4 Implementing a GPT model from scratch to generate text

This chapter covers

- Coding a GPT-like large language model (LLM) that can be trained to generate human-like text
- Normalizing layer activations to stabilize neural network training
- Adding shortcut connections in deep neural networks
- Implementing transformer blocks to create GPT models of various sizes
- Computing the number of parameters and storage requirements of GPT models

You've already learned and coded the *multi-head attention* mechanism, one of the core components of LLMs. Now, we will code the other building blocks of an LLM and assemble them into a GPT-like model that we will train in the next chapter to generate human-like text.

The LLM architecture referenced in figure 4.1, consists of several building blocks. We will begin with a top-down view of the model architecture before covering the individual components in more detail.

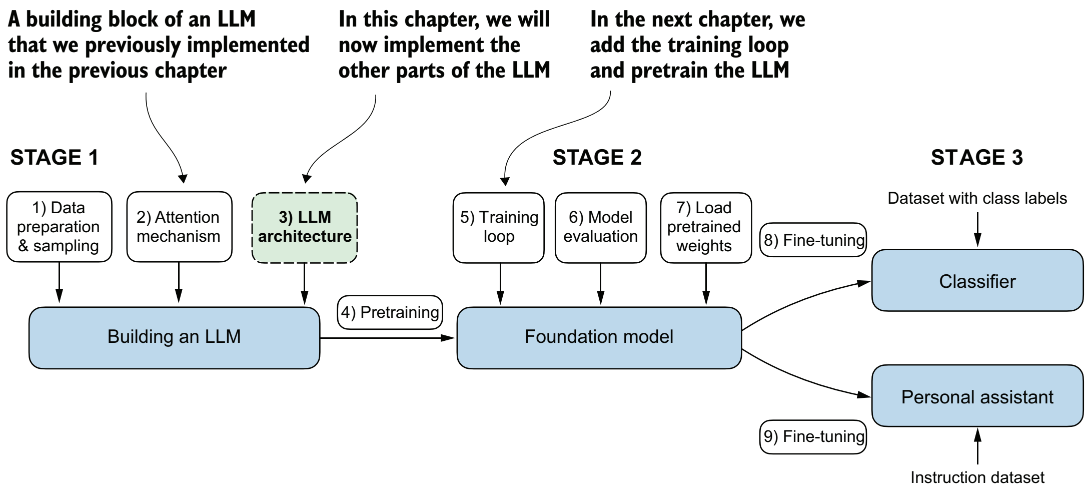

A building block of an LLM that we previously implemented in the previous chapter
In this chapter, we will now implement the other parts of the LLM
In the next chapter, we add the training loop and pretrain the LLM
STAGE 1
STAGE 2
STAGE 3
1) Data preparation & sampling
2) Attention mechanism
3) LLM architecture
5) Training loop
6) Model evaluation
7) Load pretrained weights
8) Fine-tuning
Dataset with class labels
Classifier
Building an LLM
4) Pretraining
Foundation model
Personal assistant
9) Fine-tuning
Instruction dataset

Figure 4.1 The three main stages of coding an LLM. This chapter focuses on step 3 of stage 1: implementing the LLM architecture.

4.1 Coding an LLM architecture

LLMs, such as GPT (which stands for *generative pretrained transformer*), are large deep neural network architectures designed to generate new text one word (or token) at a time. However, despite their size, the model architecture is less complicated than you might think, since many of its components are repeated, as we will see later. Figure 4.2 provides a top-down view of a GPT-like LLM, with its main components highlighted.

We have already covered several aspects of the LLM architecture, such as input tokenization and embedding and the masked multi-head attention module. Now, we will implement the core structure of the GPT model, including its *transformer blocks*, which we will later train to generate human-like text.

Previously, we used smaller embedding dimensions for simplicity, ensuring that the concepts and examples could comfortably fit on a single page. Now, we are scaling up to the size of a small GPT-2 model, specifically the smallest version with 124 million parameters, as described in "Language Models Are Unsupervised Multitask Learners," by Radford et al. (https://mng.bz/yoBq). Note that while the original report mentions 117 million parameters, this was later corrected. In chapter 6, we will focus on loading pretrained weights into our implementation and adapting it for larger GPT-2 models with 345, 762, and 1,542 million parameters.

In the context of deep learning and LLMs like GPT, the term "parameters" refers to the trainable weights of the model. These weights are essentially the internal variables of the model that are adjusted and optimized during the training process to minimize a specific loss function. This optimization allows the model to learn from the training data.

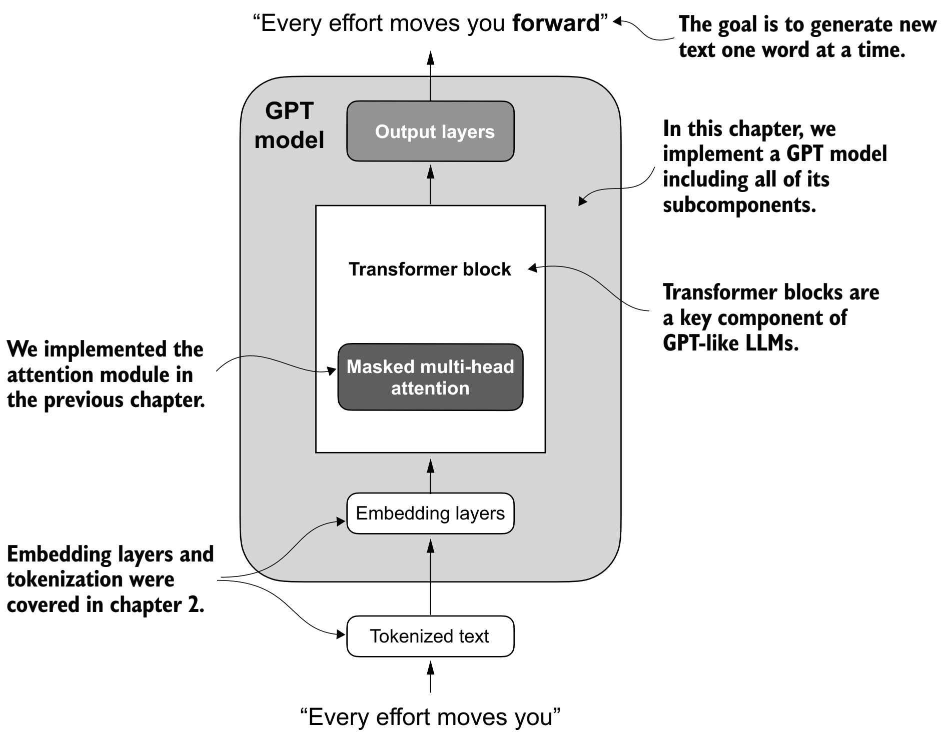

"Every effort moves you **forward**"
The goal is to generate new text one word at a time.
GPT model
Output layers
In this chapter, we implement a GPT model including all of its subcomponents.
Transformer block
Transformer blocks are a key component of GPT-like LLMs.
We implemented the attention module in the previous chapter.
Masked multi-head attention
Embedding layers
Embedding layers and tokenization were covered in chapter 2.
Tokenized text
"Every effort moves you"

Figure 4.2 A GPT model. In addition to the embedding layers, it consists of one or more transformer blocks containing the masked multi-head attention module we previously implemented.

For example, in a neural network layer that is represented by a 2,048 × 2,048–dimensional matrix (or tensor) of weights, each element of this matrix is a parameter. Since there are 2,048 rows and 2,048 columns, the total number of parameters in this layer is 2,048 multiplied by 2,048, which equals 4,194,304 parameters.

GPT-2 vs. GPT-3

Note that we are focusing on GPT-2 because OpenAI has made the weights of the pretrained model publicly available, which we will load into our implementation in chapter 6. GPT-3 is fundamentally the same in terms of model architecture, except that it is scaled up from 1.5 billion parameters in GPT-2 to 175 billion parameters in GPT-3, and it is trained on more data. As of this writing, the weights for GPT-3 are not publicly available. GPT-2 is also a better choice for learning how to implement LLMs, as it can be run on a single laptop computer, whereas GPT-3 requires a GPU cluster for training and inference. According to Lambda Labs (https://lambdalabs.com/), it would take 355 years to train GPT-3 on a single V100 datacenter GPU and 665 years on a consumer RTX 8000 GPU.

We specify the configuration of the small GPT-2 model via the following Python dictionary, which we will use in the code examples later:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,     # Vocabulary size
    "context_length": 1024,  # Context length
    "emb_dim": 768,          # Embedding dimension
    "n_heads": 12,           # Number of attention heads
    "n_layers": 12,          # Number of layers
    "drop_rate": 0.1,        # Dropout rate
    "qkv_bias": False        # Query-Key-Value bias
}
```

In the `GPT_CONFIG_124M` dictionary, we use concise variable names for clarity and to prevent long lines of code:

- `vocab_size` refers to a vocabulary of 50,257 words, as used by the BPE tokenizer (see chapter 2).
- `context_length` denotes the maximum number of input tokens the model can handle via the positional embeddings (see chapter 2).
- `emb_dim` represents the embedding size, transforming each token into a 768-dimensional vector.
- `n_heads` indicates the count of attention heads in the multi-head attention mechanism (see chapter 3).
- `n_layers` specifies the number of transformer blocks in the model, which we will cover in the upcoming discussion.
- `drop_rate` indicates the intensity of the dropout mechanism (0.1 implies a 10% random drop out of hidden units) to prevent overfitting (see chapter 3).
- `qkv_bias` determines whether to include a bias vector in the `Linear` layers of the multi-head attention for query, key, and value computations. We will initially disable this, following the norms of modern LLMs, but we will revisit it in chapter 6 when we load pretrained GPT-2 weights from OpenAI into our model (see chapter 6).

Using this configuration, we will implement a GPT placeholder architecture (`DummyGPTModel`), as shown in figure 4.3. This will provide us with a big-picture view of how everything fits together and what other components we need to code to assemble the full GPT model architecture.

The numbered boxes in figure 4.3 illustrate the order in which we tackle the individual concepts required to code the final GPT architecture. We will start with step 1, a placeholder GPT backbone we will call `DummyGPTModel`.

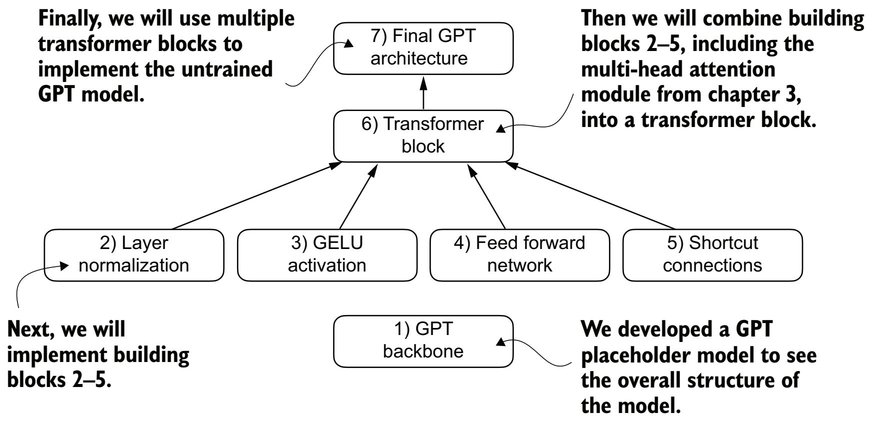

Finally, we will use multiple transformer blocks to implement the untrained GPT model.
7) Final GPT architecture
Then we will combine building blocks 2–5, including the multi-head attention module from chapter 3, into a transformer block.
6) Transformer block
2) Layer normalization
3) GELU activation
4) Feed forward network
5) Shortcut connections
Next, we will implement building blocks 2–5.
1) GPT backbone
We developed a GPT placeholder model to see the overall structure of the model.

Figure 4.3 The order in which we code the GPT architecture. We start with the GPT backbone, a placeholder architecture, before getting to the individual core pieces and eventually assembling them in a transformer block for the final GPT architecture.

Listing 4.1 A placeholder GPT model architecture class

