KVCache: A Source Code Perspective

(W7) Weekly Report

Nan Lin

Shanghai Jiao Tong University

2024-08-25

Outline

Outline

Architecture of a Transformer Encoder-Decoder Structure [1]

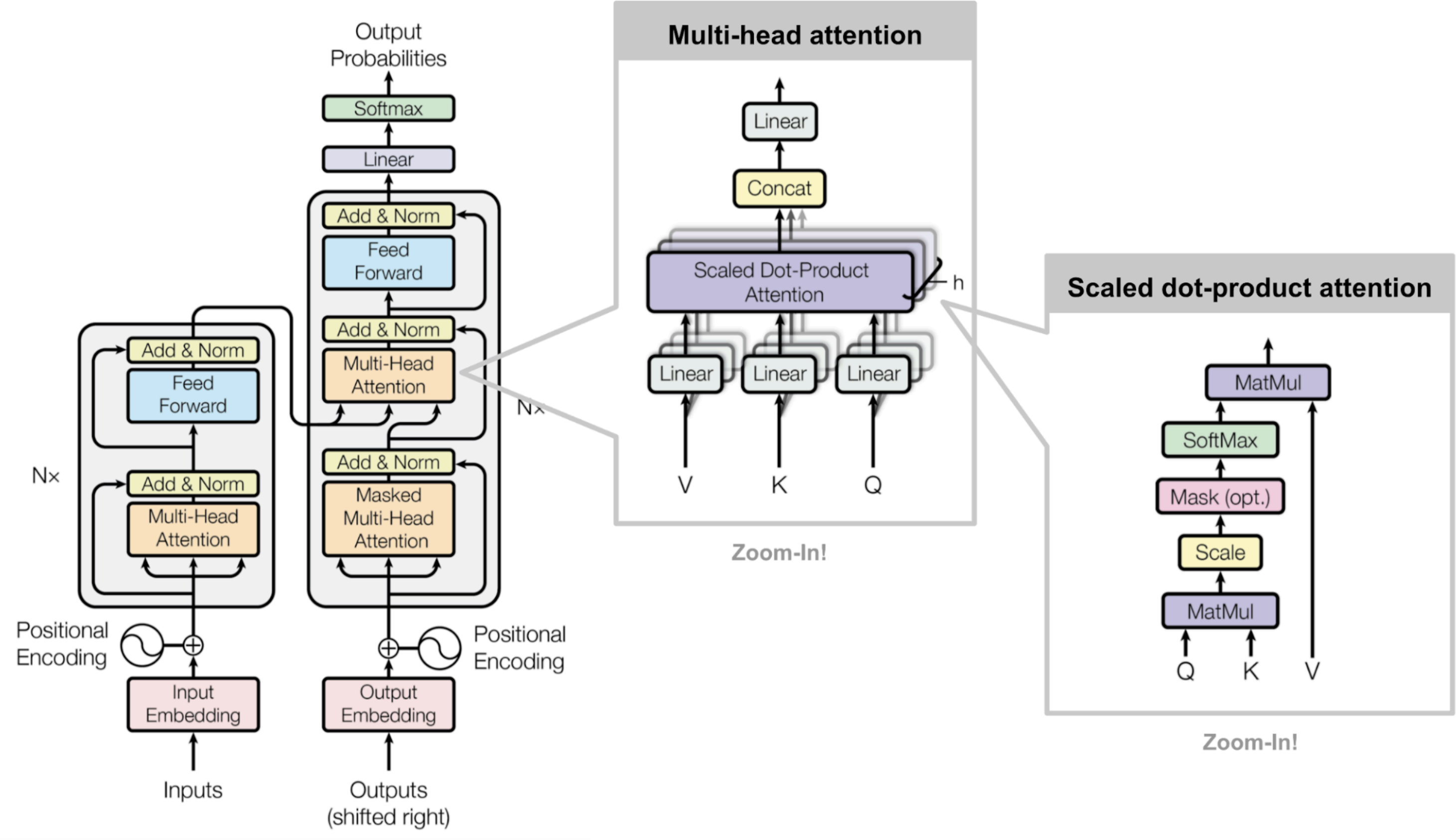

Figure 1: Archtecture of a Transformer Encoder-Decoder Structure

Architecture of GPT-2 Decoder

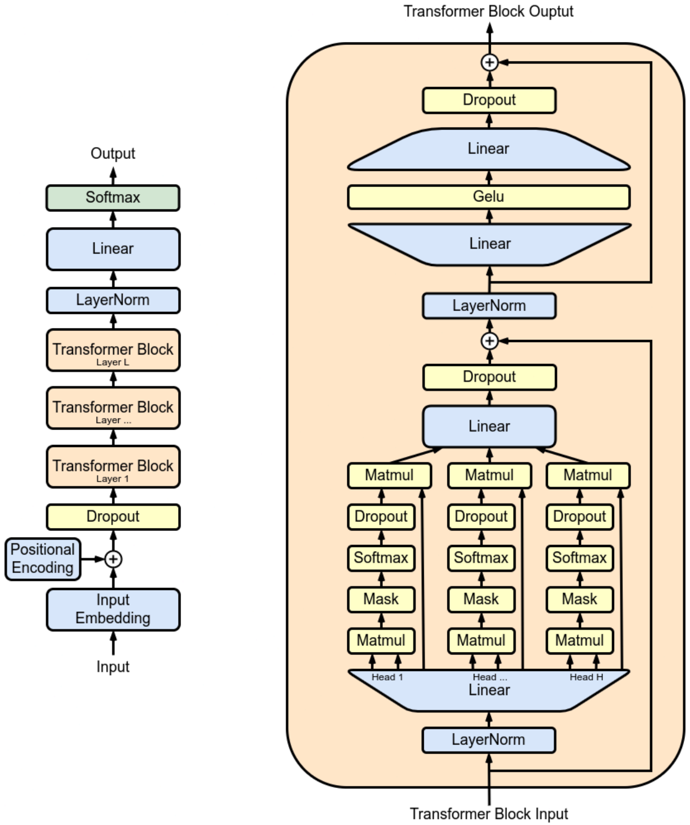

Figure 2: Architecture of GPT-2 Decoder

Implementation

Several Class:

```
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. Each `GPT2Block` consists of a `GPT2Attention` layer, a `GPT2MLP` layer and some other components)
    ├── GPT2Attention (Manages self-attention)
    └── GPT2MLP (A feedforward neural network)
```

Implementation

In `GPT2Model`: the input goes through

```
├── Embedding
