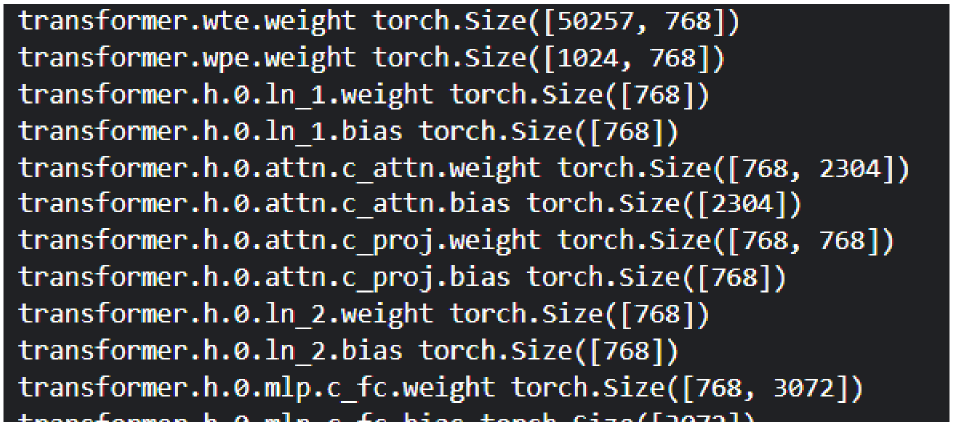
Implementing GPT-2 (124M parameter version) from scratch

The original code in OpenAI's GitHub is made using TensorFlow. However, a PyTorch implementation of GPT-2 from Hugging Face as it is the current industry standard for deep learning. More specifically the GPT2LMHeadModel model from transformers (Hugging Face) will be used as reference.

Parameters of GPT-2

Below is the number of parameters in each neuron layer in GPT-2. Only the first few lines are shown.



```
transformer.wte.weight torch.Size([50257, 768])
transformer.wpe.weight torch.Size([1024, 768])
transformer.h.0.ln_1.weight torch.Size([768])
transformer.h.0.ln_1.bias torch.Size([768])
transformer.h.0.attn.c_attn.weight torch.Size([768, 2304])
transformer.h.0.attn.c_attn.bias torch.Size([2304])
transformer.h.0.attn.c_proj.weight torch.Size([768, 768])
transformer.h.0.attn.c_proj.bias torch.Size([768])
transformer.h.0.ln_2.weight torch.Size([768])
transformer.h.0.ln_2.bias torch.Size([768])
transformer.h.0.mlp.c_fc.weight torch.Size([768, 3072])
transformer.h.0.mlp.c_fc.bias torch.Size([3072])
```

Line 1: Token embedding layer, where num_tokens=50257 and emb_dim=768.

Line 2: Positional embedding layer, where context_length=1024 and emb_dim=768.

Line 3 onward: Every neural network layer within the transformer block.

Positional Embeddings

- Size: 1024*768.
- The '1024' is the context length, which is the number of tokens each token can 'attend' to. Tokens can only attend tokens before it.
- The '768' is the number of embedding dimensions in the positional embedding space.
- In the original "Attention is all you need" paper, positional embeddings were represented by predetermined sinusoids. However, in GPT-2 the shape of these sinusoids are learned through training.

Difference between GPT2 and the original transformer model architecture

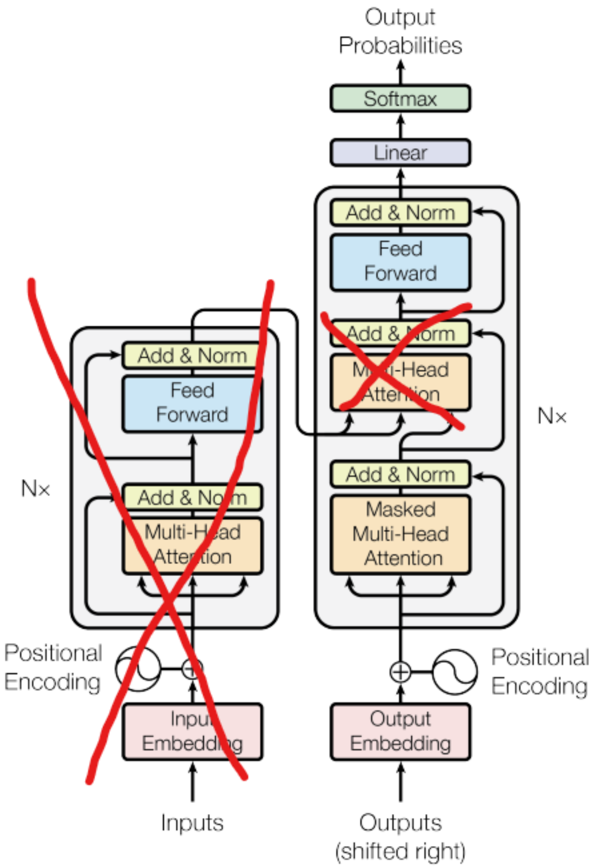
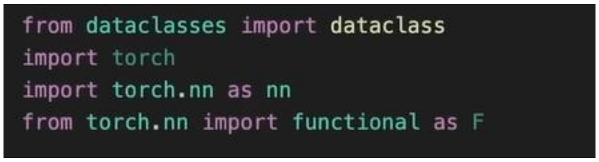


Figure 1: The Transformer - model architecture.

