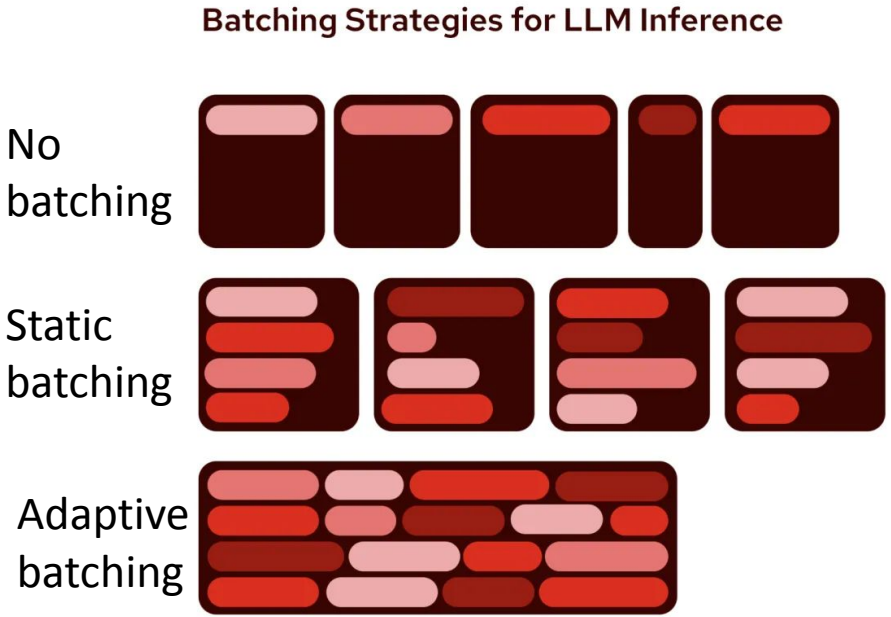
Iterate until the entire batch is over - low util
High latency for finished requests

Continuous Batching

Iterative/Continuous[1] Batching

- Token level batching
- Replace completed requests with new
- Batch size parameter:
  - Throughput vs latency tradeoff

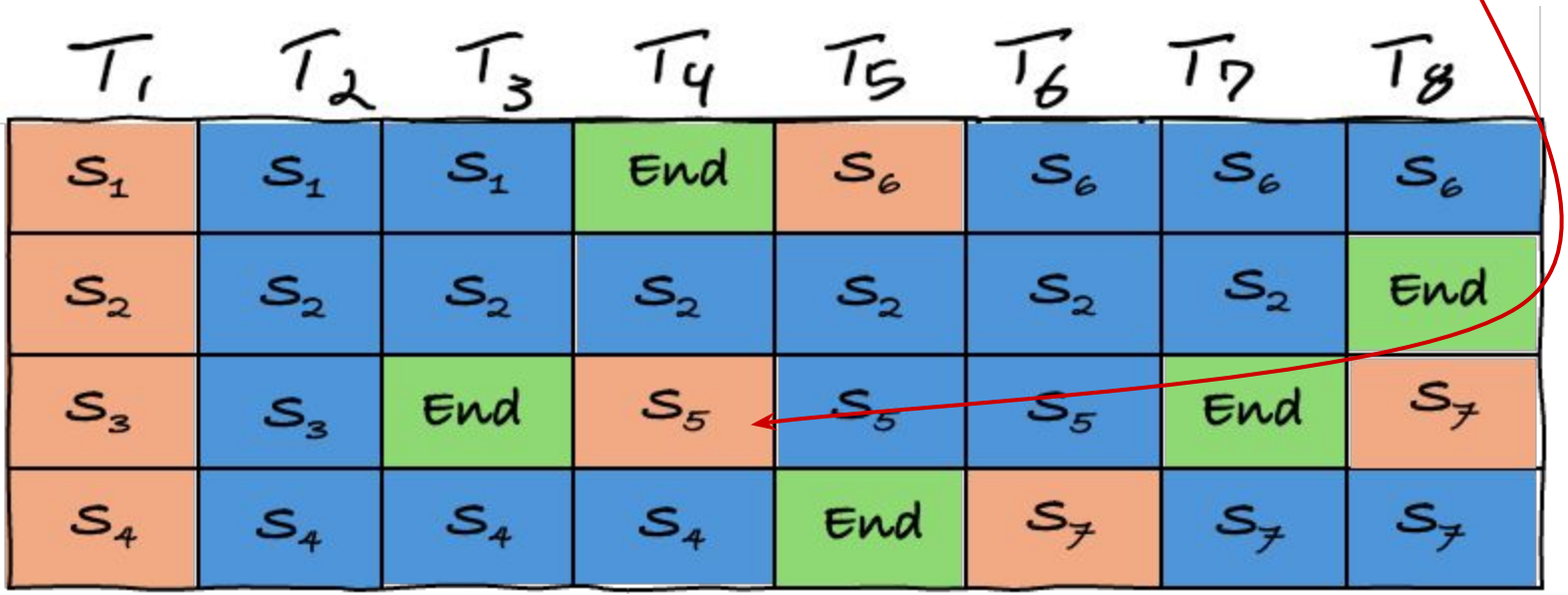
Any problems?



1. *Orca: A Distributed Serving System for Transformer-Based Generative Models* NSDI '22
2. Image source: https://www.redhat.com/en/blog/meet-vllm-faster-more-efficient-llm-inference-and-serving

Continuous batching

Next input starts executing immediately



Let’s summarize batching

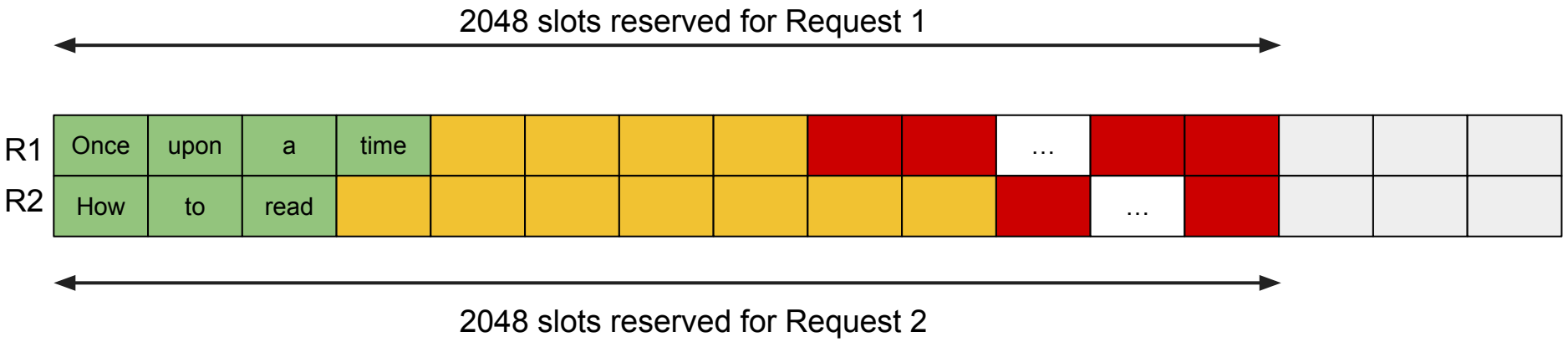
1. 2 types of batching
2. What do we gain by batching?
3. What do we lose?
    a. Multi-step scheduling