├── Multiple blocks encapsulated in GPT2Block:
        self.h = nn.ModuleList([GPT2Block(config, layer_idx=i) for i in range(config.num_hidden_layers)])
    └── For each block:
            for i, (block, layer_past) in enumerate(zip(self.h, past_key_values)):
        └──     outputs = block(
                    layer_past=layer_past, ...
                )
└── Final output: LinearNorm, transformation...
```

Outline

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

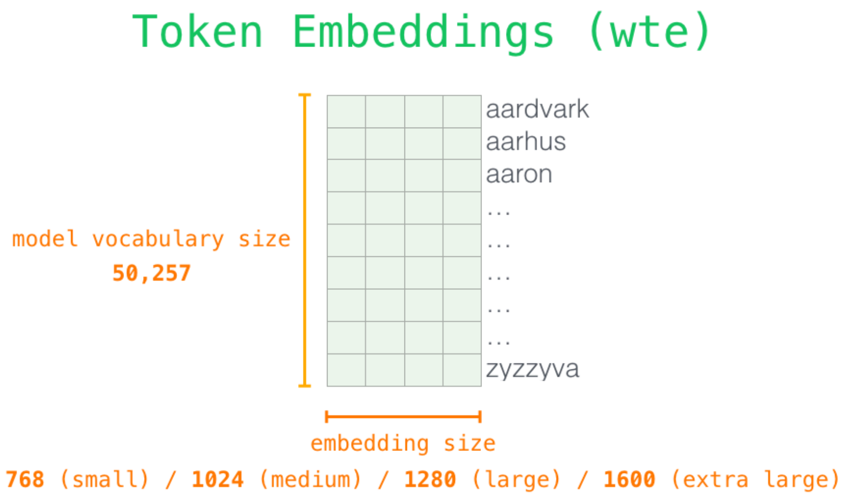

Figure 3: Each row is a word embedding: a list of numbers representing a word and capturing some of its meaning. The size of that list is different in different GPT2 model sizes. The smallest model uses an embedding size of 768 per word/token.