1. Decoder only model: GPT2 is a decoder only model. So, it doesn't have an encoder. This means that the entire left side of the transformer is removed. The attention block connecting to the encoder is also removed.
2. Revision of layer norms: Layer norms are now used before each attention block. A new layer norm is added before the final feedforward layer.

The Implementation

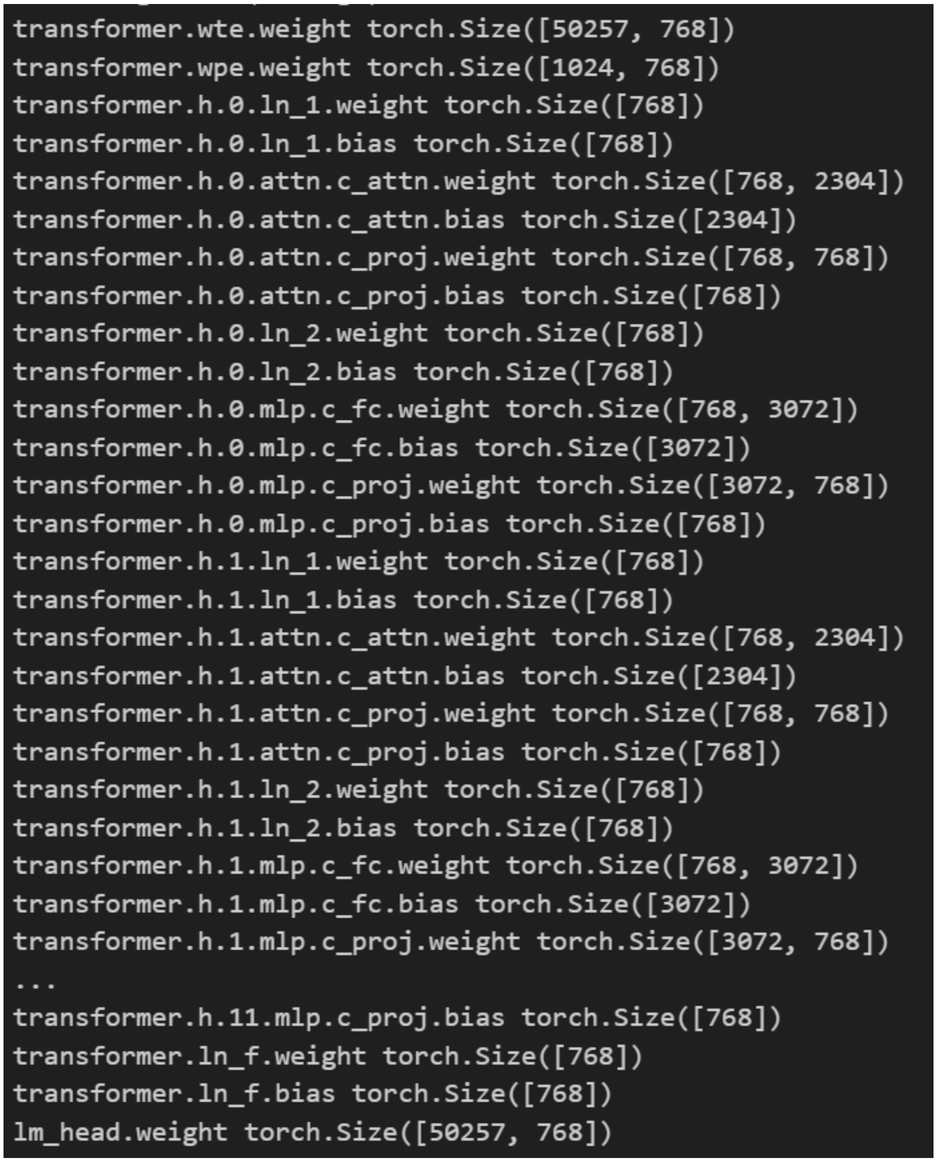
Libraries used



```
from dataclasses import dataclass
import torch
import torch.nn as nn
from torch.nn import functional as F
```