```
import torch
import torch.nn as nn

class DummyGPTModel(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(
            *[DummyTransformerBlock(cfg)
              for _ in range(cfg["n_layers"])]
        )
        self.final_norm = DummyLayerNorm(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
```

Uses a placeholder for TransformerBlock

Uses a placeholder for LayerNorm

```
class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()

    def forward(self, x):
        return x

class DummyLayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()

    def forward(self, x):
        return x
```

A simple placeholder class that will be replaced by a real TransformerBlock later

This block does nothing and just returns its input.

A simple placeholder class that will be replaced by a real LayerNorm later

The parameters here are just to mimic the LayerNorm interface.

The `DummyGPTModel` class in this code defines a simplified version of a GPT-like model using PyTorch's neural network module (`nn.Module`). The model architecture in the `DummyGPTModel` class consists of token and positional embeddings, dropout, a series of transformer blocks (`DummyTransformerBlock`), a final layer normalization (`DummyLayerNorm`), and a linear output layer (`out_head`). The configuration is passed in via a Python dictionary, for instance, the `GPT_CONFIG_124M` dictionary we created earlier.

The `forward` method describes the data flow through the model: it computes token and positional embeddings for the input indices, applies dropout, processes the data through the transformer blocks, applies normalization, and finally produces logits with the linear output layer.

The code in listing 4.1 is already functional. However, for now, note that we use placeholders (`DummyLayerNorm` and `DummyTransformerBlock`) for the transformer block and layer normalization, which we will develop later.

Next, we will prepare the input data and initialize a new GPT model to illustrate its usage. Building on our coding of the tokenizer (see chapter 2), let's now consider a high-level overview of how data flows in and out of a GPT model, as shown in figure 4.4.

To implement these steps, we tokenize a batch consisting of two text inputs for the GPT model using the tiktoken tokenizer from chapter 2:

```
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
batch = []
txt1 = "Every effort moves you"
txt2 = "Every day holds a"

batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
print(batch)
```

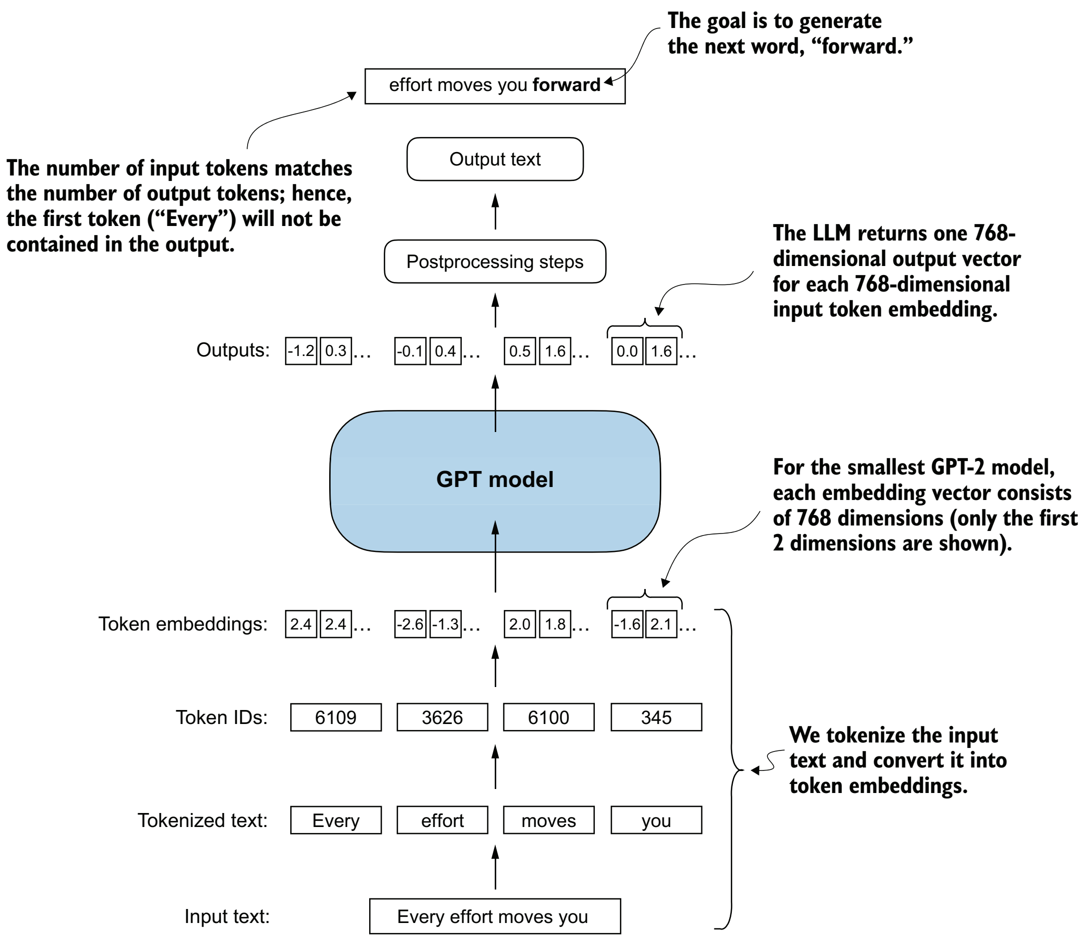

The goal is to generate the next word, "forward."
effort moves you **forward**
The number of input tokens matches the number of output tokens; hence, the first token ("Every") will not be contained in the output.
Output text
Postprocessing steps
The LLM returns one 768-dimensional output vector for each 768-dimensional input token embedding.
Outputs: -1.2 0.3 … -0.1 0.4 … 0.5 1.6 … 0.0 1.6 …
GPT model
For the smallest GPT-2 model, each embedding vector consists of 768 dimensions (only the first 2 dimensions are shown).
Token embeddings: 2.4 2.4 … -2.6 -1.3 … 2.0 1.8 … -1.6 2.1 …
Token IDs: 6109 3626 6100 345
Tokenized text: Every effort moves you
Input text: Every effort moves you
We tokenize the input text and convert it into token embeddings.

Figure 4.4 A big-picture overview showing how the input data is tokenized, embedded, and fed to the GPT model. Note that in our `DummyGPTClass` coded earlier, the token embedding is handled inside the GPT model. In LLMs, the embedded input token dimension typically matches the output dimension. The output embeddings here represent the context vectors (see chapter 3).

The resulting token IDs for the two texts are as follows:

```
tensor([[6109,  3626,  6100,   345],
        [6109,  1110,  6622,   257]])
```

The first row corresponds to the first text, and the second row corresponds to the second text.

Next, we initialize a new 124-million-parameter `DummyGPTModel` instance and feed it the tokenized `batch`:

```
torch.manual_seed(123)
model = DummyGPTModel(GPT_CONFIG_124M)
logits = model(batch)
print("Output shape:", logits.shape)
print(logits)
```

The model outputs, which are commonly referred to as logits, are as follows:

```
Output shape: torch.Size([2, 4, 50257])
tensor([[[-1.2034,  0.3201, -0.7130,  ..., -1.5548, -0.2390, -0.4667],
         [-0.1192,  0.4539, -0.4432,  ...,  0.2392,  1.3469,  1.2430],
         [ 0.5307,  1.6720, -0.4695,  ...,  1.1966,  0.0111,  0.5835],
         [ 0.0139,  1.6755, -0.3388,  ...,  1.1586, -0.0435, -1.0400]],

        [[-1.0908,  0.1798, -0.9484,  ..., -1.6047,  0.2439, -0.4530],
         [-0.7860,  0.5581, -0.0610,  ...,  0.4835, -0.0077,  1.6621],
         [ 0.3567,  1.2698, -0.6398,  ..., -0.0162, -0.1296,  0.3717],
         [-0.2407, -0.7349, -0.5102,  ...,  2.0057, -0.3694,  0.1814]]],
       grad_fn=<UnsafeViewBackward0>)
```

The output tensor has two rows corresponding to the two text samples. Each text sample consists of four tokens; each token is a 50,257-dimensional vector, which matches the size of the tokenizer's vocabulary.

The embedding has 50,257 dimensions because each of these dimensions refers to a unique token in the vocabulary. When we implement the postprocessing code, we will convert these 50,257-dimensional vectors back into token IDs, which we can then decode into words.

Now that we have taken a top-down look at the GPT architecture and its inputs and outputs, we will code the individual placeholders, starting with the real layer normalization class that will replace the `DummyLayerNorm` in the previous code.

4.2 Normalizing activations with layer normalization

Training deep neural networks with many layers can sometimes prove challenging due to problems like vanishing or exploding gradients. These problems lead to unstable training dynamics and make it difficult for the network to effectively adjust its weights, which means the learning process struggles to find a set of parameters (weights) for the neural network that minimizes the loss function. In other words, the network has difficulty learning the underlying patterns in the data to a degree that would allow it to make accurate predictions or decisions.

> **NOTE** If you are new to neural network training and the concepts of gradients, a brief introduction to these concepts can be found in section A.4 in appendix A. However, a deep mathematical understanding of gradients is not required to follow the contents of this book.

Let's now implement *layer normalization* to improve the stability and efficiency of neural network training. The main idea behind layer normalization is to adjust the activations (outputs) of a neural network layer to have a mean of 0 and a variance of 1, also known as unit variance. This adjustment speeds up the convergence to effective weights and ensures consistent, reliable training. In GPT-2 and modern transformer architectures, layer normalization is typically applied before and after the multi-head attention module, and, as we have seen with the `DummyLayerNorm` placeholder, before the final output layer. Figure 4.5 provides a visual overview of how layer normalization functions.

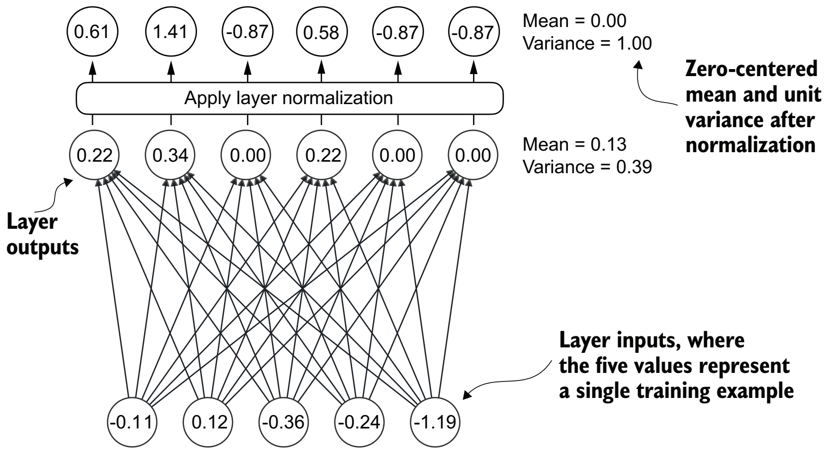

0.61
1.41
-0.87
0.58
-0.87
-0.87
Mean = 0.00
Variance = 1.00
Zero-centered mean and unit variance after normalization
Apply layer normalization
0.22
0.34
0.00
0.22
0.00
0.00
Mean = 0.13
Variance = 0.39
Layer outputs
Layer inputs, where the five values represent a single training example
-0.11
0.12
-0.36
-0.24
-1.19

Figure 4.5 An illustration of layer normalization where the six outputs of the layer, also called activations, are normalized such that they have a 0 mean and a variance of 1.

We can recreate the example shown in figure 4.5 via the following code, where we implement a neural network layer with five inputs and six outputs that we apply to two input examples:

```
torch.manual_seed(123)
batch_example = torch.randn(2, 5)
layer = nn.Sequential(nn.Linear(5, 6), nn.ReLU())
out = layer(batch_example)
print(out)
```

Creates two training examples with five dimensions (features) each

This prints the following tensor, where the first row lists the layer outputs for the first input and the second row lists the layer outputs for the second row:

```
tensor([[0.2260, 0.3470, 0.0000, 0.2216, 0.0000, 0.0000],
        [0.2133, 0.2394, 0.0000, 0.5198, 0.3297, 0.0000]],
       grad_fn=<ReluBackward0>)
```

The neural network layer we have coded consists of a `Linear` layer followed by a non-linear activation function, `ReLU` (short for rectified linear unit), which is a standard activation function in neural networks. If you are unfamiliar with `ReLU`, it simply thresholds negative inputs to 0, ensuring that a layer outputs only positive values, which explains why the resulting layer output does not contain any negative values. Later, we will use another, more sophisticated activation function in GPT.

Before we apply layer normalization to these outputs, let's examine the mean and variance:

```
mean = out.mean(dim=-1, keepdim=True)
var = out.var(dim=-1, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

The output is

```
Mean:
  tensor([[0.1324],
          [0.2170]], grad_fn=<MeanBackward1>)
Variance:
  tensor([[0.0231],
          [0.0398]], grad_fn=<VarBackward0>)
```

The first row in the mean tensor here contains the mean value for the first input row, and the second output row contains the mean for the second input row.

Using `keepdim=True` in operations like mean or variance calculation ensures that the output tensor retains the same number of dimensions as the input tensor, even though the operation reduces the tensor along the dimension specified via `dim`. For instance, without `keepdim=True`, the returned mean tensor would be a two-dimensional vector `[0.1324, 0.2170]` instead of a 2 × 1–dimensional matrix `[[0.1324], [0.2170]]`.

The `dim` parameter specifies the dimension along which the calculation of the statistic (here, mean or variance) should be performed in a tensor. As figure 4.6 explains, for

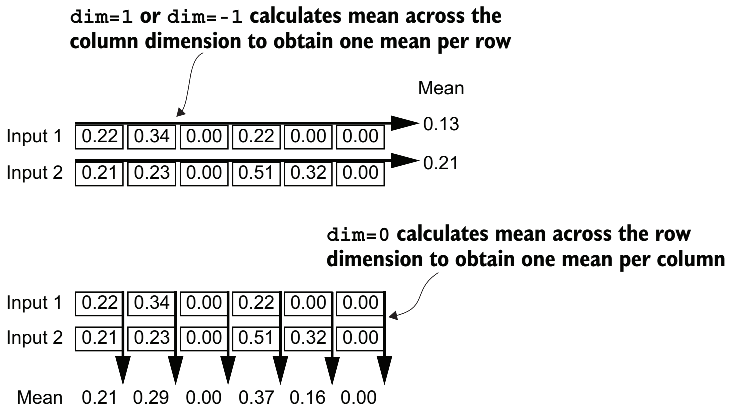

dim=1 or dim=-1 calculates mean across the column dimension to obtain one mean per row
Mean
Input 1 0.22 0.34 0.00 0.22 0.00 0.00 → 0.13
Input 2 0.21 0.23 0.00 0.51 0.32 0.00 → 0.21
dim=0 calculates mean across the row dimension to obtain one mean per column
Input 1 0.22 0.34 0.00 0.22 0.00 0.00
Input 2 0.21 0.23 0.00 0.51 0.32 0.00
Mean 0.21 0.29 0.00 0.37 0.16 0.00

Figure 4.6 An illustration of the dim parameter when calculating the mean of a tensor. For instance, if we have a two-dimensional tensor (matrix) with dimensions `[rows, columns]`, using `dim=0` will perform the operation across rows (vertically, as shown at the bottom), resulting in an output that aggregates the data for each column. Using `dim=1` or `dim=-1` will perform the operation across columns (horizontally, as shown at the top), resulting in an output aggregating the data for each row.

a two-dimensional tensor (like a matrix), using `dim=-1` for operations such as mean or variance calculation is the same as using `dim=1`. This is because `-1` refers to the tensor's last dimension, which corresponds to the columns in a two-dimensional tensor. Later, when adding layer normalization to the GPT model, which produces three-dimensional tensors with the shape `[batch_size, num_tokens, embedding_size]`, we can still use `dim=-1` for normalization across the last dimension, avoiding a change from `dim=1` to `dim=2`.

Next, let's apply layer normalization to the layer outputs we obtained earlier. The operation consists of subtracting the mean and dividing by the square root of the variance (also known as the standard deviation):

```
out_norm = (out - mean) / torch.sqrt(var)
mean = out_norm.mean(dim=-1, keepdim=True)
var = out_norm.var(dim=-1, keepdim=True)
print("Normalized layer outputs:\n", out_norm)
print("Mean:\n", mean)
print("Variance:\n", var)
```

As we can see based on the results, the normalized layer outputs, which now also contain negative values, have 0 mean and a variance of 1:

```
Normalized layer outputs:
 tensor([[ 0.6159,  1.4126, -0.8719,  0.5872, -0.8719, -0.8719],
        [-0.0189,  0.1121, -1.0876,  1.5173,  0.5647, -1.0876]],
       grad_fn=<DivBackward0>)
Mean:
 tensor([[-5.9605e-08],
        [1.9868e-08]], grad_fn=<MeanBackward1>)
Variance:
 tensor([[1.],
        [1.]], grad_fn=<VarBackward0>)
```

Note that the value –5.9605e-08 in the output tensor is the scientific notation for -5.9605×10^{-8}, which is –0.000000059605 in decimal form. This value is very close to 0, but it is not exactly 0 due to small numerical errors that can accumulate because of the finite precision with which computers represent numbers.

To improve readability, we can also turn off the scientific notation when printing tensor values by setting `sci_mode` to `False`:

```
torch.set_printoptions(sci_mode=False)
print("Mean:\n", mean)
print("Variance:\n", var)
```

The output is

```
Mean:
 tensor([[    0.0000],
        [    0.0000]], grad_fn=<MeanBackward1>)
```

```
Variance:
 tensor([[1.],
        [1.]], grad_fn=<VarBackward0>)
```

So far, we have coded and applied layer normalization in a step-by-step process. Let's now encapsulate this process in a PyTorch module that we can use in the GPT model later.

Listing 4.2 A layer normalization class

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift
```

This specific implementation of layer normalization operates on the last dimension of the input tensor x, which represents the embedding dimension (`emb_dim`). The variable `eps` is a small constant (epsilon) added to the variance to prevent division by zero during normalization. The `scale` and `shift` are two trainable parameters (of the same dimension as the input) that the LLM automatically adjusts during training if it is determined that doing so would improve the model's performance on its training task. This allows the model to learn appropriate scaling and shifting that best suit the data it is processing.

Biased variance

In our variance calculation method, we use an implementation detail by setting `unbiased=False`. For those curious about what this means, in the variance calculation, we divide by the number of inputs *n* in the variance formula. This approach does not apply Bessel's correction, which typically uses $n - 1$ instead of *n* in the denominator to adjust for bias in sample variance estimation. This decision results in a so-called biased estimate of the variance. For LLMs, where the embedding dimension *n* is significantly large, the difference between using *n* and $n - 1$ is practically negligible. I chose this approach to ensure compatibility with the GPT-2 model's normalization layers and because it reflects TensorFlow's default behavior, which was used to implement the original GPT-2 model. Using a similar setting ensures our method is compatible with the pretrained weights we will load in chapter 6.

Let's now try the `LayerNorm` module in practice and apply it to the batch input:

```
ln = LayerNorm(emb_dim=5)
out_ln = ln(batch_example)
mean = out_ln.mean(dim=-1, keepdim=True)
var = out_ln.var(dim=-1, unbiased=False, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

The results show that the layer normalization code works as expected and normalizes the values of each of the two inputs such that they have a mean of 0 and a variance of 1:

```
Mean:
 tensor([[    -0.0000],
        [     0.0000]], grad_fn=<MeanBackward1>)
Variance:
 tensor([[1.0000],
        [1.0000]], grad_fn=<VarBackward0>)
```

We have now covered two of the building blocks we will need to implement the GPT architecture, as shown in figure 4.7. Next, we will look at the GELU activation function, which is one of the activation functions used in LLMs, instead of the traditional ReLU function we used previously.

