

# KVCache: A Source Code Perspective

## (W7) Weekly Report

Nan Lin

Shanghai Jiao Tong University

2024-08-26

# Outline

Overview of the Architecture

Embedding

Transformer Decoder

Final Step

Rerun the Workflow: KVCache Perspective

Bibliography

# Outline

## Overview of the Architecture

## Embedding

## Transformer Decoder

## Final Step

## Rerun the Workflow: KVCache Perspective

## Bibliography

# Architecture of a Transformer Encoder-Decoder Structure [1]

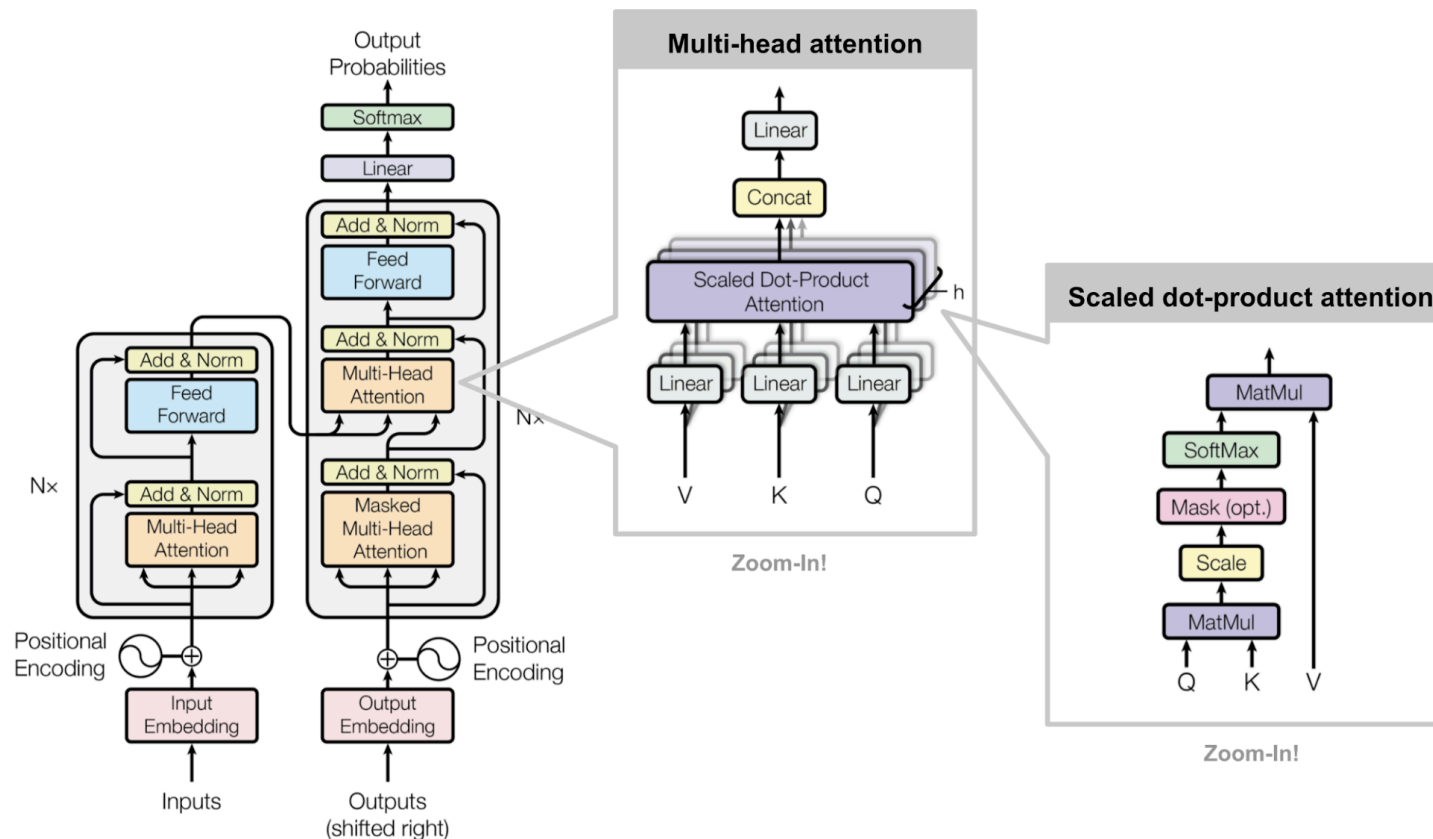


Figure 1: Architecture of a Transformer Encoder-Decoder Structure

# Architecture of GPT-2 Decoder

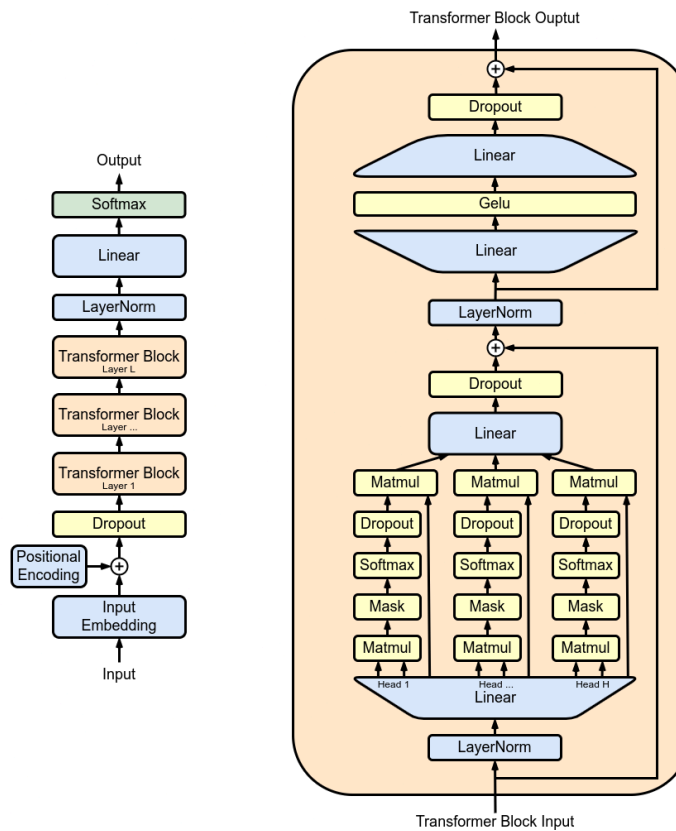


Figure 2: Architecture of GPT-2 Decoder

# Implementation

Several Class:

- 1 `GPT2Model(nn.Module)` (The overall architecture of GPT-2, which consists of multiple stacked ``GPT2Block`` layers.)
  - └─ `GPT2Block` (A modular unit within GPT-2 that processes the input data.
- 2 Each ``GPT2Block`` consists of a ``GPT2Attention`` layer, a ``GPT2MLP`` layer and some other components)
  - 3 └─ `GPT2Attention` (Manages self-attention)