Implementation reference

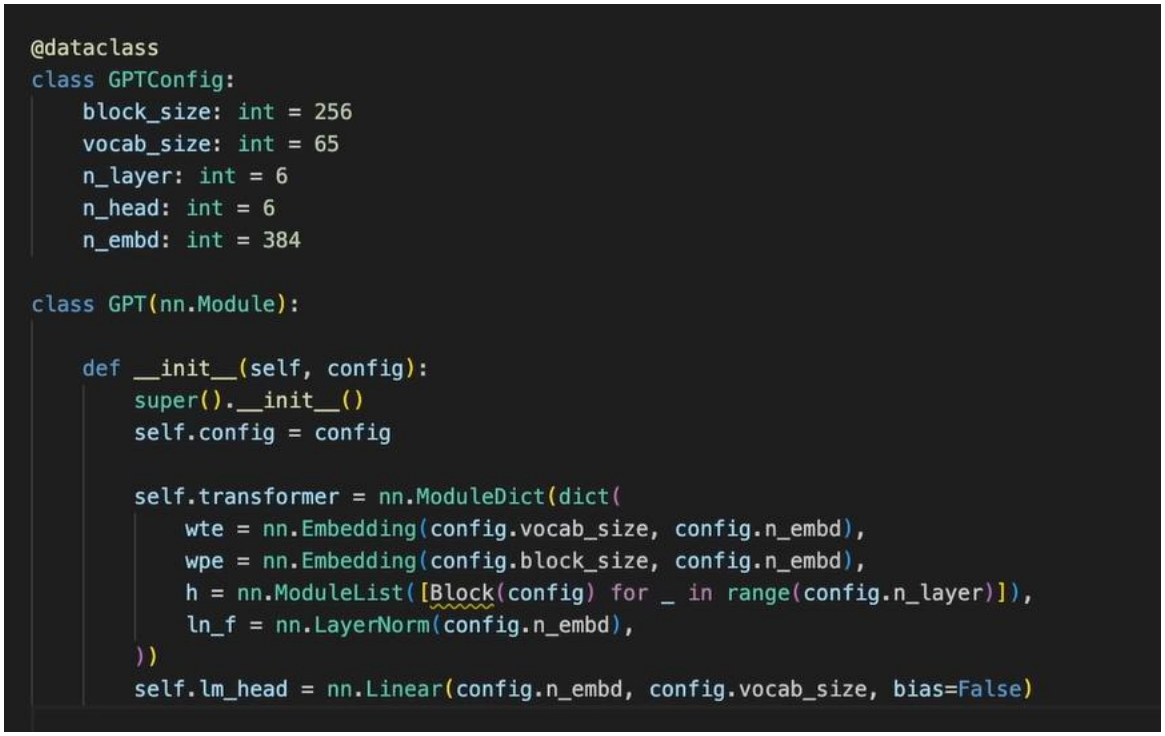
The below image contains information about each layer, which will be used as reference for our model. This image was extracted from Hugging Face's "gpt2" model.



```
transformer.wte.weight torch.Size([50257, 768])
transformer.wpe.weight torch.Size([1024, 768])
transformer.h.0.ln_1.weight torch.Size([768])
transformer.h.0.ln_1.bias torch.Size([768])
transformer.h.0.attn.c_attn.weight torch.Size([768, 2304])
transformer.h.0.attn.c_attn.bias torch.Size([2304])
transformer.h.0.attn.c_proj.weight torch.Size([768, 768])
transformer.h.0.attn.c_proj.bias torch.Size([768])
transformer.h.0.ln_2.weight torch.Size([768])
transformer.h.0.ln_2.bias torch.Size([768])
transformer.h.0.mlp.c_fc.weight torch.Size([768, 3072])
transformer.h.0.mlp.c_fc.bias torch.Size([3072])
transformer.h.0.mlp.c_proj.weight torch.Size([3072, 768])
transformer.h.0.mlp.c_proj.bias torch.Size([768])
transformer.h.1.ln_1.weight torch.Size([768])
transformer.h.1.ln_1.bias torch.Size([768])
transformer.h.1.attn.c_attn.weight torch.Size([768, 2304])
transformer.h.1.attn.c_attn.bias torch.Size([2304])
transformer.h.1.attn.c_proj.weight torch.Size([768, 768])
transformer.h.1.attn.c_proj.bias torch.Size([768])
transformer.h.1.ln_2.weight torch.Size([768])
transformer.h.1.ln_2.bias torch.Size([768])
transformer.h.1.mlp.c_fc.weight torch.Size([768, 3072])
transformer.h.1.mlp.c_fc.bias torch.Size([3072])
transformer.h.1.mlp.c_proj.weight torch.Size([3072, 768])
...
transformer.h.11.mlp.c_proj.bias torch.Size([768])
transformer.ln_f.weight torch.Size([768])
transformer.ln_f.bias torch.Size([768])
lm_head.weight torch.Size([50257, 768])
```

The GPT block

1. GPTConfig contains all the dimensions of the model.
2. The transformer block is implemented using ModuleDict, which stores nns in dictionary format.
3. Self.lm_head contains the final nn layer.

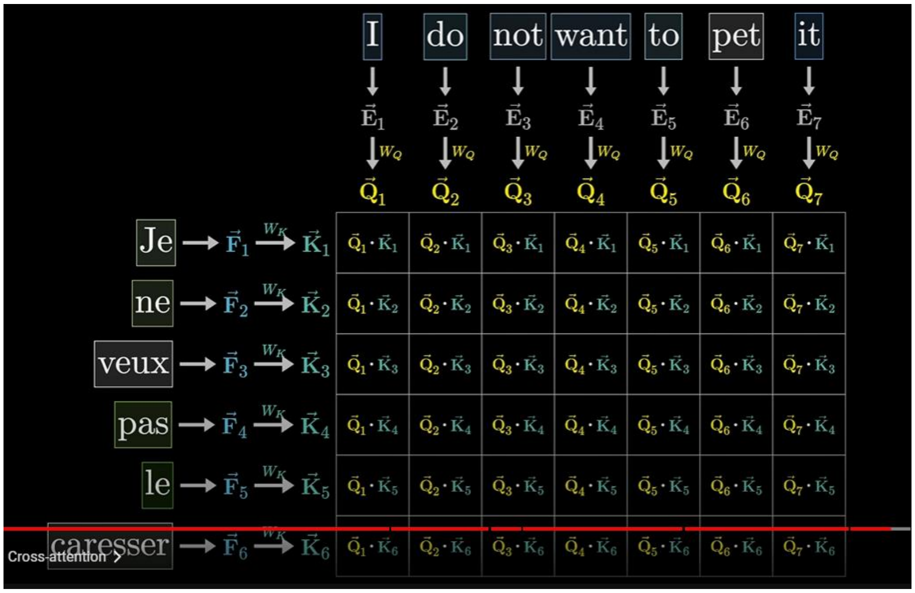


```
@dataclass
class GPTConfig:
    block_size: int = 256
    vocab_size: int = 65
    n_layer: int = 6
    n_head: int = 6
    n_embd: int = 384

class GPT(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.config = config

        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(config.vocab_size, config.n_embd),
            wpe = nn.Embedding(config.block_size, config.n_embd),
            h = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
            ln_f = nn.LayerNorm(config.n_embd),
        ))
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
```

Forward function in page 8.