Paged Attention

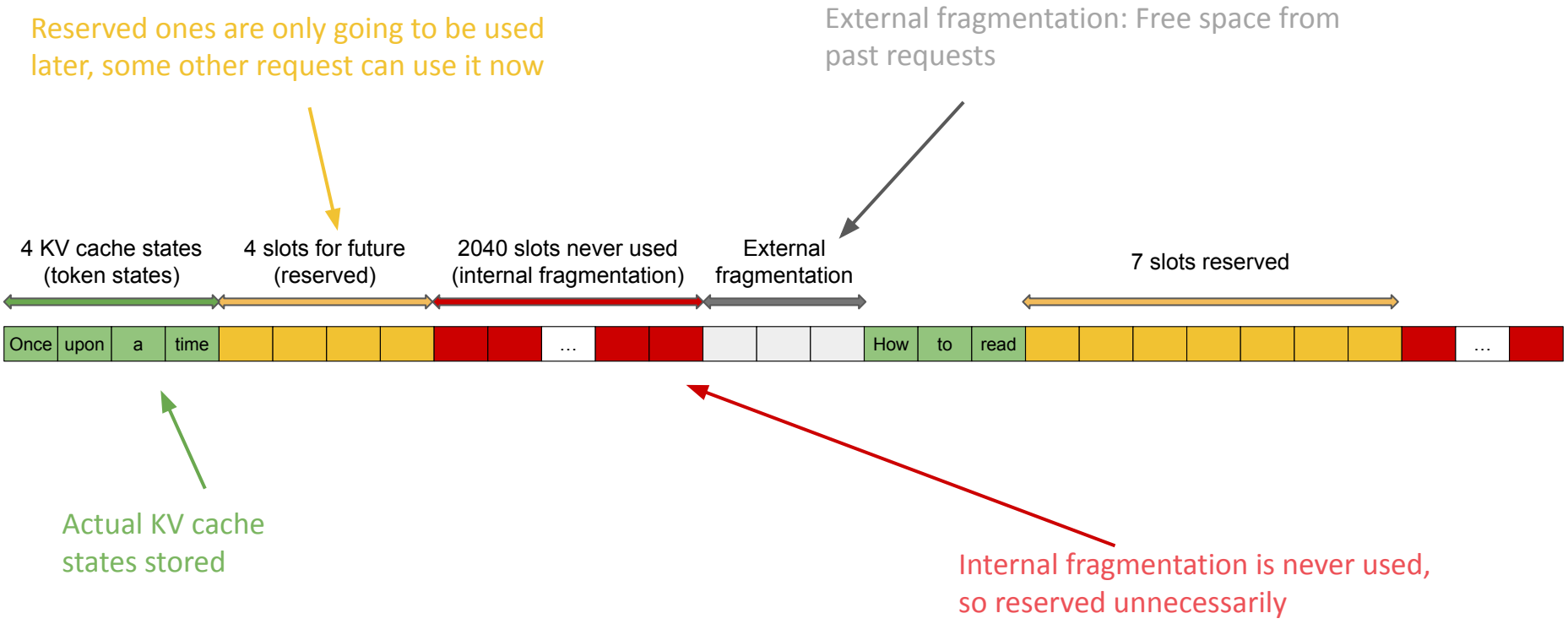
Memory management of KV Cache

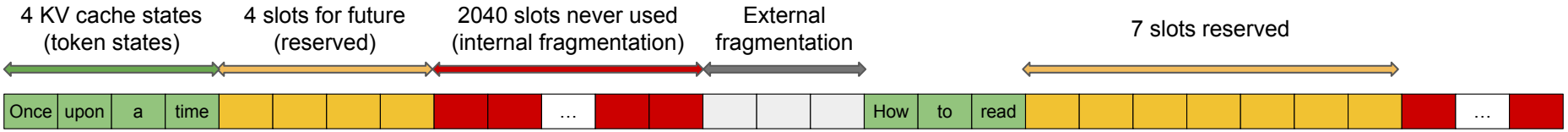
With batching and KV Caching, the KV cache is initialized as a contiguous Tensor for the max sequence length supported by the model

For a long context model (8k - 128k tokens), how much memory does this take?



PagedAttention: https://arxiv.org/pdf/2309.06180


Reserved ones are only going to be used later, some other request can use it now
External fragmentation: Free space from past requests
4 KV cache states (token states)
4 slots for future (reserved)
2040 slots never used (internal fragmentation)
External fragmentation
7 slots reserved
Once
upon
a
time
…
How
to
read
…
Actual KV cache states stored
Internal fragmentation is never used, so reserved unnecessarily



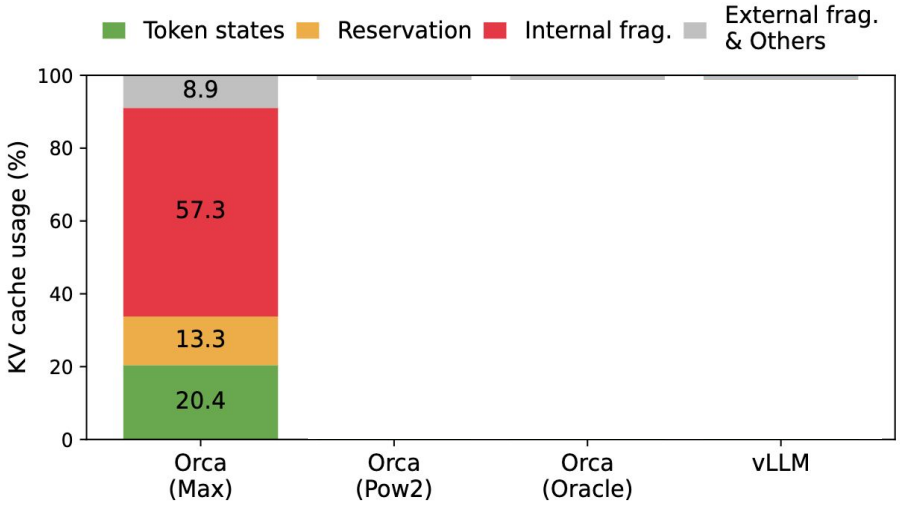
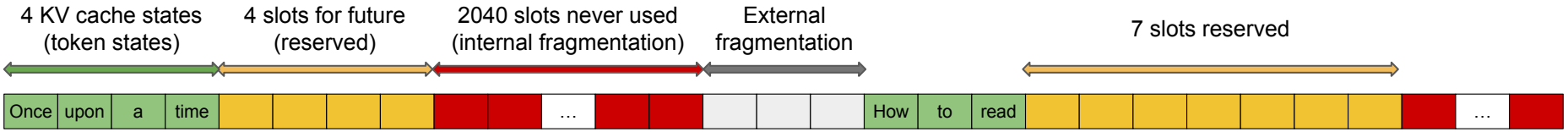


Figure 2. Average percentage of memory wastes in different LLM serving systems during the experiment in §6.2.



