Next, we will test the tokenizer further on text that contains unknown words and discuss additional special tokens that can be used to provide further context for an LLM during training.

## 2.4 Adding special context tokens

We need to modify the tokenizer to handle unknown words. We also need to address the usage and addition of special context tokens that can enhance a model's understanding of context or other relevant information in the text. These special tokens can include markers for unknown words and document boundaries, for example. In particular, we will modify the vocabulary and tokenizer, `SimpleTokenizerV2`, to support two new tokens, `<|unk|>` and `<|endoftext|>`, as illustrated in figure 2.9.

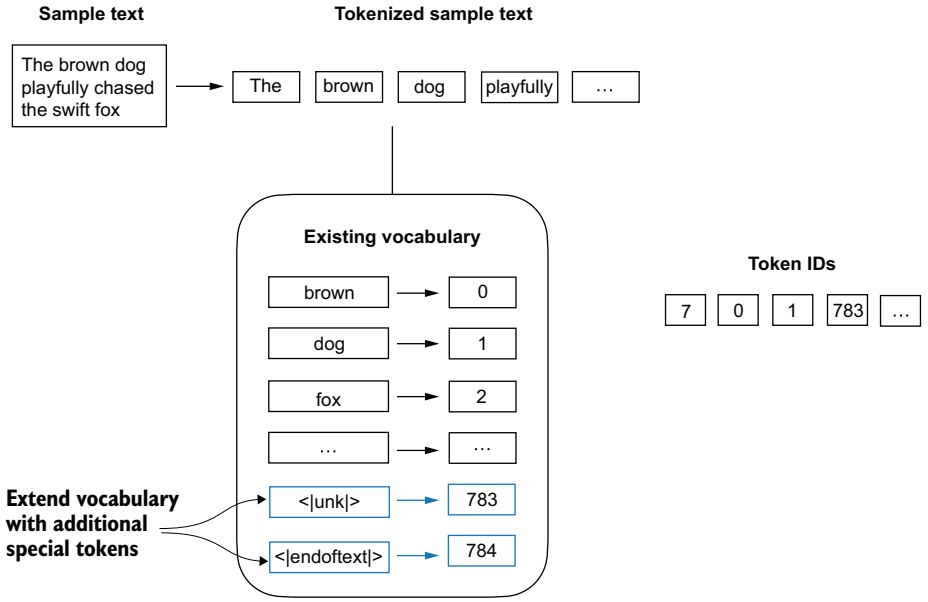



**Figure 2.9 We add special tokens to a vocabulary to deal with certain contexts. For instance, we add an `<|unk|>` token to represent new and unknown words that were not part of the training data and thus not part of the existing vocabulary. Furthermore, we add an `<|endoftext|>` token that we can use to separate two unrelated text sources.**

We can modify the tokenizer to use an `<|unk|>` token if it encounters a word that is not part of the vocabulary. Furthermore, we add a token between unrelated texts. For example, when training GPT-like LLMs on multiple independent documents or books, it is common to insert a token before each document or book that follows a previous text source, as illustrated in figure 2.10. This helps the LLM understand that although these text sources are concatenated for training, they are, in fact, unrelated.

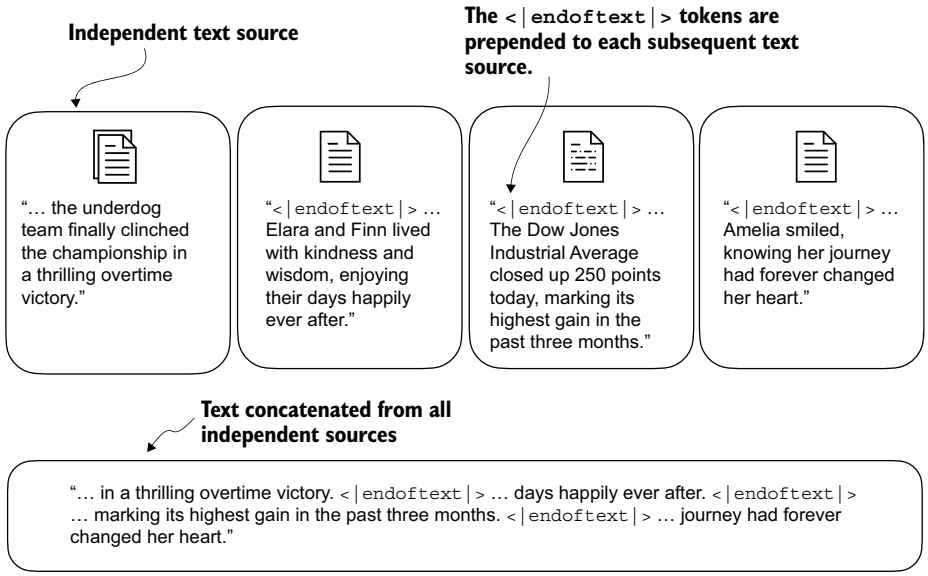



**Figure 2.10 When working with multiple independent text source, we add <|endoftext|> tokens between these texts. These <|endoftext|> tokens act as markers, signaling the start or end of a particular segment, allowing for more effective processing and understanding by the LLM.**

Let's now modify the vocabulary to include these two special tokens, <unk> and <|endoftext|>, by adding them to our list of all unique words:

```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
vocab = {token:integer for integer,token in enumerate(all_tokens)}

print(len(vocab.items()))
```

Based on the output of this print statement, the new vocabulary size is 1,132 (the previous vocabulary size was 1,130).

As an additional quick check, let's print the last five entries of the updated vocabulary:

```
for i, item in enumerate(list(vocab.items())[-5:]):
    print(item)
```

The code prints

```
('younger', 1127)
('your', 1128)
('yourself', 1129)
('<|endoftext|>', 1130)
('<|unk|>', 1131)
```

Based on the code output, we can confirm that the two new special tokens were indeed successfully incorporated into the vocabulary. Next, we adjust the tokenizer from code listing 2.3 accordingly as shown in the following listing.

**Listing 2.4 A simple text tokenizer that handles unknown words**

```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = { i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', text)
        preprocessed = [
            item.strip() for item in preprocessed if item.strip()
        ]
        preprocessed = [item if item in self.str_to_int
                        else "<|unk|>" for item in preprocessed]

        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])

        text = re.sub(r'\s+([,.:;?!"()\'])', r'\1', text)
        return text
```

Replaces unknown words by <|unk|> tokens

Replaces spaces before the specified punctuations

Compared to the `SimpleTokenizerV1` we implemented in listing 2.3, the new `SimpleTokenizerV2` replaces unknown words with `<|unk|>` tokens.

Let's now try this new tokenizer out in practice. For this, we will use a simple text sample that we concatenate from two independent and unrelated sentences:

```
text1 = "Hello, do you like tea?"
text2 = "In the sunlit terraces of the palace."
text = " <|endoftext|> ".join((text1, text2))
print(text)
```

The output is

```
Hello, do you like tea? <|endoftext|> In the sunlit terraces of
the palace.
```

Next, let's tokenize the sample text using the `SimpleTokenizerV2` on the vocab we previously created in listing 2.2:

```
tokenizer = SimpleTokenizerV2(vocab)
print(tokenizer.encode(text))
```

This prints the following token IDs:

```
[1131, 5, 355, 1126, 628, 975, 10, 1130, 55, 988, 956, 984, 722, 988, 1131, 7]
```

We can see that the list of token IDs contains `1130` for the `<|endoftext|>` separator token as well as two `1131` tokens, which are used for unknown words.

Let's detokenize the text for a quick sanity check:

```
print(tokenizer.decode(tokenizer.encode(text)))
```

The output is

```
<|unk|>, do you like tea? <|endoftext|> In the sunlit terraces of
the <|unk|>.
```

Based on comparing this detokenized text with the original input text, we know that the training dataset, Edith Wharton's short story "The Verdict," does not contain the words "Hello" and "palace."

Depending on the LLM, some researchers also consider additional special tokens such as the following:

- `[BOS]` *(beginning of sequence)*—This token marks the start of a text. It signifies to the LLM where a piece of content begins.
- `[EOS]` *(end of sequence)*—This token is positioned at the end of a text and is especially useful when concatenating multiple unrelated texts, similar to `<|endoftext|>`. For instance, when combining two different Wikipedia articles or books, the `[EOS]` token indicates where one ends and the next begins.
- `[PAD]` *(padding)*—When training LLMs with batch sizes larger than one, the batch might contain texts of varying lengths. To ensure all texts have the same length, the shorter texts are extended or "padded" using the `[PAD]` token, up to the length of the longest text in the batch.

The tokenizer used for GPT models does not need any of these tokens; it only uses an `<|endoftext|>` token for simplicity. `<|endoftext|>` is analogous to the `[EOS]` token. `<|endoftext|>` is also used for padding. However, as we'll explore in subsequent chapters, when training on batched inputs, we typically use a mask, meaning we don't attend to padded tokens. Thus, the specific token chosen for padding becomes inconsequential.

Moreover, the tokenizer used for GPT models also doesn't use an `<|unk|>` token for out-of-vocabulary words. Instead, GPT models use a *byte pair encoding* tokenizer, which breaks words down into subword units, which we will discuss next.

## 2.5 Byte pair encoding

Let's look at a more sophisticated tokenization scheme based on a concept called byte pair encoding (BPE). The BPE tokenizer was used to train LLMs such as GPT-2, GPT-3, and the original model used in ChatGPT.

Since implementing BPE can be relatively complicated, we will use an existing Python open source library called *tiktoken* (https://github.com/openai/tiktoken), which implements the BPE algorithm very efficiently based on source code in Rust. Similar to other Python libraries, we can install the tiktoken library via Python's `pip` installer from the terminal:

```
pip install tiktoken
```

The code we will use is based on tiktoken 0.7.0. You can use the following code to check the version you currently have installed:

```
from importlib.metadata import version
import tiktoken
print("tiktoken version:", version("tiktoken"))
```

Once installed, we can instantiate the BPE tokenizer from tiktoken as follows:

```
tokenizer = tiktoken.get_encoding("gpt2")
```

The usage of this tokenizer is similar to the `SimpleTokenizerV2` we implemented previously via an `encode` method:

```
text = (
    "Hello, do you like tea? <|endoftext|> In the sunlit terraces"
     "of someunknownPlace."
)
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})
print(integers)
```

The code prints the following token IDs:

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250,
 8812, 2114, 286, 617, 34680, 27271, 13]
```

We can then convert the token IDs back into text using the decode method, similar to our `SimpleTokenizerV2`:

```
strings = tokenizer.decode(integers)
print(strings)
```

The code prints

```
Hello, do you like tea? <|endoftext|> In the sunlit terraces of
 someunknownPlace.
```

We can make two noteworthy observations based on the token IDs and decoded text. First, the `<|endoftext|>` token is assigned a relatively large token ID, namely, `50256`. In fact, the BPE tokenizer, which was used to train models such as GPT-2, GPT-3, and the original model used in ChatGPT, has a total vocabulary size of 50,257, with `<|endoftext|>` being assigned the largest token ID.

Second, the BPE tokenizer encodes and decodes unknown words, such as `someunknownPlace`, correctly. The BPE tokenizer can handle any unknown word. How does it achieve this without using `<|unk|>` tokens?

The algorithm underlying BPE breaks down words that aren't in its predefined vocabulary into smaller subword units or even individual characters, enabling it to handle out-of-vocabulary words. So, thanks to the BPE algorithm, if the tokenizer encounters an unfamiliar word during tokenization, it can represent it as a sequence of subword tokens or characters, as illustrated in figure 2.11.

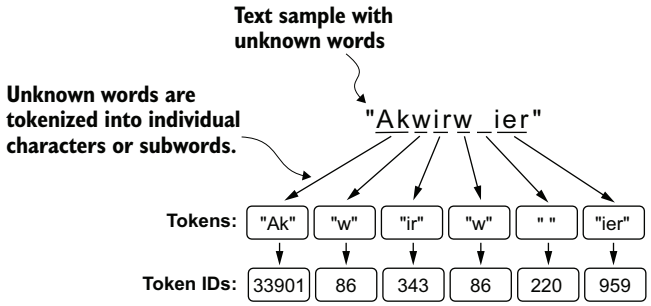



**Figure 2.11 BPE tokenizers break down unknown words into subwords and individual characters. This way, a BPE tokenizer can parse any word and doesn't need to replace unknown words with special tokens, such as `<|unk|>`.**

The ability to break down unknown words into individual characters ensures that the tokenizer and, consequently, the LLM that is trained with it can process any text, even if it contains words that were not present in its training data.

### Exercise 2.1 Byte pair encoding of unknown words

Try the BPE tokenizer from the tiktoken library on the unknown words "Akwirw ier" and print the individual token IDs. Then, call the `decode` function on each of the resulting integers in this list to reproduce the mapping shown in figure 2.11. Lastly, call the decode method on the token IDs to check whether it can reconstruct the original input, "Akwirw ier."

A detailed discussion and implementation of BPE is out of the scope of this book, but in short, it builds its vocabulary by iteratively merging frequent characters into subwords and frequent subwords into words. For example, BPE starts with adding all individual single characters to its vocabulary ("a," "b," etc.). In the next stage, it merges character combinations that frequently occur together into subwords. For example, "d" and "e" may be merged into the subword "de," which is common in many English

words like “define,” “depend,” “made,” and “hidden.” The merges are determined by a frequency cutoff.

## 2.6 Data sampling with a sliding window

The next step in creating the embeddings for the LLM is to generate the input–target pairs required for training an LLM. What do these input–target pairs look like? As we already learned, LLMs are pretrained by predicting the next word in a text, as depicted in figure 2.12.

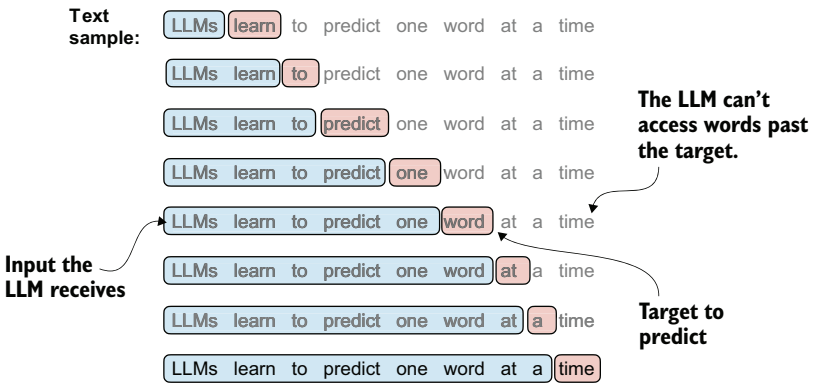



**Figure 2.12 Given a text sample, extract input blocks as subsamples that serve as input to the LLM, and the LLM’s prediction task during training is to predict the next word that follows the input block. During training, we mask out all words that are past the target. Note that the text shown in this figure must undergo tokenization before the LLM can process it; however, this figure omits the tokenization step for clarity.**

Let’s implement a data loader that fetches the input–target pairs in figure 2.12 from the training dataset using a sliding window approach. To get started, we will tokenize the whole “The Verdict” short story using the BPE tokenizer:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

enc_text = tokenizer.encode(raw_text)
print(len(enc_text))
```

Executing this code will return `5145`, the total number of tokens in the training set, after applying the BPE tokenizer.