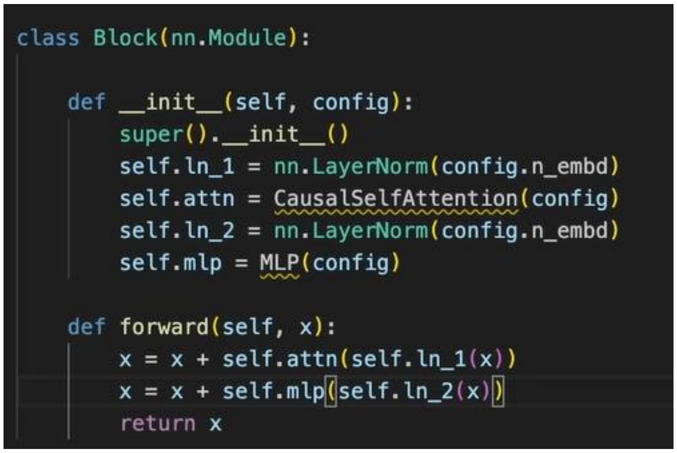
The block (within self.transformer.h)

1. Attention is a communication mechanism that allows tokens to exchange information. In the below image, keys are the tokens within context and queries are the tokens each key can pay attention to. For each key, a dot product is used to find the relationship between itself and all the queries. The dot product highlights the strength and nature of the relationship between tokens and are used as weights. The weights are used to calculate a weighted sum for query with regards to all keys within context. Therefore, attention can be thought of as a weighted sum operation or a reduce operation.



	I	do	not	want	to	pet	it
	$\vec{E}_1$	$\vec{E}_2$	$\vec{E}_3$	$\vec{E}_4$	$\vec{E}_5$	$\vec{E}_6$	$\vec{E}_7$
	W_Q	W_Q	W_Q	W_Q	W_Q	W_Q	W_Q
	$\vec{Q}_1$	$\vec{Q}_2$	$\vec{Q}_3$	$\vec{Q}_4$	$\vec{Q}_5$	$\vec{Q}_6$	$\vec{Q}_7$
Je → $\vec{F}_1 \xrightarrow{W_K} \vec{K}_1$	$\vec{Q}_1 \cdot \vec{K}_1$	$\vec{Q}_2 \cdot \vec{K}_1$	$\vec{Q}_3 \cdot \vec{K}_1$	$\vec{Q}_4 \cdot \vec{K}_1$	$\vec{Q}_5 \cdot \vec{K}_1$	$\vec{Q}_6 \cdot \vec{K}_1$	$\vec{Q}_7 \cdot \vec{K}_1$
ne → $\vec{F}_2 \xrightarrow{W_K} \vec{K}_2$	$\vec{Q}_1 \cdot \vec{K}_2$	$\vec{Q}_2 \cdot \vec{K}_2$	$\vec{Q}_3 \cdot \vec{K}_2$	$\vec{Q}_4 \cdot \vec{K}_2$	$\vec{Q}_5 \cdot \vec{K}_2$	$\vec{Q}_6 \cdot \vec{K}_2$	$\vec{Q}_7 \cdot \vec{K}_2$
veux → $\vec{F}_3 \xrightarrow{W_K} \vec{K}_3$	$\vec{Q}_1 \cdot \vec{K}_3$	$\vec{Q}_2 \cdot \vec{K}_3$	$\vec{Q}_3 \cdot \vec{K}_3$	$\vec{Q}_4 \cdot \vec{K}_3$	$\vec{Q}_5 \cdot \vec{K}_3$	$\vec{Q}_6 \cdot \vec{K}_3$	$\vec{Q}_7 \cdot \vec{K}_3$
pas → $\vec{F}_4 \xrightarrow{W_K} \vec{K}_4$	$\vec{Q}_1 \cdot \vec{K}_4$	$\vec{Q}_2 \cdot \vec{K}_4$	$\vec{Q}_3 \cdot \vec{K}_4$	$\vec{Q}_4 \cdot \vec{K}_4$	$\vec{Q}_5 \cdot \vec{K}_4$	$\vec{Q}_6 \cdot \vec{K}_4$	$\vec{Q}_7 \cdot \vec{K}_4$
le → $\vec{F}_5 \xrightarrow{W_K} \vec{K}_5$	$\vec{Q}_1 \cdot \vec{K}_5$	$\vec{Q}_2 \cdot \vec{K}_5$	$\vec{Q}_3 \cdot \vec{K}_5$	$\vec{Q}_4 \cdot \vec{K}_5$	$\vec{Q}_5 \cdot \vec{K}_5$	$\vec{Q}_6 \cdot \vec{K}_5$	$\vec{Q}_7 \cdot \vec{K}_5$
caresser → $\vec{F}_6 \xrightarrow{W_K} \vec{K}_6$	$\vec{Q}_1 \cdot \vec{K}_6$	$\vec{Q}_2 \cdot \vec{K}_6$	$\vec{Q}_3 \cdot \vec{K}_6$	$\vec{Q}_4 \cdot \vec{K}_6$	$\vec{Q}_5 \cdot \vec{K}_6$	$\vec{Q}_6 \cdot \vec{K}_6$	$\vec{Q}_7 \cdot \vec{K}_6$