  - 4 └─ `GPT2MLP` (A feedforward neural network)

# Implementation

In GPT2Model: the input goes through

```
1 |— Embedding
2 |— Multiple blocks encapsulated in GPT2Block:
3     self.h = nn.ModuleList([GPT2Block(config, layer_idx=i) for i in
4     range(config.num_hidden_layers)])
5     |— For each block:
6         for i, (block, layer_past) in enumerate(zip(self.h,
7         past_key_values)):
8             |— outputs = block(
9             layer_past=layer_past, ...
10             )
11     |— Final output: LinearNorm, transformation...
```

# Outline

Overview of the Architecture

**Embedding**

Transformer Decoder

Final Step

Rerun the Workflow: KVCache Perspective

Bibliography



# Token Embedding and Positional Embedding [2], [3]

## Token Embeddings (wte)

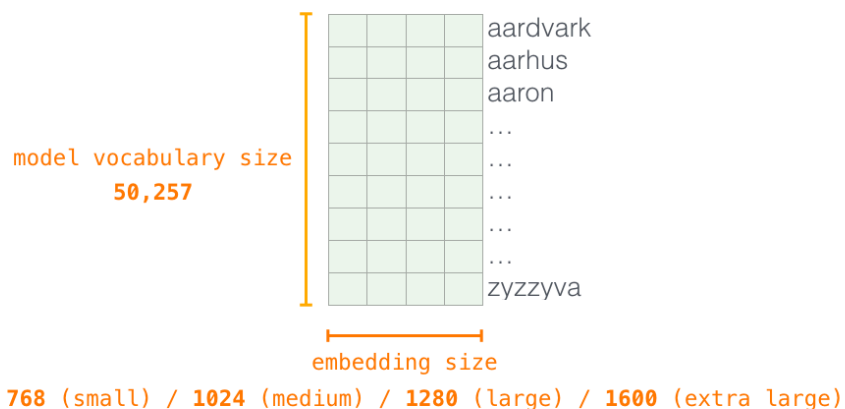


Figure 3: Each row is a word embedding: a list of numbers representing a word and capturing some of its meaning. The size of that list is different in different GPT2 model sizes. The smallest model uses an embedding size of 768 per word/token.