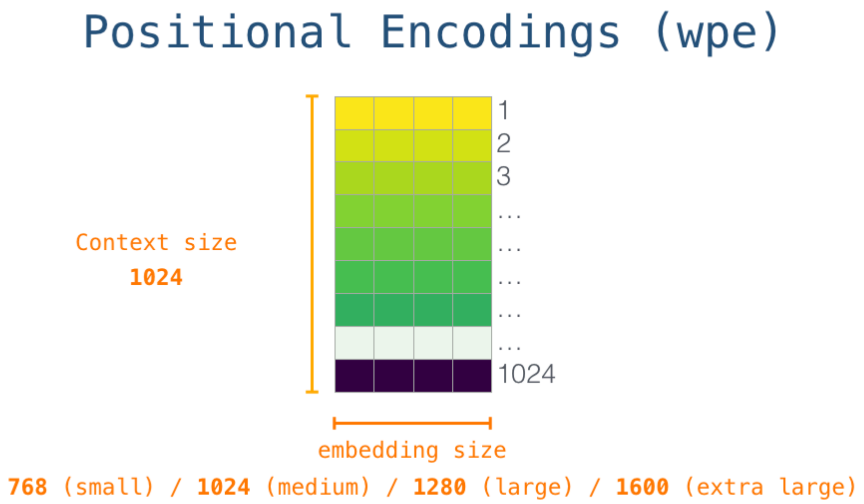

Figure 4: Positional encoding – a signal that indicates the order of the words in the sequence to the transformer blocks. Part of the trained model is a matrix that contains a positional encoding vector for each of the 1024 positions in the input.

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]

```python
class GPT2Model(GPT2PreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.wte = nn.Embedding(config.vocab_size, self.embed_dim)
        self.wpe = nn.Embedding(config.max_position_embeddings, self.embed_dim)

    def forward(
        input_ids: Optional[torch.LongTensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        ...
    ):
```

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

```
        if position_ids are none:
            position_ids = torch.arange(past_length, input_shape[-1] + past_length, dtype=torch.long, device=device)
            position_ids = position_ids.unsqueeze(0)

        if inputs_embeds is none:
            inputs_embeds = self.wte(input_ids)

        position_embeds = self.wpe(position_ids)
        hidden_states = inputs_embeds + position_embeds
        ...
```

Outline

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

```python
class GPT2Model(GPT2PreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.h = nn.ModuleList([GPT2Block(config, layer_idx=i) for i in range(config.num_hidden_layers)])

    def forward(
        self,
        past_key_values: Optional[Tuple[Tuple[torch.Tensor]]] = None,
        encoder_hidden_states: Optional[torch.Tensor] = None,
        use_cache: Optional[bool] = None,
        output_hidden_states: Optional[bool] = None,
```

Decoder Block

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

Decoder Block

```
            past_length = 0
            past_key_values = tuple([None] * len(self.h))
        presents = () if use_cache else None

        for i, (block, layer_past) in enumerate(zip(self.h, past_key_values)):
            """
            layer_past is an element of KVCache, each representing a single block
            """
            outputs = block(
                    hidden_states,
```

Decoder Block

```
                layer_past=layer_past,
                use_cache=use_cache,
                output_attentions=output_attentions,
                ...
            )

        hidden_states = outputs[0]
        if use_cache is True:
            presents = presents + (outputs[1],)
```