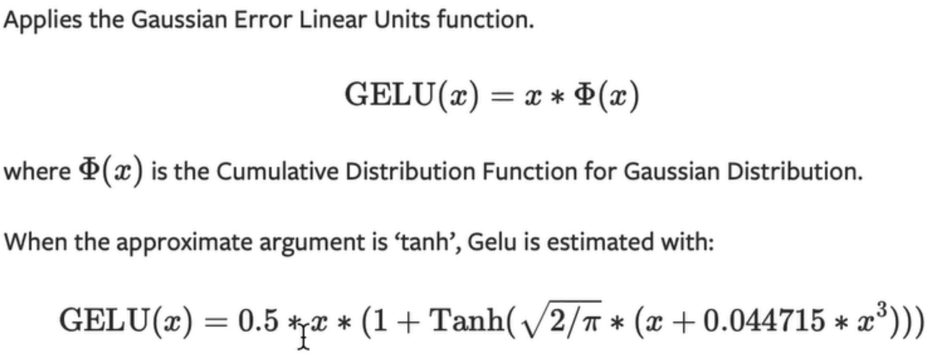
2. The linear layer is map operation that maps the embedding space.
3. There the transformer block can be thought of as a series of repeated reduce and map operations.



```
class Block(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.ln_1 = nn.LayerNorm(config.n_embd)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = nn.LayerNorm(config.n_embd)
        self.mlp = MLP(config)

    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x
```

The MLP

1. A GELU activation function is sandwiched between two linear layers.



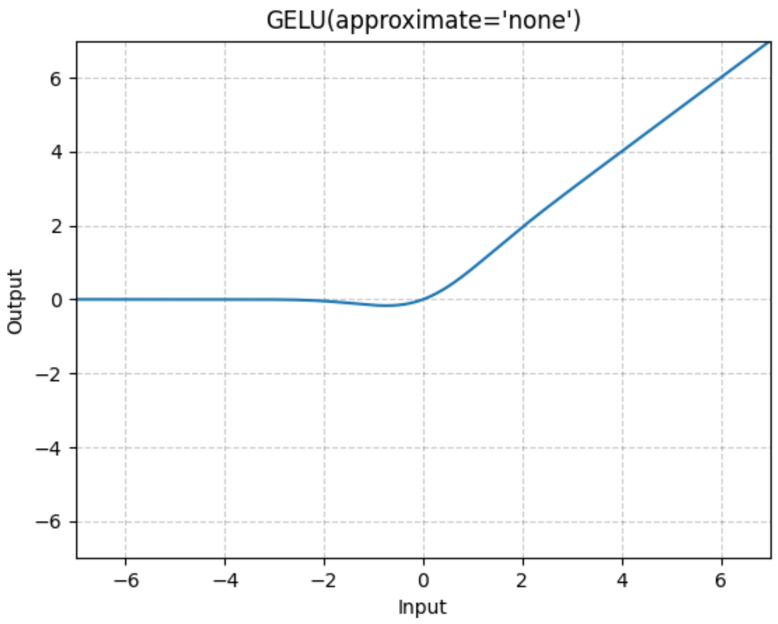
Applies the Gaussian Error Linear Units function.

$$\text{GELU}(x) = x * \Phi(x)$$

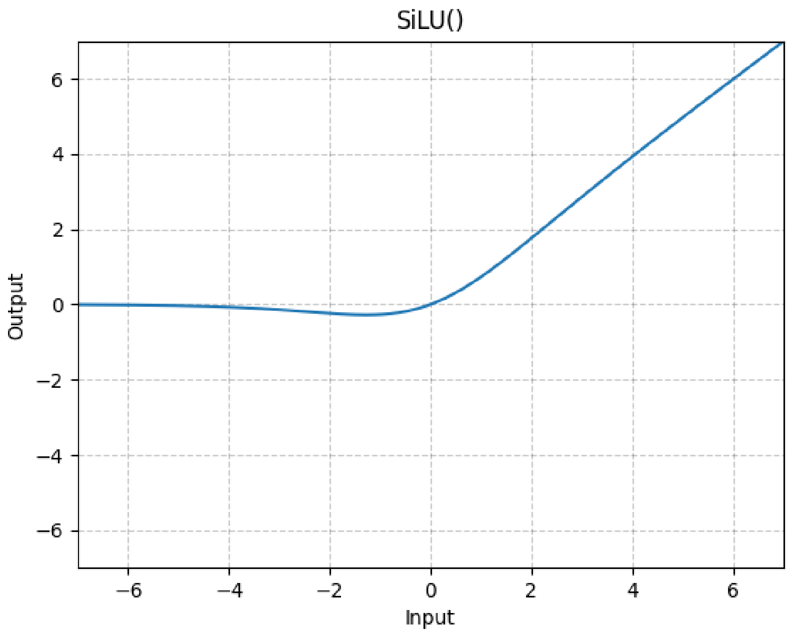
where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.