## Positional Encodings (wpe)

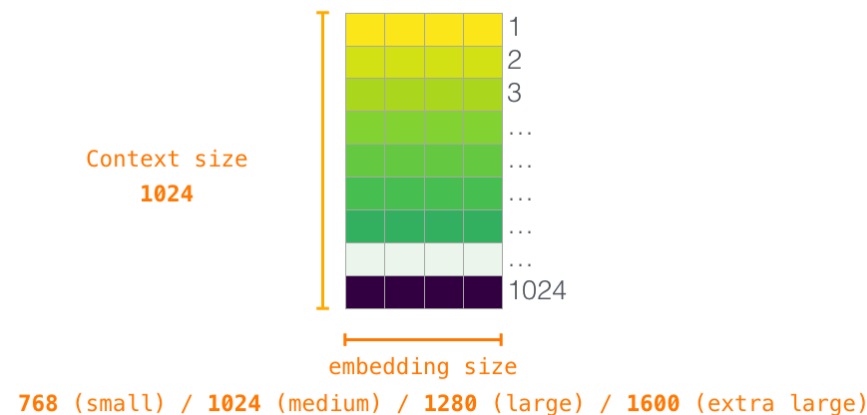


Figure 4: Positional encoding – a signal that indicates the order of the words in the sequence to the transformer blocks. Part of the trained model is a matrix that contains a positional encoding vector for each of the 1024 positions in the input.

# Token Embedding and Positional Embedding [2], [3]



Figure 5: Sending a word to the first transformer block means looking up its embedding and adding up the positional encoding vector for position #1.

# Token Embedding and Positional Embedding [2], [3]

```
1  class GPT2Model(GPT2PreTrainedModel):python
2      def __init__(self, config):
3          super().__init__(config)
4          self.wte = nn.Embedding(config.vocab_size, self.embed_dim)
5          self.wpe = nn.Embedding(config.max_position_embeddings,
6          self.embed_dim)
7
8      def forward(
9          input_ids: Optional[torch.LongTensor] = None,
10         position_ids: Optional[torch.LongTensor] = None,
11         ...
12     ):
```

# Token Embedding and Positional Embedding [2], [3]

```
12         if position_ids are none:
13             position_ids = torch.arange(past_length, input_shape[-1] +
past_length, dtype=torch.long, device=device)
14             position_ids = position_ids.unsqueeze(0)
15
16         if inputs_embeds is none:
17             inputs_embeds = self.wte(input_ids)
18
19         position_embeds = self.wpe(position_ids)
20         hidden_states = inputs_embeds + position_embeds
21         ...
```

# Outline

Overview of the Architecture

Embedding

**Transformer Decoder**

Final Step

Rerun the Workflow: KVCache Perspective

Bibliography

# Decoder Block

Several explanation:

- `past_key_values` is the representation of KVCache
- `layer_past` is an element of KVCache, each representing the calculation result of the last block. `layer_past[0]` is the K-cache and `layer_past[1]` is the V-cache
- `presents` is updated by `presents = presents + (outputs[1], )`

# Decoder Block

```
1  class GPT2Model(GPT2PreTrainedModel):python
2      def __init__(self, config):
3          super().__init__(config)
4          self.h = nn.ModuleList([GPT2Block(config, layer_idx=i) for i in
5                                  range(config.num_hidden_layers)])
6
7      def forward(
8          self,
9          past_key_values: Optional[Tuple[Tuple[torch.Tensor]]] = None,
10         encoder_hidden_states: Optional[torch.Tensor] = None,
11         use_cache: Optional[bool] = None,
12         output_hidden_states: Optional[bool] = None,
```

# Decoder Block

```
12         return_dict: Optional[bool] = None,
13         ...
14     ) -> Union[Tuple, BaseModelOutputWithPastAndCrossAttentions]:
15         # After embedding .....
16         use_cache = use_cache if use_cache is not None else
self.config.use_cache
17         """
18         past_key_values is the representation of KVCache
19         If it is the first iteration, we should first create the KVCache
variable list which dimension should be [None] times num_layer.
20         """
21         if past_key_values is None:
```



# Decoder Block

```
22         past_length = 0
23         past_key_values = tuple([None] * len(self.h))
24         presents = () if use_cache else None
25
26         for i, (block, layer_past) in enumerate(zip(self.h,
27             past_key_values)):
28             """
29             layer_past is an element of KVCache, each representing a single
30             block
31             """
32             outputs = block(
33                 hidden_states,
```

# Decoder Block

```
32         layer_past=layer_past,  
33         use_cache=use_cache,  
34         output_attentions=output_attentions,  
35         ...  
36     )  
37  
38     hidden_states = outputs[0]  
39     if use_cache is True:  
40         presents = presents + (outputs[1],)
```

