Implementing a GPT model from Scratch To Generate Text

Hamza Fatnaoui

Hicham Ibn Issaghyr

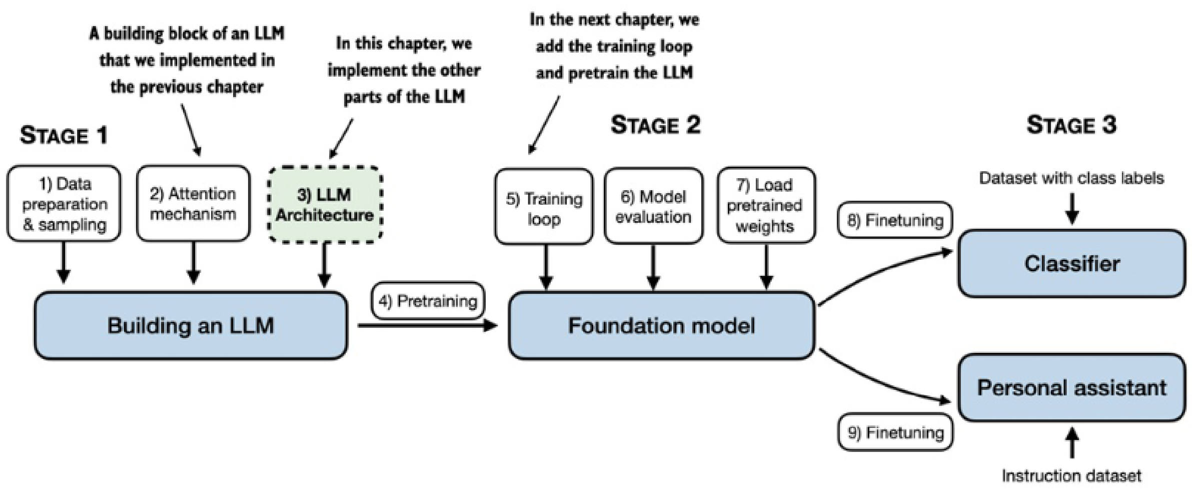
Ikram Arif

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 to train models more effectively.
- Computing the number of parameters and storage requirements of GPT models.

In the previous chapter, you learned and coded the multi-head attention mechanism, one of the core components of LLMs. In this chapter, we will now 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.

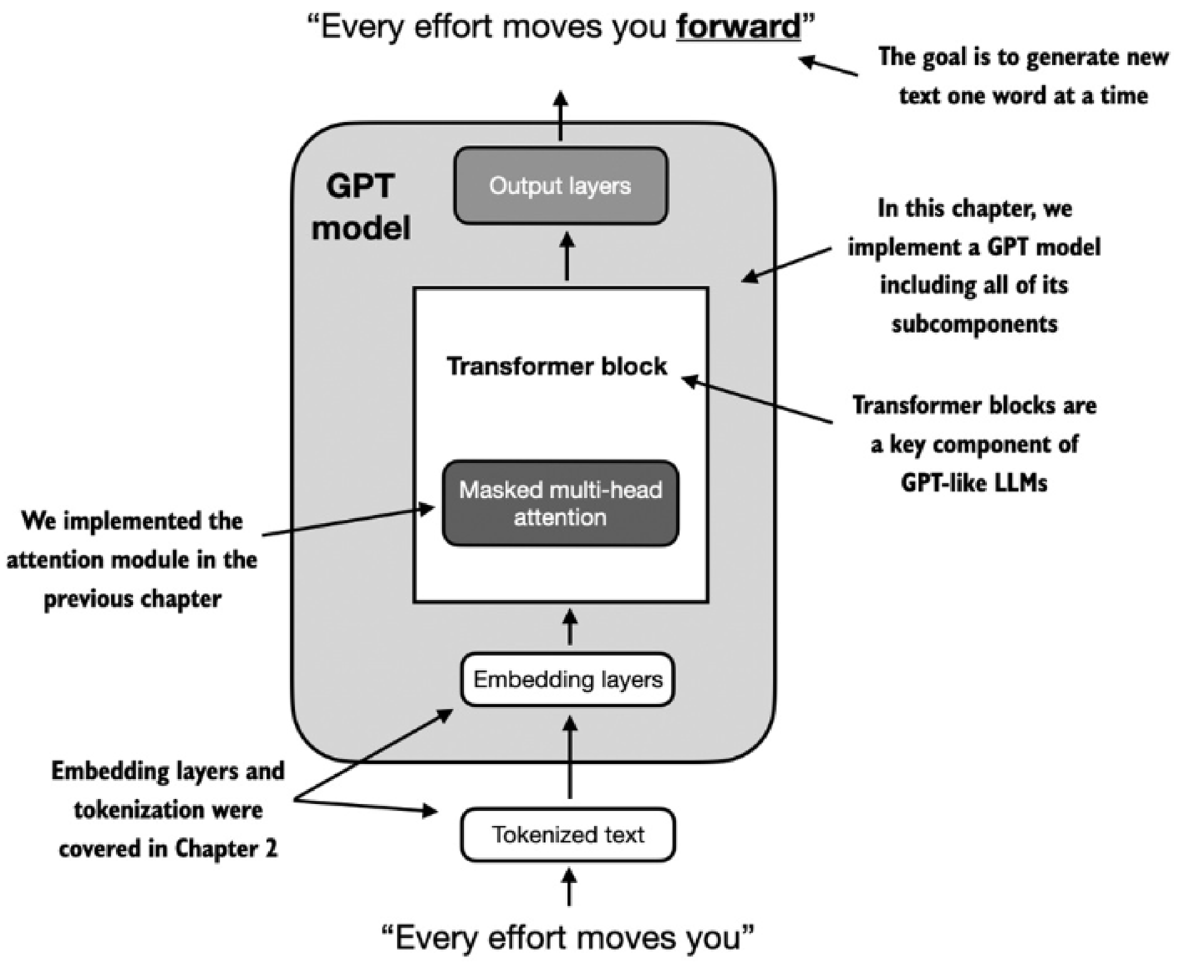
A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset, and finetuning it on a labeled dataset. This chapter focuses on implementing the LLM architecture, which we will train in the next chapter.


A building block of an LLM that we implemented in the previous chapter
In this chapter, we 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
Building an LLM
4) Pretraining
5) Training loop
6) Model evaluation
7) Load pretrained weights
Foundation model
8) Finetuning
9) Finetuning
Dataset with class labels
Classifier
Personal assistant
Instruction dataset

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.

A mental model of a GPT model. Next to the embedding layers, it consists of one or more transformer blocks containing the masked multi-head attention module we implemented in the previous chapter.