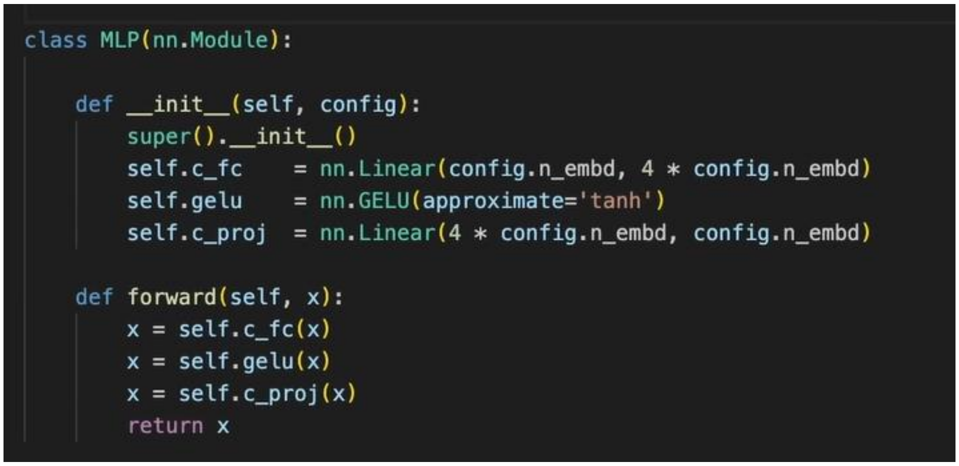
When the approximate argument is 'tanh', Gelu is estimated with:

$$\text{GELU}(x) = 0.5 * x * (1 + \text{Tanh}(\sqrt{2/\pi} * (x + 0.044715 * x^3)))$$



2. In GPT2 and Bert an approximation of GELU was used as at that time it was inefficient to computer GELU.
3. Why not use RELU?: It's because of the dead RELU neuron problem. In RELU if the input is less than 0, the output will always be 0. This causes the model to not learn anything in a lot of situations. GELU solves this by adding a slight curve for when the input is less than 0 (The bump slowly approaches 0). This allows the model to learn something instead of nothing. The research papers of GPT2 and Bert highlight the effectiveness in swapping to GELU.
4. Additional: In newer models such as LLaMA 3.1, this activation function further changes into SiLU. SiLU approaches 0 much slower and hence has a smoother bump.

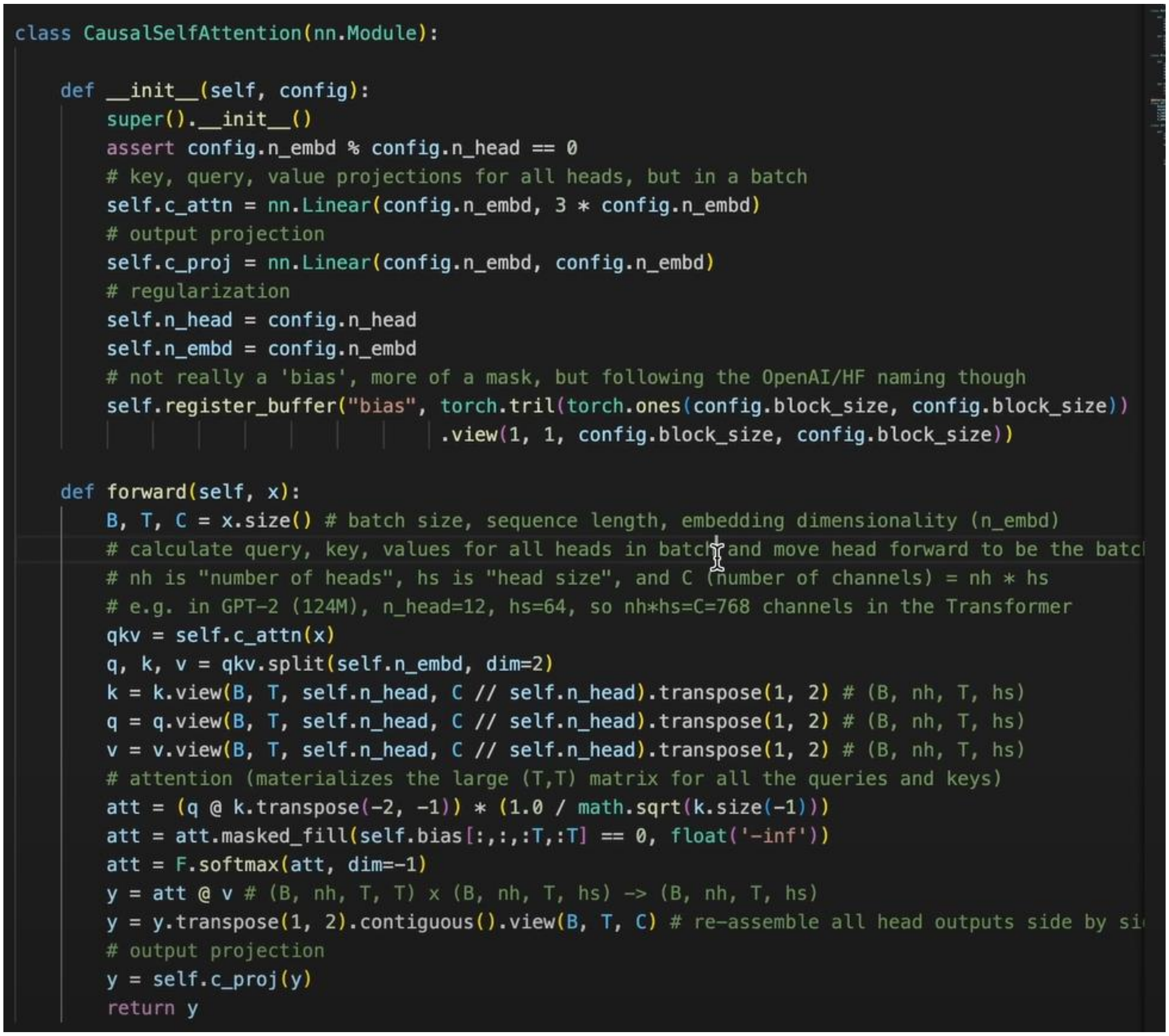




```python
class MLP(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.c_fc    = nn.Linear(config.n_embd, 4 * config.n_embd)
        self.gelu    = nn.GELU(approximate='tanh')
        self.c_proj  = nn.Linear(4 * config.n_embd, config.n_embd)

    def forward(self, x):
        x = self.c_fc(x)
        x = self.gelu(x)
        x = self.c_proj(x)
        return x
```

CasualSelfAttention

1. **IMPORTANT:** All variable names follow the hugging face GPT2 naming convention.