# Decoder Block

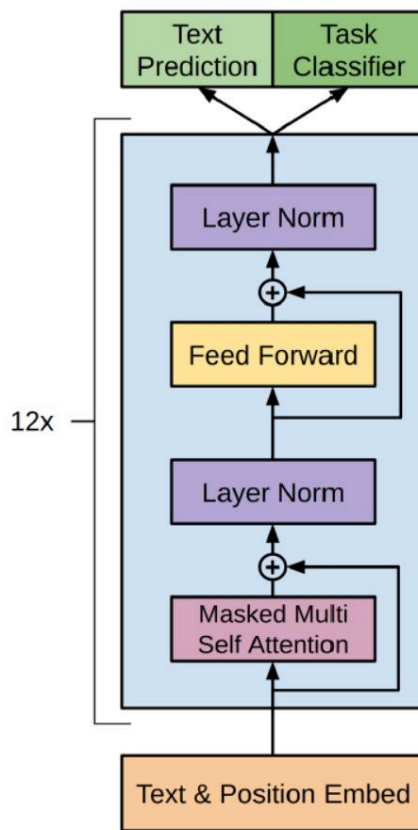


Figure 6: Decoder Unit

# Decoder Block

```
1 class GPT2Block(nn.Module):  
2     def __init__(self, config, layer_idx=None):  
3         super().__init__()  
4         hidden_size = config.hidden_size  
5         inner_dim = config.n_inner if config.n_inner is not none else 4 *  
        hidden_size  
6         attention_class =  
        GPT2_ATTENTION_CLASSES[config._attn_implementation]  
7         self.ln_1 = nn.LayerNorm(hidden_size, eps=config.layer_norm_epsilon)  
8         self.attn = attention_class(config=config, layer_idx=layer_idx)  
9         self.ln_2 = nn.LayerNorm(hidden_size, eps=config.layer_norm_epsilon)  
10        self.mlp = GPT2MLP(inner_dim, config)
```

python

# Decoder Block

```
11     def forward(  
12         self,  
13         hidden_states: Optional[Tuple[torch.FloatTensor]],  
14         layer_past: Optional[Tuple[torch.Tensor]] = None,  
15         use_cache: Optional[bool] = False,  
16         ...  
17     ) -> Union[Tuple[torch.Tensor], Optional[Tuple[torch.Tensor,  
18         Tuple[torch.FloatTensor, ...]]]]:  
19         # First residual unit  
20         residual = hidden_states  
21         hidden_states = self.ln_1(hidden_states)  
22         attn_outputs = self.attn(  

```

# Decoder Block

```
22         hidden_states,
23         layer_past=layer_past,
24         use_cache=use_cache,
25         ...
26     )
27     attn_output = attn_outputs[0] # output_attn: a, present,
    (attentions)
28     outputs = attn_outputs[1:]
29     # residual connection
30     hidden_states = attn_output + residual
31
32     # Second residual unit
```

# Decoder Block

```
33     residual = hidden_states
34     hidden_states = self.ln_2(hidden_states)
35     feed_forward_hidden_states = self.mlp(hidden_states)
36     # residual connection
37     hidden_states = residual + feed_forward_hidden_states
38
39     if use_cache:
40         outputs = (hidden_states,) + outputs
41     else:
42         outputs = (hidden_states,) + outputs[1:]
43     return outputs # hidden_states, present, (attentions,
cross_attentions)
```

# QKV Illustration [2], [3]

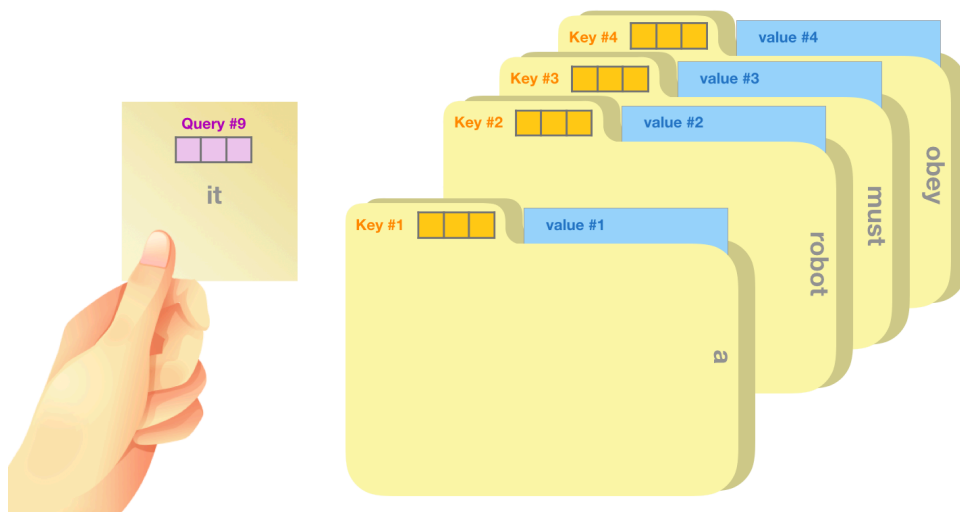


Figure 7: A crude analogy is to think of it like searching through a filing cabinet. The query is like a sticky note with the topic you're researching. The keys are like the labels of the folders inside the cabinet. When you match the tag with a sticky note, we take out the contents of that folder, these contents are the value vector. Except you're not only looking for one value, but a blend of values from a blend of folders.