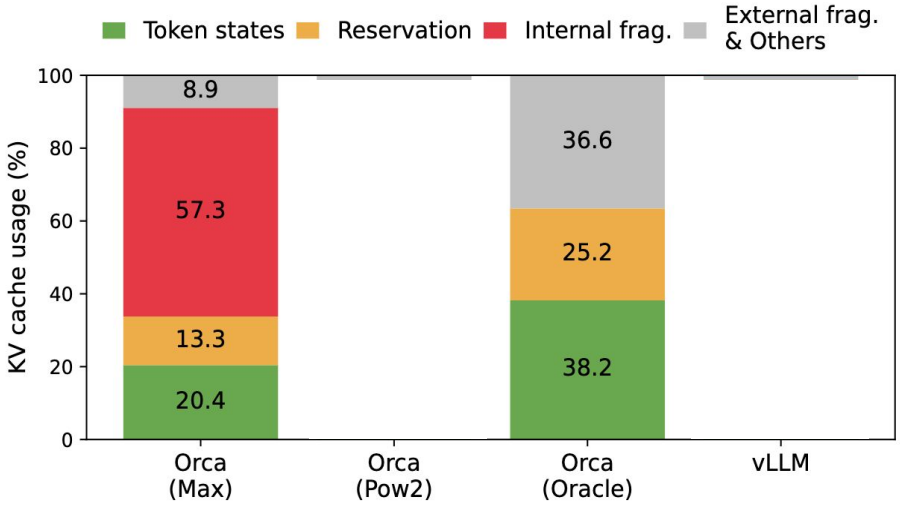
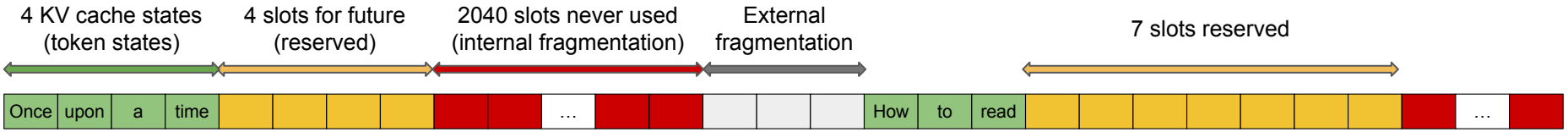


Figure 2. Average percentage of memory wastes in different LLM serving systems during the experiment in §6.2.



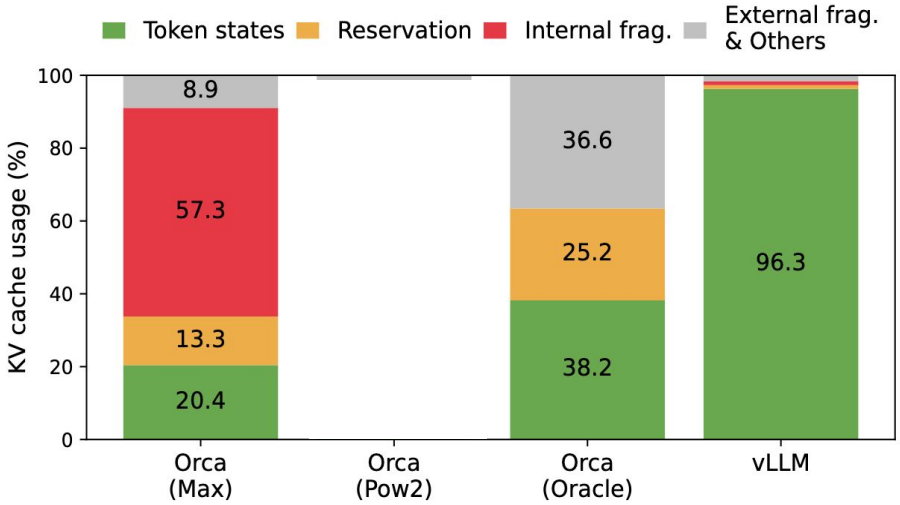
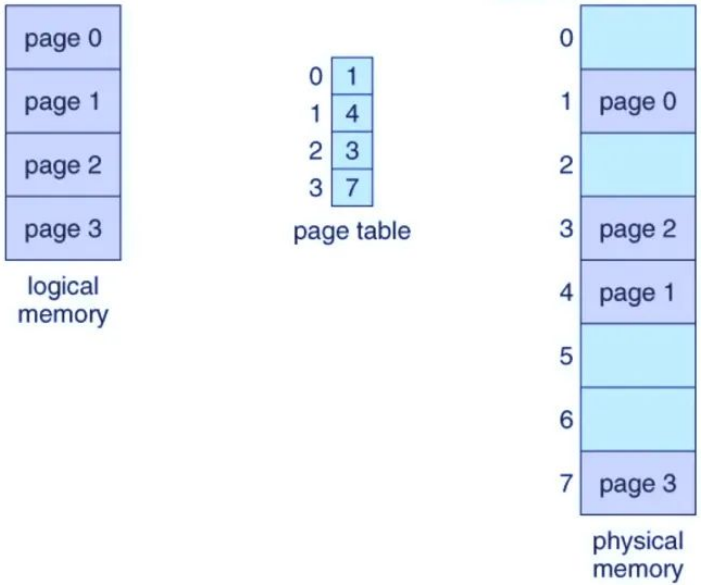
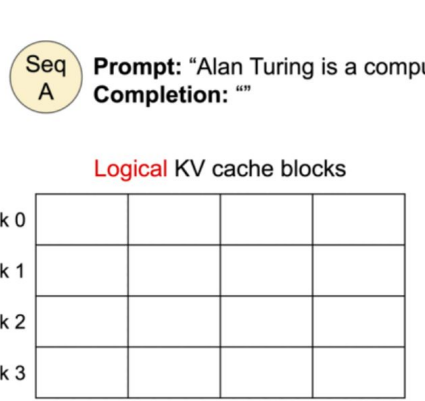
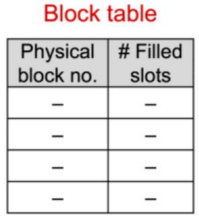


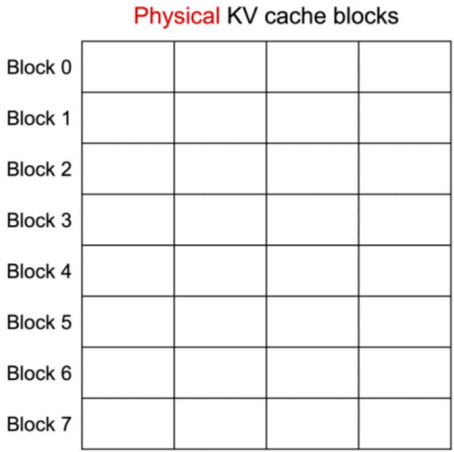
Figure 2. Average percentage of memory wastes in different LLM serving systems during the experiment in §6.2.