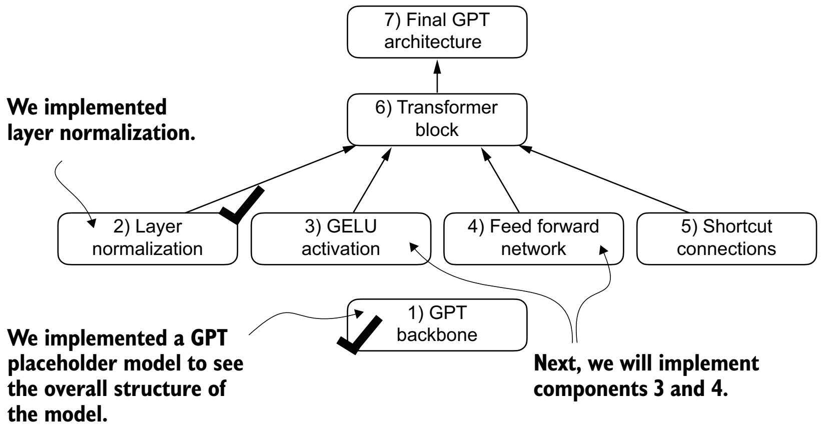

7) Final GPT architecture
6) Transformer block
We implemented layer normalization.
2) Layer normalization
3) GELU activation
4) Feed forward network
5) Shortcut connections
1) GPT backbone
We implemented a GPT placeholder model to see the overall structure of the model.
Next, we will implement components 3 and 4.

Figure 4.7 The building blocks necessary to build the GPT architecture. So far, we have completed the GPT backbone and layer normalization. Next, we will focus on GELU activation and the feed forward network.

Layer normalization vs. batch normalization

If you are familiar with batch normalization, a common and traditional normalization method for neural networks, you may wonder how it compares to layer normalization. Unlike batch normalization, which normalizes across the batch dimension, layer normalization normalizes across the feature dimension. LLMs often require significant computational resources, and the available hardware or the specific use case can dictate the batch size during training or inference. Since layer normalization normalizes each input independently of the batch size, it offers more flexibility and stability in these scenarios. This is particularly beneficial for distributed training or when deploying models in environments where resources are constrained.

4.3 Implementing a feed forward network with GELU activations

Next, we will implement a small neural network submodule used as part of the transformer block in LLMs. We begin by implementing the *GELU* activation function, which plays a crucial role in this neural network submodule.

NOTE For additional information on implementing neural networks in PyTorch, see section A.5 in appendix A.

Historically, the ReLU activation function has been commonly used in deep learning due to its simplicity and effectiveness across various neural network architectures. However, in LLMs, several other activation functions are employed beyond the traditional ReLU. Two notable examples are GELU (*Gaussian error linear unit*) and SwiGLU (*Swish-gated linear unit*).

GELU and SwiGLU are more complex and smooth activation functions incorporating Gaussian and sigmoid-gated linear units, respectively. They offer improved performance for deep learning models, unlike the simpler ReLU.

The GELU activation function can be implemented in several ways; the exact version is defined as GELU(x) = x⋅Φ(x), where Φ(x) is the cumulative distribution function of the standard Gaussian distribution. In practice, however, it's common to implement a computationally cheaper approximation (the original GPT-2 model was also trained with this approximation, which was found via curve fitting):

$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + tanh\left[\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right)$$

In code, we can implement this function as a PyTorch module.

Listing 4.3 An implementation of the GELU activation function

```
class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0 / torch.pi)) *
            (x + 0.044715 * torch.pow(x, 3))
        ))
```

Next, to get an idea of what this GELU function looks like and how it compares to the ReLU function, let's plot these functions side by side:

```
import matplotlib.pyplot as plt
gelu, relu = GELU(), nn.ReLU()

x = torch.linspace(-3, 3, 100)     <-- Creates 100 sample data points in the range –3 to 3
y_gelu, y_relu = gelu(x), relu(x)
plt.figure(figsize=(8, 3))
for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "ReLU"]), 1):
    plt.subplot(1, 2, i)
    plt.plot(x, y)
    plt.title(f"{label} activation function")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)
plt.tight_layout()
plt.show()
```

As we can see in the resulting plot in figure 4.8, ReLU (right) is a piecewise linear function that outputs the input directly if it is positive; otherwise, it outputs zero. GELU (left) is a smooth, nonlinear function that approximates ReLU but with a non-zero gradient for almost all negative values (except at approximately $x = -0.75$).

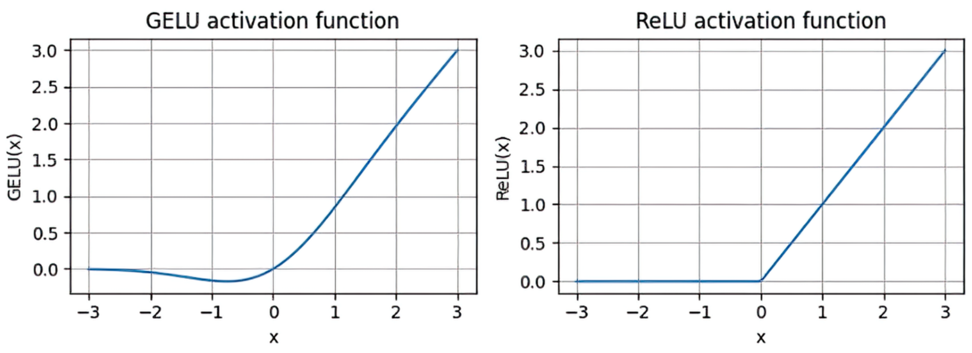


Figure 4.8 The output of the GELU and ReLU plots using matplotlib. The x-axis shows the function inputs and the y-axis shows the function outputs.

The smoothness of GELU can lead to better optimization properties during training, as it allows for more nuanced adjustments to the model's parameters. In contrast, ReLU has a sharp corner at zero (figure 4.18, right), which can sometimes make optimization harder, especially in networks that are very deep or have complex architectures. Moreover, unlike ReLU, which outputs zero for any negative input, GELU allows for a small, non-zero output for negative values. This characteristic means that during the training process, neurons that receive negative input can still contribute to the learning process, albeit to a lesser extent than positive inputs.

Next, let's use the GELU function to implement the small neural network module, `FeedForward`, that we will be using in the LLM's transformer block later.

Listing 4.4 A feed forward neural network module

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

    def forward(self, x):
        return self.layers(x)
```

As we can see, the `FeedForward` module is a small neural network consisting of two `Linear` layers and a `GELU` activation function. In the 124-million-parameter GPT model, it receives the input batches with tokens that have an embedding size of 768 each via the `GPT_CONFIG_124M` dictionary where `GPT_CONFIG_ 124M["emb_dim"] = 768`. Figure 4.9 shows how the embedding size is manipulated inside this small feed forward neural network when we pass it some inputs.

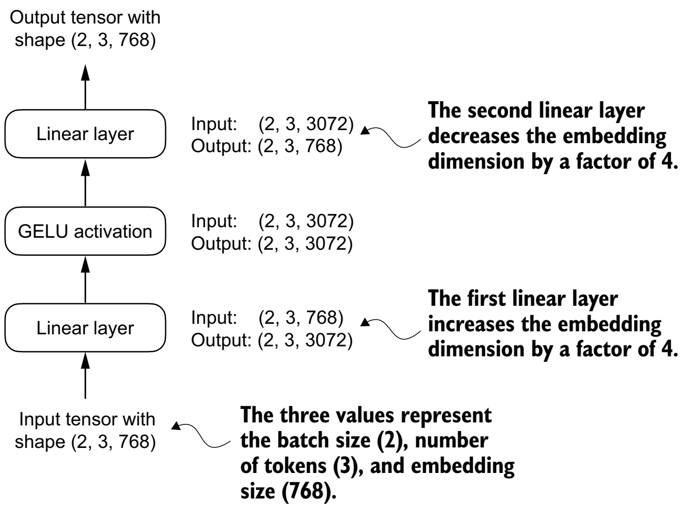

Output tensor with shape (2, 3, 768)
Linear layer
Input: (2, 3, 3072)
Output: (2, 3, 768)
The second linear layer decreases the embedding dimension by a factor of 4.
GELU activation
Input: (2, 3, 3072)
Output: (2, 3, 3072)
Linear layer
Input: (2, 3, 768)
Output: (2, 3, 3072)
The first linear layer increases the embedding dimension by a factor of 4.
Input tensor with shape (2, 3, 768)
The three values represent the batch size (2), number of tokens (3), and embedding size (768).

Figure 4.9 An overview of the connections between the layers of the feed forward neural network. This neural network can accommodate variable batch sizes and numbers of tokens in the input. However, the embedding size for each token is determined and fixed when initializing the weights.

Following the example in figure 4.9, let's initialize a new `FeedForward` module with a token embedding size of 768 and feed it a batch input with two samples and three tokens each:

```
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768)          <-- Creates sample input
out = ffn(x)                           with batch dimension 2
print(out.shape)
```

As we can see, the shape of the output tensor is the same as that of the input tensor:

```
torch.Size([2, 3, 768])
```

The `FeedForward` module plays a crucial role in enhancing the model's ability to learn from and generalize the data. Although the input and output dimensions of this module are the same, it internally expands the embedding dimension into a higher-dimensional space through the first linear layer, as illustrated in figure 4.10. This expansion is followed by a nonlinear GELU activation and then a contraction back to the original dimension with the second linear transformation. Such a design allows for the exploration of a richer representation space.

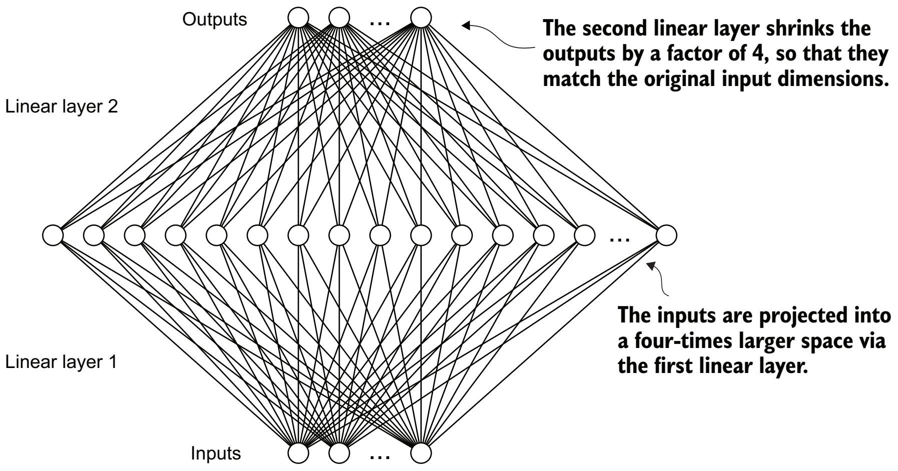

Outputs
The second linear layer shrinks the outputs by a factor of 4, so that they match the original input dimensions.
Linear layer 2
The inputs are projected into a four-times larger space via the first linear layer.
Linear layer 1
Inputs

Figure 4.10 An illustration of the expansion and contraction of the layer outputs in the feed forward neural network. First, the inputs expand by a factor of 4 from 768 to 3,072 values. Then, the second layer compresses the 3,072 values back into a 768-dimensional representation.

Moreover, the uniformity in input and output dimensions simplifies the architecture by enabling the stacking of multiple layers, as we will do later, without the need to adjust dimensions between them, thus making the model more scalable.

As figure 4.11 shows, we have now implemented most of the LLM's building blocks. Next, we will go over the concept of shortcut connections that we insert between different layers of a neural network, which are important for improving the training performance in deep neural network architectures.

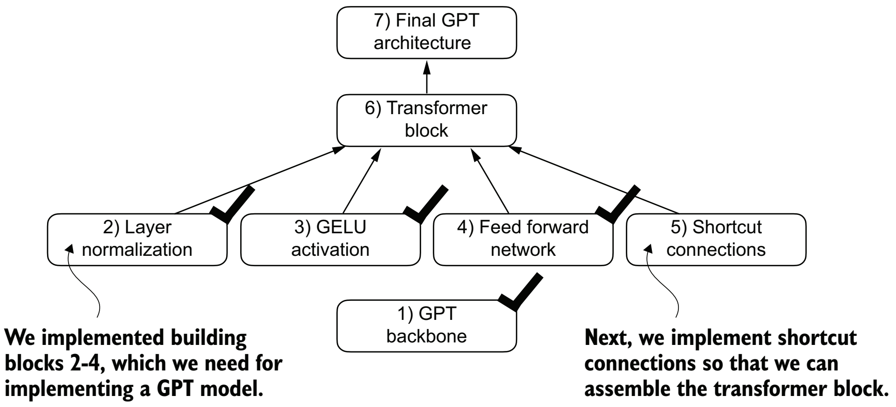

7) Final GPT architecture
6) Transformer block
2) Layer normalization
3) GELU activation
4) Feed forward network
5) Shortcut connections
1) GPT backbone
We implemented building blocks 2-4, which we need for implementing a GPT model.
Next, we implement shortcut connections so that we can assemble the transformer block.

Figure 4.11 The building blocks necessary to build the GPT architecture. The black checkmarks indicating those we have already covered.

4.4 Adding shortcut connections

Let's discuss the concept behind *shortcut connections*, also known as skip or residual connections. Originally, shortcut connections were proposed for deep networks in computer vision (specifically, in residual networks) to mitigate the challenge of vanishing gradients. The vanishing gradient problem refers to the issue where gradients (which guide weight updates during training) become progressively smaller as they propagate backward through the layers, making it difficult to effectively train earlier layers.

Figure 4.12 shows that a shortcut connection creates an alternative, shorter path for the gradient to flow through the network by skipping one or more layers, which is achieved by adding the output of one layer to the output of a later layer. This is why these connections are also known as skip connections. They play a crucial role in preserving the flow of gradients during the backward pass in training.

In the following list, we implement the neural network in figure 4.12 to see how we can add shortcut connections in the `forward` method.

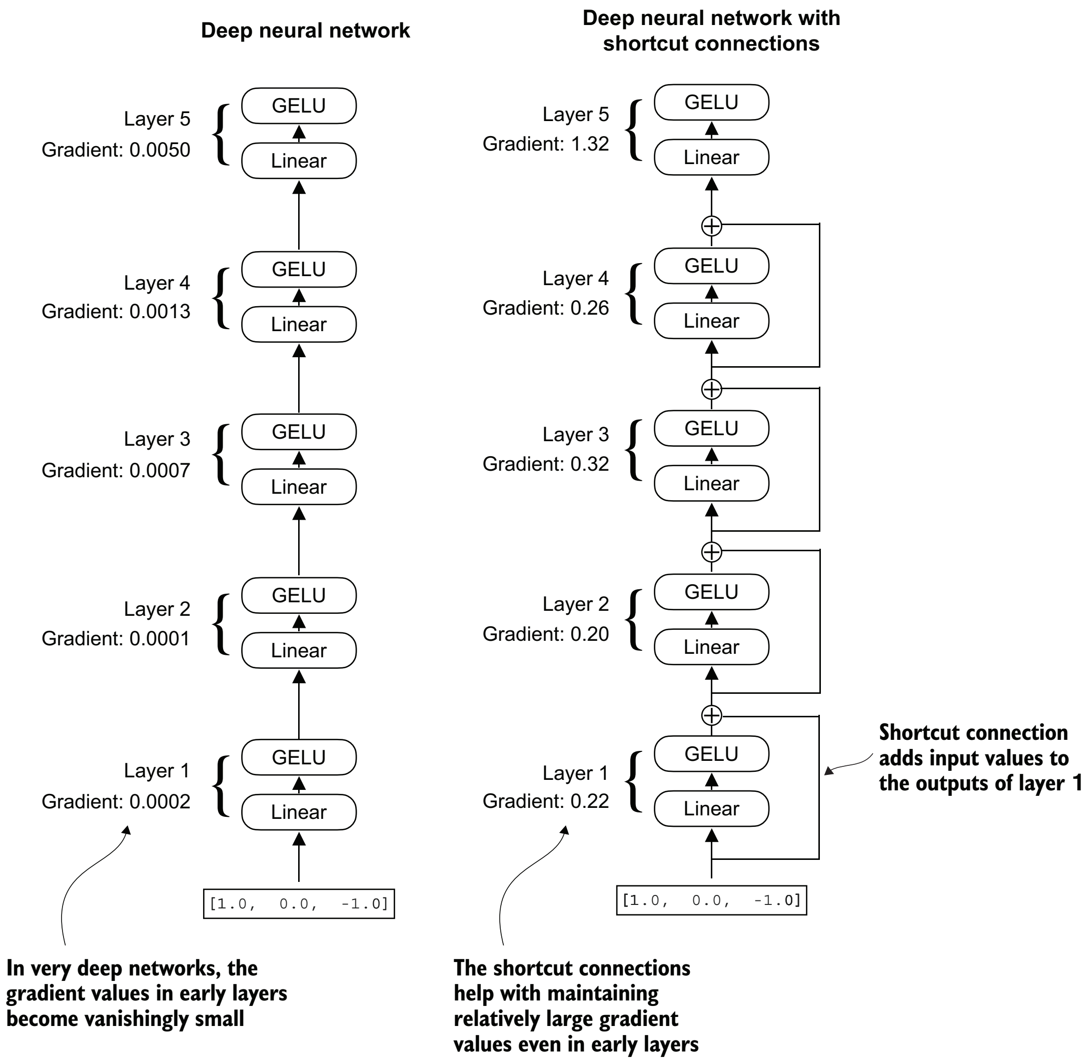

Deep neural network
Layer 5
Gradient: 0.0050
Layer 4
Gradient: 0.0013
Layer 3
Gradient: 0.0007
Layer 2
Gradient: 0.0001
Layer 1
Gradient: 0.0002
GELU
Linear
[1.0, 0.0, -1.0]
In very deep networks, the gradient values in early layers become vanishingly small
Deep neural network with shortcut connections
Layer 5
Gradient: 1.32
Layer 4
Gradient: 0.26
Layer 3
Gradient: 0.32
Layer 2
Gradient: 0.20
Layer 1
Gradient: 0.22
Shortcut connection adds input values to the outputs of layer 1
The shortcut connections help with maintaining relatively large gradient values even in early layers

Figure 4.12 A comparison between a deep neural network consisting of five layers without (left) and with shortcut connections (right). Shortcut connections involve adding the inputs of a layer to its outputs, effectively creating an alternate path that bypasses certain layers. The gradients denote the mean absolute gradient at each layer, which we compute in listing 4.5.

Listing 4.5 A neural network to illustrate shortcut connections

```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([          <- Implements five layers
            nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5]),
                          GELU())
        ])

    def forward(self, x):
        for layer in self.layers:
            layer_output = layer(x)
            if self.use_shortcut and x.shape == layer_output.shape:
                x = x + layer_output
            else:
                x = layer_output
        return x
```

Compute the output of the current layer

Check if shortcut can be applied

The code implements a deep neural network with five layers, each consisting of a `Linear` layer and a `GELU` activation function. In the forward pass, we iteratively pass the input through the layers and optionally add the shortcut connections if the `self.use_shortcut` attribute is set to `True`.

Let's use this code to initialize a neural network without shortcut connections. Each layer will be initialized such that it accepts an example with three input values and returns three output values. The last layer returns a single output value:

```
layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]])
torch.manual_seed(123)
model_without_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)
```

Specifies random seed for the initial weights for reproducibility

Next, we implement a function that computes the gradients in the model's backward pass:

```
def print_gradients(model, x):
    output = model(x)
    target = torch.tensor([[0.]])

    loss = nn.MSELoss()
    loss = loss(output, target)

    loss.backward()
```

Forward pass

Calculates loss based on how close the target and output are

Backward pass to calculate the gradients

```
    for name, param in model.named_parameters():
        if 'weight' in name:
            print(f"{name} has gradient mean of {param.grad.abs().mean().item()}")
```

This code specifies a loss function that computes how close the model output and a user-specified target (here, for simplicity, the value 0) are. Then, when calling `loss.backward()`, PyTorch computes the loss gradient for each layer in the model. We can iterate through the weight parameters via `model.named_parameters()`. Suppose we have a 3 × 3 weight parameter matrix for a given layer. In that case, this layer will have 3 × 3 gradient values, and we print the mean absolute gradient of these 3 × 3 gradient values to obtain a single gradient value per layer to compare the gradients between layers more easily.

In short, the `.backward()` method is a convenient method in PyTorch that computes loss gradients, which are required during model training, without implementing the math for the gradient calculation ourselves, thereby making working with deep neural networks much more accessible.

NOTE If you are unfamiliar with the concept of gradients and neural network training, I recommend reading sections A.4 and A.7 in appendix A.

Let's now use the `print_gradients` function and apply it to the model without skip connections:

```
print_gradients(model_without_shortcut, sample_input)
```

The output is

```
layers.0.0.weight has gradient mean of 0.00020173587836325169
layers.1.0.weight has gradient mean of 0.0001201116101583466
layers.2.0.weight has gradient mean of 0.0007152041653171182
layers.3.0.weight has gradient mean of 0.001398873864673078
layers.4.0.weight has gradient mean of 0.005049646366387606
```

The output of the `print_gradients` function shows, the gradients become smaller as we progress from the last layer (`layers.4`) to the first layer (`layers.0`), which is a phenomenon called the *vanishing gradient problem*.

Let's now instantiate a model with skip connections and see how it compares:

```
torch.manual_seed(123)
model_with_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=True
)
print_gradients(model_with_shortcut, sample_input)
```

The output is