**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.



# QKV Illustration [2], [3]

```
1 class GPT2Attention(nn.Module):python
2     def __init__(self, config, layer_idx=None):
3         super().__init__()
4         self.embed_dim = config.hidden_size
5         self.split_size = self.embed_dim
6         self.c_attn = Conv1D(3 * self.embed_dim, self.embed_dim)
7     def forward(
8         self,
9         hidden_states: Optional[Tuple[torch.FloatTensor]], ...
10    ) -> Tuple[Union[torch.Tensor, Tuple[torch.Tensor]], ...]:
11         query, key, value =
        self.c_attn(hidden_states).split(self.split_size, dim=2)
```

# Multi-head Implementation [4]

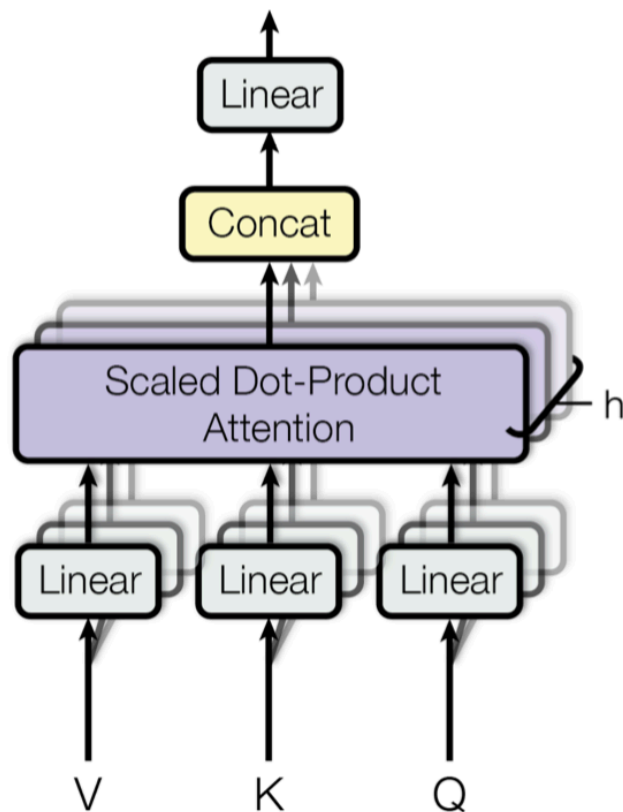


Figure 8: Multi-head Architecture Illustration

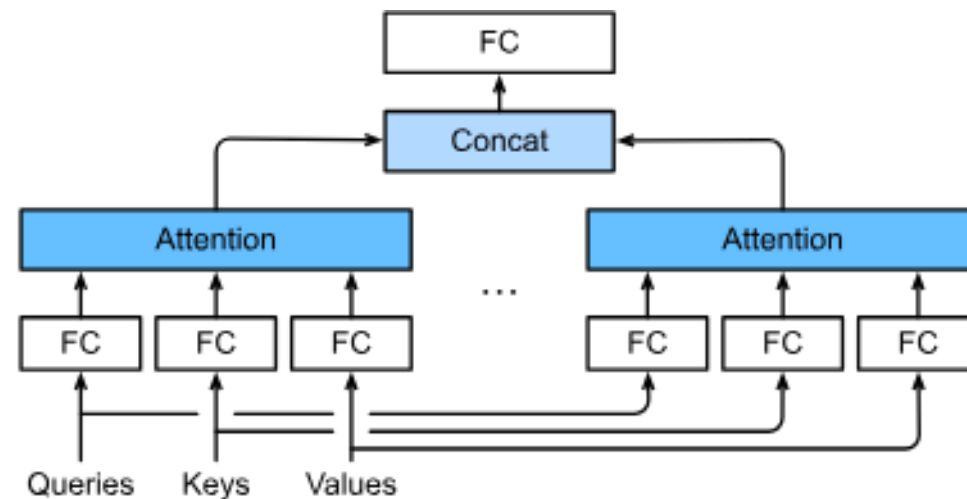


Figure 9: In the Transformer, the Attention module repeats its computations multiple times in parallel. Each of these is called an Attention Head. The Attention module splits its Query, Key, and Value parameters N-ways and passes each split independently through a separate Head. All of these similar Attention calculations are then combined together to produce a final Attention score.