Decoder Block

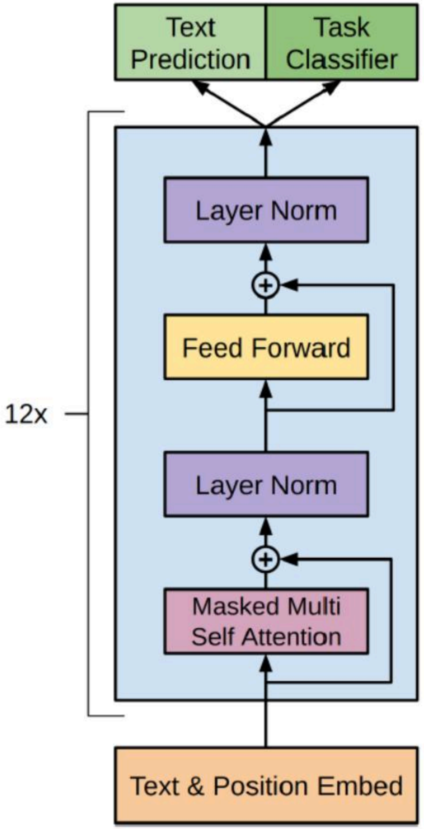

Figure 6: Decoder Unit

Decoder Block

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

Decoder Block

```python
    def forward(
        self,
        hidden_states: Optional[Tuple[torch.FloatTensor]],
        layer_past: Optional[Tuple[torch.Tensor]] = None,
        use_cache: Optional[bool] = False,
        ...
    ) -> Union[Tuple[torch.Tensor], Optional[Tuple[torch.Tensor, Tuple[torch.FloatTensor, ...]]]]:
        # First residual unit
        residual = hidden_states
        hidden_states = self.ln_1(hidden_states)
        attn_outputs = self.attn(
```

Decoder Block

```
            hidden_states,
            layer_past=layer_past,
            use_cache=use_cache,
            ...
        )
        attn_output = attn_outputs[0]  # output_attn: a, present, (attentions)
        outputs = attn_outputs[1:]
        # residual connection
        hidden_states = attn_output + residual

        # Second residual unit
```

Decoder Block

```
        residual = hidden_states
        hidden_states = self.ln_2(hidden_states)
        feed_forward_hidden_states = self.mlp(hidden_states)
        # residual connection
        hidden_states = residual + feed_forward_hidden_states

        if use_cache:
            outputs = (hidden_states,) + outputs
        else:
            outputs = (hidden_states,) + outputs[1:]
        return outputs  # hidden_states, present, (attentions, cross_attentions)
```

QKV Illustration [2], [3]

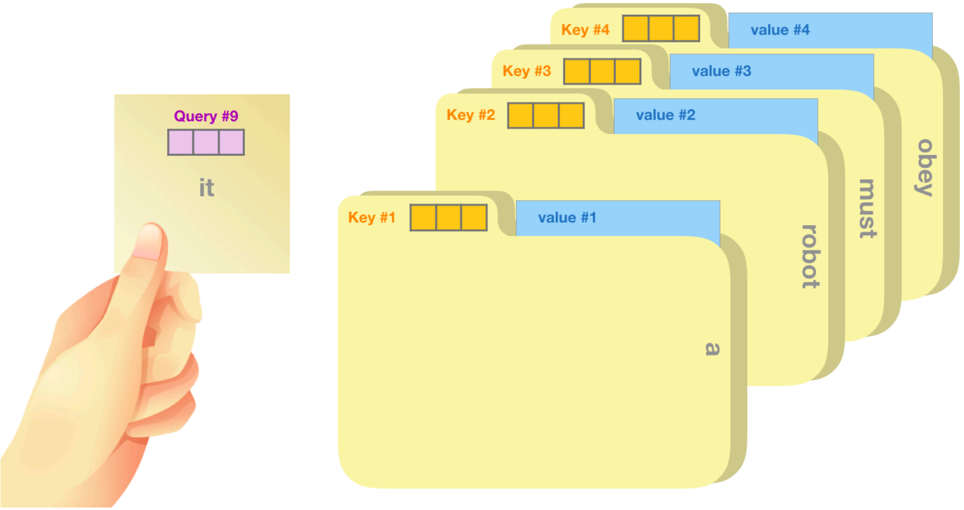

Figure 7: A crude analogy is to think of it like searching through a filing cabinet. The query is like a sticky note with the topic you're researching. The keys are like the labels of the folders inside the cabinet. When you match the tag with a sticky note, we take out the contents of that folder, these 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]

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

Multi-head Implementation [4]