"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
Masked multi-head attention
We implemented the attention module in the previous chapter
Embedding layers
Embedding layers and tokenization were covered in Chapter 2
Tokenized text
"Every effort moves you"

GPT-2 versus 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 later. 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, it would take 355 years to train GPT-3 on a single V100 datacenter GPU, and 665 years on a consumer RTX 8000 GPU.

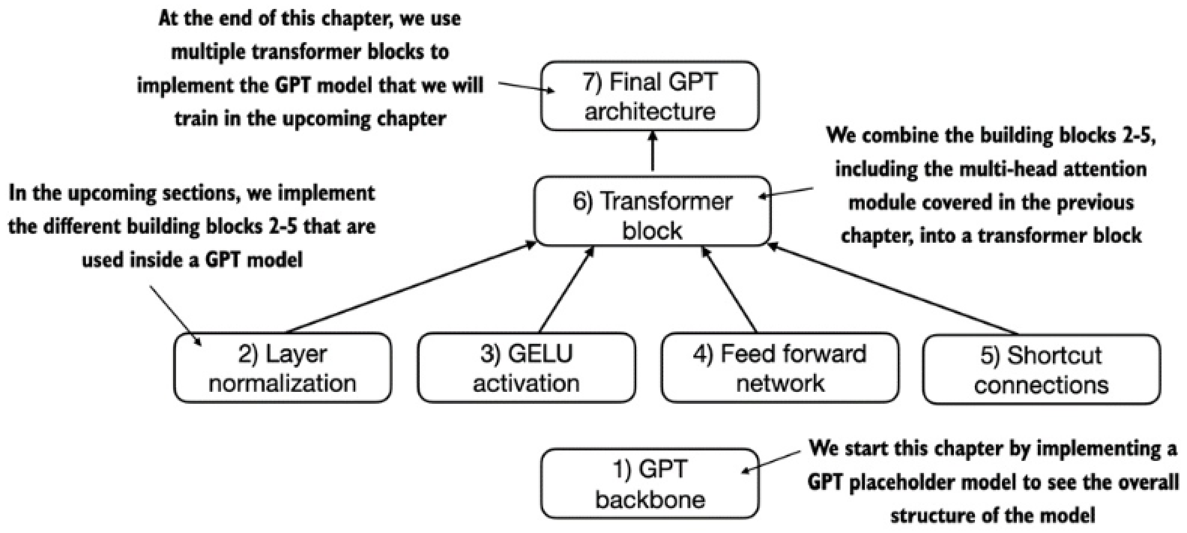
We specify the configuration of the small GPT-2 model via the following Python dictionary, which we will use in the code examples later:

```
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
- **context_length** denotes the maximum number of input tokens the model can handle, via the positional embeddings
- **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, as implemented in chapter 3.
- **n_layers** specifies the number of transformer blocks in the model, which will be elaborated on in upcoming sections.
- **drop_rate** indicates the intensity of the dropout mechanism (0.1 implies a 10% drop of hidden units) to prevent overfitting
- **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 will revisit it in chapter 6 when we load pretrained GPT-2 weights from OpenAI into our model.

A mental model outlining the order in which we code the GPT architecture. In this chapter, we will start with the GPT backbone, a placeholder architecture, before we get to the individual core pieces and eventually assemble them in a transformer block for the final GPT architecture.


At the end of this chapter, we use multiple transformer blocks to implement the GPT model that we will train in the upcoming chapter
7) Final GPT architecture
We combine the building blocks 2-5, including the multi-head attention module covered in the previous chapter, into a transformer block
In the upcoming sections, we implement the different building blocks 2-5 that are used inside a GPT model
6) Transformer block
2) Layer normalization
3) GELU activation
4) Feed forward network
5) Shortcut connections
1) GPT backbone
We start this chapter by implementing a GPT placeholder model to see the overall structure of the model

A placeholder GPT model architecture class

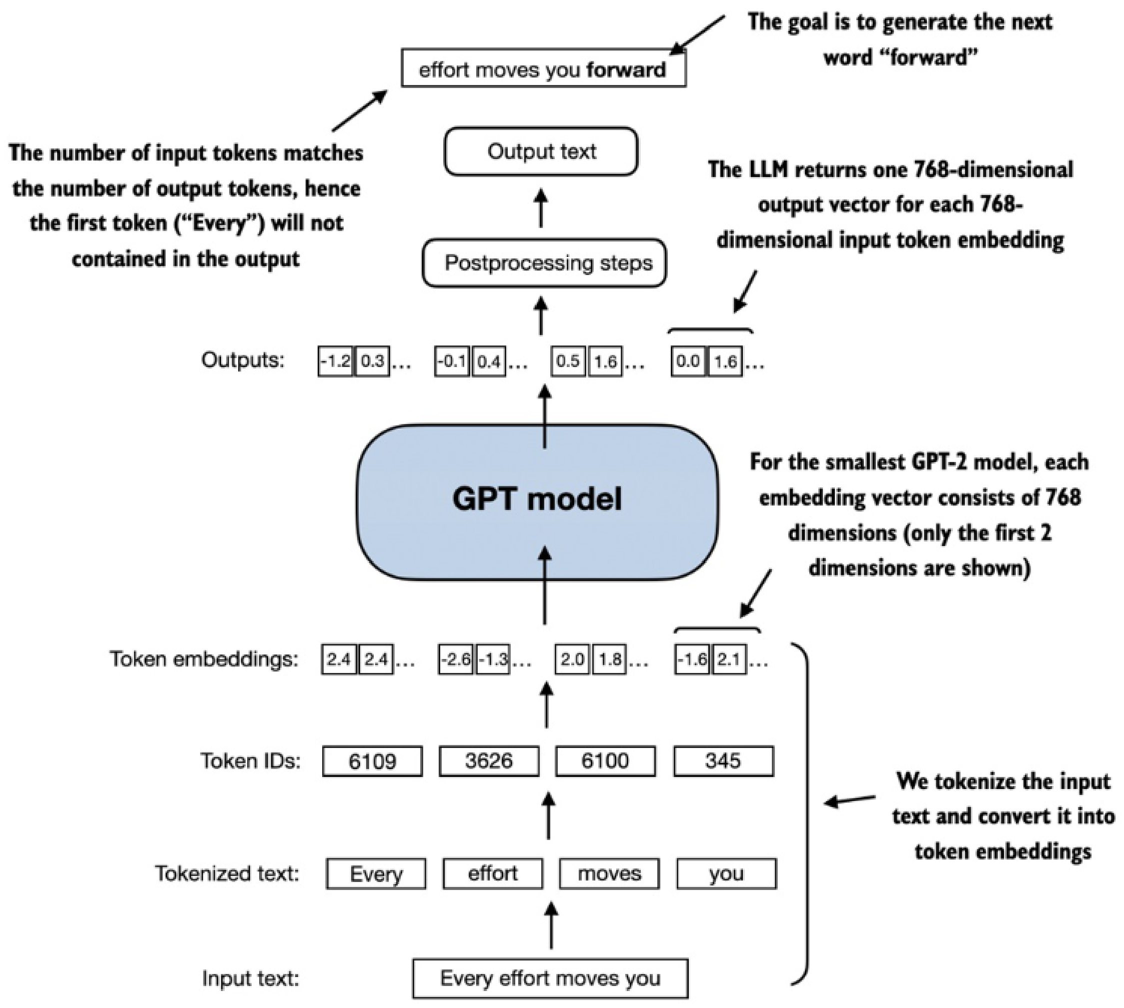
```
class DummyGPTModel(nn.Module):
  def __init__(self, cfg):
    super().__init__()
    self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim'
    self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_
    self.drop_emb = nn.Dropout(cfg["drop_rate"])
    self.trf_blocks = nn.Sequential(
      *[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_i
    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