```
layers.0.0.weight has gradient mean of 0.22169792652130127
layers.1.0.weight has gradient mean of 0.20694105327129364
layers.2.0.weight has gradient mean of 0.32896995544433594
layers.3.0.weight has gradient mean of 0.2665732502937317
layers.4.0.weight has gradient mean of 1.3258541822433472
```

The last layer `(layers.4)` still has a larger gradient than the other layers. However, the gradient value stabilizes as we progress toward the first layer (`layers.0`) and doesn't shrink to a vanishingly small value.

In conclusion, shortcut connections are important for overcoming the limitations posed by the vanishing gradient problem in deep neural networks. Shortcut connections are a core building block of very large models such as LLMs, and they will help facilitate more effective training by ensuring consistent gradient flow across layers when we train the GPT model in the next chapter.

Next, we'll connect all of the previously covered concepts (layer normalization, GELU activations, feed forward module, and shortcut connections) in a transformer block, which is the final building block we need to code the GPT architecture.

4.5 Connecting attention and linear layers in a transformer block

Now, let's implement the *transformer block*, a fundamental building block of GPT and other LLM architectures. This block, which is repeated a dozen times in the 124-million-parameter GPT-2 architecture, combines several concepts we have previously covered: multi-head attention, layer normalization, dropout, feed forward layers, and GELU activations. Later, we will connect this transformer block to the remaining parts of the GPT architecture.

Figure 4.13 shows a transformer block that combines several components, including the masked multi-head attention module (see chapter 3) and the `FeedForward` module we previously implemented (see section 4.3). When a transformer block processes an input sequence, each element in the sequence (for example, a word or subword token) is represented by a fixed-size vector (in this case, 768 dimensions). The operations within the transformer block, including multi-head attention and feed forward layers, are designed to transform these vectors in a way that preserves their dimensionality.

The idea is that the self-attention mechanism in the multi-head attention block identifies and analyzes relationships between elements in the input sequence. In contrast, the feed forward network modifies the data individually at each position. This combination not only enables a more nuanced understanding and processing of the input but also enhances the model's overall capacity for handling complex data patterns.

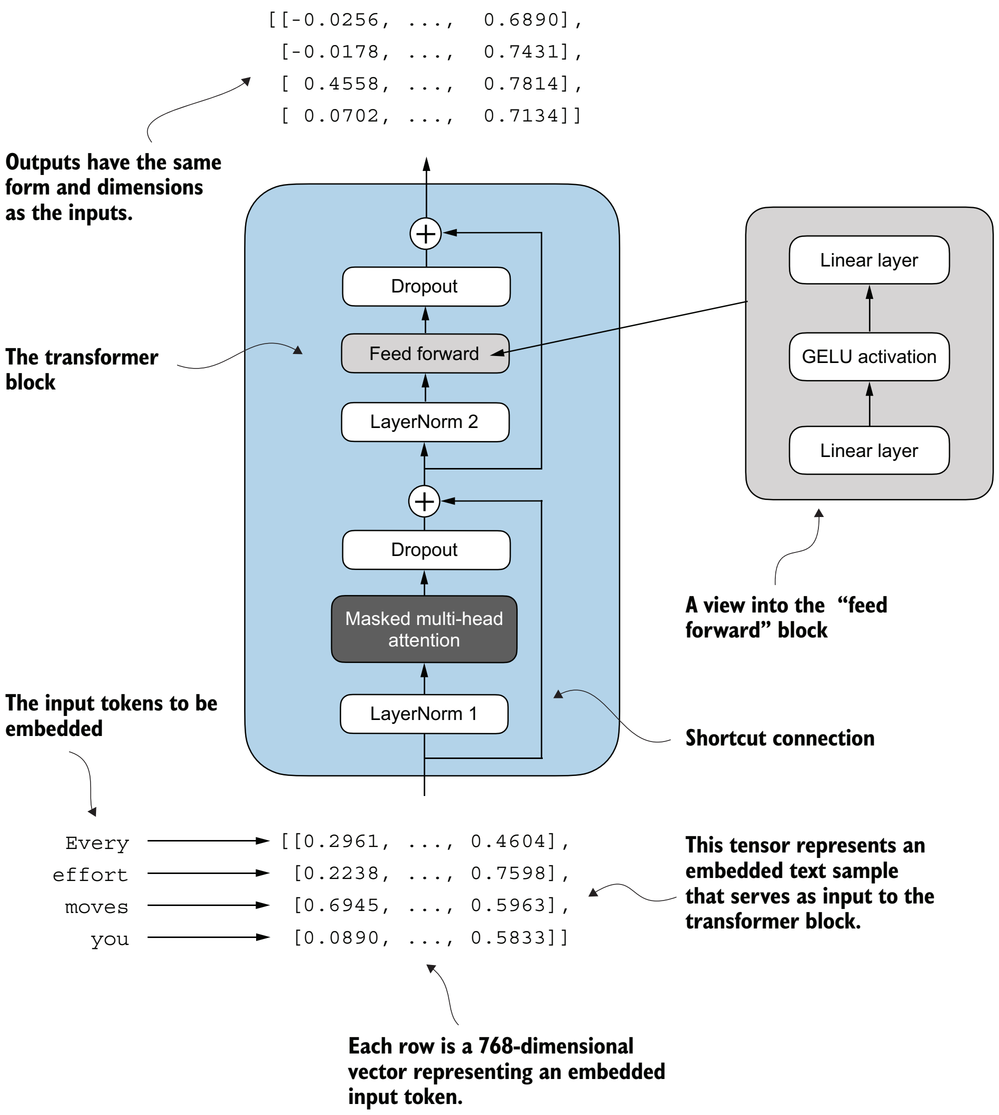


Figure 4.13 An illustration of a transformer block. Input tokens have been embedded into 768-dimensional vectors. Each row corresponds to one token's vector representation. The outputs of the transformer block are vectors of the same dimension as the input, which can then be fed into subsequent layers in an LLM.

We can create the `TransformerBlock` in code.

Listing 4.6 The transformer block component of GPT

```
from chapter03 import MultiHeadAttention

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
                                        # Shortcut connection for attention block
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut                # Add the original input back

        shortcut = x                    # Shortcut connection for feed forward block
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut                # Adds the original input back
        return x
```

The given code defines a `TransformerBlock` class in PyTorch that includes a multi-head attention mechanism (`MultiHeadAttention`) and a feed forward network (`FeedForward`), both configured based on a provided configuration dictionary (`cfg`), such as `GPT_CONFIG_124M`.

Layer normalization (`LayerNorm`) is applied before each of these two components, and dropout is applied after them to regularize the model and prevent overfitting. This is also known as *Pre-LayerNorm*. Older architectures, such as the original transformer model, applied layer normalization after the self-attention and feed forward networks instead, known as *Post-LayerNorm*, which often leads to worse training dynamics.

The class also implements the forward pass, where each component is followed by a shortcut connection that adds the input of the block to its output. This critical feature helps gradients flow through the network during training and improves the learning of deep models (see section 4.4).

Using the `GPT_CONFIG_124M` dictionary we defined earlier, let's instantiate a transformer block and feed it some sample data:

```
torch.manual_seed(123)
x = torch.rand(2, 4, 768)
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)

print("Input shape:", x.shape)
print("Output shape:", output.shape)
```

Creates sample input of shape [batch_size, num_tokens, emb_dim]

The output is

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])
```

As we can see, the transformer block maintains the input dimensions in its output, indicating that the transformer architecture processes sequences of data without altering their shape throughout the network.

The preservation of shape throughout the transformer block architecture is not incidental but a crucial aspect of its design. This design enables its effective application across a wide range of sequence-to-sequence tasks, where each output vector directly corresponds to an input vector, maintaining a one-to-one relationship. However, the output is a context vector that encapsulates information from the entire input sequence (see chapter 3). This means that while the physical dimensions of the sequence (length and feature size) remain unchanged as it passes through the transformer block, the content of each output vector is re-encoded to integrate contextual information from across the entire input sequence.

With the transformer block implemented, we now have all the building blocks needed to implement the GPT architecture. As illustrated in figure 4.14, the transformer block combines layer normalization, the feed forward network, GELU activations, and shortcut connections. As we will eventually see, this transformer block will make up the main component of the GPT architecture.

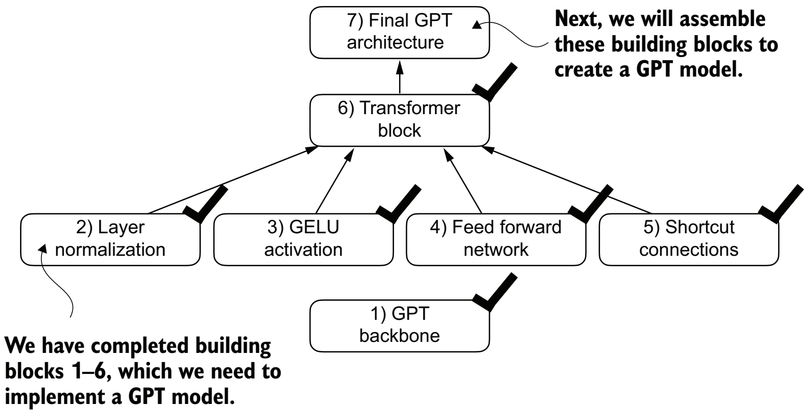

7) Final GPT architecture
Next, we will assemble these building blocks to create a GPT model.
6) Transformer block
2) Layer normalization
3) GELU activation
4) Feed forward network
5) Shortcut connections
1) GPT backbone
We have completed building blocks 1–6, which we need to implement a GPT model.

Figure 4.14 The building blocks necessary to build the GPT architecture. The black checks indicate the blocks we have completed.