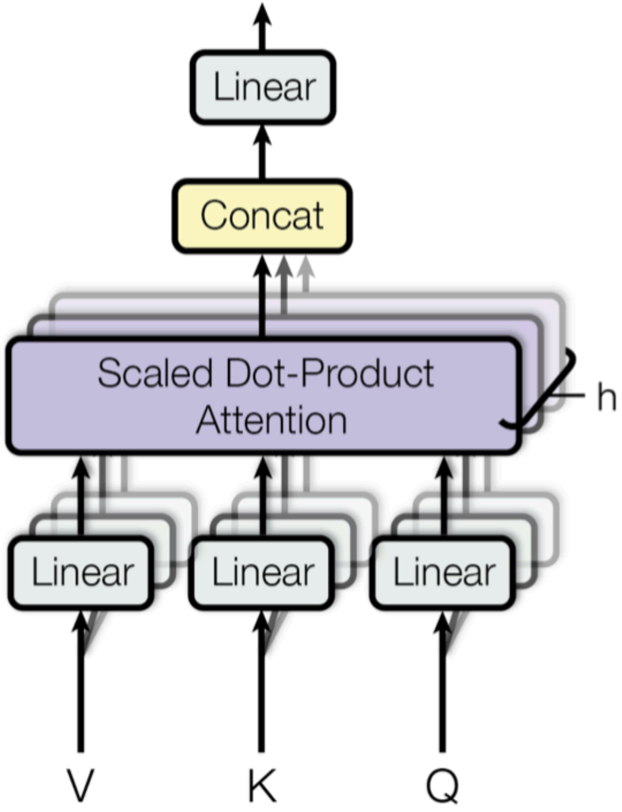

Figure 8: Multi-head Architecture Illustration

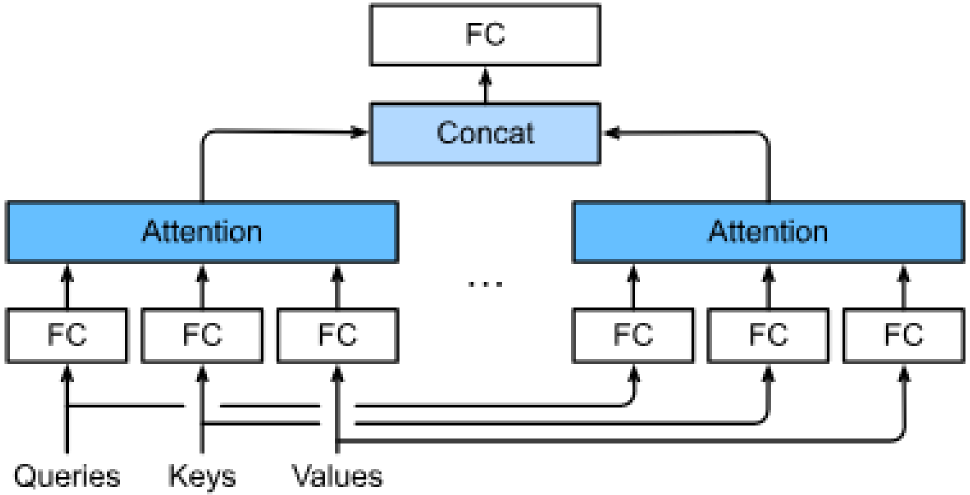

Figure 9: In the Transformer, the Attention module repeats its computations multiple times in parallel. Each of these is called an Attention Head. The Attention module splits its Query, Key, and Value parameters N-ways and passes each split independently through a separate Head. All of these similar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

Multi-head Implementation [4]

python

```python
class GPT2Attention(nn.Module):
    def __init__(self, config, layer_idx=None):
        super().__init__()
        ...
        self.embed_dim = config.hidden_size
        self.num_heads = config.num_attention_heads
        self.head_dim = self.embed_dim // self.num_heads
        if self.head_dim * self.num_heads != self.embed_dim:
            raise ValueError(
                f"`embed_dim` must be divisible by num_heads (got `embed_dim`: {self.embed_dim} and `num_heads`:"
                f" {self.num_heads})."
```

Multi-head Implementation [4]

```python
            )
    def forward(
        self,
        hidden_states: Optional[Tuple[torch.FloatTensor]], ...
    ) -> Tuple[Union[torch.Tensor, Tuple[torch.Tensor]], ...]:
        ... # (code in the previous section)
        query, key, value = self.c_attn(hidden_states).split(self.split_size, dim=2)
        # Split into multiple heads
        query = self._split_heads(query, self.num_heads, self.head_dim)
        key = self._split_heads(key, self.num_heads, self.head_dim)
        value = self._split_heads(value, self.num_heads, self.head_dim)
```

Attention Calculation

Attention Formula:

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

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

Attention Calculation

python

```python
class GPT2Attention(nn.Module):
    def _attn(self, query, key, value, attention_mask=None, head_mask=None):
        ###---------------Q@K--------------------
        attn_weights = torch.matmul(query, key.transpose(-1, -2))

        ###---------------/sqrt(d_k)-------------------
        if self.scale_attn_weights:
            attn_weights = attn_weights / torch.full(
                [], value.size(-1) ** 0.5, dtype=attn_weights.dtype, device=attn_weights.device
            )