Next, we remove the first 50 tokens from the dataset for demonstration purposes, as it results in a slightly more interesting text passage in the next steps:

```
enc_sample = enc_text[50:]
```

One of the easiest and most intuitive ways to create the input–target pairs for the next-word prediction task is to create two variables, `x` and `y`, where `x` contains the input tokens and `y` contains the targets, which are the inputs shifted by 1:

```
context_size = 4
x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]
print(f"x: {x}")
print(f"y:      {y}")
```

**The context size determines how many tokens are included in the input.**

Running the previous code prints the following output:

```
x: [290, 4920, 2241, 287]
y:      [4920, 2241, 287, 257]
```

By processing the inputs along with the targets, which are the inputs shifted by one position, we can create the next-word prediction tasks (see figure 2.12), as follows:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(context, "---->", desired)
```

The code prints

```
[290] ----> 4920
[290, 4920] ----> 2241
[290, 4920, 2241] ----> 287
[290, 4920, 2241, 287] ----> 257
```

Everything left of the arrow (`---->`) refers to the input an LLM would receive, and the token ID on the right side of the arrow represents the target token ID that the LLM is supposed to predict. Let's repeat the previous code but convert the token IDs into text:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(tokenizer.decode(context), "---->", tokenizer.decode([desired]))
```

The following outputs show how the input and outputs look in text format:

```
 and ---->  established
 and established ---->  himself
 and established himself ---->  in
 and established himself in ---->  a
```

We've now created the input–target pairs that we can use for LLM training.

There's only one more task before we can turn the tokens into embeddings: implementing an efficient data loader that iterates over the input dataset and returns the

inputs and targets as PyTorch tensors, which can be thought of as multidimensional arrays. In particular, we are interested in returning two tensors: an input tensor containing the text that the LLM sees and a target tensor that includes the targets for the LLM to predict, as depicted in figure 2.13. While the figure shows the tokens in string format for illustration purposes, the code implementation will operate on token IDs directly since the `encode` method of the BPE tokenizer performs both tokenization and conversion into token IDs as a single step.

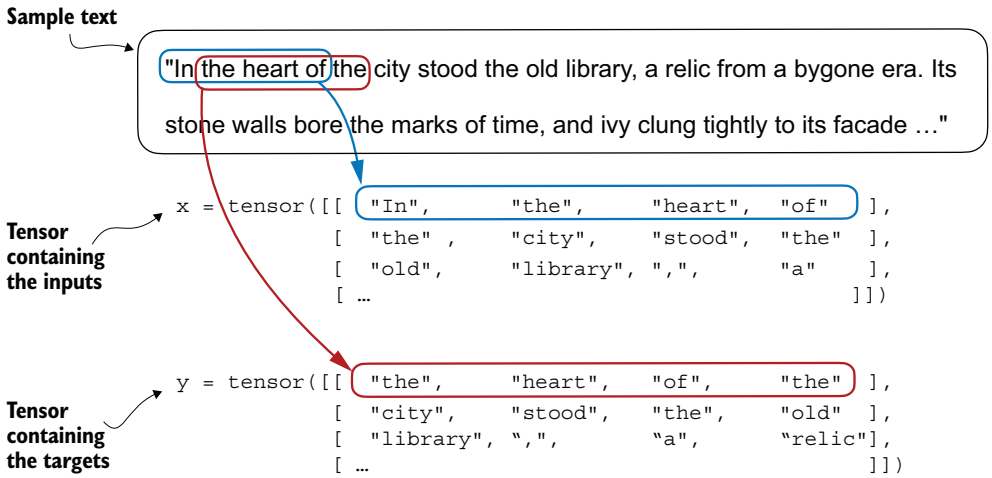



**Figure 2.13 To implement efficient data loaders, we collect the inputs in a tensor, `x`, where each row represents one input context. A second tensor, `y`, contains the corresponding prediction targets (next words), which are created by shifting the input by one position.**

> **NOTE** For the efficient data loader implementation, we will use PyTorch's built-in `Dataset` and `DataLoader` classes. For additional information and guidance on installing PyTorch, please see section A.2.1.3 in appendix A.

The code for the dataset class is shown in the following listing.

**Listing 2.5 A dataset for batched inputs and targets**

```
import torch
from torch.utils.data import Dataset, DataLoader

class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.input_ids = []
        self.target_ids = []

        token_ids = tokenizer.encode(txt)
```

Tokenizes the entire text

```
        for i in range(0, len(token_ids) - max_length, stride):
            input_chunk = token_ids[i:i + max_length]
            target_chunk = token_ids[i + 1: i + max_length + 1]
            self.input_ids.append(torch.tensor(input_chunk))
            self.target_ids.append(torch.tensor(target_chunk))

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return self.input_ids[idx], self.target_ids[idx]
```

**Uses a sliding window to chunk the book into overlapping sequences of max_length**

**Returns a single row from the dataset**

**Returns the total number of rows in the dataset**

The `GPTDatasetV1` class is based on the PyTorch `Dataset` class and defines how individual rows are fetched from the dataset, where each row consists of a number of token IDs (based on a `max_length`) assigned to an `input_chunk` tensor. The `target_chunk` tensor contains the corresponding targets. I recommend reading on to see what the data returned from this dataset looks like when we combine the dataset with a PyTorch `DataLoader`—this will bring additional intuition and clarity.

> **NOTE** If you are new to the structure of PyTorch `Dataset` classes, such as shown in listing 2.5, refer to section A.6 in appendix A, which explains the general structure and usage of PyTorch `Dataset` and `DataLoader` classes.

The following code uses the `GPTDatasetV1` to load the inputs in batches via a PyTorch `DataLoader`.

**Listing 2.6 A data loader to generate batches with input-with pairs**

```
def create_dataloader_v1(txt, batch_size=4, max_length=256,
                         stride=128, shuffle=True, drop_last=True,
                         num_workers=0):
    tokenizer = tiktoken.get_encoding("gpt2")
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        drop_last=drop_last,
        num_workers=num_workers
    )

    return dataloader
```

**Initializes the tokenizer**

**Creates dataset**

**drop_last=True drops the last batch if it is shorter than the specified batch_size to prevent loss spikes during training.**

**The number of CPU processes to use for preprocessing**

Let's test the `dataloader` with a batch size of 1 for an LLM with a context size of 4 to develop an intuition of how the `GPTDatasetV1` class from listing 2.5 and the `create_dataloader_v1` function from listing 2.6 work together:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

dataloader = create_dataloader_v1(
    raw_text, batch_size=1, max_length=4, stride=1, shuffle=False)
data_iter = iter(dataloader)
first_batch = next(data_iter)
print(first_batch)
```

**Converts dataloader into a Python iterator to fetch the next entry via Python's built-in next() function**

Executing the preceding code prints the following:

```
[tensor([[  40,  367, 2885, 1464]]), tensor([[ 367, 2885, 1464, 1807]])]
```

The `first_batch` variable contains two tensors: the first tensor stores the input token IDs, and the second tensor stores the target token IDs. Since the `max_length` is set to 4, each of the two tensors contains four token IDs. Note that an input size of 4 is quite small and only chosen for simplicity. It is common to train LLMs with input sizes of at least 256.

To understand the meaning of `stride=1`, let's fetch another batch from this dataset:

```
second_batch = next(data_iter)
print(second_batch)
```

The second batch has the following contents:

```
[tensor([[ 367, 2885, 1464, 1807]]), tensor([[2885, 1464, 1807, 3619]])]
```

If we compare the first and second batches, we can see that the second batch's token IDs are shifted by one position (for example, the second ID in the first batch's input is 367, which is the first ID of the second batch's input). The `stride` setting dictates the number of positions the inputs shift across batches, emulating a sliding window approach, as demonstrated in figure 2.14.

**Exercise 2.2 Data loaders with different strides and context sizes**

To develop more intuition for how the data loader works, try to run it with different settings such as `max_length=2` and `stride=2`, and `max_length=8` and `stride=2`.

Batch sizes of 1, such as we have sampled from the data loader so far, are useful for illustration purposes. If you have previous experience with deep learning, you may know that small batch sizes require less memory during training but lead to more

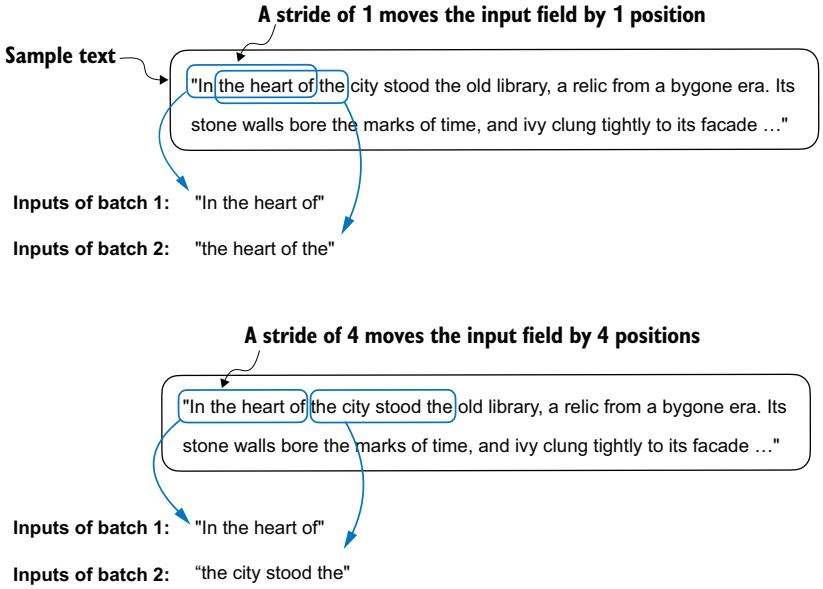



**Figure 2.14 When creating multiple batches from the input dataset, we slide an input window across the text. If the stride is set to 1, we shift the input window by one position when creating the next batch. If we set the stride equal to the input window size, we can prevent overlaps between the batches.**

noisy model updates. Just like in regular deep learning, the batch size is a tradeoff and a hyperparameter to experiment with when training LLMs.

Let's look briefly at how we can use the data loader to sample with a batch size greater than 1:

```
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=4, stride=4,
    shuffle=False
)

data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Inputs:\n", inputs)
print("\nTargets:\n", targets)
```

This prints

```
Inputs:
 tensor([[   40,   367,  2885,  1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438,  2016,   257],
        [  922,  5891,  1576,   438],
        [  568,   340,   373,   645],
```

```
        [ 1049,  5975,   284,   502],
        [  284,  3285,   326,    11]])

Targets:
 tensor([[  367,  2885,  1464,  1807],
        [ 3619,   402,   271, 10899],
        [ 2138,   257,  7026, 15632],
        [  438,  2016,   257,   922],
        [ 5891,  1576,   438,   568],
        [  340,   373,   645,  1049],
        [ 5975,   284,   502,   284],
        [ 3285,   326,    11,   287]])
```

Note that we increase the stride to 4 to utilize the data set fully (we don't skip a single word). This avoids any overlap between the batches since more overlap could lead to increased overfitting.

## 2.7 Creating token embeddings

The last step in preparing the input text for LLM training is to convert the token IDs into embedding vectors, as shown in figure 2.15. As a preliminary step, we must initialize

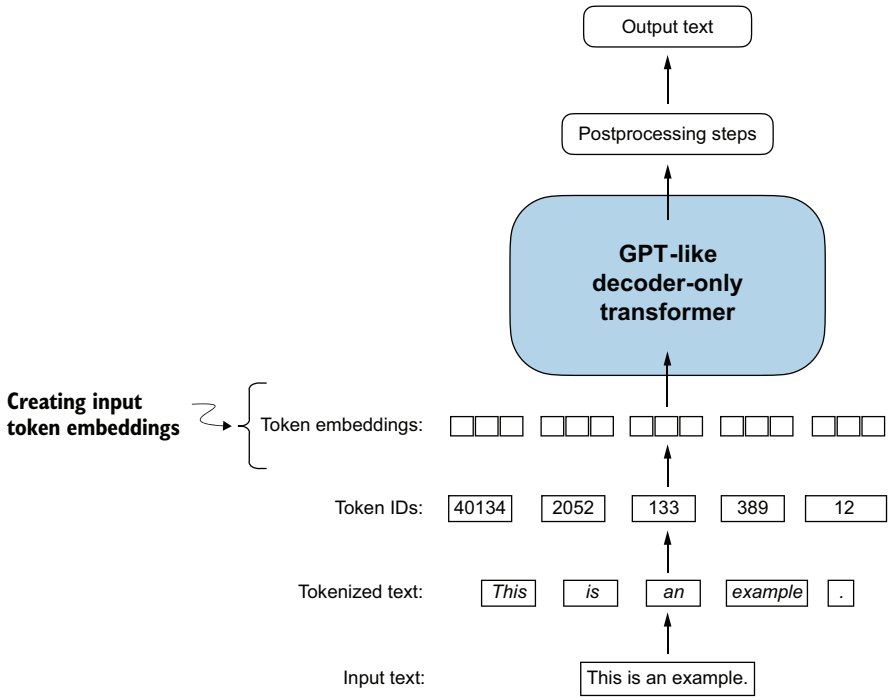



**Figure 2.15 Preparation involves tokenizing text, converting text tokens to token IDs, and converting token IDs into embedding vectors. Here, we consider the previously created token IDs to create the token embedding vectors.**

these embedding weights with random values. This initialization serves as the starting point for the LLM's learning process. In chapter 5, we will optimize the embedding weights as part of the LLM training.

A continuous vector representation, or embedding, is necessary since GPT-like LLMs are deep neural networks trained with the backpropagation algorithm.

> **NOTE** If you are unfamiliar with how neural networks are trained with backpropagation, please read section A.4 in appendix A.

Let's see how the token ID to embedding vector conversion works with a hands-on example. Suppose we have the following four input tokens with IDs 2, 3, 5, and 1:

```
input_ids = torch.tensor([2, 3, 5, 1])
```

For the sake of simplicity, suppose we have a small vocabulary of only 6 words (instead of the 50,257 words in the BPE tokenizer vocabulary), and we want to create embeddings of size 3 (in GPT-3, the embedding size is 12,288 dimensions):

```
vocab_size = 6
output_dim = 3
```

Using the `vocab_size` and `output_dim`, we can instantiate an embedding layer in PyTorch, setting the random seed to `123` for reproducibility purposes:

```
torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
print(embedding_layer.weight)
```

The print statement prints the embedding layer's underlying weight matrix:

```
Parameter containing:
tensor([[ 0.3374, -0.1778, -0.1690],
        [ 0.9178,  1.5810,  1.3010],
        [ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-1.1589,  0.3255, -0.6315],
        [-2.8400, -0.7849, -1.4096]], requires_grad=True)
```

The weight matrix of the embedding layer contains small, random values. These values are optimized during LLM training as part of the LLM optimization itself. Moreover, we can see that the weight matrix has six rows and three columns. There is one row for each of the six possible tokens in the vocabulary, and there is one column for each of the three embedding dimensions.

Now, let's apply it to a token ID to obtain the embedding vector:

```
print(embedding_layer(torch.tensor([3])))
```

The returned embedding vector is

```
tensor([[-0.4015,  0.9666, -1.1481]], grad_fn=<EmbeddingBackward0>)
```

If we compare the embedding vector for token ID 3 to the previous embedding matrix, we see that it is identical to the fourth row (Python starts with a zero index, so it's the row corresponding to index 3). In other words, the embedding layer is essentially a lookup operation that retrieves rows from the embedding layer's weight matrix via a token ID.

> **NOTE** For those who are familiar with one-hot encoding, the embedding layer approach described here is essentially just a more efficient way of implementing one-hot encoding followed by matrix multiplication in a fully connected layer, which is illustrated in the supplementary code on GitHub at https://mng.bz/ZEB5. Because the embedding layer is just a more efficient implementation equivalent to the one-hot encoding and matrix-multiplication approach, it can be seen as a neural network layer that can be optimized via backpropagation.

We've seen how to convert a single token ID into a three-dimensional embedding vector. Let's now apply that to all four input IDs (`torch.tensor([2, 3, 5, 1])`):

```
print(embedding_layer(input_ids))
```

The print output reveals that this results in a 4 × 3 matrix:

```
tensor([[ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-2.8400, -0.7849, -1.4096],
        [ 0.9178,  1.5810,  1.3010]], grad_fn=<EmbeddingBackward0>)
```

Each row in this output matrix is obtained via a lookup operation from the embedding weight matrix, as illustrated in figure 2.16.

Having now created embedding vectors from token IDs, next we'll add a small modification to these embedding vectors to encode positional information about a token within a text.

## 2.8 Encoding word positions

In principle, token embeddings are a suitable input for an LLM. However, a minor shortcoming of LLMs is that their self-attention mechanism (see chapter 3) doesn't have a notion of position or order for the tokens within a sequence. The way the previously introduced embedding layer works is that the same token ID always gets mapped to the same vector representation, regardless of where the token ID is positioned in the input sequence, as shown in figure 2.17.

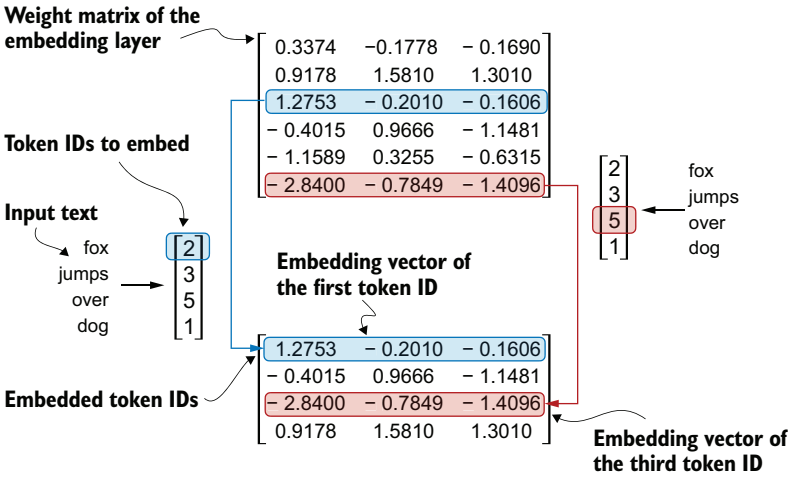



**Figure 2.16** Embedding layers perform a lookup operation, retrieving the embedding vector corresponding to the token ID from the embedding layer's weight matrix. For instance, the embedding vector of the token ID 5 is the sixth row of the embedding layer weight matrix (it is the sixth instead of the fifth row because Python starts counting at 0). We assume that the token IDs were produced by the small vocabulary from section 2.3.

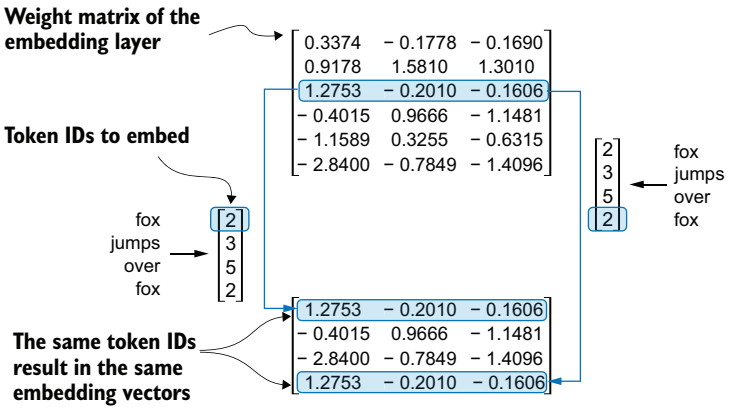



**Figure 2.17** The embedding layer converts a token ID into the same vector representation regardless of where it is located in the input sequence. For example, the token ID 5, whether it's in the first or fourth position in the token ID input vector, will result in the same embedding vector.

In principle, the deterministic, position-independent embedding of the token ID is good for reproducibility purposes. However, since the self-attention mechanism of LLMs itself is also position-agnostic, it is helpful to inject additional position information into the LLM.

To achieve this, we can use two broad categories of position-aware embeddings: relative positional embeddings and absolute positional embeddings. Absolute positional embeddings are directly associated with specific positions in a sequence. For each position in the input sequence, a unique embedding is added to the token's embedding to convey its exact location. For instance, the first token will have a specific positional embedding, the second token another distinct embedding, and so on, as illustrated in figure 2.18.

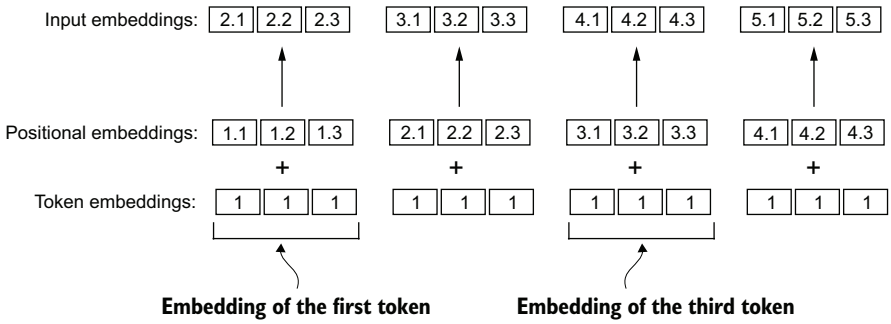



**Figure 2.18 Positional embeddings are added to the token embedding vector to create the input embeddings for an LLM. The positional vectors have the same dimension as the original token embeddings. The token embeddings are shown with value 1 for simplicity.**

Instead of focusing on the absolute position of a token, the emphasis of relative positional embeddings is on the relative position or distance between tokens. This means the model learns the relationships in terms of "how far apart" rather than "at which exact position." The advantage here is that the model can generalize better to sequences of varying lengths, even if it hasn't seen such lengths during training.

Both types of positional embeddings aim to augment the capacity of LLMs to understand the order and relationships between tokens, ensuring more accurate and context-aware predictions. The choice between them often depends on the specific application and the nature of the data being processed.

OpenAI's GPT models use absolute positional embeddings that are optimized during the training process rather than being fixed or predefined like the positional encodings in the original transformer model. This optimization process is part of the model training itself. For now, let's create the initial positional embeddings to create the LLM inputs.

Previously, we focused on very small embedding sizes for simplicity. Now, let's consider more realistic and useful embedding sizes and encode the input tokens into a 256-dimensional vector representation, which is smaller than what the original GPT-3 model used (in GPT-3, the embedding size is 12,288 dimensions) but still reasonable for experimentation. Furthermore, we assume that the token IDs were created by the BPE tokenizer we implemented earlier, which has a vocabulary size of 50,257:

```
vocab_size = 50257
output_dim = 256
token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
```

Using the previous `token_embedding_layer`, if we sample data from the data loader, we embed each token in each batch into a 256-dimensional vector. If we have a batch size of 8 with four tokens each, the result will be an $8 \times 4 \times 256$ tensor.

Let's instantiate the data loader (see section 2.6) first:

```
max_length = 4
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=max_length,
   stride=max_length, shuffle=False
)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Token IDs:\n", inputs)
print("\nInputs shape:\n", inputs.shape)
```

This code prints

```
Token IDs:
 tensor([[   40,   367,  2885,  1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438,  2016,   257],
        [  922,  5891,  1576,   438],
        [  568,   340,   373,   645],
        [ 1049,  5975,   284,   502],
        [  284,  3285,   326,    11]])

Inputs shape:
 torch.Size([8, 4])
```

As we can see, the token ID tensor is $8 \times 4$ dimensional, meaning that the data batch consists of eight text samples with four tokens each.

Let's now use the embedding layer to embed these token IDs into 256-dimensional vectors:

```
token_embeddings = token_embedding_layer(inputs)
print(token_embeddings.shape)
```

The print function call returns

```
torch.Size([8, 4, 256])
```

The 8 × 4 × 256–dimensional tensor output shows that each token ID is now embedded as a 256-dimensional vector.

For a GPT model's absolute embedding approach, we just need to create another embedding layer that has the same embedding dimension as the `token_embedding_layer`:

```
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_length, output_dim)
pos_embeddings = pos_embedding_layer(torch.arange(context_length))
print(pos_embeddings.shape)
```

The input to the `pos_embeddings` is usually a placeholder vector `torch.arange(context_length)`, which contains a sequence of numbers 0, 1, ..., up to the maximum input length –1. The `context_length` is a variable that represents the supported input size of the LLM. Here, we choose it similar to the maximum length of the input text. In practice, input text can be longer than the supported context length, in which case we have to truncate the text.

The output of the print statement is

```
torch.Size([4, 256])
```

As we can see, the positional embedding tensor consists of four 256-dimensional vectors. We can now add these directly to the token embeddings, where PyTorch will add the 4 × 256–dimensional `pos_embeddings` tensor to each 4 × 256–dimensional token embedding tensor in each of the eight batches:

```
input_embeddings = token_embeddings + pos_embeddings
print(input_embeddings.shape)
```

The print output is