4.6 Coding the GPT model

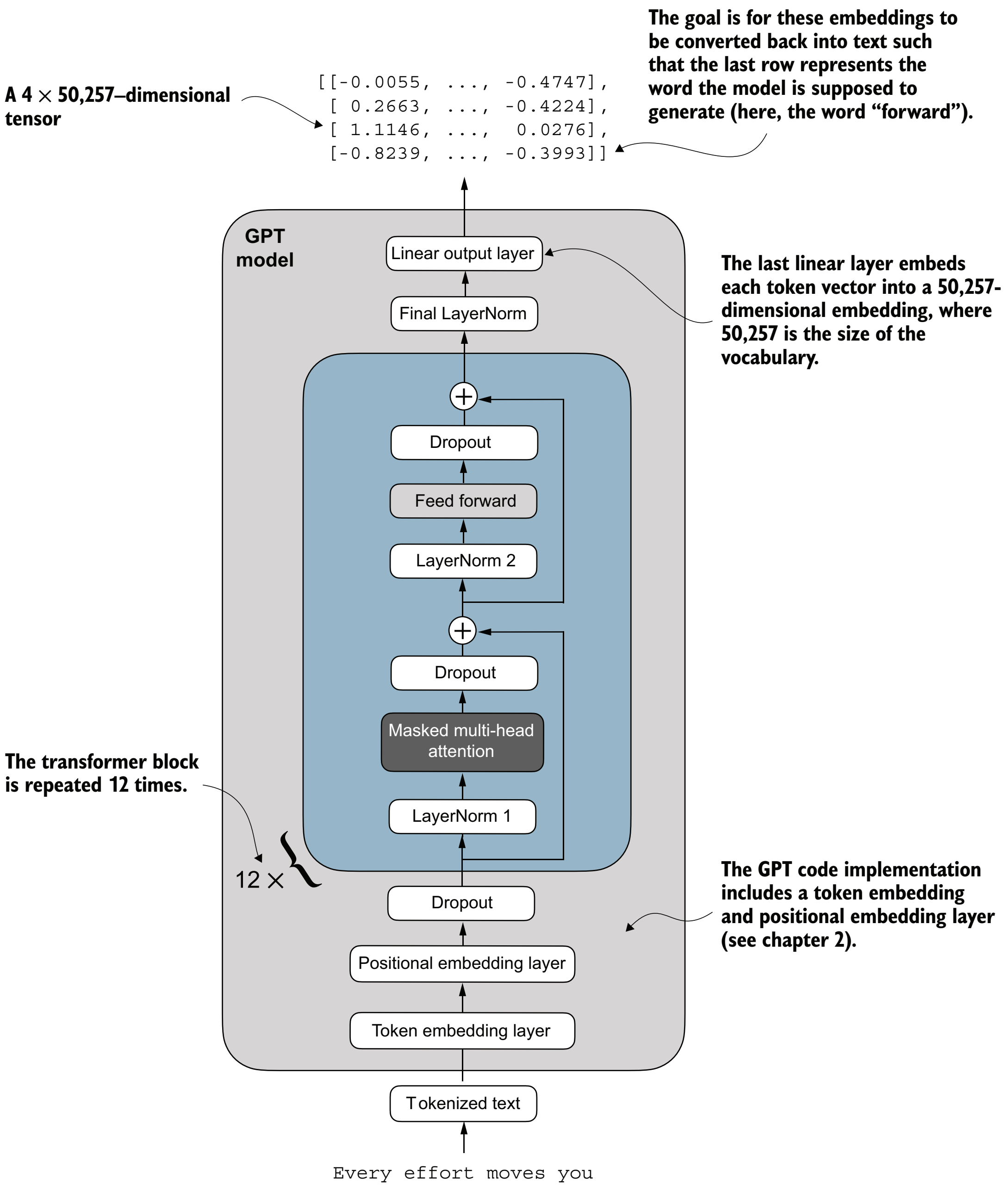
We started this chapter with a big-picture overview of a GPT architecture that we called `DummyGPTModel`. In this `DummyGPTModel` code implementation, we showed the input and outputs to the GPT model, but its building blocks remained a black box using a `DummyTransformerBlock` and `DummyLayerNorm` class as placeholders.

Let's now replace the `DummyTransformerBlock` and `DummyLayerNorm` placeholders with the real `TransformerBlock` and `LayerNorm` classes we coded previously to assemble a fully working version of the original 124-million-parameter version of GPT-2. In chapter 5, we will pretrain a GPT-2 model, and in chapter 6, we will load in the pretrained weights from OpenAI.

Before we assemble the GPT-2 model in code, let's look at its overall structure, as shown in figure 4.15, which includes all the concepts we have covered so far. As we can see, the transformer block is repeated many times throughout a GPT model architecture. In the case of the 124-million-parameter GPT-2 model, it's repeated 12 times, which we specify via the `n_layers` entry in the `GPT_CONFIG_124M` dictionary. This transform block is repeated 48 times in the largest GPT-2 model with 1,542 million parameters.

The output from the final transformer block then goes through a final layer normalization step before reaching the linear output layer. This layer maps the transformer's output to a high-dimensional space (in this case, 50,257 dimensions, corresponding to the model's vocabulary size) to predict the next token in the sequence.

Let's now code the architecture in figure 4.15.



The goal is for these embeddings to be converted back into text such that the last row represents the word the model is supposed to generate (here, the word "forward").
A 4 × 50,257–dimensional tensor
[[-0.0055, ..., -0.4747],
[0.2663, ..., -0.4224],
[1.1146, ..., 0.0276],
[-0.8239, ..., -0.3993]]
GPT model
Linear output layer
The last linear layer embeds each token vector into a 50,257-dimensional embedding, where 50,257 is the size of the vocabulary.
Final LayerNorm
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
The transformer block is repeated 12 times.
LayerNorm 1
12 ×
Dropout
The GPT code implementation includes a token embedding and positional embedding layer (see chapter 2).
Positional embedding layer
Token embedding layer
Tokenized text
Every effort moves you

Figure 4.15 An overview of the GPT model architecture showing the flow of data through the GPT model. Starting from the bottom, tokenized text is first converted into token embeddings, which are then augmented with positional embeddings. This combined information forms a tensor that is passed through a series of transformer blocks shown in the center (each containing multi-head attention and feed forward neural network layers with dropout and layer normalization), which are stacked on top of each other and repeated 12 times.

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],        Token IDs of text 1
         [6109,  1110,  6622,   257]])       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.

Lastly, let's compute the memory requirements of the 163 million parameters in our `GPTModel` object:

```
total_size_bytes = total_params * 4
total_size_mb = total_size_bytes / (1024 * 1024)
print(f"Total size of the model: {total_size_mb:.2f} MB")
```

Calculates the total size in bytes (assuming float32, 4 bytes per parameter)

Converts to megabytes

The result is

```
Total size of the model: 621.83 MB
```

In conclusion, by calculating the memory requirements for the 163 million parameters in our `GPTModel` object and assuming each parameter is a 32-bit float taking up 4 bytes, we find that the total size of the model amounts to 621.83 MB, illustrating the relatively large storage capacity required to accommodate even relatively small LLMs.

Now that we've implemented the `GPTModel` architecture and saw that it outputs numeric tensors of shape `[batch_size, num_tokens, vocab_size]`, let's write the code to convert these output tensors into text.

Exercise 4.2 Initializing larger GPT models

We initialized a 124-million-parameter GPT model, which is known as "GPT-2 small." Without making any code modifications besides updating the configuration file, use the `GPTModel` class to implement GPT-2 medium (using 1,024-dimensional embeddings, 24 transformer blocks, 16 multi-head attention heads), GPT-2 large (1,280-dimensional embeddings, 36 transformer blocks, 20 multi-head attention heads), and GPT-2 XL (1,600-dimensional embeddings, 48 transformer blocks, 25 multi-head attention heads). As a bonus, calculate the total number of parameters in each GPT model.

4.7 Generating text

We will now implement the code that converts the tensor outputs of the GPT model back into text. Before we get started, let's briefly review how a generative model like an LLM generates text one word (or token) at a time.

Figure 4.16 illustrates the step-by-step process by which a GPT model generates text given an input context, such as "Hello, I am." With each iteration, the input context grows, allowing the model to generate coherent and contextually appropriate text. By the sixth iteration, the model has constructed a complete sentence: "Hello, I am a model ready to help." We've seen that our current `GPTModel` implementation outputs tensors with shape `[batch_size, num_token, vocab_size]`. Now the question is: How does a GPT model go from these output tensors to the generated text?

The process by which a GPT model goes from output tensors to generated text involves several steps, as illustrated in figure 4.17. These steps include decoding the

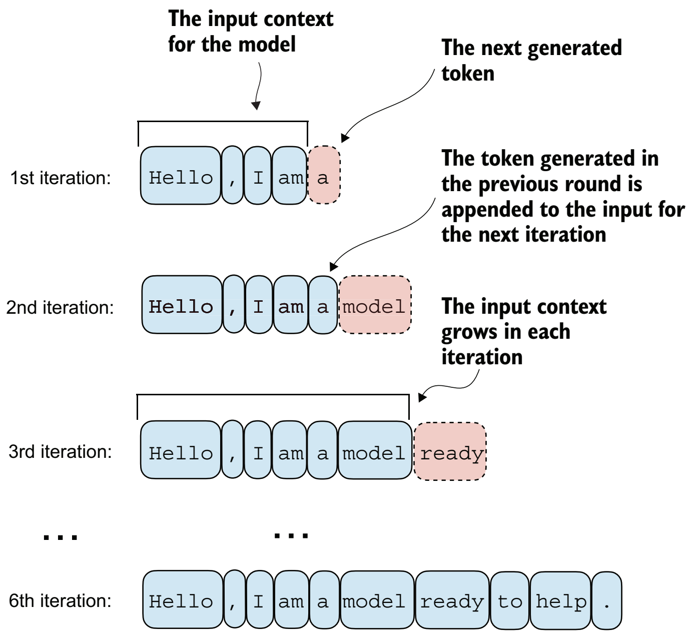

The input context for the model
The next generated token
1st iteration:
Hello , I am a
The token generated in the previous round is appended to the input for the next iteration
2nd iteration:
Hello , I am a model
The input context grows in each iteration
3rd iteration:
Hello , I am a model ready
...
6th iteration:
Hello , I am a model ready to help .

Figure 4.16 The step-by-step process by which an LLM generates text, one token at a time. Starting with an initial input context ("Hello, I am"), the model predicts a subsequent token during each iteration, appending it to the input context for the next round of prediction. As shown, the first iteration adds "a," the second "model," and the third "ready," progressively building the sentence.

output tensors, selecting tokens based on a probability distribution, and converting these tokens into human-readable text.

The next-token generation process detailed in figure 4.17 illustrates a single step where the GPT model generates the next token given its input. In each step, the model outputs a matrix with vectors representing potential next tokens. The vector corresponding to the next token is extracted and converted into a probability distribution via the `softmax` function. Within the vector containing the resulting probability scores, the index of the highest value is located, which translates to the token ID. This token ID is then decoded back into text, producing the next token in the sequence. Finally, this token is appended to the previous inputs, forming a new input sequence for the subsequent iteration. This step-by-step process enables the model to generate text sequentially, building coherent phrases and sentences from the initial input context.

In practice, we repeat this process over many iterations, such as shown in figure 4.16, until we reach a user-specified number of generated tokens. In code, we can implement the token-generation process as shown in the following listing.