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
```

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 above is already functional, as we will see later in this section after we prepare the input data. However, for now, note in the code above that we have used placeholders (DummyLayerNorm and DummyTransformerBlock) for the transformer block and layer normalization, which we will develop in later sections.

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


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 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
We tokenize the input text and convert it into token embeddings
Tokenized text:
Every
effort
moves
you
Input text:
Every effort moves you

To implement the steps shown in the figure, we tokenize a batch consisting of two text inputs for the GPT model using the tiktoken tokenizer

```
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)
```

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

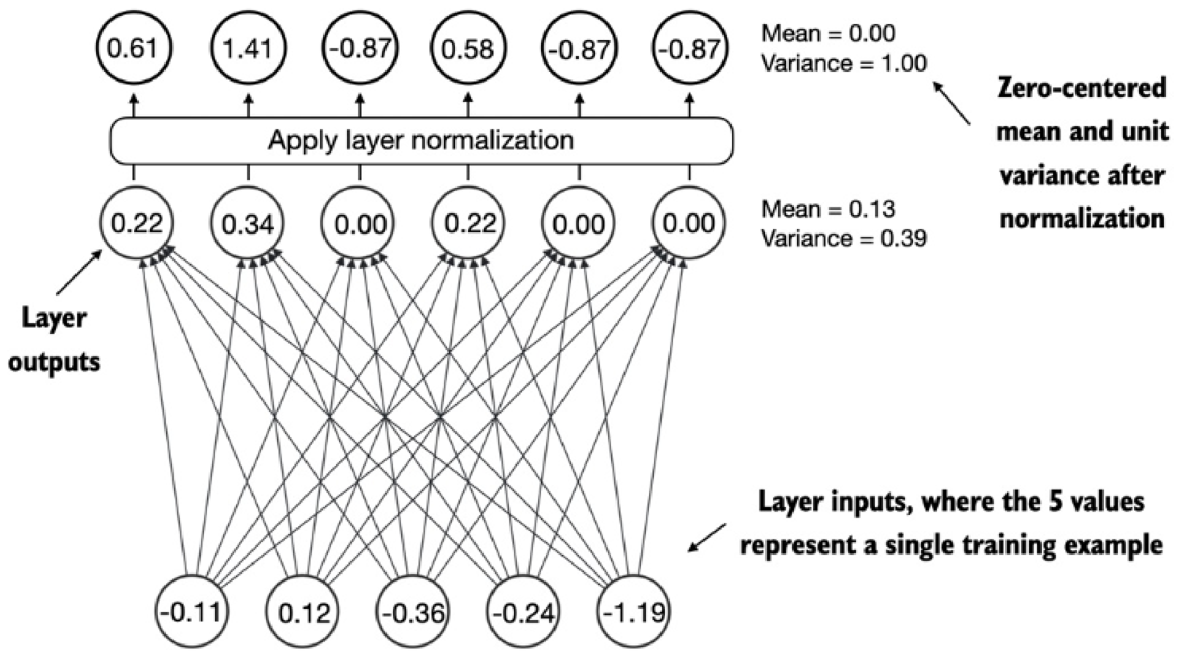
Normalizing activations with layer normalization

Training deep neural networks with many layers can sometimes prove challenging due to issues like vanishing or exploding gradients. These issues 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.

In this section, we will 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. As we have seen in the previous section, based on the DummyLayerNorm placeholder, in GPT-2 and modern transformer architectures, layer normalization is typically applied before and after the multi-head attention module and before the final output layer.

An illustration of layer normalization where the 5 layer outputs, also called activations, are normalized such that they have a zero mean and variance of 1.


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 5 values represent a single training example
-0.11
0.12
-0.36
-0.24
-1.19

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 have opted for 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 large-scale language models (LLMs), where the embedding dimension n is significantly large, the difference between using n and n-1 is practically negligible. We 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

Layer normalization versus 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.

Implementing a feed forward network with GELU activations

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 (Sigmoid-Weighted Linear Unit).

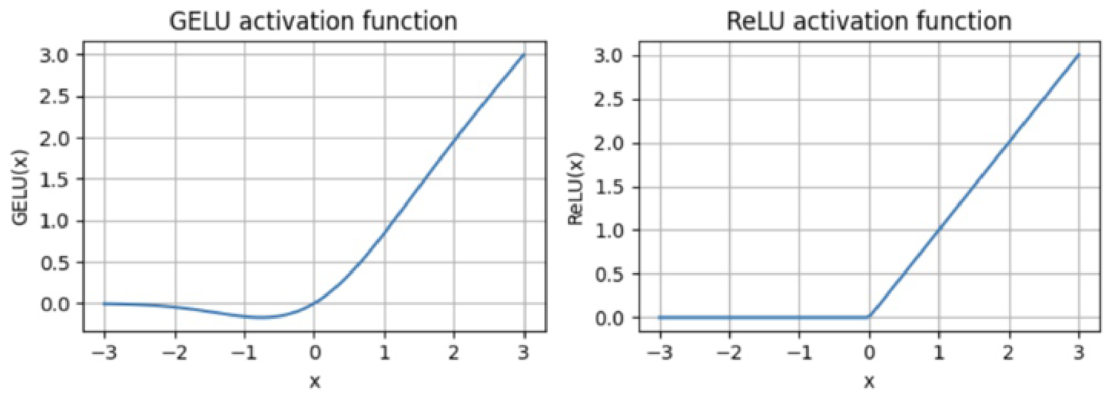
GELU and SwiGLU are more complex and smooth activation functions incorporating Gaussian and sigmoid-gated linear units, respectively. They offer improved performance for deep learning models, unlike the simpler ReLU

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):

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

In code, we can implement this function as PyTorch module as follows