```
torch.Size([8, 4, 256])
```

The `input_embeddings` we created, as summarized in figure 2.19, are the embedded input examples that can now be processed by the main LLM modules, which we will begin implementing in the next chapter.

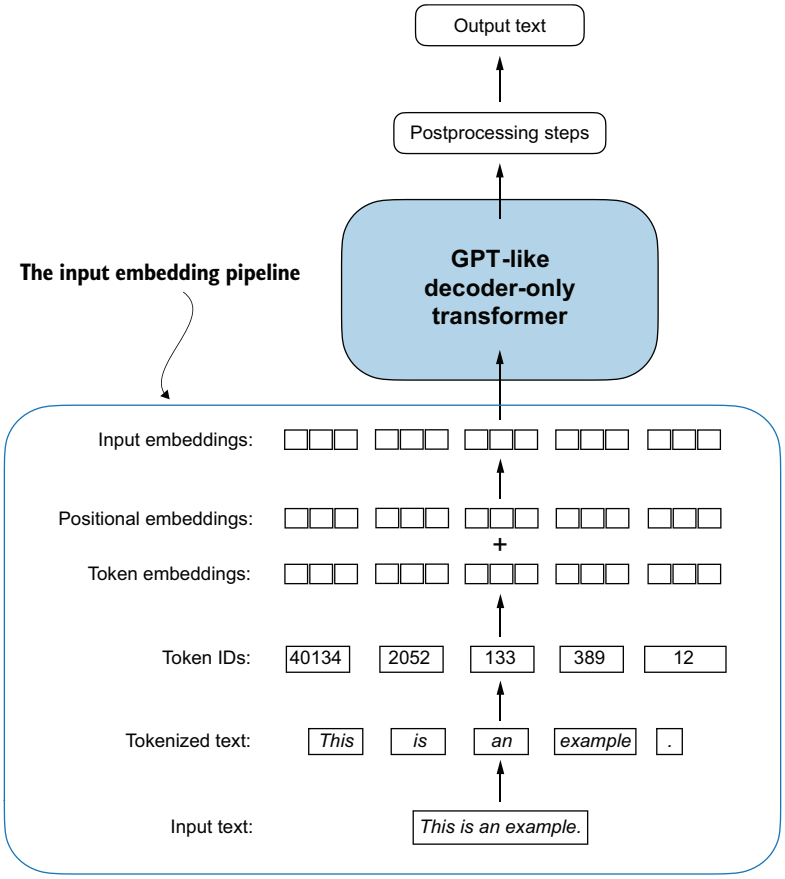



**Figure 2.19 As part of the input processing pipeline, input text is first broken up into individual tokens. These tokens are then converted into token IDs using a vocabulary. The token IDs are converted into embedding vectors to which positional embeddings of a similar size are added, resulting in input embeddings that are used as input for the main LLM layers.**

## Summary

- LLMs require textual data to be converted into numerical vectors, known as embeddings, since they can't process raw text. Embeddings transform discrete data (like words or images) into continuous vector spaces, making them compatible with neural network operations.
- As the first step, raw text is broken into tokens, which can be words or characters. Then, the tokens are converted into integer representations, termed token IDs.
- Special tokens, such as `<|unk|>` and `<|endoftext|>`, can be added to enhance the model's understanding and handle various contexts, such as unknown words or marking the boundary between unrelated texts.

- The byte pair encoding (BPE) tokenizer used for LLMs like GPT-2 and GPT-3 can efficiently handle unknown words by breaking them down into subword units or individual characters.
- We use a sliding window approach on tokenized data to generate input–target pairs for LLM training.
- Embedding layers in PyTorch function as a lookup operation, retrieving vectors corresponding to token IDs. The resulting embedding vectors provide continuous representations of tokens, which is crucial for training deep learning models like LLMs.
- While token embeddings provide consistent vector representations for each token, they lack a sense of the token's position in a sequence. To rectify this, two main types of positional embeddings exist: absolute and relative. OpenAI's GPT models utilize absolute positional embeddings, which are added to the token embedding vectors and are optimized during the model training.

# 3 Coding attention mechanisms

***This chapter covers***

- The reasons for using attention mechanisms in neural networks
- A basic self-attention framework, progressing to an enhanced self-attention mechanism
- A causal attention module that allows LLMs to generate one token at a time
- Masking randomly selected attention weights with dropout to reduce overfitting
- Stacking multiple causal attention modules into a multi-head attention module

At this point, you know how to prepare the input text for training LLMs by splitting text into individual word and subword tokens, which can be encoded into vector representations, embeddings, for the LLM.

Now, we will look at an integral part of the LLM architecture itself, attention mechanisms, as illustrated in figure 3.1. We will largely look at attention mechanisms in isolation and focus on them at a mechanistic level. Then we will code the remaining

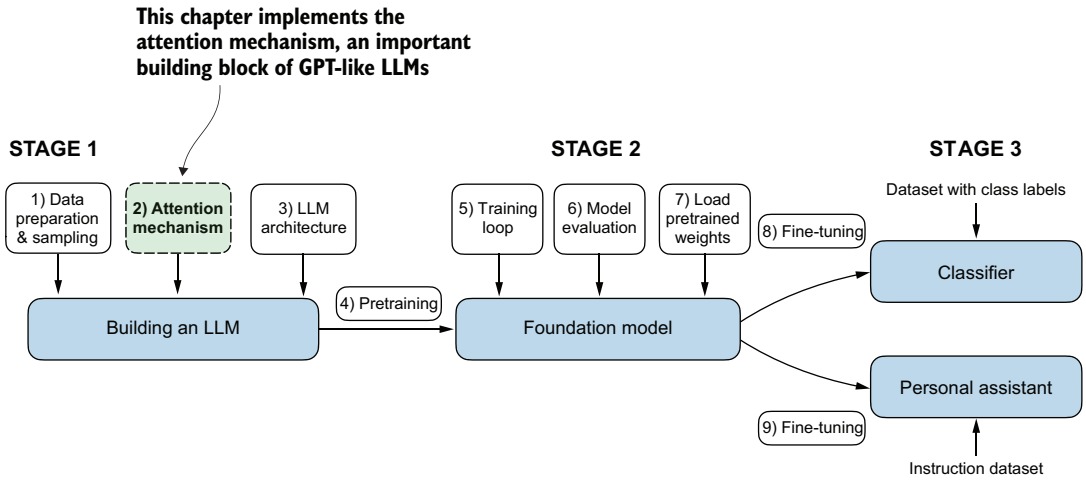



**Figure 3.1 The three main stages of coding an LLM. This chapter focuses on step 2 of stage 1: implementing attention mechanisms, which are an integral part of the LLM architecture.**

parts of the LLM surrounding the self-attention mechanism to see it in action and to create a model to generate text.

We will implement four different variants of attention mechanisms, as illustrated in figure 3.2. These different attention variants build on each other, and the goal is to

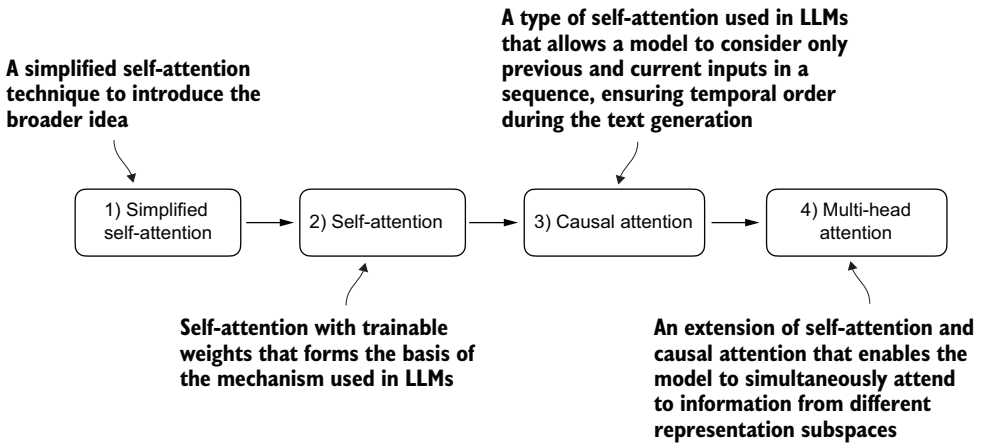



**Figure 3.2 The figure depicts different attention mechanisms we will code in this chapter, starting with a simplified version of self-attention before adding the trainable weights. The causal attention mechanism adds a mask to self-attention that allows the LLM to generate one word at a time. Finally, multi-head attention organizes the attention mechanism into multiple heads, allowing the model to capture various aspects of the input data in parallel.**

arrive at a compact and efficient implementation of multi-head attention that we can then plug into the LLM architecture we will code in the next chapter.

## 3.1 The problem with modeling long sequences

Before we dive into the *self-attention* mechanism at the heart of LLMs, let's consider the problem with pre-LLM architectures that do not include attention mechanisms. Suppose we want to develop a language translation model that translates text from one language into another. As shown in figure 3.3, we can't simply translate a text word by word due to the grammatical structures in the source and target language.

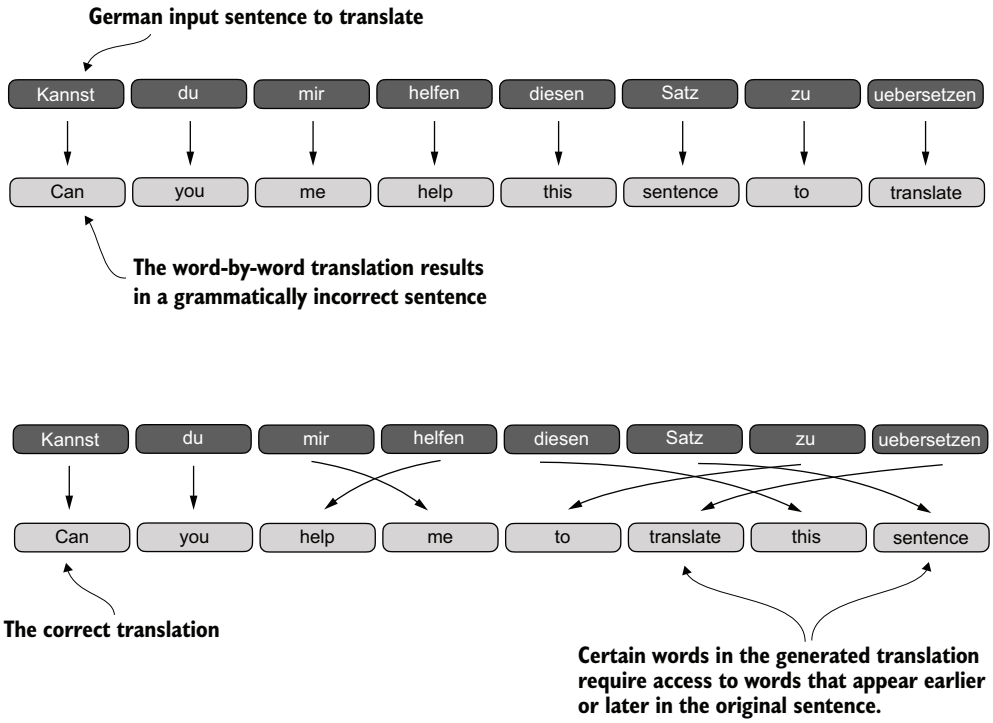



**Figure 3.3 When translating text from one language to another, such as German to English, it's not possible to merely translate word by word. Instead, the translation process requires contextual understanding and grammatical alignment.**

To address this problem, it is common to use a deep neural network with two submodules, an *encoder* and a *decoder*. The job of the encoder is to first read in and process the entire text, and the decoder then produces the translated text.

Before the advent of transformers, *recurrent neural networks* (RNNs) were the most popular encoder–decoder architecture for language translation. An RNN is a type of neural network where outputs from previous steps are fed as inputs to the current

step, making them well-suited for sequential data like text. If you are unfamiliar with RNNs, don't worry—you don't need to know the detailed workings of RNNs to follow this discussion; our focus here is more on the general concept of the encoder–decoder setup.

In an encoder–decoder RNN, the input text is fed into the encoder, which processes it sequentially. The encoder updates its hidden state (the internal values at the hidden layers) at each step, trying to capture the entire meaning of the input sentence in the final hidden state, as illustrated in figure 3.4. The decoder then takes this final hidden state to start generating the translated sentence, one word at a time. It also updates its hidden state at each step, which is supposed to carry the context necessary for the next-word prediction.

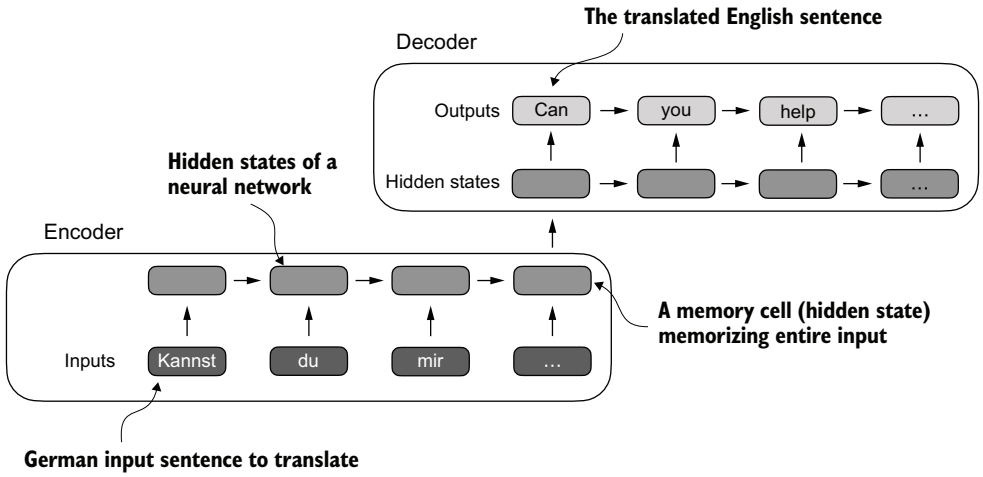



**Figure 3.4 Before the advent of transformer models, encoder–decoder RNNs were a popular choice for machine translation. The encoder takes a sequence of tokens from the source language as input, where a hidden state (an intermediate neural network layer) of the encoder encodes a compressed representation of the entire input sequence. Then, the decoder uses its current hidden state to begin the translation, token by token.**

While we don't need to know the inner workings of these encoder–decoder RNNs, the key idea here is that the encoder part processes the entire input text into a hidden state (memory cell). The decoder then takes in this hidden state to produce the output. You can think of this hidden state as an embedding vector, a concept we discussed in chapter 2.

The big limitation of encoder–decoder RNNs is that the RNN can't directly access earlier hidden states from the encoder during the decoding phase. Consequently, it relies solely on the current hidden state, which encapsulates all relevant information. This can lead to a loss of context, especially in complex sentences where dependencies might span long distances.

Fortunately, it is not essential to understand RNNs to build an LLM. Just remember that encoder–decoder RNNs had a shortcoming that motivated the design of attention mechanisms.

## 3.2 Capturing data dependencies with attention mechanisms

Although RNNs work fine for translating short sentences, they don't work well for longer texts as they don't have direct access to previous words in the input. One major shortcoming in this approach is that the RNN must remember the entire encoded input in a single hidden state before passing it to the decoder (figure 3.4).

Hence, researchers developed the *Bahdanau attention* mechanism for RNNs in 2014 (named after the first author of the respective paper; for more information, see appendix B), which modifies the encoder–decoder RNN such that the decoder can selectively access different parts of the input sequence at each decoding step as illustrated in figure 3.5.

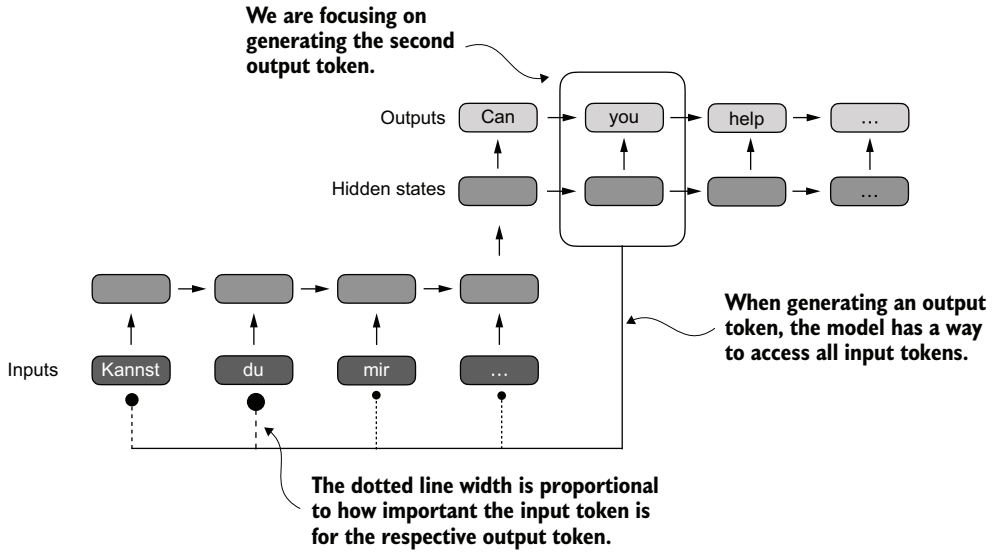



**Figure 3.5 Using an attention mechanism, the text-generating decoder part of the network can access all input tokens selectively. This means that some input tokens are more important than others for generating a given output token. The importance is determined by the attention weights, which we will compute later. Note that this figure shows the general idea behind attention and does not depict the exact implementation of the Bahdanau mechanism, which is an RNN method outside this book's scope.**

Interestingly, only three years later, researchers found that RNN architectures are not required for building deep neural networks for natural language processing and

proposed the original *transformer* architecture (discussed in chapter 1) including a self-attention mechanism inspired by the Bahdanau attention mechanism.

Self-attention is a mechanism that allows each position in the input sequence to consider the relevancy of, or "attend to," all other positions in the same sequence when computing the representation of a sequence. Self-attention is a key component of contemporary LLMs based on the transformer architecture, such as the GPT series.

This chapter focuses on coding and understanding this self-attention mechanism used in GPT-like models, as illustrated in figure 3.6. In the next chapter, we will code the remaining parts of the LLM.

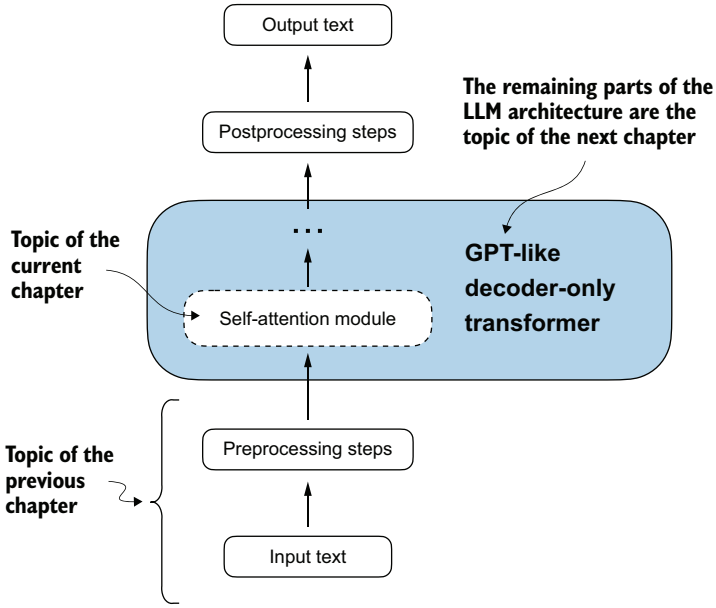



**Figure 3.6 Self-attention is a mechanism in transformers used to compute more efficient input representations by allowing each position in a sequence to interact with and weigh the importance of all other positions within the same sequence. In this chapter, we will code this self-attention mechanism from the ground up before we code the remaining parts of the GPT-like LLM in the following chapter.**

## 3.3 Attending to different parts of the input with self-attention

We'll now cover the inner workings of the self-attention mechanism and learn how to code it from the ground up. Self-attention serves as the cornerstone of every LLM based on the transformer architecture. This topic may require a lot of focus and attention (no pun intended), but once you grasp its fundamentals, you will have conquered one of the toughest aspects of this book and LLM implementation in general.

### The "self" in self-attention

In self-attention, the "self" refers to the mechanism's ability to compute attention weights by relating different positions within a single input sequence. It assesses and learns the relationships and dependencies between various parts of the input itself, such as words in a sentence or pixels in an image.

This is in contrast to traditional attention mechanisms, where the focus is on the relationships between elements of two different sequences, such as in sequence-to-sequence models where the attention might be between an input sequence and an output sequence, such as the example depicted in figure 3.5.

Since self-attention can appear complex, especially if you are encountering it for the first time, we will begin by examining a simplified version of it. Then we will implement the self-attention mechanism with trainable weights used in LLMs.

### 3.3.1 A simple self-attention mechanism without trainable weights

Let's begin by implementing a simplified variant of self-attention, free from any trainable weights, as summarized in figure 3.7. The goal is to illustrate a few key concepts in self-attention before adding trainable weights.

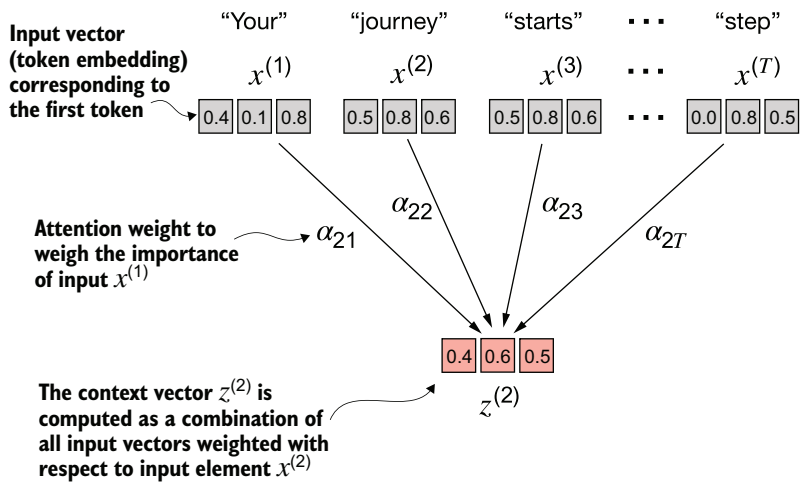



**Figure 3.7 The goal of self-attention is to compute a context vector for each input element that combines information from all other input elements. In this example, we compute the context vector $z^{(2)}$. The importance or contribution of each input element for computing $z^{(2)}$ is determined by the attention weights $\alpha_{21}$ to $\alpha_{2T}$. When computing $z^{(2)}$, the attention weights are calculated with respect to input element $x^{(2)}$ and all other inputs.**

Figure 3.7 shows an input sequence, denoted as $x$, consisting of $T$ elements represented as $x^{(1)}$ to $x^{(T)}$. This sequence typically represents text, such as a sentence, that has already been transformed into token embeddings.

For example, consider an input text like "Your journey starts with one step." In this case, each element of the sequence, such as $x^{(1)}$, corresponds to a $d$-dimensional embedding vector representing a specific token, like "Your." Figure 3.7 shows these input vectors as three-dimensional embeddings.

In self-attention, our goal is to calculate context vectors $z^{(i)}$ for each element $x^{(i)}$ in the input sequence. A *context vector* can be interpreted as an enriched embedding vector.

To illustrate this concept, let's focus on the embedding vector of the second input element, $x^{(2)}$ (which corresponds to the token "journey"), and the corresponding context vector, $z^{(2)}$, shown at the bottom of figure 3.7. This enhanced context vector, $z^{(2)}$, is an embedding that contains information about $x^{(2)}$ and all other input elements, $x^{(1)}$ to $x^{(T)}$.

Context vectors play a crucial role in self-attention. Their purpose is to create enriched representations of each element in an input sequence (like a sentence) by incorporating information from all other elements in the sequence (figure 3.7). This is essential in LLMs, which need to understand the relationship and relevance of words in a sentence to each other. Later, we will add trainable weights that help an LLM learn to construct these context vectors so that they are relevant for the LLM to generate the next token. But first, let's implement a simplified self-attention mechanism to compute these weights and the resulting context vector one step at a time.

Consider the following input sentence, which has already been embedded into three-dimensional vectors (see chapter 2). I've chosen a small embedding dimension to ensure it fits on the page without line breaks:

```
import torch
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
   [0.55, 0.87, 0.66], # journey  (x^2)
   [0.57, 0.85, 0.64], # starts   (x^3)
   [0.22, 0.58, 0.33], # with     (x^4)
   [0.77, 0.25, 0.10], # one      (x^5)
   [0.05, 0.80, 0.55]] # step     (x^6)
)
```

The first step of implementing self-attention is to compute the intermediate values $\omega$, referred to as attention scores, as illustrated in figure 3.8. Due to spatial constraints, the figure displays the values of the preceding `inputs` tensor in a truncated version; for example, 0.87 is truncated to 0.8. In this truncated version, the embeddings of the words "journey" and "starts" may appear similar by random chance.

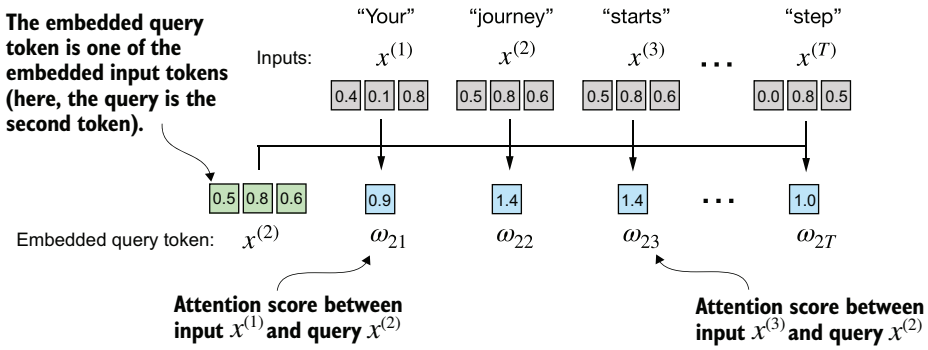



**Figure 3.8 The overall goal is to illustrate the computation of the context vector $z^{(2)}$ using the second input element, $x^{(2)}$ as a query. This figure shows the first intermediate step, computing the attention scores $\omega$ between the query $x^{(2)}$ and all other input elements as a dot product. (Note that the numbers are truncated to one digit after the decimal point to reduce visual clutter.)**

Figure 3.8 illustrates how we calculate the intermediate attention scores between the query token and each input token. We determine these scores by computing the dot product of the query, $x^{(2)}$, with every other input token:

```
query = inputs[1]
attn_scores_2 = torch.empty(inputs.shape[0])
for i, x_i in enumerate(inputs):
    attn_scores_2[i] = torch.dot(x_i, query)
print(attn_scores_2)
```

**The second input token serves as the query.**

The computed attention scores are

```
tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
```

### Understanding dot products

A dot product is essentially a concise way of multiplying two vectors element-wise and then summing the products, which can be demonstrated as follows:

```
res = 0.
for idx, element in enumerate(inputs[0]):
    res += inputs[0][idx] * query[idx]
print(res)
print(torch.dot(inputs[0], query))
```

The output confirms that the sum of the element-wise multiplication gives the same results as the dot product:

```
tensor(0.9544)
tensor(0.9544)
```

Beyond viewing the dot product operation as a mathematical tool that combines two vectors to yield a scalar value, the dot product is a measure of similarity because it quantifies how closely two vectors are aligned: a higher dot product indicates a greater degree of alignment or similarity between the vectors. In the context of self-attention mechanisms, the dot product determines the extent to which each element in a sequence focuses on, or "attends to," any other element: the higher the dot product, the higher the similarity and attention score between two elements.

In the next step, as shown in figure 3.9, we normalize each of the attention scores we computed previously. The main goal behind the normalization is to obtain attention weights that sum up to 1. This normalization is a convention that is useful for interpretation and maintaining training stability in an LLM. Here's a straightforward method for achieving this normalization step:

```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum()
print("Attention weights:", attn_weights_2_tmp)
print("Sum:", attn_weights_2_tmp.sum())
```

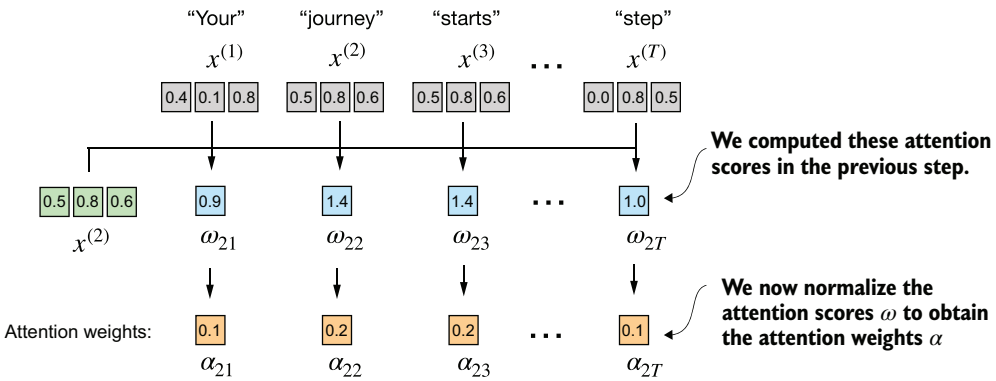



**Figure 3.9 After computing the attention scores $\omega_{21}$ to $\omega_{2T}$ with respect to the input query $x^{(2)}$, the next step is to obtain the attention weights $\alpha_{21}$ to $\alpha_{2T}$ by normalizing the attention scores.**

As the output shows, the attention weights now sum to 1:

```
Attention weights: tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656])
Sum: tensor(1.0000)
```

In practice, it's more common and advisable to use the softmax function for normalization. This approach is better at managing extreme values and offers more favorable

gradient properties during training. The following is a basic implementation of the softmax function for normalizing the attention scores:

```
def softmax_naive(x):
    return torch.exp(x) / torch.exp(x).sum(dim=0)

attn_weights_2_naive = softmax_naive(attn_scores_2)
print("Attention weights:", attn_weights_2_naive)
print("Sum:", attn_weights_2_naive.sum())
```

As the output shows, the softmax function also meets the objective and normalizes the attention weights such that they sum to 1:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum: tensor(1.)
```

In addition, the softmax function ensures that the attention weights are always positive. This makes the output interpretable as probabilities or relative importance, where higher weights indicate greater importance.