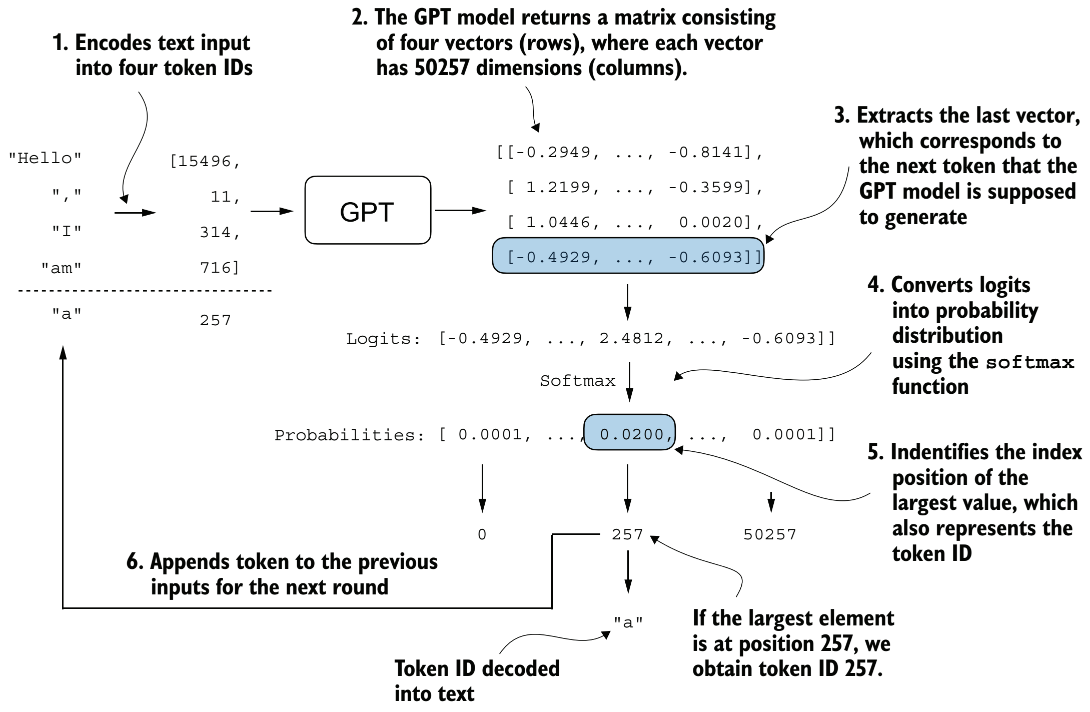

1. Encodes text input into four token IDs
2. The GPT model returns a matrix consisting of four vectors (rows), where each vector has 50257 dimensions (columns).
3. Extracts the last vector, which corresponds to the next token that the GPT model is supposed to generate
"Hello"
","
"I"
"am"
"a"
[15496,
11,
314,
716]
257
GPT
[[-0.2949, ..., -0.8141],
[1.2199, ..., -0.3599],
[1.0446, ..., 0.0020],
[-0.4929, ..., -0.6093]]
4. Converts logits into probability distribution using the softmax function
Logits: [-0.4929, ..., 2.4812, ..., -0.6093]]
Softmax
Probabilities: [0.0001, ..., 0.0200, ..., 0.0001]]
5. Indentifies the index position of the largest value, which also represents the token ID
0
257
50257
6. Appends token to the previous inputs for the next round
"a"
Token ID decoded into text
If the largest element is at position 257, we obtain token ID 257.

Figure 4.17 The mechanics of text generation in a GPT model by showing a single iteration in the token generation process. The process begins by encoding the input text into token IDs, which are then fed into the GPT model. The outputs of the model are then converted back into text and appended to the original input text.

Listing 4.8 A function for the GPT model to generate text

```
def generate_text_simple(model, idx,
                         max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :]
        probas = torch.softmax(logits, dim=-1)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)

    return idx
```

Crops current context if it exceeds the supported context size, e.g., if LLM supports only 5 tokens, and the context size is 10, then only the last 5 tokens are used as context

idx is a (batch, n_tokens) array of indices in the current context.

Focuses only on the last time step, so that (batch, n_token, vocab_size) becomes (batch, vocab_size)

probas has shape (batch, vocab_size).

idx_next has shape (batch, 1).

Appends sampled index to the running sequence, where idx has shape (batch, n_tokens+1)

This code demonstrates a simple implementation of a generative loop for a language model using PyTorch. It iterates for a specified number of new tokens to be generated, crops the current context to fit the model's maximum context size, computes predictions, and then selects the next token based on the highest probability prediction.

To code the `generate_text_simple` function, we use a `softmax` function to convert the logits into a probability distribution from which we identify the position with the highest value via `torch.argmax`. The `softmax` function is monotonic, meaning it preserves the order of its inputs when transformed into outputs. So, in practice, the softmax step is redundant since the position with the highest score in the softmax output tensor is the same position in the logit tensor. In other words, we could apply the `torch.argmax` function to the logits tensor directly and get identical results. However, I provide the code for the conversion to illustrate the full process of transforming logits to probabilities, which can add additional intuition so that the model generates the most likely next token, which is known as *greedy decoding*.

When we implement the GPT training code in the next chapter, we will use additional sampling techniques to modify the softmax outputs such that the model doesn't always select the most likely token. This introduces variability and creativity in the generated text.

This process of generating one token ID at a time and appending it to the context using the `generate_text_simple` function is further illustrated in figure 4.18. (The token ID generation process for each iteration is detailed in figure 4.17.) We generate the token IDs in an iterative fashion. For instance, in iteration 1, the model is provided with the tokens corresponding to "Hello, I am," predicts the next token (with ID 257, which is "a"), and appends it to the input. This process is repeated until the model produces the complete sentence "Hello, I am a model ready to help" after six iterations.

Let's now try out the `generate_text_simple` function with the `"Hello, I am"` context as model input. First, we encode the input context into token IDs:

```
start_context = "Hello, I am"
encoded = tokenizer.encode(start_context)
print("encoded:", encoded)
encoded_tensor = torch.tensor(encoded).unsqueeze(0)      # Adds batch dimension
print("encoded_tensor.shape:", encoded_tensor.shape)
```

The encoded IDs are

```
encoded: [15496, 11, 314, 716]
encoded_tensor.shape: torch.Size([1, 4])
```

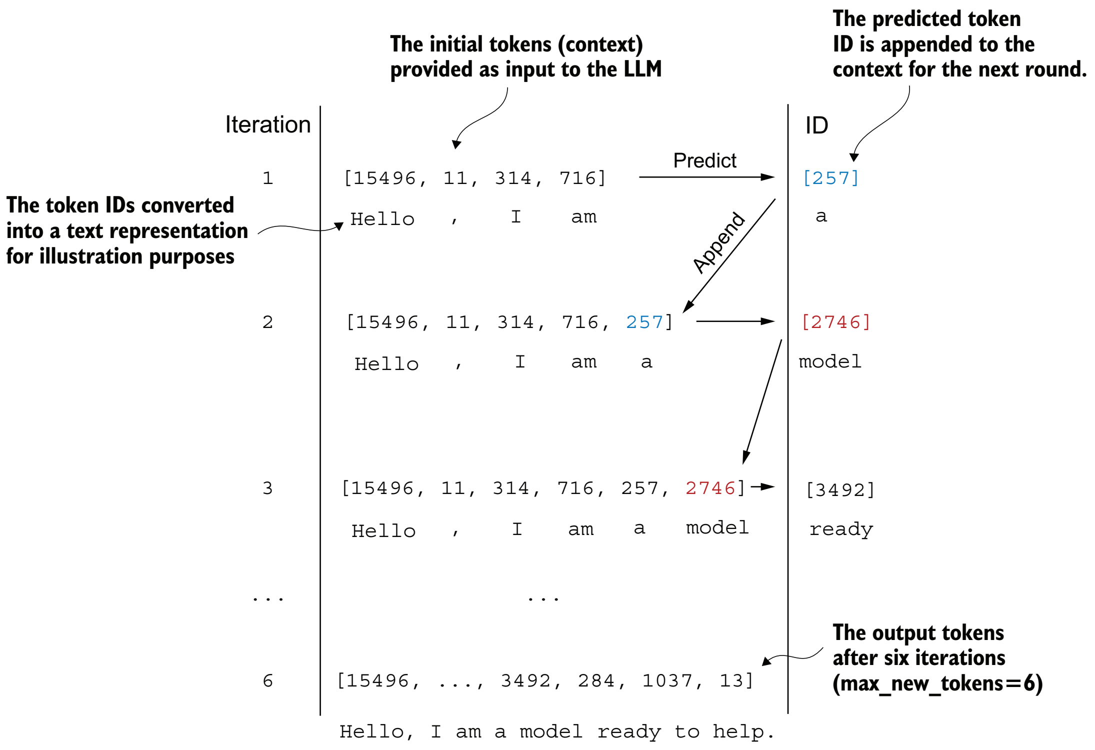

The initial tokens (context) provided as input to the LLM
The predicted token ID is appended to the context for the next round.
Iteration
ID
1
[15496, 11, 314, 716]
Predict
[257]
The token IDs converted into a text representation for illustration purposes
Hello , I am
a
Append
2
[15496, 11, 314, 716, 257]
[2746]
Hello , I am a
model
3
[15496, 11, 314, 716, 257, 2746]
[3492]
Hello , I am a model
ready
...
...
The output tokens after six iterations (max_new_tokens=6)
6
[15496, ..., 3492, 284, 1037, 13]
Hello, I am a model ready to help.

Figure 4.18 The six iterations of a token prediction cycle, where the model takes a sequence of initial token IDs as input, predicts the next token, and appends this token to the input sequence for the next iteration. (The token IDs are also translated into their corresponding text for better understanding.)

Next, we put the model into `.eval()` mode. This disables random components like dropout, which are only used during training, and use the `generate_text_simple` function on the encoded input tensor:

```
model.eval()                                   Disables dropout since
out = generate_text_simple(                    we are not training
    model=model,                               the model
    idx=encoded_tensor,
    max_new_tokens=6,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output:", out)
print("Output length:", len(out[0]))
```

The resulting output token IDs are

```
Output: tensor([[15496,    11,   314,   716, 27018, 24086, 47843,
30961, 42348,  7267]])
Output length: 10
```

Using the `.decode` method of the tokenizer, we can convert the IDs back into text:

```
decoded_text = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_text)
```

The model output in text format is

```
Hello, I am Featureiman Byeswickattribute argue
```

As we can see, the model generated gibberish, which is not at all like the coherent text `Hello, I am a model ready to help`. What happened? The reason the model is unable to produce coherent text is that we haven't trained it yet. So far, we have only implemented the GPT architecture and initialized a GPT model instance with initial random weights. Model training is a large topic in itself, and we will tackle it in the next chapter.

Exercise 4.3 Using separate dropout parameters

At the beginning of this chapter, we defined a global `drop_rate` setting in the `GPT_CONFIG_124M` dictionary to set the dropout rate in various places throughout the `GPTModel` architecture. Change the code to specify a separate dropout value for the various dropout layers throughout the model architecture. (Hint: there are three distinct places where we used dropout layers: the embedding layer, shortcut layer, and multi-head attention module.)

Summary

- Layer normalization stabilizes training by ensuring that each layer's outputs have a consistent mean and variance.
- Shortcut connections are connections that skip one or more layers by feeding the output of one layer directly to a deeper layer, which helps mitigate the vanishing gradient problem when training deep neural networks, such as LLMs.
- Transformer blocks are a core structural component of GPT models, combining masked multi-head attention modules with fully connected feed forward networks that use the GELU activation function.
- GPT models are LLMs with many repeated transformer blocks that have millions to billions of parameters.
- GPT models come in various sizes, for example, 124, 345, 762, and 1,542 million parameters, which we can implement with the same `GPTModel` Python class.
- The text-generation capability of a GPT-like LLM involves decoding output tensors into human-readable text by sequentially predicting one token at a time based on a given input context.
- Without training, a GPT model generates incoherent text, which underscores the importance of model training for coherent text generation.