# Multi-head Implementation [4]

```
1  class GPT2Attention(nn.Module):python
2      def __init__(self, config, layer_idx=None):
3          super().__init__()
4          ...
5          self.embed_dim = config.hidden_size
6          self.num_heads = config.num_attention_heads
7          self.head_dim = self.embed_dim // self.num_heads
8          if self.head_dim * self.num_heads != self.embed_dim:
9              raise ValueError(
10                 f"`embed_dim` must be divisible by num_heads (got
11                 `embed_dim`: {self.embed_dim} and `num_heads`: "
12                 f"{self.num_heads})."
```

# Multi-head Implementation [4]

```
12         )
13     def forward(
14         self,
15         hidden_states: Optional[Tuple[torch.FloatTensor]], ...
16     ) -> Tuple[Union[torch.Tensor, Tuple[torch.Tensor]], ...]:
17         ... # (code in the previous section)
18         query, key, value =
19     self.c_attn(hidden_states).split(self.split_size, dim=2)
19         # Split into multiple heads
20         query = self._split_heads(query, self.num_heads, self.head_dim)
21         key = self._split_heads(key, self.num_heads, self.head_dim)
22         value = self._split_heads(value, self.num_heads, self.head_dim)
```

# Attention Calculation

Attention Formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

```
1 class GPT2Attention(nn.Module):  
2     def forward(  
3         self,  
4         hidden_states: Optional[Tuple[torch.FloatTensor]], ...  
5     ) -> Tuple[Union[torch.Tensor, Tuple[torch.Tensor]], ...]:  
6         ... # (code in previous section)  
7         attn_output, attn_weights = self._attn(query, key, value,  
         attention_mask, head_mask)
```

[python](#)

# Attention Calculation

```
1  class GPT2Attention(nn.Module):python
2      def _attn(self, query, key, value, attention_mask=None, head_mask=None):
3          ###-----Q@K-----
4          attn_weights = torch.matmul(query, key.transpose(-1, -2))
5
6          ###-----/sqrt(d_k)-----
7          if self.scale_attn_weights:
8              attn_weights = attn_weights / torch.full(
9                  [], value.size(-1) ** 0.5, dtype=attn_weights.dtype,
10                 device=attn_weights.device
11            )
```

# Attention Calculation

```
12         # Layer-wise attention scaling
13         if self.scale_attn_by_inverse_layer_idx:
14             attn_weights = attn_weights / float(self.layer_idx + 1)
15
16         ###-----Masking-----
17         # if only "normal" attention layer implements causal mask
18         query_length, key_length = query.size(-2), key.size(-2)
19         causal_mask = self.bias[:, :, key_length - query_length :
key_length, :key_length]
20         mask_value = torch.finfo(attn_weights.dtype).min
21         # Need to be a tensor, otherwise we get error: `RuntimeError:
expected scalar type float but found double`.
```

# Attention Calculation

```
22         # Need to be on the same device, otherwise `RuntimeError: ..., x and
    y to be on the same device`
23         mask_value = torch.full([], mask_value, dtype=attn_weights.dtype,
    device=attn_weights.device)
24         attn_weights = torch.where(causal_mask,
    attn_weights.to(attn_weights.dtype), mask_value)
25
26         if attention_mask is not None:
27             # Apply the attention mask
28             attn_weights = attn_weights + attention_mask
29
30         attn_weights = nn.functional.softmax(attn_weights, dim=-1)
```



# Attention Calculation

```
31
32     # Downcast (if necessary) back to V's dtype (if in mixed-precision)
33     -- No-Op otherwise
34     attn_weights = attn_weights.type(value.dtype)
35     attn_weights = self.attn_dropout(attn_weights)
36
37     # Mask heads if we want to
38     if head_mask is not None:
39         attn_weights = attn_weights * head_mask
40
41     ###-----previous_value@V-----
42     attn_output = torch.matmul(attn_weights, value)
```

# MLP Layer

```
1  class GPT2MLP(nn.Module):python
2      def __init__(self, intermediate_size, config):
3          super().__init__()
4          embed_dim = config.hidden_size
5          self.c_fc = Conv1D(intermediate_size, embed_dim)
6          self.c_proj = Conv1D(embed_dim, intermediate_size)
7          self.act = ACT2FN[config.activation_function]
8          self.dropout = nn.Dropout(config.resid_pdrop)
9
10         def forward(self, hidden_states: Optional[Tuple[torch.FloatTensor]]) ->
            torch.FloatTensor:
11             hidden_states = self.c_fc(hidden_states)
```

# MLP Layer

```
12         hidden_states = self.act(hidden_states)
13         hidden_states = self.c_proj(hidden_states)
14         hidden_states = self.dropout(hidden_states)
15         return hidden_states
```

# Outline

Overview of the Architecture

Embedding

Transformer Decoder

**Final Step**

Rerun the Workflow: KVCache Perspective

Bibliography

```
1 class GPT2Model(GPT2PreTrainedModel):  
2     def forward(...):  
3         # hidden_states is the output returned from all of the previous  
         transformer decoder layers  
4         hidden_states = outputs[0]  
5         hidden_states = self.ln_f(hidden_states)  
6         hidden_states = hidden_states.view(output_shape)
```

python

# Outline

Overview of the Architecture

Embedding

Transformer Decoder

Final Step

**Rerun the Workflow: KVCache Perspective**

Bibliography

# Quick Reminder of Variables

`hidden_states` is the representation of input. Its first appearance is in the `GPT2Model` class, which is the sum of `inputs_embeds` and `position_embeds`.