Note that this naive softmax implementation (`softmax_naive`) may encounter numerical instability problems, such as overflow and underflow, when dealing with large or small input values. Therefore, in practice, it's advisable to use the PyTorch implementation of softmax, which has been extensively optimized for performance:

```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0)
print("Attention weights:", attn_weights_2)
print("Sum:", attn_weights_2.sum())
```

In this case, it yields the same results as our previous `softmax_naive` function:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum: tensor(1.)
```

Now that we have computed the normalized attention weights, we are ready for the final step, as shown in figure 3.10: calculating the context vector $z^{(2)}$ by multiplying the embedded input tokens, $x^{(i)}$, with the corresponding attention weights and then summing the resulting vectors. Thus, context vector $z^{(2)}$ is the weighted sum of all input vectors, obtained by multiplying each input vector by its corresponding attention weight:

```
query = inputs[1]                          <-- The second input token is the query.
context_vec_2 = torch.zeros(query.shape)
for i,x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
```

The results of this computation are

```
tensor([0.4419, 0.6515, 0.5683])
```

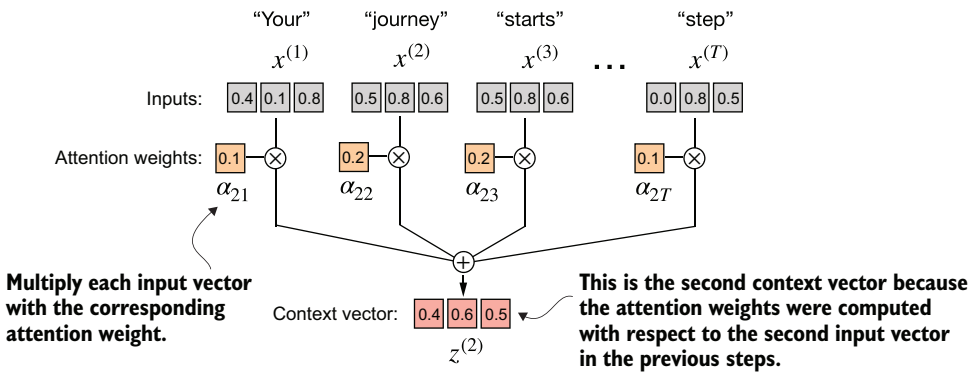



**Figure 3.10 The final step, after calculating and normalizing the attention scores to obtain the attention weights for query $x^{(2)}$, is to compute the context vector $z^{(2)}$. This context vector is a combination of all input vectors $x^{(1)}$ to $x^{(T)}$ weighted by the attention weights.**

Next, we will generalize this procedure for computing context vectors to calculate all context vectors simultaneously.

### 3.3.2 Computing attention weights for all input tokens

So far, we have computed attention weights and the context vector for input 2, as shown in the highlighted row in figure 3.11. Now let's extend this computation to calculate attention weights and context vectors for all inputs.

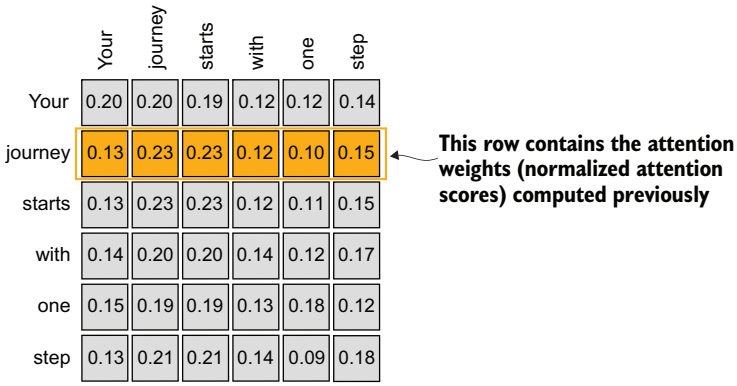



**Figure 3.11 The highlighted row shows the attention weights for the second input element as a query. Now we will generalize the computation to obtain all other attention weights. (Please note that the numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter. The values in each row should add up to 1.0 or 100%.)**

We follow the same three steps as before (see figure 3.12), except that we make a few modifications in the code to compute all context vectors instead of only the second one, $z^{(2)}$:

```
attn_scores = torch.empty(6, 6)
for i, x_i in enumerate(inputs):
    for j, x_j in enumerate(inputs):
        attn_scores[i, j] = torch.dot(x_i, x_j)
print(attn_scores)
```

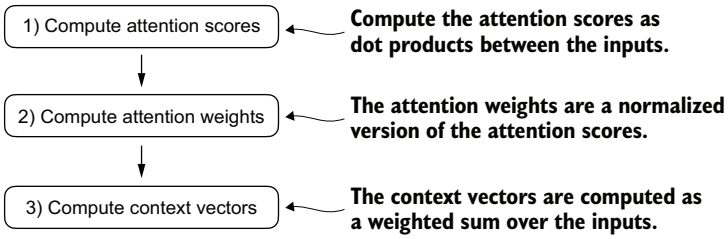



**Figure 3.12 In step 1, we add an additional `for` loop to compute the dot products for all pairs of inputs.**

The resulting attention scores are as follows:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
        [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
        [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
        [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
        [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
        [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

Each element in the tensor represents an attention score between each pair of inputs, as we saw in figure 3.11. Note that the values in that figure are normalized, which is why they differ from the unnormalized attention scores in the preceding tensor. We will take care of the normalization later.

When computing the preceding attention score tensor, we used `for` loops in Python. However, `for` loops are generally slow, and we can achieve the same results using matrix multiplication:

```
attn_scores = inputs @ inputs.T
print(attn_scores)
```

We can visually confirm that the results are the same as before:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
        [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
        [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
        [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
```

```
        [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
        [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

In step 2 of figure 3.12, we normalize each row so that the values in each row sum to 1:

```
attn_weights = torch.softmax(attn_scores, dim=-1)
print(attn_weights)
```

This returns the following attention weight tensor that matches the values shown in figure 3.10:

```
tensor([[0.2098, 0.2006, 0.1981, 0.1242, 0.1220, 0.1452],
        [0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581],
        [0.1390, 0.2369, 0.2326, 0.1242, 0.1108, 0.1565],
        [0.1435, 0.2074, 0.2046, 0.1462, 0.1263, 0.1720],
        [0.1526, 0.1958, 0.1975, 0.1367, 0.1879, 0.1295],
        [0.1385, 0.2184, 0.2128, 0.1420, 0.0988, 0.1896]])
```

In the context of using PyTorch, the dim parameter in functions like `torch.softmax` specifies the dimension of the input tensor along which the function will be computed. By setting `dim=-1`, we are instructing the `softmax` function to apply the normalization along the last dimension of the `attn_scores` tensor. If `attn_scores` is a two-dimensional tensor (for example, with a shape of [rows, columns]), it will normalize across the columns so that the values in each row (summing over the column dimension) sum up to 1.

We can verify that the rows indeed all sum to 1:

```
row_2_sum = sum([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
print("Row 2 sum:", row_2_sum)
print("All row sums:", attn_weights.sum(dim=-1))
```

The result is

```
Row 2 sum: 1.0
All row sums: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

In the third and final step of figure 3.12, we use these attention weights to compute all context vectors via matrix multiplication:

```
all_context_vecs = attn_weights @ inputs
print(all_context_vecs)
```

In the resulting output tensor, each row contains a three-dimensional context vector:

```
tensor([[0.4421, 0.5931, 0.5790],
        [0.4419, 0.6515, 0.5683],
        [0.4431, 0.6496, 0.5671],
        [0.4304, 0.6298, 0.5510],
        [0.4671, 0.5910, 0.5266],
        [0.4177, 0.6503, 0.5645]])
```

We can double-check that the code is correct by comparing the second row with the context vector $z^{(2)}$ that we computed in section 3.3.1:

```
print("Previous 2nd context vector:", context_vec_2)
```

Based on the result, we can see that the previously calculated `context_vec_2` matches the second row in the previous tensor exactly:

```
Previous 2nd context vector: tensor([0.4419, 0.6515, 0.5683])
```

This concludes the code walkthrough of a simple self-attention mechanism. Next, we will add trainable weights, enabling the LLM to learn from data and improve its performance on specific tasks.

## 3.4 Implementing self-attention with trainable weights

Our next step will be to implement the self-attention mechanism used in the original transformer architecture, the GPT models, and most other popular LLMs. This self-attention mechanism is also called *scaled dot-product attention*. Figure 3.13 shows how this self-attention mechanism fits into the broader context of implementing an LLM.

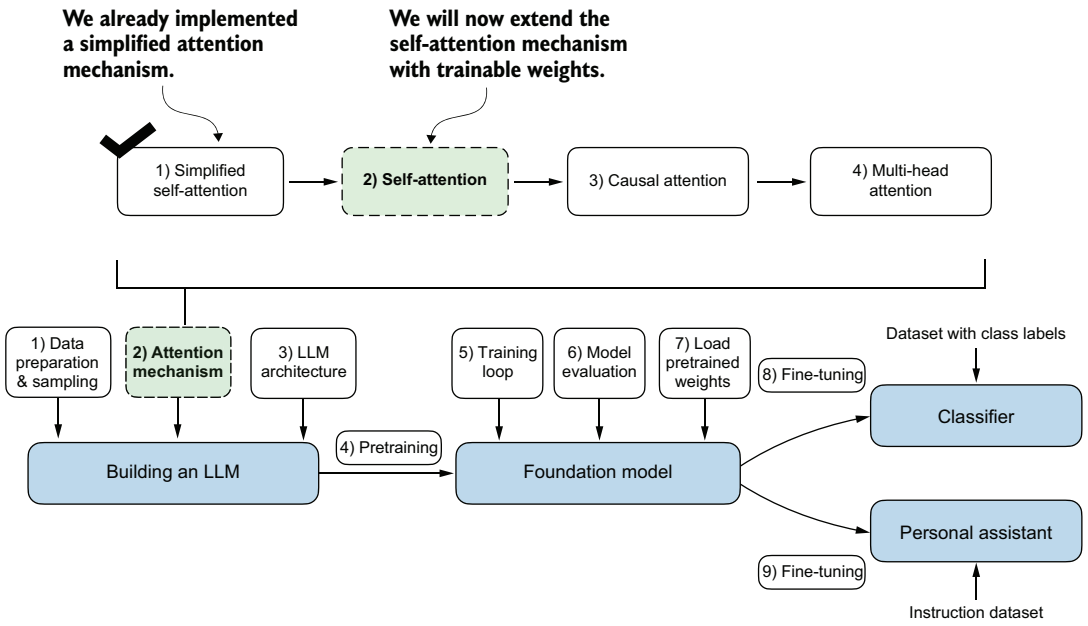



**Figure 3.13 Previously, we coded a simplified attention mechanism to understand the basic mechanism behind attention mechanisms. Now, we add trainable weights to this attention mechanism. Later, we will extend this self-attention mechanism by adding a causal mask and multiple heads.**

As illustrated in figure 3.13, the self-attention mechanism with trainable weights builds on the previous concepts: we want to compute context vectors as weighted sums over the input vectors specific to a certain input element. As you will see, there are only slight differences compared to the basic self-attention mechanism we coded earlier.

The most notable difference is the introduction of weight matrices that are updated during model training. These trainable weight matrices are crucial so that the model (specifically, the attention module inside the model) can learn to produce "good" context vectors. (We will train the LLM in chapter 5.)

We will tackle this self-attention mechanism in the two subsections. First, we will code it step by step as before. Second, we will organize the code into a compact Python class that can be imported into the LLM architecture.

### 3.4.1 Computing the attention weights step by step

We will implement the self-attention mechanism step by step by introducing the three trainable weight matrices $W_q$, $W_k$, and $W_v$. These three matrices are used to project the embedded input tokens, $x^{(i)}$, into query, key, and value vectors, respectively, as illustrated in figure 3.14.

