

**Listing 4.7 The GPT model architecture implementation**

```

class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)

        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits

```

The device setting will allow us to train the model on a CPU or GPU, depending on which device the input data sits on.

Thanks to the `TransformerBlock` class, the `GPTModel` class is relatively small and compact.

The `__init__` constructor of this `GPTModel` class initializes the token and positional embedding layers using the configurations passed in via a Python dictionary, `cfg`. These embedding layers are responsible for converting input token indices into dense vectors and adding positional information (see chapter 2).

Next, the `__init__` method creates a sequential stack of `TransformerBlock` modules equal to the number of layers specified in `cfg`. Following the transformer blocks, a `LayerNorm` layer is applied, standardizing the outputs from the transformer blocks to stabilize the learning process. Finally, a linear output head without bias is defined, which projects the transformer's output into the vocabulary space of the tokenizer to generate logits for each token in the vocabulary.

The forward method takes a batch of input token indices, computes their embeddings, applies the positional embeddings, passes the sequence through the transformer blocks, normalizes the final output, and then computes the logits, representing the next token's unnormalized probabilities. We will convert these logits into tokens and text outputs in the next section.

Let's now initialize the 124-million-parameter GPT model using the `GPT_CONFIG_124M` dictionary we pass into the `cfg` parameter and feed it with the batch text input we previously created:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)

out = model(batch)
print("Input batch:\n", batch)
print("\nOutput shape:", out.shape)
print(out)
```

This code prints the contents of the input batch followed by the output tensor:

```
Input batch:
tensor([[6109, 3626, 6100, 345],
       [6109, 1110, 6622, 257]])           ↪ Token IDs of text 1
                                         ↪ Token IDs of text 2

Output shape: torch.Size([2, 4, 50257])
tensor([[[ 0.3613, 0.4222, -0.0711, ..., 0.3483, 0.4661, -0.2838],
         [-0.1792, -0.5660, -0.9485, ..., 0.0477, 0.5181, -0.3168],
         [ 0.7120, 0.0332, 0.1085, ..., 0.1018, -0.4327, -0.2553],
         [-1.0076, 0.3418, -0.1190, ..., 0.7195, 0.4023, 0.0532]],

        [[-0.2564, 0.0900, 0.0335, ..., 0.2659, 0.4454, -0.6806],
         [ 0.1230, 0.3653, -0.2074, ..., 0.7705, 0.2710, 0.2246],
         [ 1.0558, 1.0318, -0.2800, ..., 0.6936, 0.3205, -0.3178],
         [-0.1565, 0.3926, 0.3288, ..., 1.2630, -0.1858, 0.0388]]],  
grad_fn=<UnsafeViewBackward0>)
```

As we can see, the output tensor has the shape `[2, 4, 50257]`, since we passed in two input texts with four tokens each. The last dimension, 50257, corresponds to the vocabulary size of the tokenizer. Later, we will see how to convert each of these 50,257-dimensional output vectors back into tokens.

Before we move on to coding the function that converts the model outputs into text, let's spend a bit more time with the model architecture itself and analyze its size. Using the `numel()` method, short for "number of elements," we can collect the total number of parameters in the model's parameter tensors:

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Total number of parameters: {total_params:,}")
```

The result is

```
Total number of parameters: 163,009,536
```

Now, a curious reader might notice a discrepancy. Earlier, we spoke of initializing a 124-million-parameter GPT model, so why is the actual number of parameters 163 million?

The reason is a concept called *weight tying*, which was used in the original GPT-2 architecture. It means that the original GPT-2 architecture reuses the weights from the token embedding layer in its output layer. To understand better, let's take a look at the shapes of the token embedding layer and linear output layer that we initialized on the `model` via the `GPTModel` earlier:

```
print("Token embedding layer shape:", model.tok_emb.weight.shape)
print("Output layer shape:", model.out_head.weight.shape)
```

As we can see from the print outputs, the weight tensors for both these layers have the same shape:

```
Token embedding layer shape: torch.Size([50257, 768])
Output layer shape: torch.Size([50257, 768])
```

The token embedding and output layers are very large due to the number of rows for the 50,257 in the tokenizer's vocabulary. Let's remove the output layer parameter count from the total GPT-2 model count according to the weight tying:

```
total_params_gpt2 = (
    total_params - sum(p.numel()
        for p in model.out_head.parameters())
)
print(f"Number of trainable parameters "
      f"considering weight tying: {total_params_gpt2:, }"
)
```

The output is

```
Number of trainable parameters considering weight tying: 124,412,160
```

As we can see, the model is now only 124 million parameters large, matching the original size of the GPT-2 model.

Weight tying reduces the overall memory footprint and computational complexity of the model. However, in my experience, using separate token embedding and output layers results in better training and model performance; hence, we use separate layers in our `GPTModel` implementation. The same is true for modern LLMs. However, we will revisit and implement the weight tying concept later in chapter 6 when we load the pretrained weights from OpenAI.

### **Exercise 4.1 Number of parameters in feed forward and attention modules**

Calculate and compare the number of parameters that are contained in the feed forward module and those that are contained in the multi-head attention module.