```
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))
        ))
```



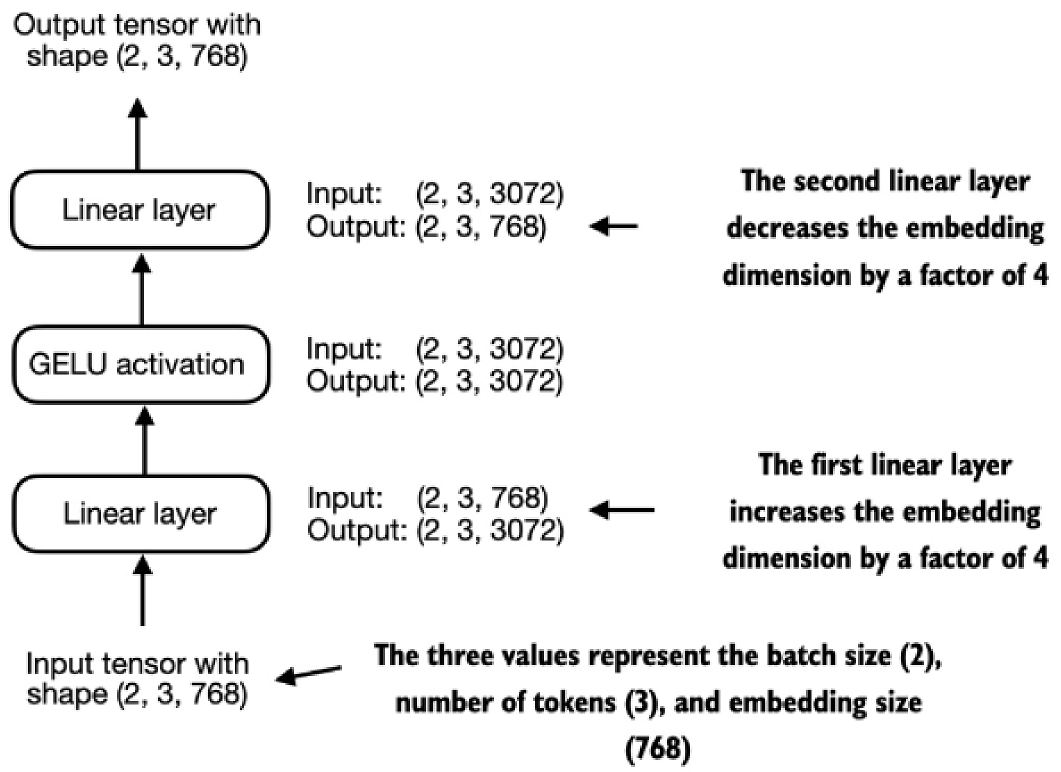
The output of the GELU and ReLU plots using matplotlib. The x-axis shows the function inputs and the y-axis shows the function outputs

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, 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.

```
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 in the preceding code, 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.

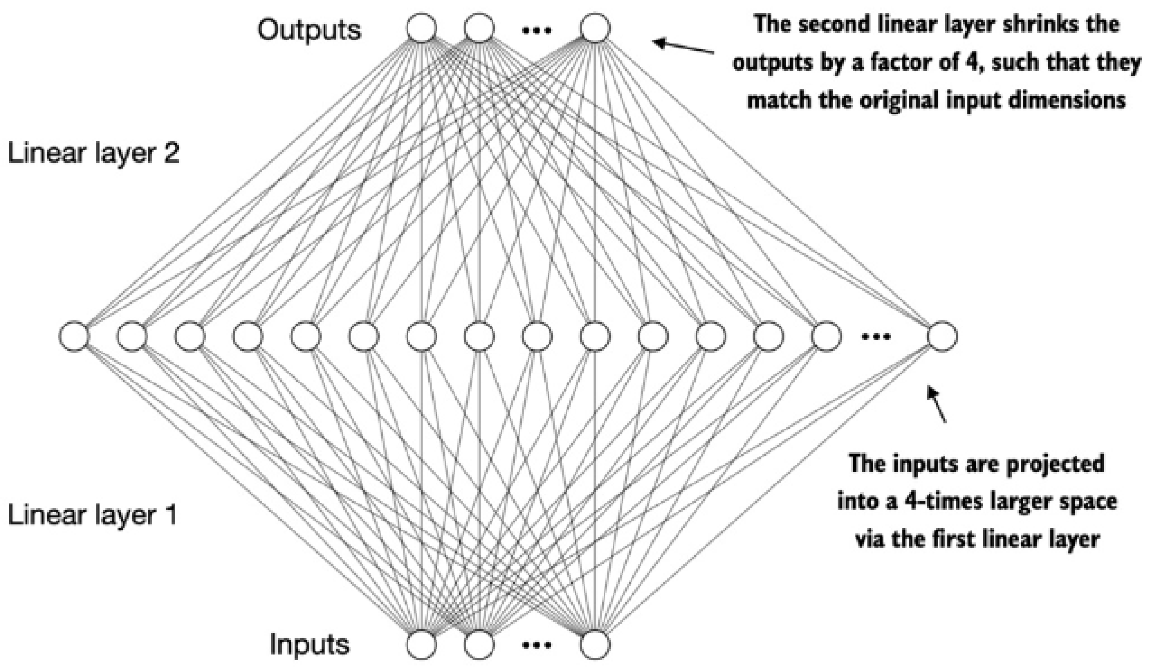
provides a visual overview of the connections between the layers of the feed forward neural network. It is important to note that this neural network can accommodate variable batch sizes and numbers of tokens in the input. However, the embedding size for each token is determined and fixed when initializing the weights.


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)

```
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768) #A
out = ffn(x)
print(out.shape)
```

The FeedForward module we implemented in this section 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. This expansion is followed by a non-linear 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.