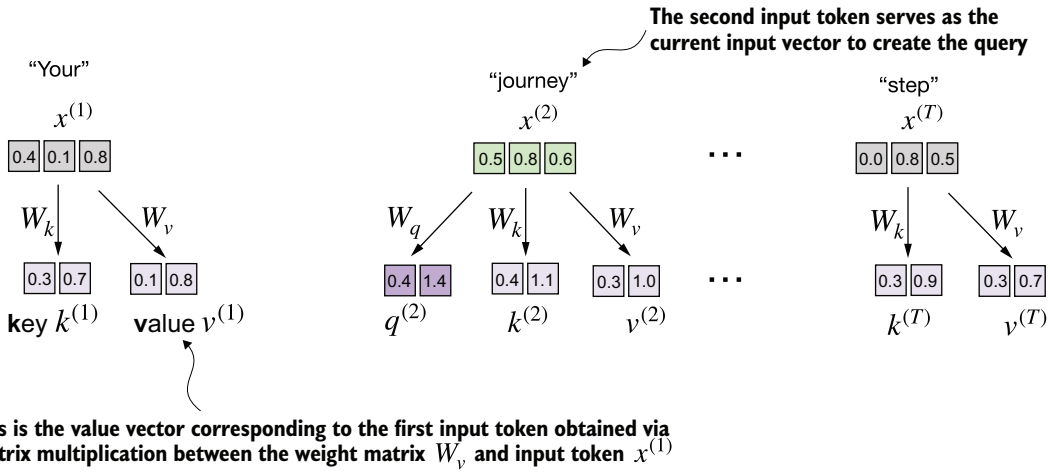
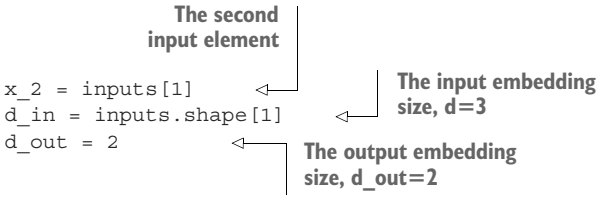



**Figure 3.14 In the first step of the self-attention mechanism with trainable weight matrices, we compute query (*q*), key (*k*), and value (*v*) vectors for input elements *x*. Similar to previous sections, we designate the second input, $x^{(2)}$, as the query input. The query vector $q^{(2)}$ is obtained via matrix multiplication between the input $x^{(2)}$ and the weight matrix $W_q$. Similarly, we obtain the key and value vectors via matrix multiplication involving the weight matrices $W_k$ and $W_v$.**

Earlier, we defined the second input element $x^{(2)}$ as the query when we computed the simplified attention weights to compute the context vector $z^{(2)}$. Then we generalized this to compute all context vectors $z^{(1)}$ ... $z^{(T)}$ for the six-word input sentence "Your journey starts with one step."

Similarly, we start here by computing only one context vector, $z^{(2)}$, for illustration purposes. We will then modify this code to calculate all context vectors.

Let's begin by defining a few variables:




Note that in GPT-like models, the input and output dimensions are usually the same, but to better follow the computation, we'll use different input (`d_in=3`) and output (`d_out=2`) dimensions here.

Next, we initialize the three weight matrices $W_q$, $W_k$, and $W_v$ shown in figure 3.14:

```
torch.manual_seed(123)
W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_key   = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
```

We set `requires_grad=False` to reduce clutter in the outputs, but if we were to use the weight matrices for model training, we would set `requires_grad=True` to update these matrices during model training.

Next, we compute the query, key, and value vectors:

```
query_2 = x_2 @ W_query
key_2 = x_2 @ W_key
value_2 = x_2 @ W_value
print(query_2)
```

The output for the query results in a two-dimensional vector since we set the number of columns of the corresponding weight matrix, via `d_out`, to 2:

```
tensor([0.4306, 1.4551])
```

### Weight parameters vs. attention weights

In the weight matrices *W*, the term "weight" is short for "weight parameters," the values of a neural network that are optimized during training. This is not to be confused with the attention weights. As we already saw, attention weights determine the extent to which a context vector depends on the different parts of the input (i.e., to what extent the network focuses on different parts of the input).

In summary, weight parameters are the fundamental, learned coefficients that define the network's connections, while attention weights are dynamic, context-specific values.

Even though our temporary goal is only to compute the one context vector, $z^{(2)}$, we still require the key and value vectors for all input elements as they are involved in computing the attention weights with respect to the query $q^{(2)}$ (see figure 3.14).

We can obtain all keys and values via matrix multiplication:

```
keys = inputs @ W_key
values = inputs @ W_value
print("keys.shape:", keys.shape)
print("values.shape:", values.shape)
```

As we can tell from the outputs, we successfully projected the six input tokens from a three-dimensional onto a two-dimensional embedding space:

```
keys.shape: torch.Size([6, 2])
values.shape: torch.Size([6, 2])
```

The second step is to compute the attention scores, as shown in figure 3.15.

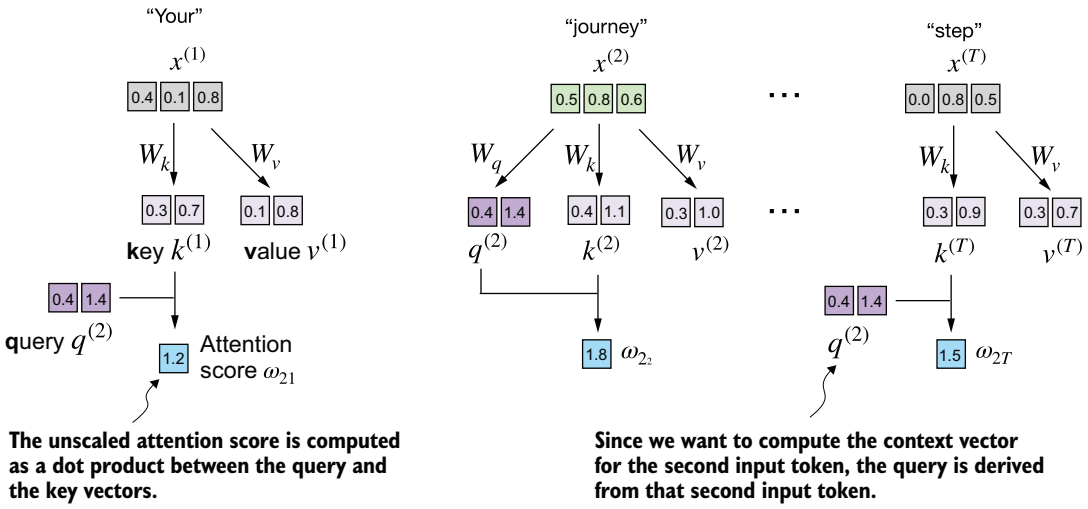



**Figure 3.15 The attention score computation is a dot-product computation similar to what we used in the simplified self-attention mechanism in section 3.3. The new aspect here is that we are not directly computing the dot-product between the input elements but using the query and key obtained by transforming the inputs via the respective weight matrices.**

First, let's compute the attention score $\omega_{22}$:

```
keys_2 = keys[1]
attn_score_22 = query_2.dot(keys_2)
print(attn_score_22)
```

Remember that Python starts indexing at 0.

The result for the unnormalized attention score is

```
tensor(1.8524)
```

Again, we can generalize this computation to all attention scores via matrix multiplication:

```
attn_scores_2 = query_2 @ keys.T
print(attn_scores_2)
```

**All attention scores for given query**

As we can see, as a quick check, the second element in the output matches the `attn_score_22` we computed previously:

```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```

Now, we want to go from the attention scores to the attention weights, as illustrated in figure 3.16. We compute the attention weights by scaling the attention scores and using the softmax function. However, now we scale the attention scores by dividing them by the square root of the embedding dimension of the keys (taking the square root is mathematically the same as exponentiating by 0.5):

```
d_k = keys.shape[-1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
```

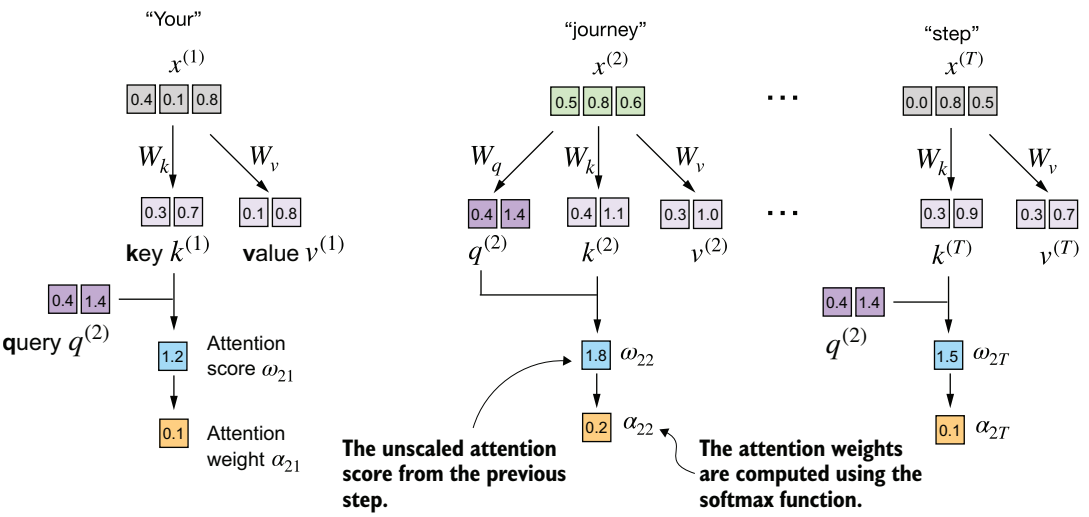



**Figure 3.16 After computing the attention scores ω, the next step is to normalize these scores using the softmax function to obtain the attention weights α.**

The resulting attention weights are

```
tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

### The rationale behind scaled-dot product attention

The reason for the normalization by the embedding dimension size is to improve the training performance by avoiding small gradients. For instance, when scaling up the embedding dimension, which is typically greater than 1,000 for GPT-like LLMs, large dot products can result in very small gradients during backpropagation due to the softmax function applied to them. As dot products increase, the softmax function behaves more like a step function, resulting in gradients nearing zero. These small gradients can drastically slow down learning or cause training to stagnate.

The scaling by the square root of the embedding dimension is the reason why this self-attention mechanism is also called scaled-dot product attention.

Now, the final step is to compute the context vectors, as illustrated in figure 3.17.

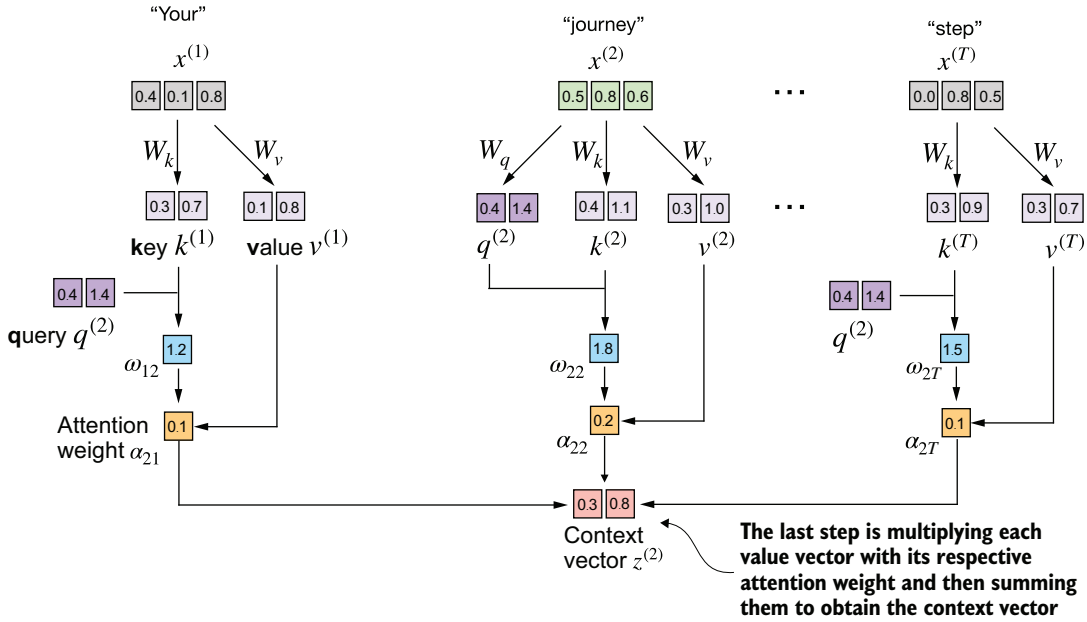



**Figure 3.17 In the final step of the self-attention computation, we compute the context vector by combining all value vectors via the attention weights.**

Similar to when we computed the context vector as a weighted sum over the input vectors (see section 3.3), we now compute the context vector as a weighted sum over the value vectors. Here, the attention weights serve as a weighting factor that weighs

the respective importance of each value vector. Also as before, we can use matrix multiplication to obtain the output in one step:

```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```

The contents of the resulting vector are as follows:

```
tensor([0.3061, 0.8210])
```

So far, we've only computed a single context vector, $z^{(2)}$. Next, we will generalize the code to compute all context vectors in the input sequence, $z^{(1)}$ to $z^{(T)}$.

**Why query, key, and value?**

The terms "key," "query," and "value" in the context of attention mechanisms are borrowed from the domain of information retrieval and databases, where similar concepts are used to store, search, and retrieve information.

A *query* is analogous to a search query in a database. It represents the current item (e.g., a word or token in a sentence) the model focuses on or tries to understand. The query is used to probe the other parts of the input sequence to determine how much attention to pay to them.

The *key* is like a database key used for indexing and searching. In the attention mechanism, each item in the input sequence (e.g., each word in a sentence) has an associated key. These keys are used to match the query.

The *value* in this context is similar to the value in a key-value pair in a database. It represents the actual content or representation of the input items. Once the model determines which keys (and thus which parts of the input) are most relevant to the query (the current focus item), it retrieves the corresponding values.

### 3.4.2 Implementing a compact self-attention Python class

At this point, we have gone through a lot of steps to compute the self-attention outputs. We did so mainly for illustration purposes so we could go through one step at a time. In practice, with the LLM implementation in the next chapter in mind, it is helpful to organize this code into a Python class, as shown in the following listing.

**Listing 3.1 A compact self-attention class**

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key   = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))
```

```
    def forward(self, x):
        keys = x @ self.W_key
        queries = x @ self.W_query
        values = x @ self.W_value
        attn_scores = queries @ keys.T # omega
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        context_vec = attn_weights @ values
        return context_vec
```

In this PyTorch code, `SelfAttention_v1` is a class derived from `nn.Module`, which is a fundamental building block of PyTorch models that provides necessary functionalities for model layer creation and management.