```

Attention Calculation

```
        # Layer-wise attention scaling
        if self.scale_attn_by_inverse_layer_idx:
            attn_weights = attn_weights / float(self.layer_idx + 1)

        ###---------------Masking--------------------
        # if only "normal" attention layer implements causal mask
        query_length, key_length = query.size(-2), key.size(-2)
        causal_mask = self.bias[:, :, key_length - query_length :
key_length, :key_length]
        mask_value = torch.finfo(attn_weights.dtype).min
        # Need to be a tensor, otherwise we get error: `RuntimeError:
expected scalar type float but found double`.
```

Attention Calculation

```python
        # Need to be on the same device, otherwise `RuntimeError: ..., x and y to be on the same device`
        mask_value = torch.full([], mask_value, dtype=attn_weights.dtype, device=attn_weights.device)
        attn_weights = torch.where(causal_mask, attn_weights.to(attn_weights.dtype), mask_value)

        if attention_mask is not none:
            # Apply the attention mask
            attn_weights = attn_weights + attention_mask

        attn_weights = nn.functional.softmax(attn_weights, dim=-1)
```

Attention Calculation

```

        # Downcast (if necessary) back to V's dtype (if in mixed-precision) -- No-Op otherwise
        attn_weights = attn_weights.type(value.dtype)
        attn_weights = self.attn_dropout(attn_weights)

        # Mask heads if we want to
        if head_mask is not none:
            attn_weights = attn_weights * head_mask

        ###------------previous_value@V------------------
        attn_output = torch.matmul(attn_weights, value)
```

MLP Layer

python

```python
class GPT2MLP(nn.Module):
    def __init__(self, intermediate_size, config):
        super().__init__()
        embed_dim = config.hidden_size
        self.c_fc = Conv1D(intermediate_size, embed_dim)
        self.c_proj = Conv1D(embed_dim, intermediate_size)
        self.act = ACT2FN[config.activation_function]
        self.dropout = nn.Dropout(config.resid_pdrop)

    def forward(self, hidden_states: Optional[Tuple[torch.FloatTensor]]) ->
torch.FloatTensor:
        hidden_states = self.c_fc(hidden_states)
```

MLP Layer

```
        hidden_states = self.act(hidden_states)
        hidden_states = self.c_proj(hidden_states)
        hidden_states = self.dropout(hidden_states)
        return hidden_states
```

Outline

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

Outline

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`)