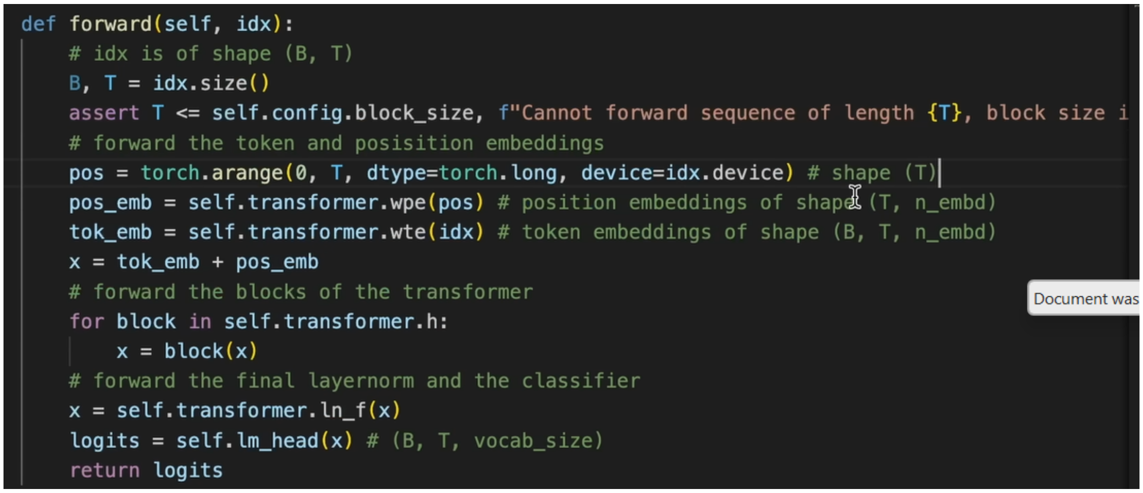
```python
class CausalSelfAttention(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd)
        # regularization
        self.n_head = config.n_head
        self.n_embd = config.n_embd
        # not really a 'bias', more of a mask, but following the OpenAI/HF naming though
        self.register_buffer("bias", torch.tril(torch.ones(config.block_size, config.block_size))
                                     .view(1, 1, config.block_size, config.block_size))

    def forward(self, x):
        B, T, C = x.size() # batch size, sequence length, embedding dimensionality (n_embd)
        # calculate query, key, values for all heads in batch and move head forward to be the batc
        # nh is "number of heads", hs is "head size", and C (number of channels) = nh * hs
        # e.g. in GPT-2 (124M), n_head=12, hs=64, so nh*hs=C=768 channels in the Transformer
        qkv = self.c_attn(x)
        q, k, v = qkv.split(self.n_embd, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        # attention (materializes the large (T,T) matrix for all the queries and keys)
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
        att = F.softmax(att, dim=-1)
        y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
        y = y.transpose(1, 2).contiguous().view(B, T, C) # re-assemble all head outputs side by si
        # output projection
        y = self.c_proj(y)
        return y
```

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

What to do next?:

1. Fully understand the above code.
2. Implement it with minimal references to code. Try to only refer to the notes and hugging face variable names.
3. Continue video (Should be the part where he downloads and uploads the weights from hugging face.)

GPT block: Forward function

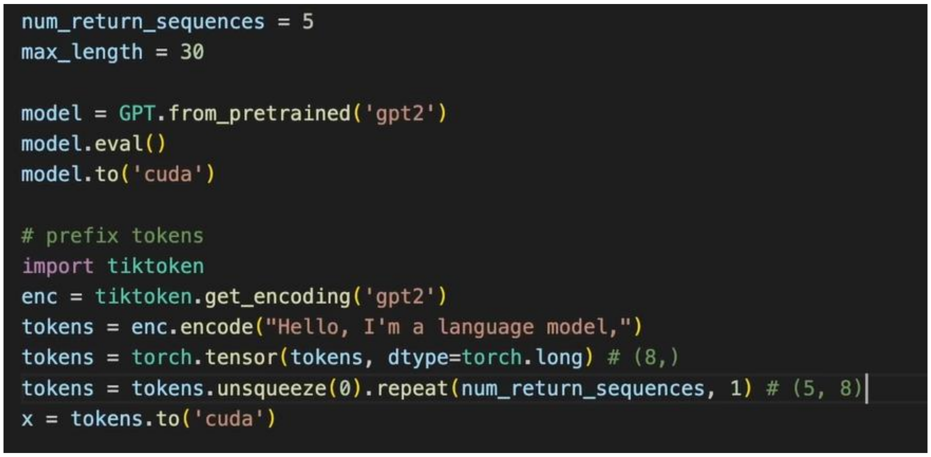


```
def forward(self, idx):
    # idx is of shape (B, T)
    B, T = idx.size()
    assert T <= self.config.block_size, f"Cannot forward sequence of length {T}, block size i
    # forward the token and posisition embeddings
    pos = torch.arange(0, T, dtype=torch.long, device=idx.device) # shape (T)
    pos_emb = self.transformer.wpe(pos) # position embeddings of shape (T, n_embd)
    tok_emb = self.transformer.wte(idx) # token embeddings of shape (B, T, n_embd)
    x = tok_emb + pos_emb
    # forward the blocks of the transformer
    for block in self.transformer.h:
        x = block(x)
    # forward the final layernorm and the classifier
    x = self.transformer.ln_f(x)
    logits = self.lm_head(x) # (B, T, vocab_size)
    return logits
```