How do they do this? Blocks/Pages

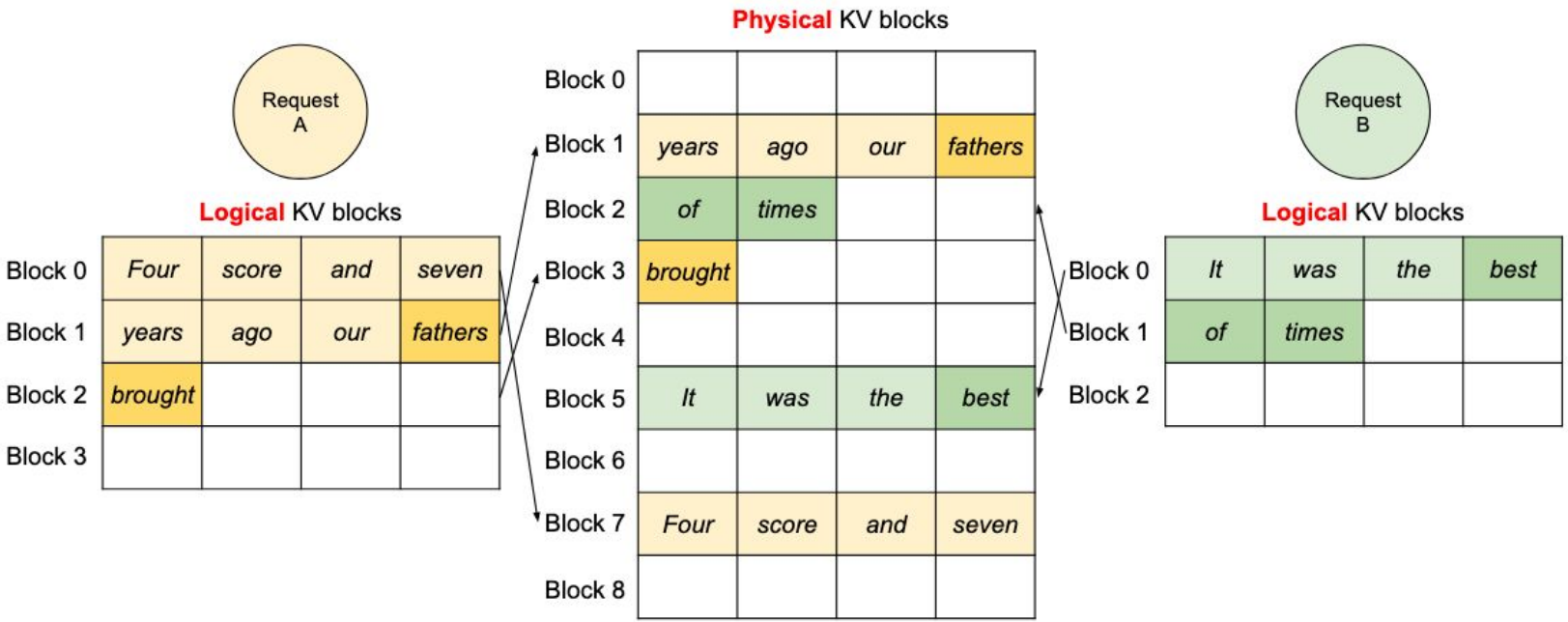


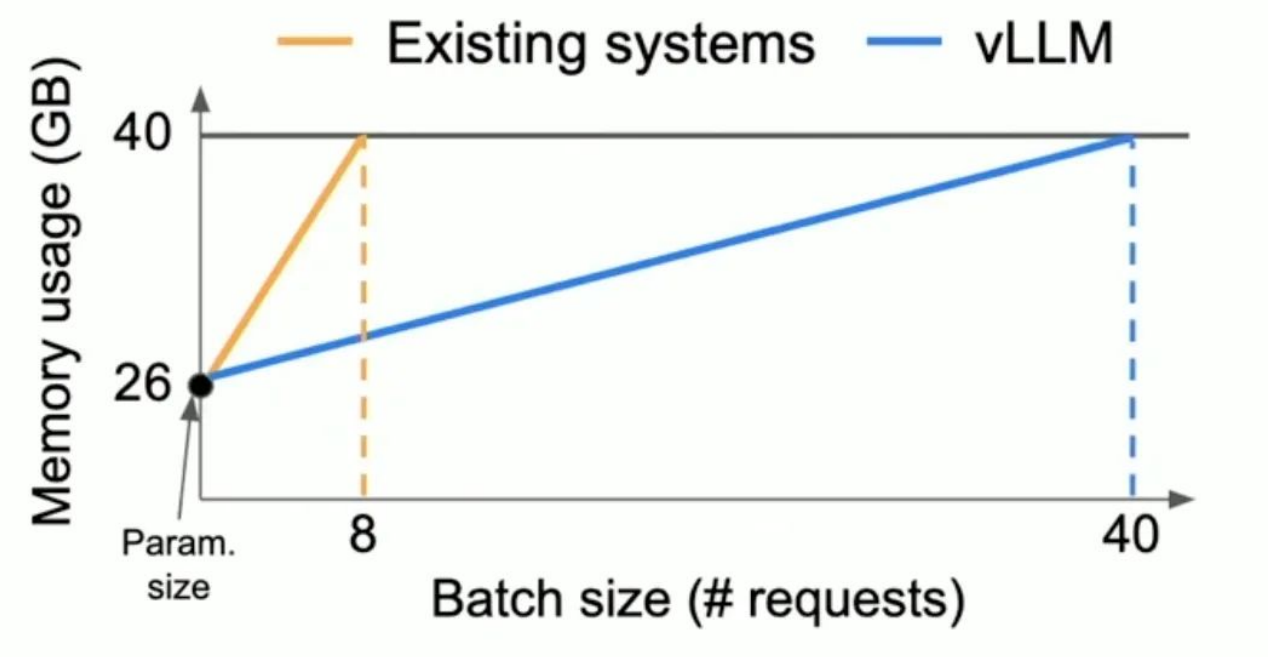






Block - KV Cache data for a few tokens


Request A
Logical KV blocks
Block 0
Four
score
and
seven
Block 1
years
ago
our
fathers
Block 2
brought
Block 3
Physical KV blocks
Block 0
Block 1
years
ago
our
fathers
Block 2
of
times
Block 3
brought
Block 4
Block 5
It
was
the
best
Block 6
Block 7
Four
score
and
seven
Block 8
Request B
Logical KV blocks
Block 0
It
was
the
best
Block 1
of
times
Block 2



2 - 4x throughput gains

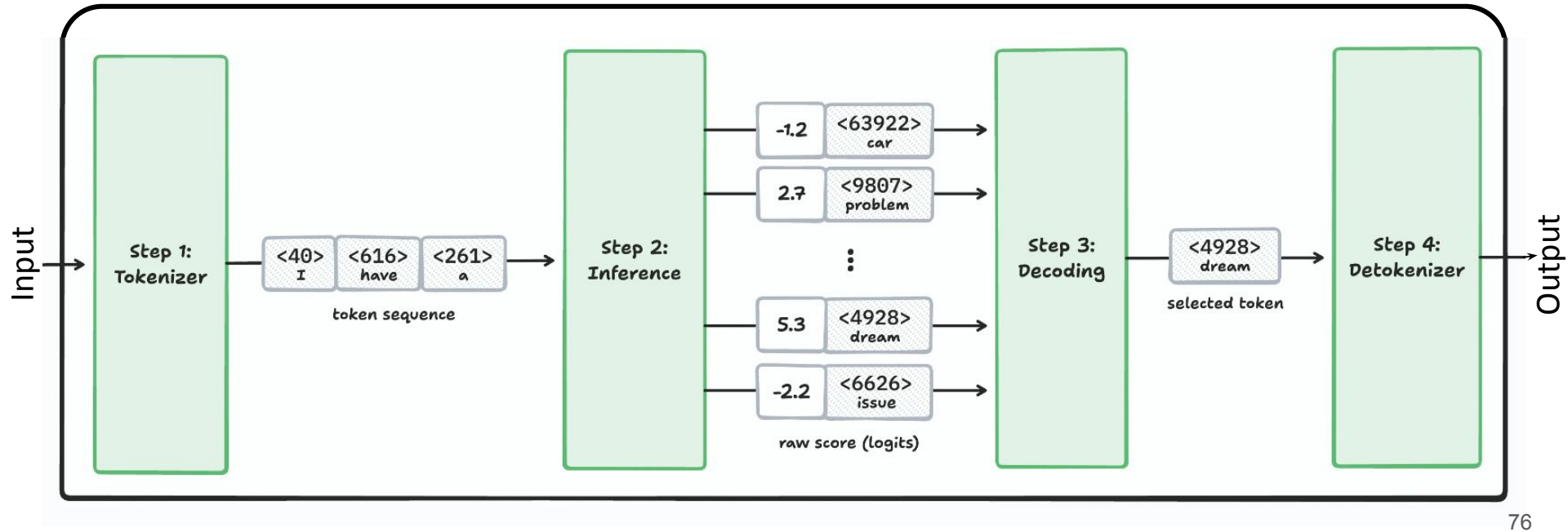
https://www.youtube.com/watch?v=5ZlavKF_98U

Decoding strategies

What is decoding?