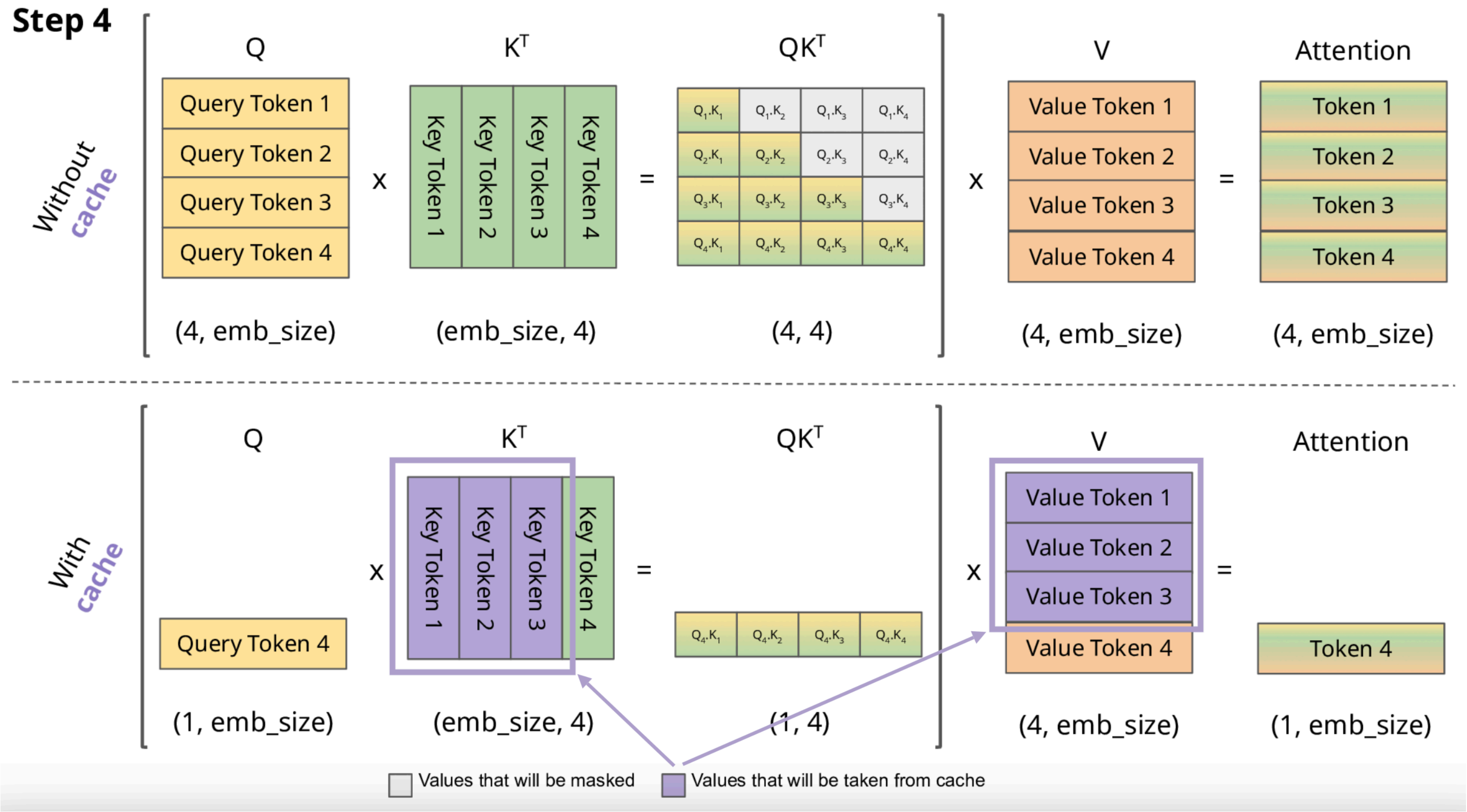

Figure 10: For full gif, visit https://miro.medium.com/v2/resize:fit:1400/format:webp/1*uyuyOW1VBqmF5Gtv225XHQ.gif

In the Attention Block (GPT2Attention)

```python
class GPT2Attention(nn.Module):
    def forward(
        layer_past: Optional[Tuple[torch.Tensor]] = None, ...):
        ...
        # query, key and value extracted and has been allocated into different heads
        if layer_past is not None:
            past_key, past_value = layer_past
            key = torch.cat((past_key, key), dim=-2)
            value = torch.cat((past_value, value), dim=-2)
        present = (key, value)
        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)

```python
class GPT2Block(nn.Module):
    def forward(
        self,
        hidden_states: Optional[Tuple[torch.FloatTensor]],
        layer_past: Optional[Tuple[torch.Tensor]] = None,):
        ... # some calculation
        attn_outputs = self.attn(...) # attn block output exported
        attn_output = attn_outputs[0] # =attn_output
        outputs = attn_outputs[1:] # =present
        ... # some calculation
        outputs = (hidden_states,) + outputs
        return outputs  # hidden_states, present
```

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)

python

```python
class GPT2Model(GPT2PreTrainedModel):
    def forward(
      past_key_values: Optional[Tuple[Tuple[torch.Tensor]]] = None,):
        # Initialization
        if past_key_values is None:
            past_length = 0
            past_key_values = tuple([None] * len(self.h))
        presents = () if use_cache else None

        for i, (block, layer_past) in enumerate(zip(self.h, past_key_values)):
            outputs = block(
```

In the Whole Process (GPT2Model)

```
            hidden_states,
            layer_past=layer_past,
            ...
          ) # (hidden_states, present)

        hidden_states = outputs[0]
        presents = presents + (outputs[1],)

        ... # Further execution of hidden_states
        return BaseModelOutputWithPastAndCrossAttentions(
          last_hidden_state=hidden_states,
          past_key_values=presents,
```

In the Whole Process (`GPT2Model`)

```
            hidden_states=all_hidden_states,
            attentions=all_self_attentions,
            cross_attentions=all_cross_attentions,
        )
```

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

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