Important notes:

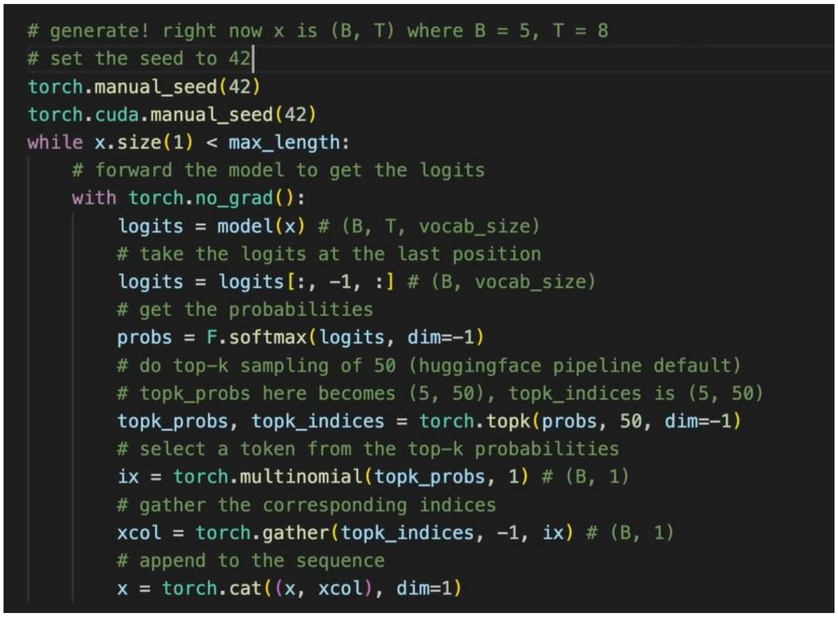
1. In the line where `pos` is defined, device=idx.device ensures that the tensor is stored in the GPU.
2. In the line where `x` is defined, tok_emb is broadcasted so that's its dimensions are compatible with pos_emb. The pos_embs are identical for each batch.
3. The forward function returns logits (not probabilities). These logits are later converted into probabilities during text generation.

<u>**Generating Text**</u>



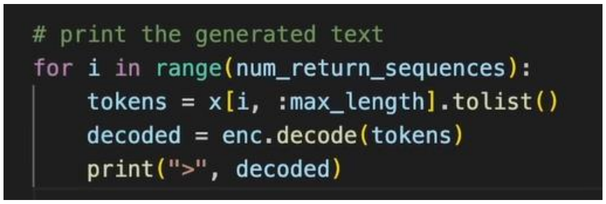
```python
num_return_sequences = 5
max_length = 30

model = GPT.from_pretrained('gpt2')
model.eval()
model.to('cuda')

# prefix tokens
import tiktoken
enc = tiktoken.get_encoding('gpt2')
tokens = enc.encode("Hello, I'm a language model,")
tokens = torch.tensor(tokens, dtype=torch.long) # (8,)
tokens = tokens.unsqueeze(0).repeat(num_return_sequences, 1) # (5, 8)
x = tokens.to('cuda')
```

1. Note: To generate a new model, initialize it as GPT(GPTconfig()).
2. tiktoken is the tokenizer used in GPT-2.
3. In this code, “Hello, I’m a language model” is the prompt for all 5 batches.
4. Set model to `eval()` mode.



```python
# generate! right now x is (B, T) where B = 5, T = 8
# set the seed to 42
torch.manual_seed(42)
torch.cuda.manual_seed(42)
while x.size(1) < max_length:
    # forward the model to get the logits
    with torch.no_grad():
        logits = model(x) # (B, T, vocab_size)
        # take the logits at the last position
        logits = logits[:, -1, :] # (B, vocab_size)
        # get the probabilities
        probs = F.softmax(logits, dim=-1)
        # do top-k sampling of 50 (huggingface pipeline default)
        # topk_probs here becomes (5, 50), topk_indices is (5, 50)
        topk_probs, topk_indices = torch.topk(probs, 50, dim=-1)
        # select a token from the top-k probabilities
        ix = torch.multinomial(topk_probs, 1) # (B, 1)
        # gather the corresponding indices
        xcol = torch.gather(topk_indices, -1, ix) # (B, 1)
        # append to the sequence
        x = torch.cat((x, xcol), dim=1)
```

5. max_length is total length of the `prompt + generated_text`.
6. `torch.no_grad()` indicates to PyTorch that no backpropagation takes place. This reduces the amount of space used by PyTorch in caching the intermediate tensors needed for backpropagation.
7. `torch.topk`, where k=50, is used to filter the 50 tokens with the highest probability and renormalize the probability distribution.
8. The next token is sampled using `torch.multinomial()`.



```python
# print the generated text
for i in range(num_return_sequences):
    tokens = x[i, :max_length].tolist()
    decoded = enc.decode(tokens)
    print(">", decoded)
```

9. Decode and print the generated text of length max_length tokens.

Continue - 46:01