# In the Attention Block (GPT2Attention)

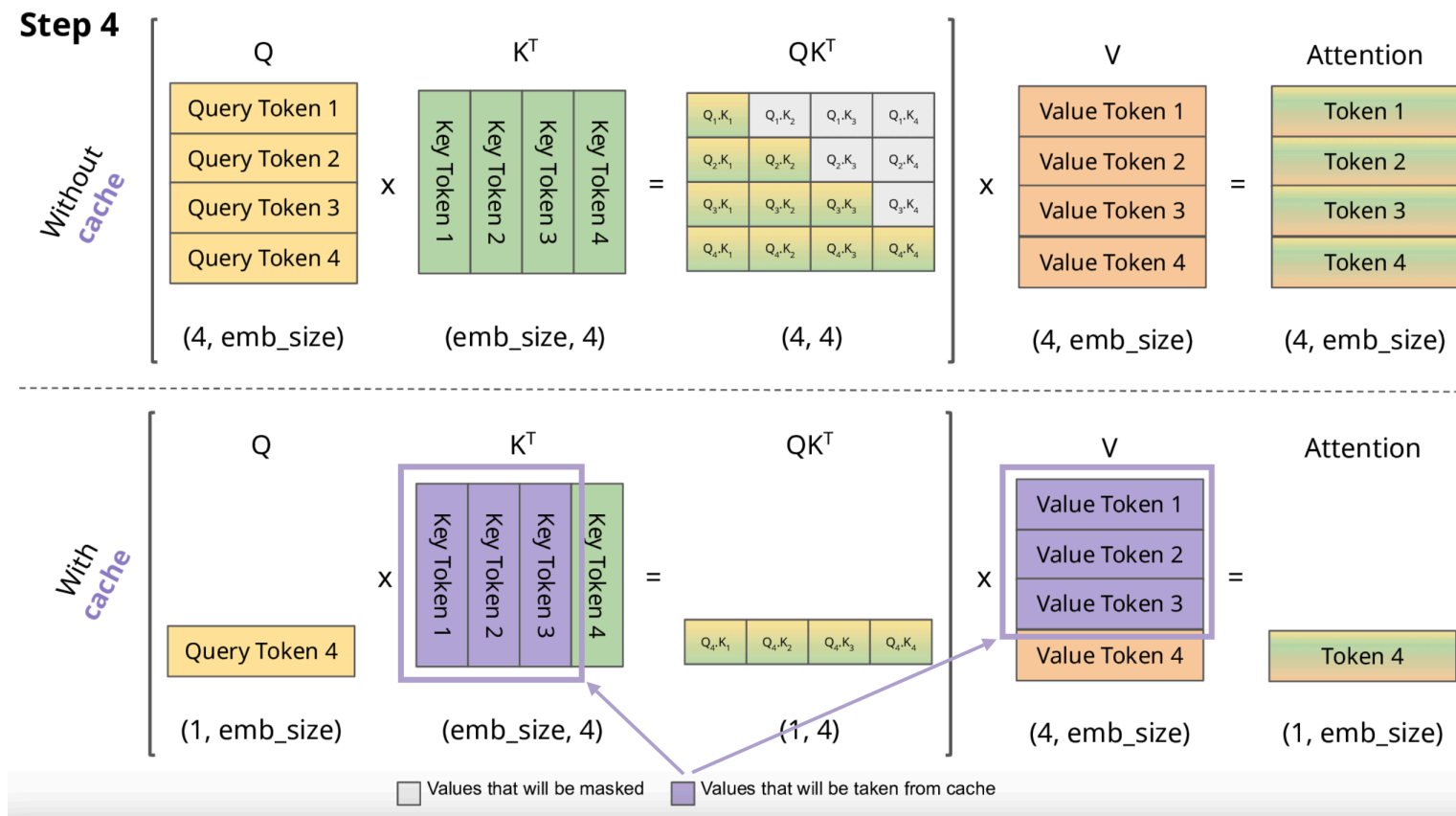


Figure 10: For full gif, visit [https://miro.medium.com/v2/resize:fit:1400/format:webp/1\\*uyuyOW1VBqmF5Gtv225XHQ.gif](https://miro.medium.com/v2/resize:fit:1400/format:webp/1*uyuyOW1VBqmF5Gtv225XHQ.gif)