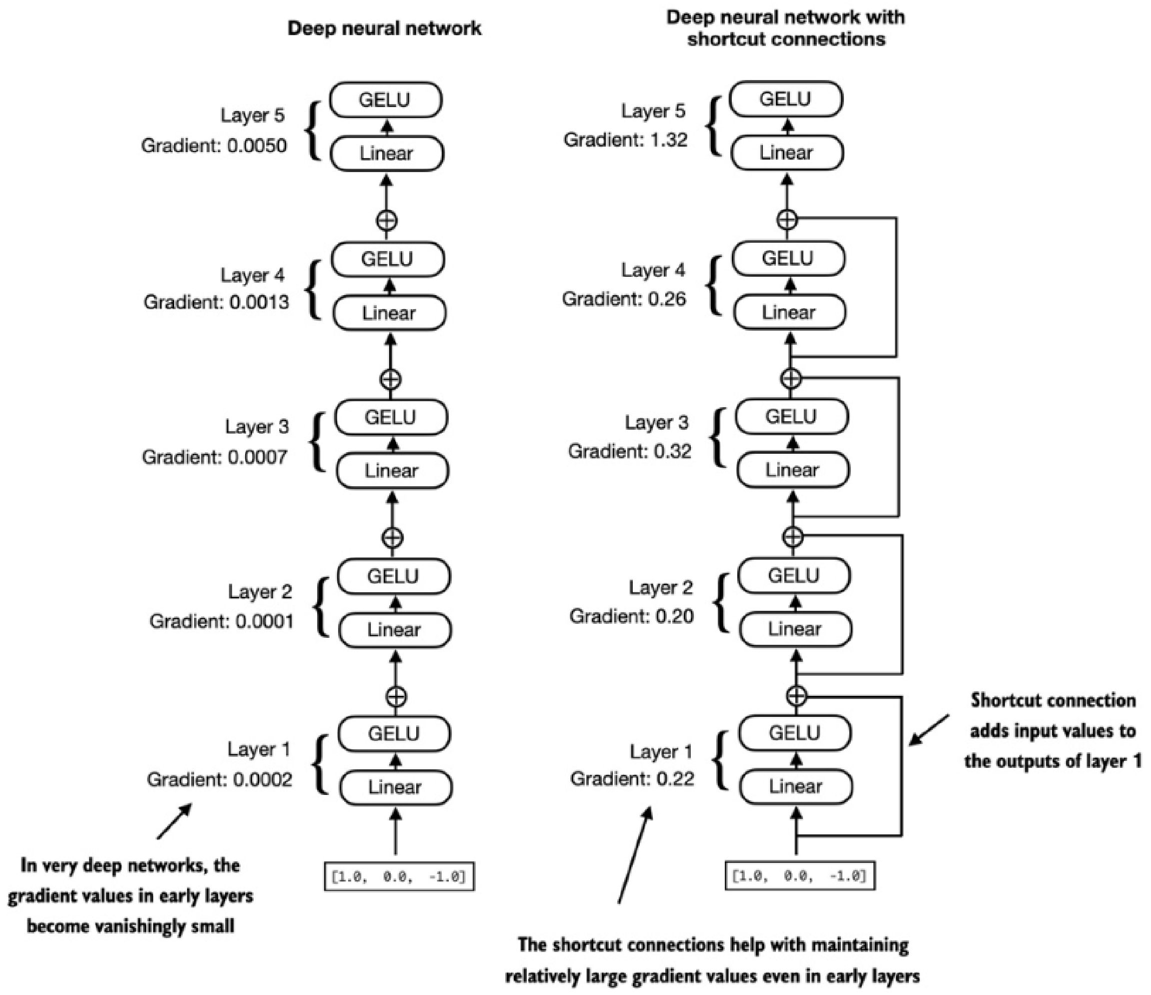
An illustration of the expansion and contraction of the layer outputs in the feed forward neural network. First, the inputs expand by a factor of 4 from 768 to 3072 values. Then, the second layer compresses the 3072 values back into a 768-dimensional representation


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

Adding shortcut connections

Next, 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

A comparison between a deep neural network consisting of 5 layers without (on the left) and with shortcut connections (on the right). Shortcut connections involve adding the inputs of a layer to its outputs, effectively creating an alternate path that bypasses certain layers. The gradient illustrated in Figure 1.1 denotes the mean absolute gradient at each layer, which we will compute in the code example that follows.