What is the output of inference? Not a token!

Logits: A degree of similarity to each token in the vocabulary



https://pm.dartus.fr/posts/2025/how-llm-generate-text/

Decoding Strategies

Which token to predict as response?

```
logits = model(inputs, positions)
print(logits.shape)
logits = logits[-1, :]
```

[x, 50257]

How to convert these logits to a token?

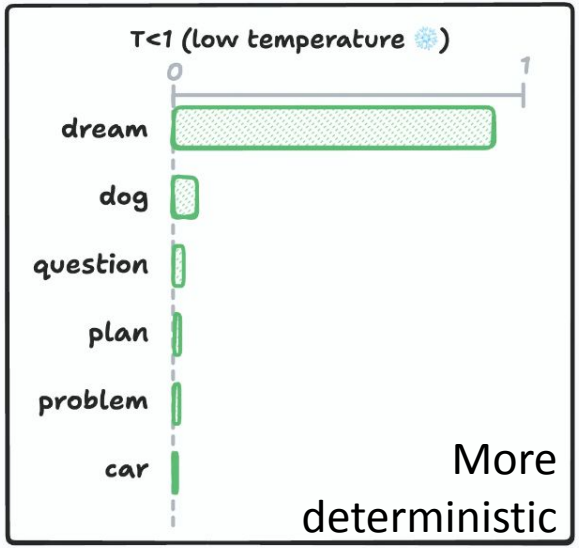
https://huggingface.co/blog/mlabonne/decoding-strategies

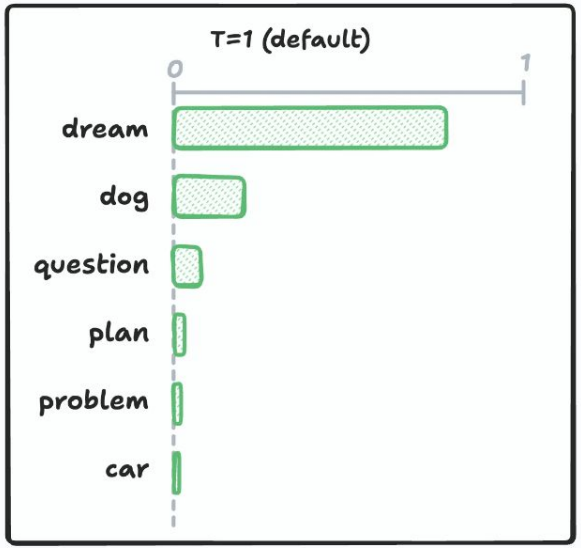
Softmax

Logit (similarity score) -> Probability score (that adds up to 1)

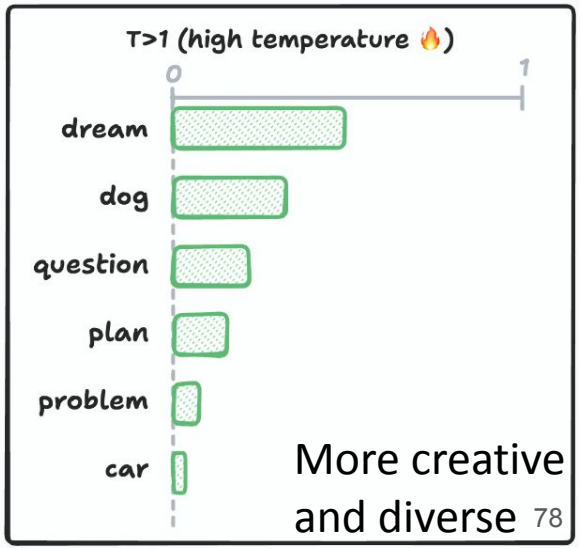
This is done via a softmax normalization function which has a “temperature” parameter which scales the logits value.

Makes values more extreme

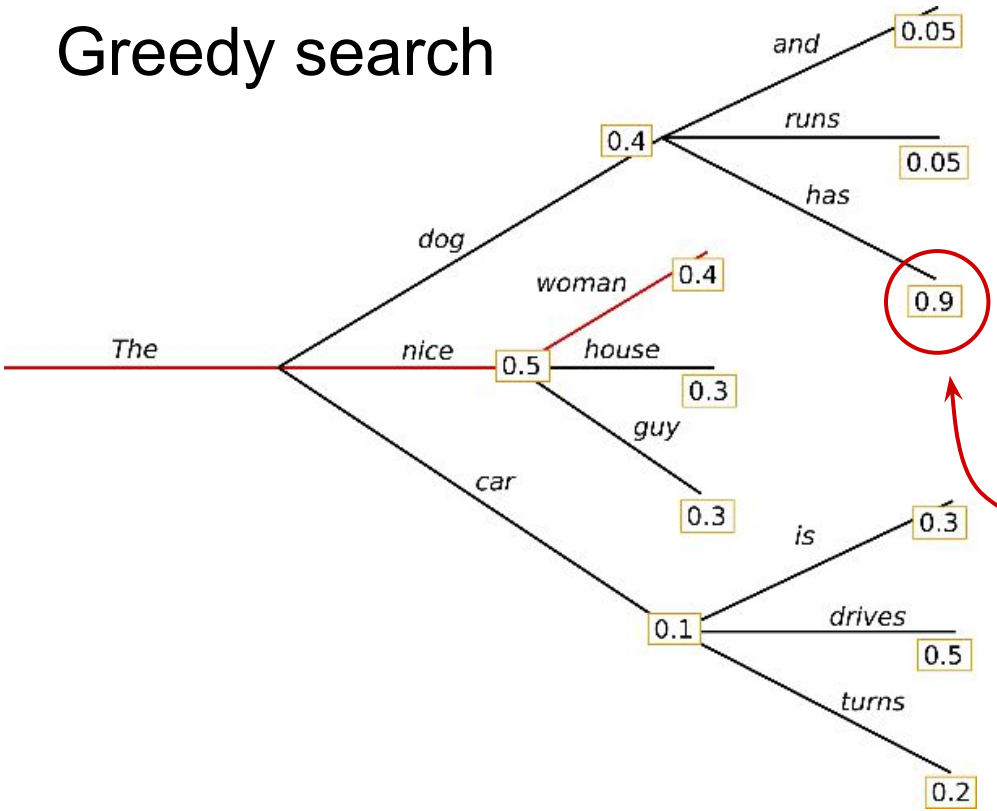




Makes values more centered



Greedy search



Choose word with highest probability at every step

Problems:

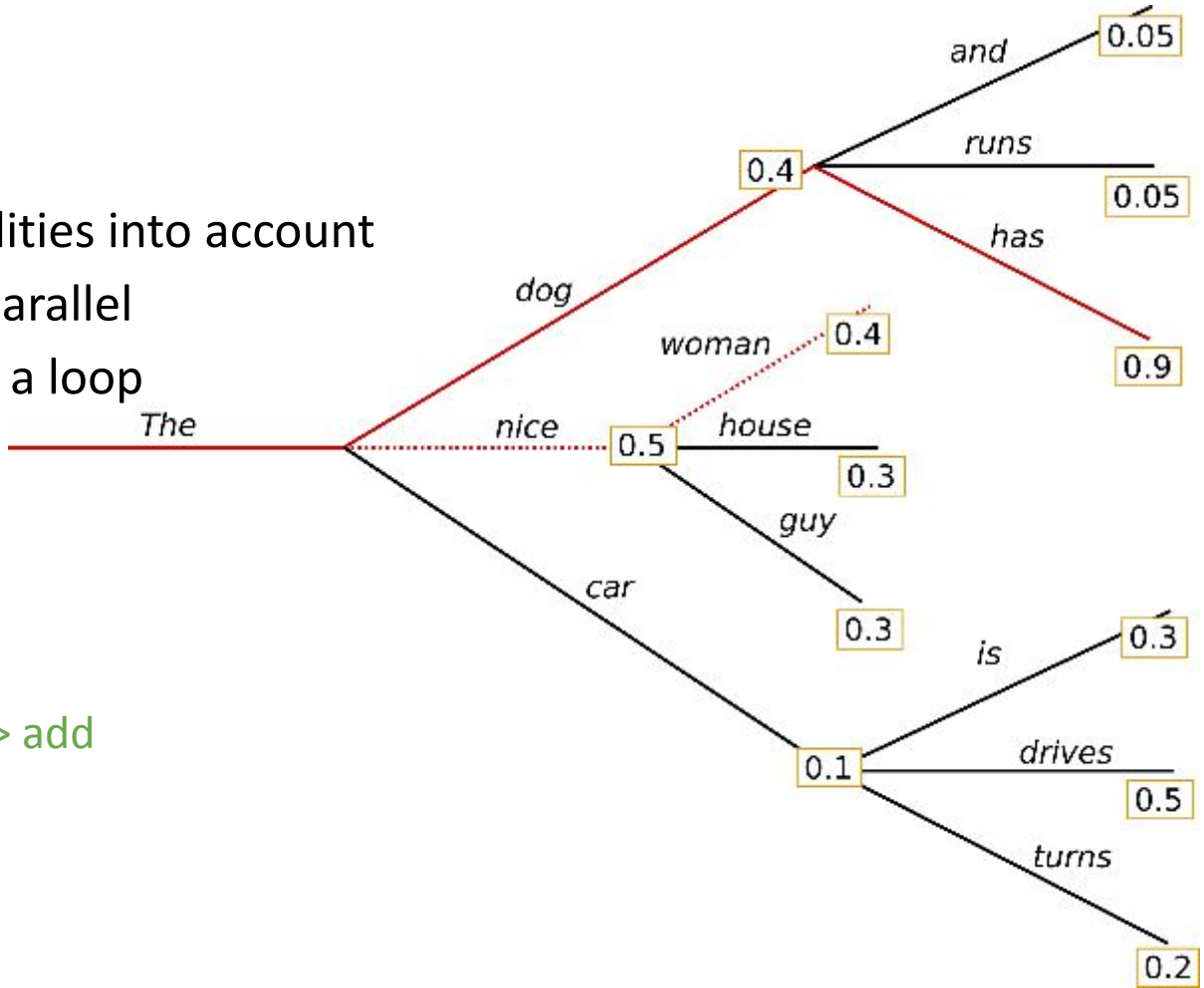
1. It starts repeating itself after some time
2. Misses high prob. words hidden behind low prob. words

```
next_token = torch.argmax(logits, dim=-1, keepdim=True)
```


https://huggingface.co/blog/how-to-generate

Beam search

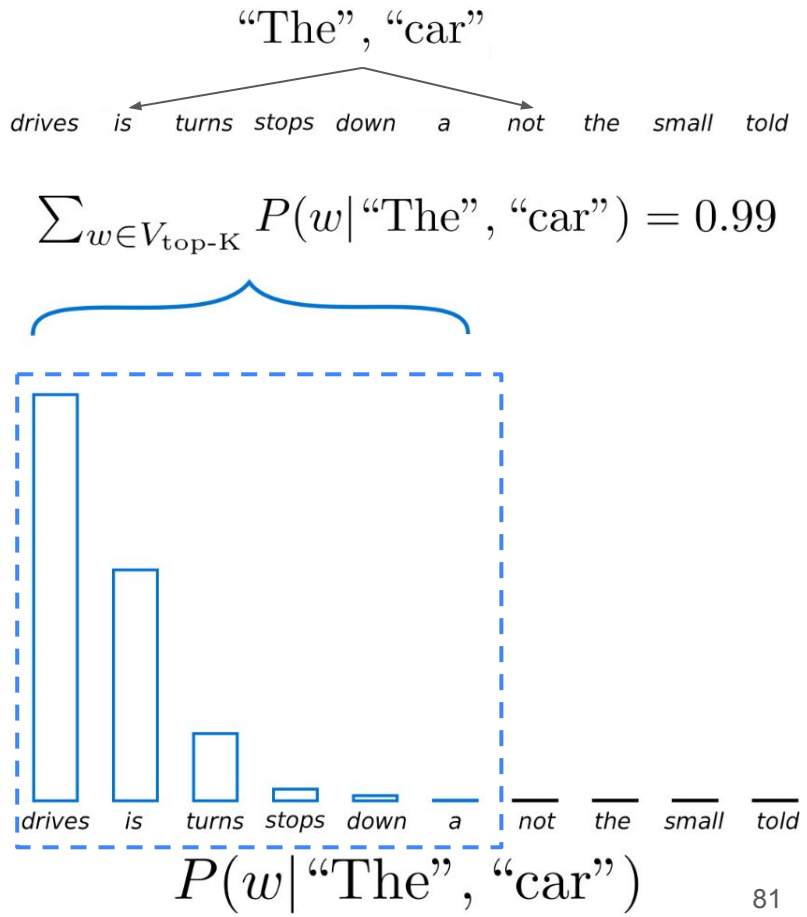
- Takes n top probabilities into account
- Runs n decodes in parallel
- Can still get stuck in a loop