# In the Attention Block (GPT2Attention)

```
1  class GPT2Attention(nn.Module):python
2      def forward(
3          layer_past: Optional[Tuple[torch.Tensor]] = None, ...):
4          ...
5          # query, key and value extracted and has been allocated into
           different heads
6          if layer_past is not None:
7              past_key, past_value = layer_past
8              key = torch.cat((past_key, key), dim=-2)
9              value = torch.cat((past_value, value), dim=-2)
10             present = (key, value)
11             outputs = (attn_output, present)
```

# In the Attention Block (GPT2Attention)

Conclusion:

- `layer_past` contains `past_key` and `past_value`, where `layer_past[0] = past_key` and `layer_past[1] = past_value`.
- In each iteration, the calculated key and value are written into the `present` variable.
- `GPT2Attention` requires `layer_past` as input, and returns both (`attn_output`, `present`)

# In the Transformer Block (GPT2Block)

```
1  class GPT2Block(nn.Module):
2      def forward(
3          self,
4          hidden_states: Optional[Tuple[torch.FloatTensor]],
5          layer_past: Optional[Tuple[torch.Tensor]] = None,):
6          ... # some calculation
7          attn_outputs = self.attn(...) # attn block output exported
8          attn_output = attn_outputs[0] # =attn_output
9          outputs = attn_outputs[1:] # =present
10         ... # some calculation
11         outputs = (hidden_states,) + outputs
12         return outputs # hidden_states, present
```

python

# In the Transformer Block (GPT2Block)

Conclusion:

- The computed results in the middle layer continue passing to the next level.
- GPT2Block requires `layer_past` as input and returns both (`hidden_states`, `present`)

# In the Whole Process (GPT2Model)

```
1  class GPT2Model(GPT2PreTrainedModel):python
2      def forward(
3          past_key_values: Optional[Tuple[Tuple[torch.Tensor]]] = None,):
4          # Initialization
5          if past_key_values is None:
6              past_length = 0
7              past_key_values = tuple([None] * len(self.h))
8          else:
9              # 关键的一行
10             past_length = past_key_values[0][0].size(-2)
11             if position_ids is None:
```

## In the Whole Process (GPT2Model)

```
12         position_ids = torch.arange(past_length, input_shape[-1] +  
    past_length, dtype=torch.long, device=device)  
13         position_ids = position_ids.unsqueeze(0)  
14  
15         # Embedding 层, 这里只相加了 token embedding 和 position embeddding, 忽  
    略了 token type embedding  
16         inputs_embed = self.wte(input_ids)  
17         position_embed = self.wpe(position_ids)  
18         # 如果是预填充阶段, 那么 inputs_embed 的维度是: [bs, seq_len,  
    embed_dim], position_embed 的维度是 [1, seq_len, embed_dim]  
19         # 如果是使用 KVCache 阶段, 那么 inputs_embed 的维度是: [bs, 1,  
    embed_dim], position_embed 的维度是 [1, 1, embed_dim]
```

## In the Whole Process (GPT2Model)

```
20
21     for i, (block, layer_past) in enumerate(zip(self.h,
past_key_values)):
22         outputs = block(
23             hidden_states,
24             layer_past=layer_past,
25             ...
26         ) # (hidden_states, present)
27
28         hidden_states = outputs[0]
29         presents = presents + (outputs[1],)
30
```

## In the Whole Process (GPT2Model)

```
31         ... # Further execution of hidden_states
32         return BaseModelOutputWithPastAndCrossAttentions(
33             last_hidden_state=hidden_states,
34             past_key_values=presents,
35             hidden_states=all_hidden_states,
36             attentions=all_self_attentions,
37             cross_attentions=all_cross_attentions,
38         )
```



# In the Whole Process (GPT2Model)

## Conclusion:

- `hidden_states` are read from the output of the consecutive blocks
- In each iteration, the present result is added to the global `presents` variable
- `past_key_values` = `presents`, which means that it contains all of the history KV-cache including the one generated in this iteration. It is then passed to the next iteration as input.

# Outline

Overview of the Architecture

Embedding

Transformer Decoder

Final Step

Rerun the Workflow: KVCache Perspective

**Bibliography**

# Bibliography

- [1] L. Weng, “Attention? Attention!,” *lilianweng.github.io*, 2018, [Online]. Available: <https://lilianweng.github.io/posts/2018-06-24-attention/>
- [2] J. Alammar, “The Illustrated GPT-2,” *jalammar.github.io*, 2024, [Online]. Available: <https://jalammar.github.io/illustrated-gpt2/>
- [3] Languisher, “GPT-2: The Complete Guide,” *languisher.icu*, 2024, [Online]. Available: <https://www.languisher.icu/blog/gpt-2/>
- [4] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, “Multi-Head Attention,” *d2l.ai*, 2024, [Online]. Available: [https://d2l.ai/chapter\\_attention-mechanisms-and-transformers/multihead-attention.html](https://d2l.ai/chapter_attention-mechanisms-and-transformers/multihead-attention.html)