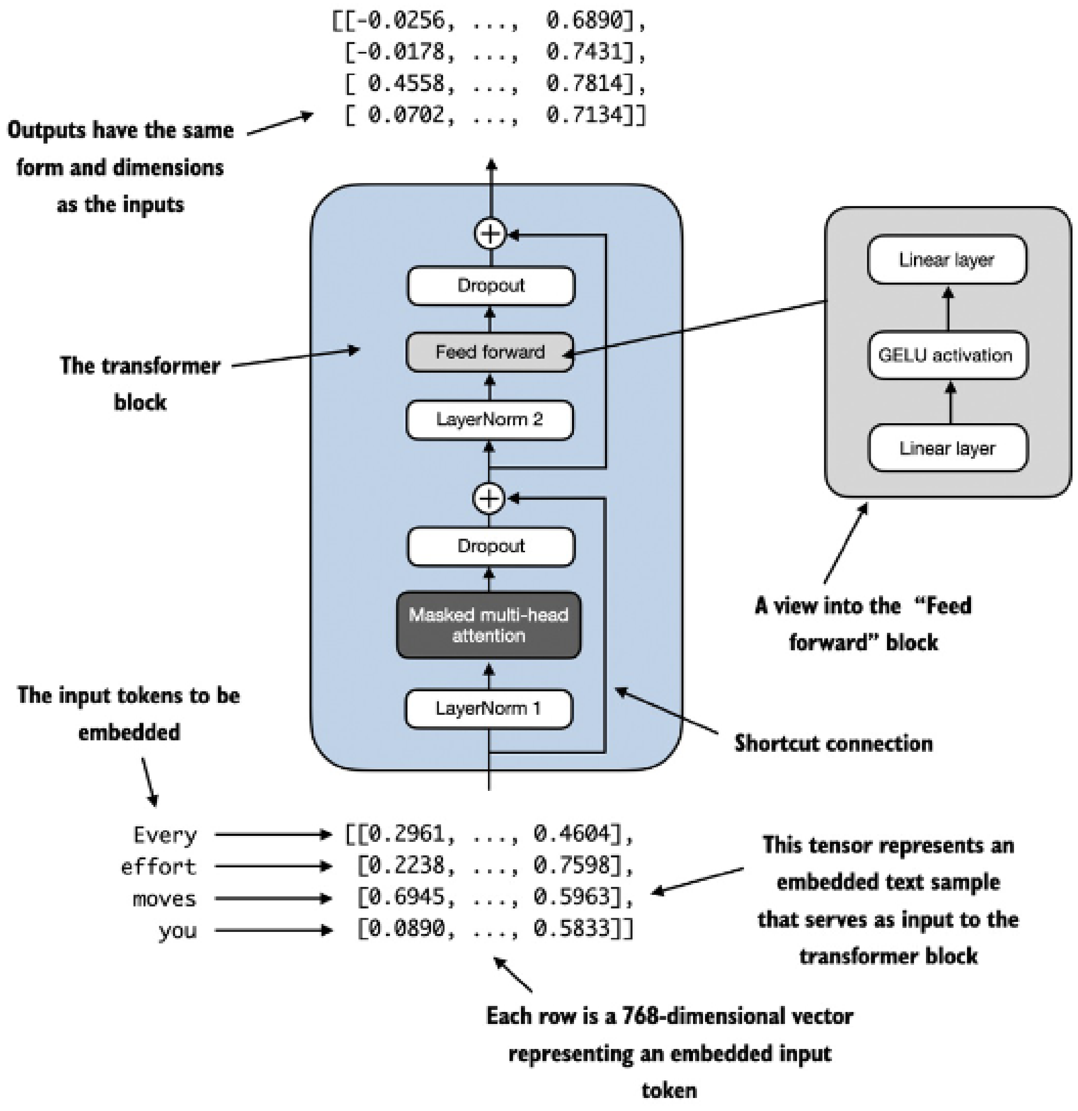

Deep neural network
Deep neural network with shortcut connections
Layer 5 Gradient: 0.0050
Layer 4 Gradient: 0.0013
Layer 3 Gradient: 0.0007
Layer 2 Gradient: 0.0001
Layer 1 Gradient: 0.0002
Layer 5 Gradient: 1.32
Layer 4 Gradient: 0.26
Layer 3 Gradient: 0.32
Layer 2 Gradient: 0.20
Layer 1 Gradient: 0.22
GELU
Linear
[1.0, 0.0, -1.0]
In very deep networks, the gradient values in early layers become vanishingly small
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

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.

Connecting attention and linear layers in a transformer block

In this section, we are implementing 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

An illustration of a transformer block. The bottom of the diagram shows input tokens that 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.



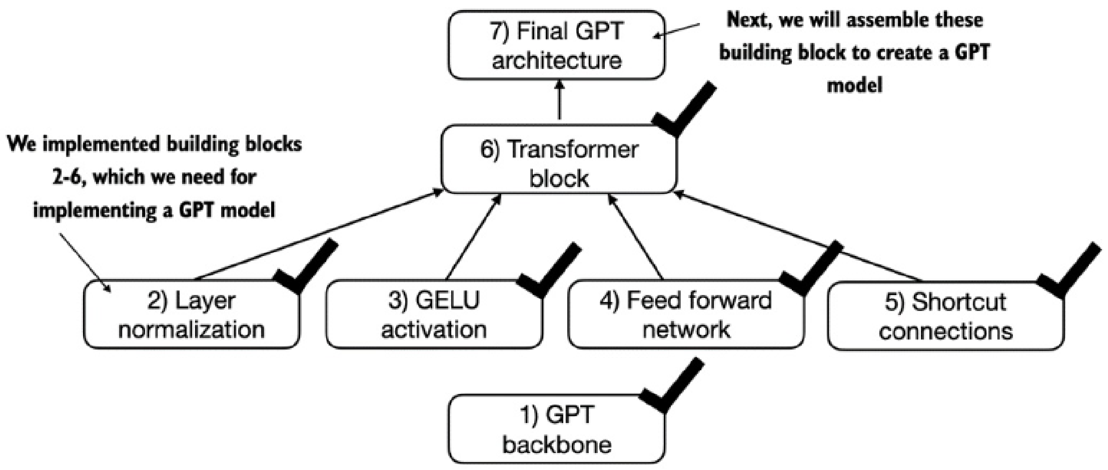
The idea is that the self-attention mechanism in the multi-head attention block identifies and analyzes relationships between elements in the input sequence. In contrast, the feed forward network modifies the data individually at each position. This combination not only enables a more
nuanced understanding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

```
from previous_chapters 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"],
            block_size=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_resid = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_resid(x)
        x = x + shortcut # Add the original input back
        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_resid(x)
        x = x + shortcut
        return x
```

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 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, as we learned in 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.

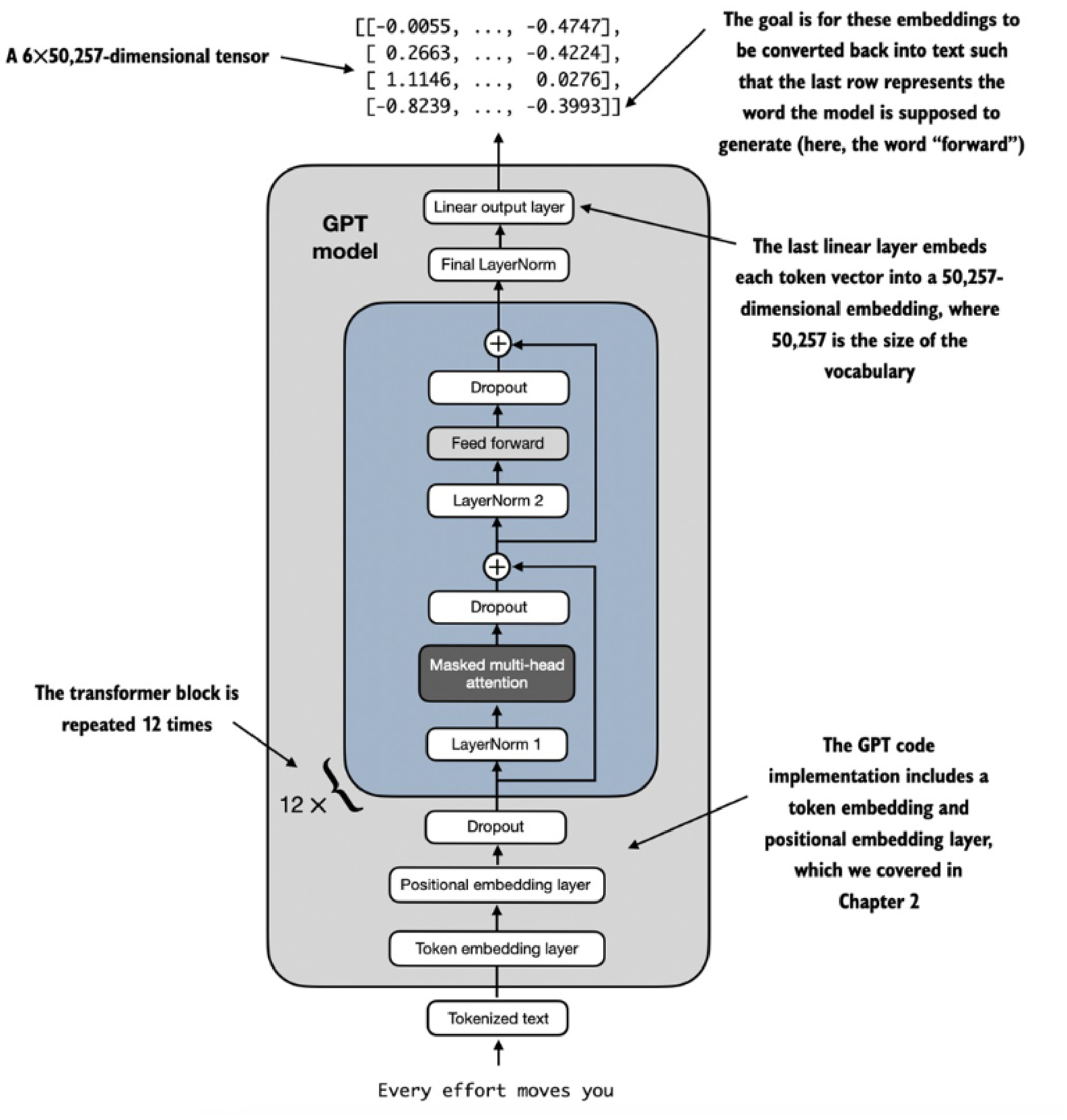
A mental model of the different concepts we have implemented in this chapter so far.