How to avoid repetition -> add randomness

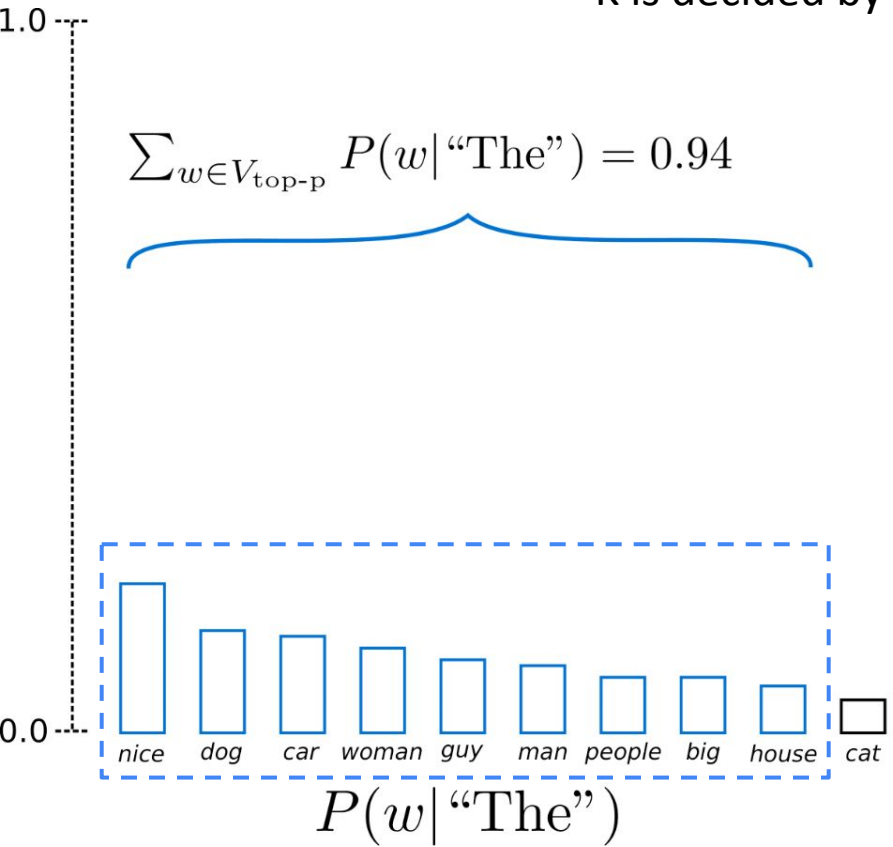
Top-K Sampling

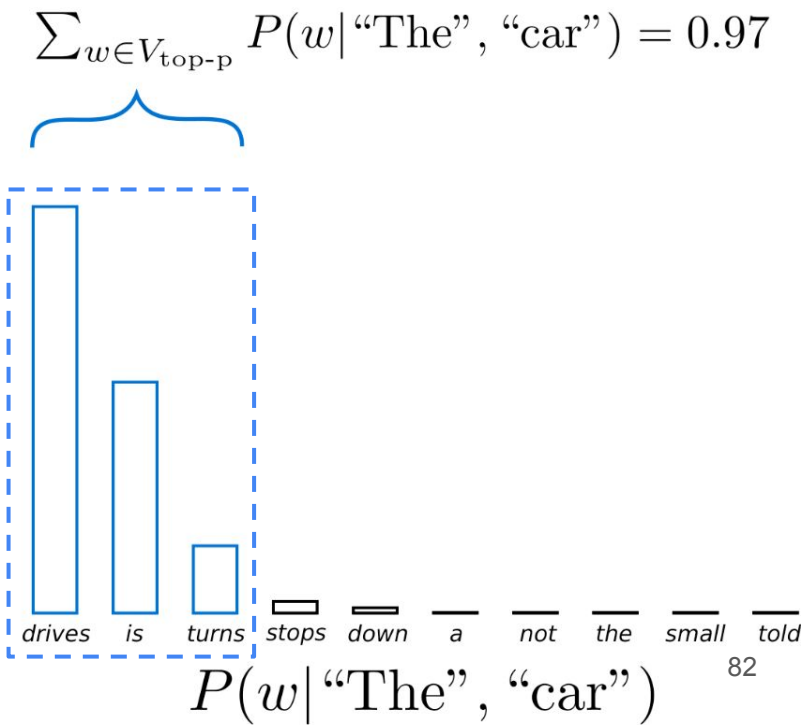
- Randomly pick one of the output tokens
- But this could bias towards the long tail of irrelevant options
- Select only top-K of those and normalize the probability
- How to decide K?



Top-p sampling

K is decided by number of options that add up to a probability p





Let’s see effect of temperature

```
from transformers import pipeline
prompt = "The quick brown fox jumps over the"
generator(prompt, max_new_tokens=20, do_sample=True, temperature=0.7)
generator(prompt, max_new_tokens=20, do_sample=True, temperature=1.5)
generator(prompt, max_new_tokens=20, do_sample=True,
temperature=0.00001)
```

Research survey

Prefix Caching

If prefix is the SAME for multiple requests, they can share the KV Cache.

Techniques to efficiently match prefix and load its KV Cache

- Hash based
- Radix Tree based

Should be fast and not affect the latency in case of non-match

Where do you think this would be useful?

KV Cache size reduction

Problem: KV Cache can grow several times larger than model size

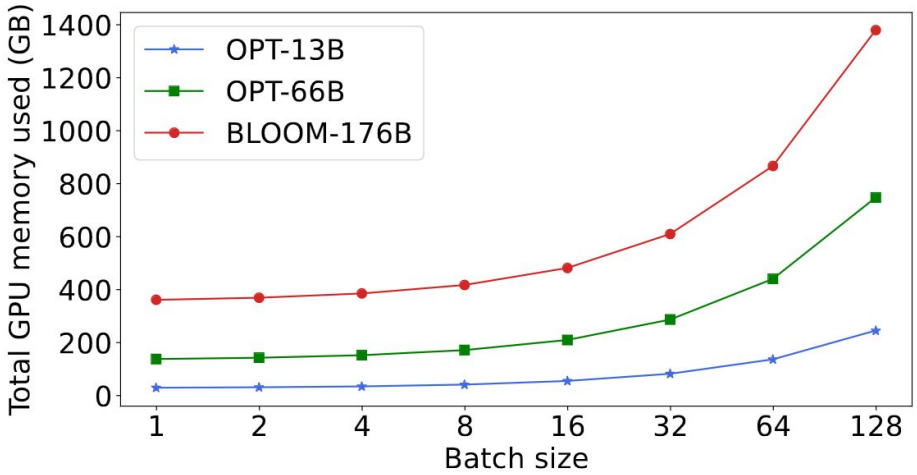


Figure 1. Memory footprint of serving various LLMs with 2K sequence length (input + generated tokens) and half precision (fp16).

DejaVu

How to reduce KV Cache size: Any ideas?

Sparsification:

- Not all values are equally important - Use attention mask to determine which tokens actually "influence" others

Sliding window attention:

- Only consider last N tokens

Speculative Decoding (Draft model)

Problem: Auto-regressive nature of generation means only 1 token is decoded in one iteration

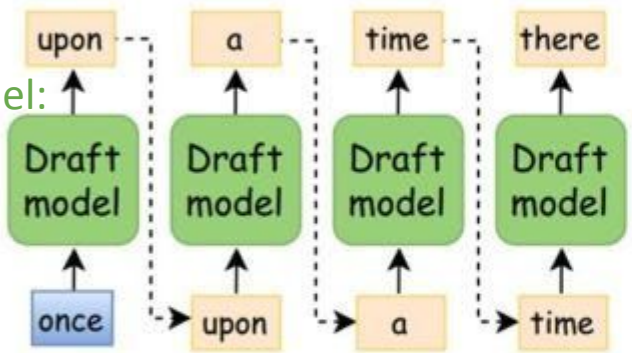
Reduce latency without compromising output correctness!

Speculative Decoding (Draft model)

Problem: Auto-regressive nature of generation means only 1 token is decoded in one iteration

Reduce latency without compromising output correctness!