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

Coding the GPT model

In this section, we are now replacing the DummyTransformerBlock and DummyLayerNorm placeholders with the real TransformerBlock and LayerNorm classes we coded later in this chapter to assemble a fully working version of the original 124 million parameter version of GPT-2. Later, we will pretrain a GPT-2 model, and we will load in the pretrained weights from OpenAI.

An overview of the GPT model architecture. This figure illustrates 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.



A 6×50,257-dimensional tensor
[[-0.0055, ..., -0.4747],
[0.2663, ..., -0.4224],
[1.1146, ..., 0.0276],
[-0.8239, ..., -0.3993]]
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")
GPT model
Linear output layer
Final LayerNorm
The last linear layer embeds each token vector into a 50,257-dimensional embedding, where 50,257 is the size of the vocabulary
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
LayerNorm 1
The transformer block is repeated 12 times
12 ×
Dropout
The GPT code implementation includes a token embedding and positional embedding layer, which we covered in Chapter 2
Positional embedding layer
Token embedding layer
Tokenized text
Every effort moves you

The transformer block we coded 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. In the case of the largest GPT-2 model with 1,542 million parameters, this transformer block is repeated 36 times.

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_
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["
        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=i
        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 **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

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 created at the beginning of this chapter:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
out = model(batch)
print("Input batch:\n", batch)
print("\nOutput shape:", out.shape)
print(out)
```

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

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, as shown in the preceding code output?

The reason is a concept called weight tying that is used in the original GPT-2 architecture, which means that the original GPT-2 architecture is reusing the weights from the token embedding layer inits output layer

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 are using 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.

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

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

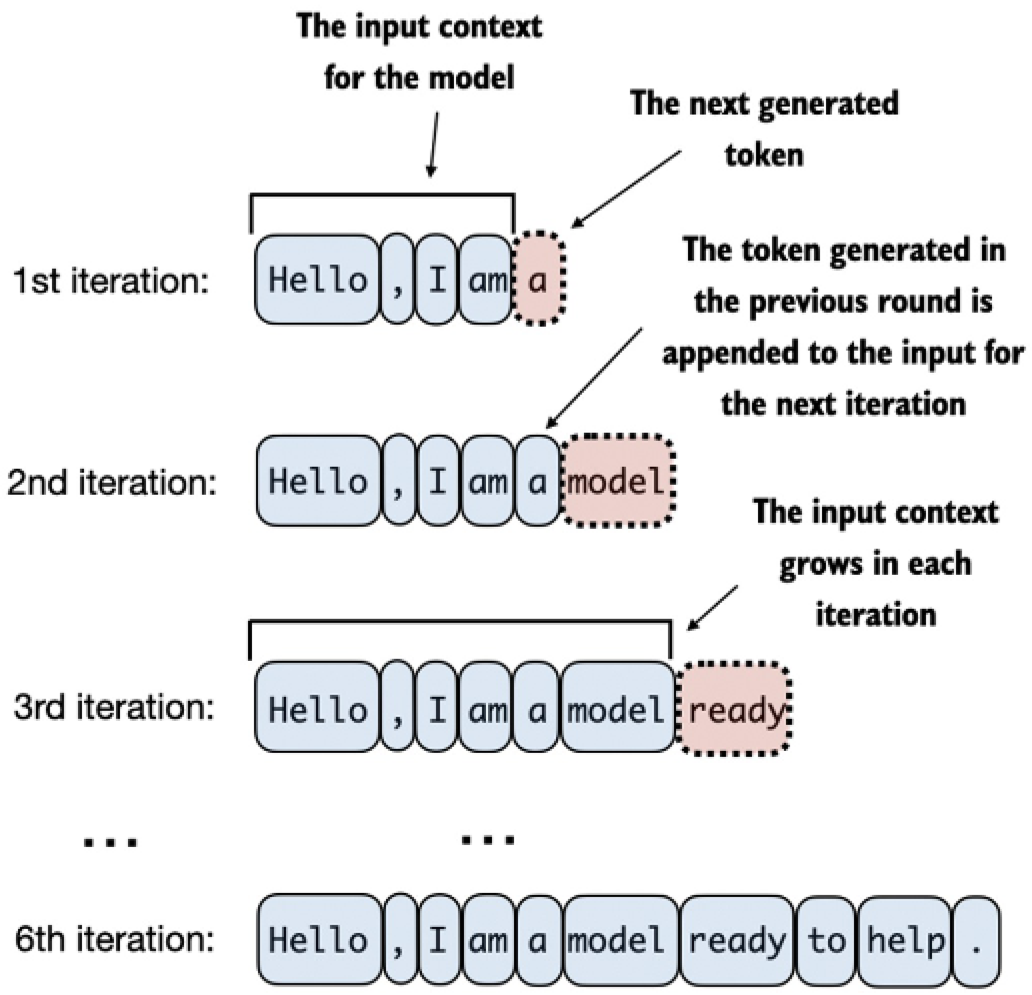
```
Total size of the model: 621.83 MB
```

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.

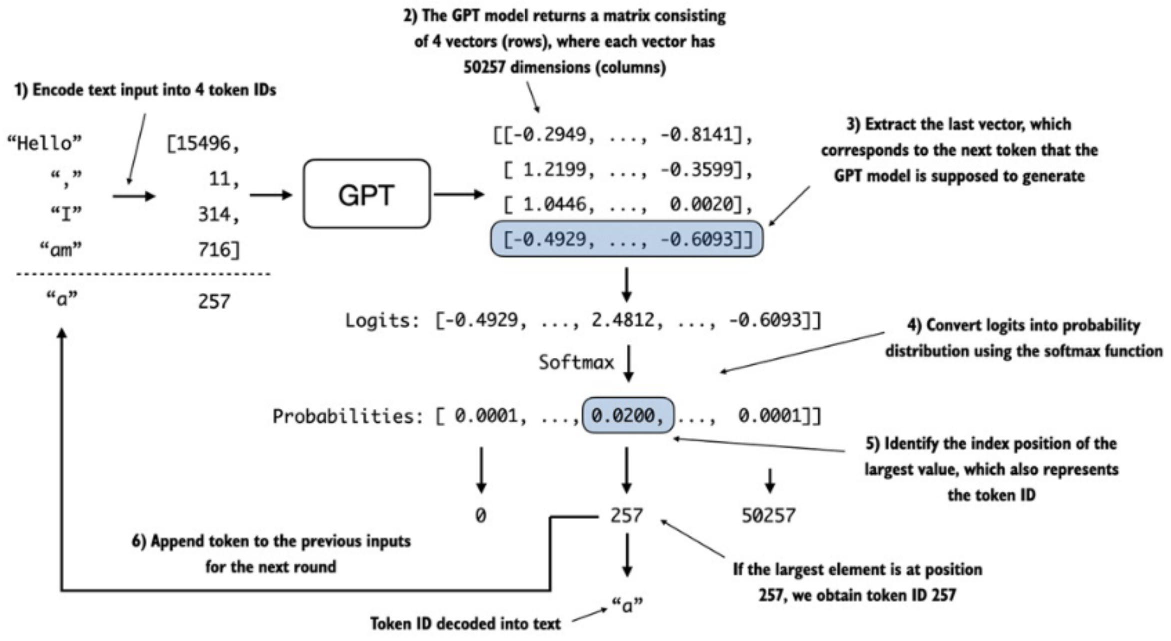
Generating text

In this final section of this chapter, we will 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

This diagram illustrates 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.


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

Details 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.


2) The GPT model returns a matrix consisting of 4 vectors (rows), where each vector has 50257 dimensions (columns)
1) Encode text input into 4 token IDs
"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]]
3) Extract the last vector, which corresponds to the next token that the GPT model is supposed to generate
Logits: [-0.4929, ..., 2.4812, ..., -0.6093]]
Softmax
4) Convert logits into probability distribution using the softmax function
Probabilities: [0.0001, ..., 0.0200, ..., 0.0001]]
5) Identify the index position of the largest value, which also represents the token ID
0
257
50257
6) Append token to the previous inputs for the next round
If the largest element is at position 257, we obtain token ID 257
Token ID decoded into text
"a"

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.

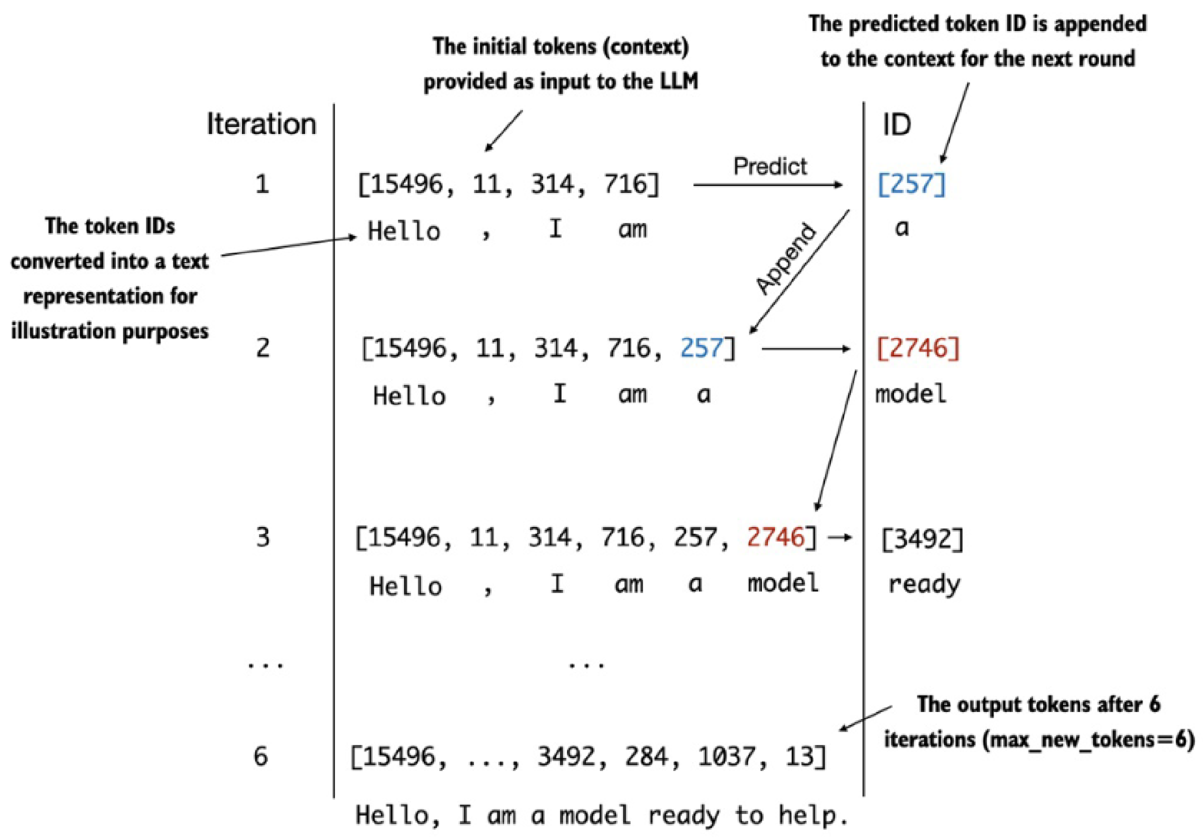
```
def generate_text_simple(model, idx, max_new_tokens, context_si
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
            with torch.no_grad():
        logits = model(idx_cond)
        logits = logits[:, -1, :]
        probas = torch.softmax(logits, dim=-1)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)
    return idx
```

The code snippet provided 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.

In the preceeding 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, we coded the conversion to illustrate the full process of transforming logits to probabilities, which can add additional intuition, such as that the model generates the most likely next token, which is known as greedy decoding.

In the next chapter, when we will implement the GPT training code, we will also introduce additional sampling techniques where we modify the softmax outputs such that the model doesn't always select the most likely token, which introduces variability and creativity in the generated text.

An illustration showing 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.)


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]
Hello , I am
a
The token IDs converted into a text representation for illustration purposes
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 6 iterations (max_new_tokens=6)
6
[15496, ..., 3492, 284, 1037, 13]
Hello, I am a model ready to help.

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 1542 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, which is the topic of subsequent chapters.