Small model: 50 - 200M params



Step 1: Draft model quickly generates 4-5 tokens (auto-regressively)

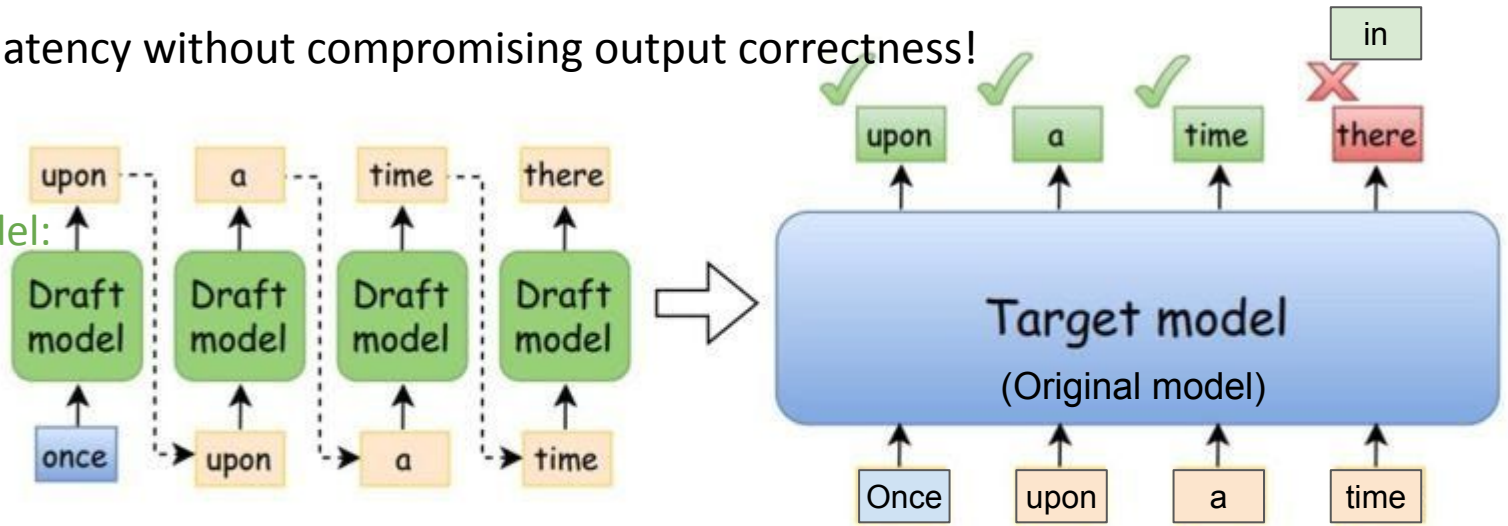
https://medium.com/@genai.works/speed-up-llm-inference-with-speculative-decoding-1fc79701e9d6

Speculative Decoding (Draft model)

Problem: Auto-regressive nature of generation means only 1 token is decoded in one iteration

Reduce latency without compromising output correctness!

Small model: 50 - 200M params



Step 1: Draft model quickly generates 4-5 tokens (auto-regressively)

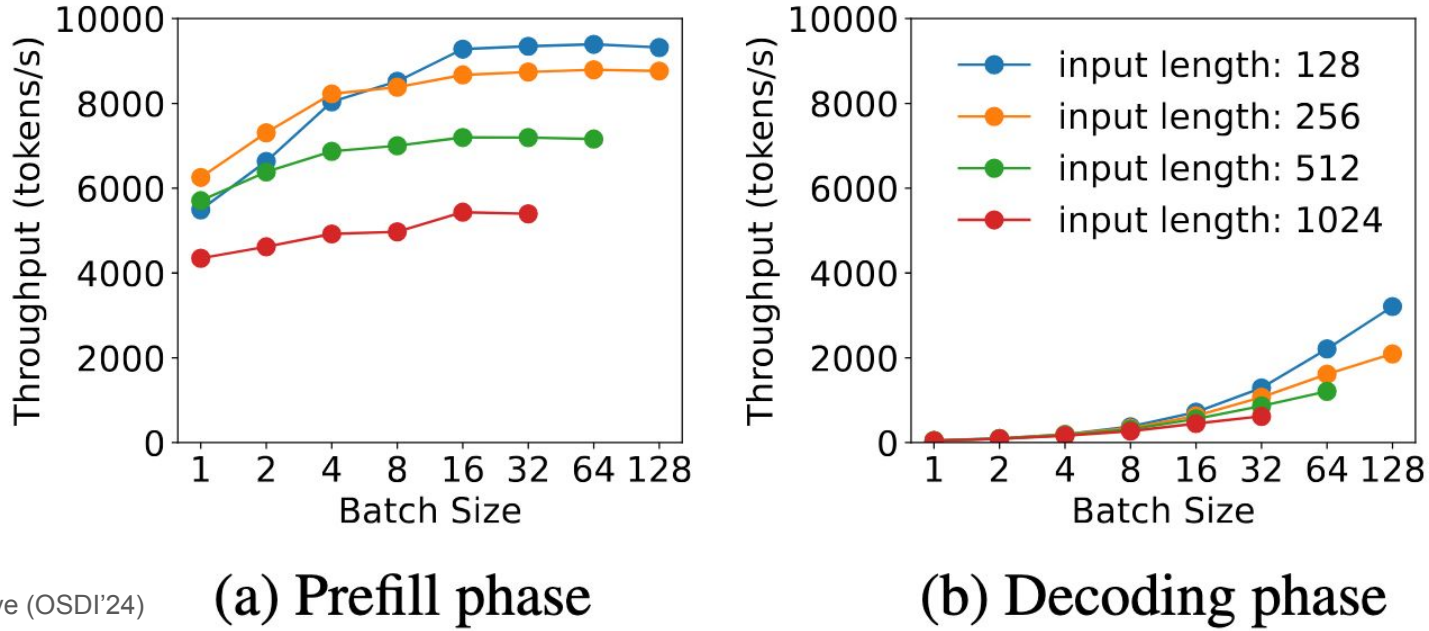
Step 2: Target model quickly verifies these tokens in parallel. Worst case, re-run from that point

https://medium.com/@genai.works/speed-up-llm-inference-with-speculative-decoding-1fc79701e9d6

Prefill-Decode Disaggregation

Problem: Prefill is compute intensive, decode is memory intensive

Not optimal to run both with similar parameters / on the same system

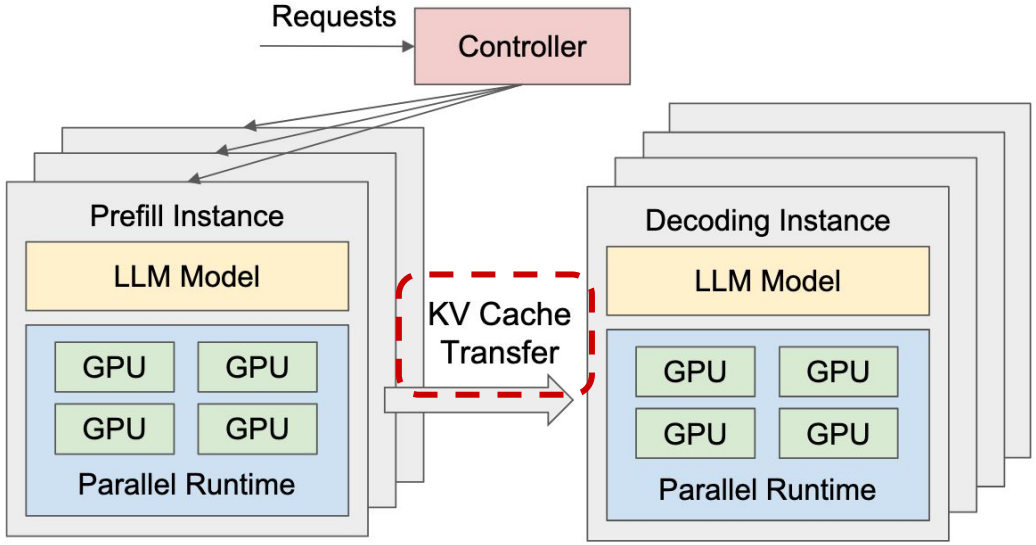


(a) Prefill phase (b) Decoding phase

Graph from DistServe (OSDI'24)

Prefill-Decode Disaggregation

Solution: Run them on different machines



Overlap communication with computation
Stream the KV Cache

Figure from DistServe (OSDI'24)

Several prior art: DistServe, Llumnix, LMCache, DejaVu