The `__init__` method initializes trainable weight matrices (`W_query`, `W_key`, and `W_value`) for queries, keys, and values, each transforming the input dimension `d_in` to an output dimension `d_out`.

During the forward pass, using the forward method, we compute the attention scores (`attn_scores`) by multiplying queries and keys, normalizing these scores using softmax. Finally, we create a context vector by weighting the values with these normalized attention scores.

We can use this class as follows:

```
torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))
```

Since `inputs` contains six embedding vectors, this results in a matrix storing the six context vectors:

```
tensor([[0.2996, 0.8053],
        [0.3061, 0.8210],
        [0.3058, 0.8203],
        [0.2948, 0.7939],
        [0.2927, 0.7891],
        [0.2990, 0.8040]], grad_fn=<MmBackward0>)
```

As a quick check, notice that the second row (`[0.3061, 0.8210]`) matches the contents of `context_vec_2` in the previous section. Figure 3.18 summarizes the self-attention mechanism we just implemented.

Self-attention involves the trainable weight matrices $W_q$, $W_k$, and $W_v$. These matrices transform input data into queries, keys, and values, respectively, which are crucial components of the attention mechanism. As the model is exposed to more data during training, it adjusts these trainable weights, as we will see in upcoming chapters.

We can improve the `SelfAttention_v1` implementation further by utilizing PyTorch's `nn.Linear` layers, which effectively perform matrix multiplication when the bias units are disabled. Additionally, a significant advantage of using `nn.Linear`

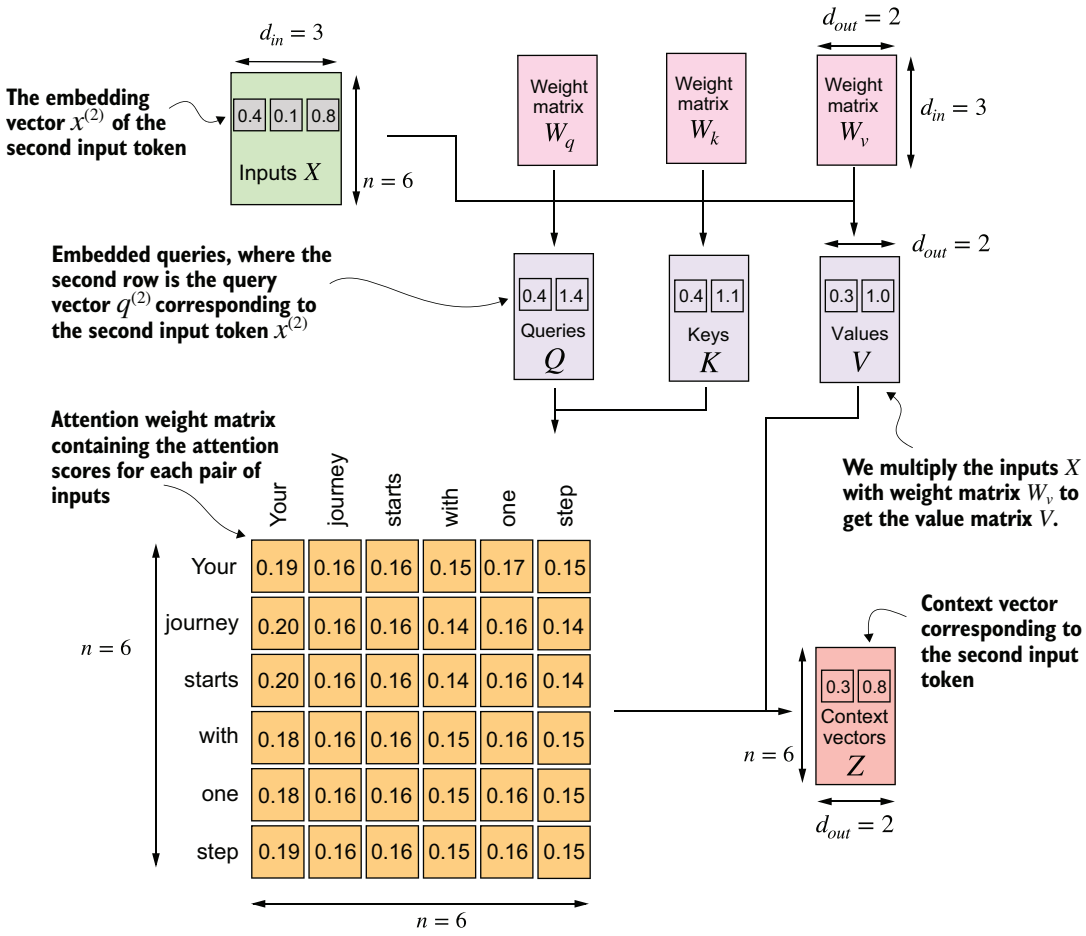



**Figure 3.18** In self-attention, we transform the input vectors in the input matrix X with the three weight matrices, $W_q$, $W_k$, and $W_v$. Then we compute the attention weight matrix based on the resulting queries (Q) and keys (K). Using the attention weights and values (V), we then compute the context vectors (Z). For visual clarity, we focus on a single input text with *n* tokens, not a batch of multiple inputs. Consequently, the three-dimensional input tensor is simplified to a two-dimensional matrix in this context. This approach allows for a more straightforward visualization and understanding of the processes involved. For consistency with later figures, the values in the attention matrix do not depict the real attention weights. (The numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter. The values in each row should add up to 1.0 or 100%.)

instead of manually implementing `nn.Parameter(torch.rand(...))` is that `nn.Linear` has an optimized weight initialization scheme, contributing to more stable and effective model training.

**Listing 3.2 A self-attention class using PyTorch's Linear layers**

```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
```

```
        super().__init__()
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        context_vec = attn_weights @ values
        return context_vec
```

You can use the `SelfAttention_v2` similar to `SelfAttention_v1`:

```
torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
```

The output is

```
tensor([[-0.0739,  0.0713],
        [-0.0748,  0.0703],
        [-0.0749,  0.0702],
        [-0.0760,  0.0685],
        [-0.0763,  0.0679],
        [-0.0754,  0.0693]], grad_fn=<MmBackward0>)
```

Note that `SelfAttention_v1` and `SelfAttention_v2` give different outputs because they use different initial weights for the weight matrices since `nn.Linear` uses a more sophisticated weight initialization scheme.

### Exercise 3.1 Comparing SelfAttention_v1 and SelfAttention_v2

Note that `nn.Linear` in `SelfAttention_v2` uses a different weight initialization scheme as `nn.Parameter(torch.rand(d_in, d_out))` used in `SelfAttention_v1`, which causes both mechanisms to produce different results. To check that both implementations, `SelfAttention_v1` and `SelfAttention_v2`, are otherwise similar, we can transfer the weight matrices from a `SelfAttention_v2` object to a `SelfAttention_v1`, such that both objects then produce the same results.

Your task is to correctly assign the weights from an instance of `SelfAttention_v2` to an instance of `SelfAttention_v1`. To do this, you need to understand the relationship between the weights in both versions. (Hint: `nn.Linear` stores the weight matrix in a transposed form.) After the assignment, you should observe that both instances produce the same outputs.

Next, we will make enhancements to the self-attention mechanism, focusing specifically on incorporating causal and multi-head elements. The causal aspect involves modifying the attention mechanism to prevent the model from accessing future information in the sequence, which is crucial for tasks like language modeling, where each word prediction should only depend on previous words.

The multi-head component involves splitting the attention mechanism into multiple "heads." Each head learns different aspects of the data, allowing the model to simultaneously attend to information from different representation subspaces at different positions. This improves the model's performance in complex tasks.

## 3.5 Hiding future words with causal attention

For many LLM tasks, you will want the self-attention mechanism to consider only the tokens that appear prior to the current position when predicting the next token in a sequence. Causal attention, also known as *masked attention*, is a specialized form of self-attention. It restricts a model to only consider previous and current inputs in a sequence when processing any given token when computing attention scores. This is in contrast to the standard self-attention mechanism, which allows access to the entire input sequence at once.

Now, we will modify the standard self-attention mechanism to create a *causal attention* mechanism, which is essential for developing an LLM in the subsequent chapters. To achieve this in GPT-like LLMs, for each token processed, we mask out the future tokens, which come after the current token in the input text, as illustrated in figure 3.19. We mask out the attention weights above the diagonal, and we

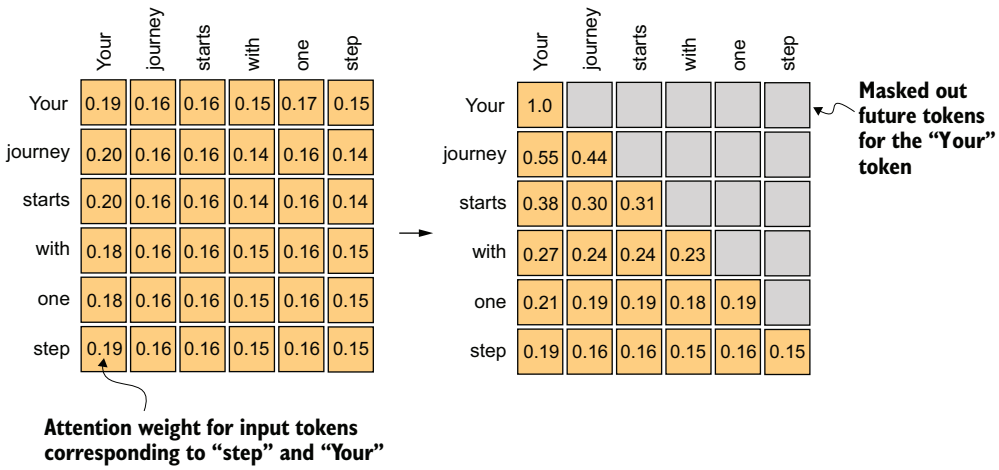



**Figure 3.19 In causal attention, we mask out the attention weights above the diagonal such that for a given input, the LLM can't access future tokens when computing the context vectors using the attention weights. For example, for the word "journey" in the second row, we only keep the attention weights for the words before ("Your") and in the current position ("journey").**

normalize the nonmasked attention weights such that the attention weights sum to 1 in each row. Later, we will implement this masking and normalization procedure in code.

### 3.5.1 Applying a causal attention mask

Our next step is to implement the causal attention mask in code. To implement the steps to apply a causal attention mask to obtain the masked attention weights, as summarized in figure 3.20, let's work with the attention scores and weights from the previous section to code the causal attention mechanism.

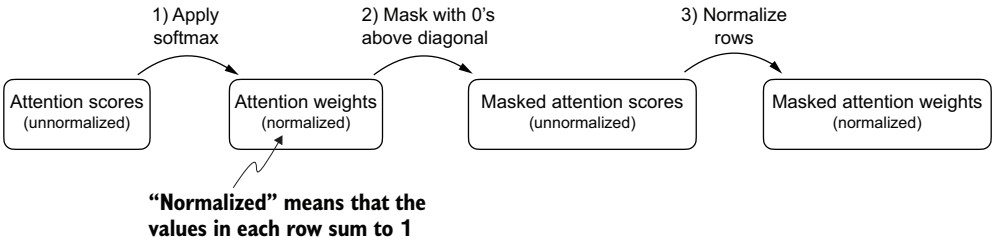



**Figure 3.20 One way to obtain the masked attention weight matrix in causal attention is to apply the softmax function to the attention scores, zeroing out the elements above the diagonal and normalizing the resulting matrix.**

In the first step, we compute the attention weights using the softmax function as we have done previously:

```
queries = sa_v2.W_query(inputs)
keys = sa_v2.W_key(inputs)
attn_scores = queries @ keys.T
attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
print(attn_weights)
```

**Reuses the query and key weight matrices of the SelfAttention_v2 object from the previous section for convenience**

This results in the following attention weights:

```
tensor([[0.1921, 0.1646, 0.1652, 0.1550, 0.1721, 0.1510],
        [0.2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477],
        [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.1661, 0.1564],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.1585],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<SoftmaxBackward0>)
```

We can implement the second step using PyTorch's `tril` function to create a mask where the values above the diagonal are zero:

```
context_length = attn_scores.shape[0]
mask_simple = torch.tril(torch.ones(context_length, context_length))
print(mask_simple)
```

The resulting mask is

```
tensor([[1., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0.],
        [1., 1., 1., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1., 1.]])
```

Now, we can multiply this mask with the attention weights to zero-out the values above the diagonal:

```
masked_simple = attn_weights*mask_simple
print(masked_simple)
```

As we can see, the elements above the diagonal are successfully zeroed out:

```
tensor([[0.1921, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2041, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2036, 0.1659, 0.1662, 0.0000, 0.0000, 0.0000],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.0000, 0.0000],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<MulBackward0>)
```

The third step is to renormalize the attention weights to sum up to 1 again in each row. We can achieve this by dividing each element in each row by the sum in each row:

```
row_sums = masked_simple.sum(dim=-1, keepdim=True)
masked_simple_norm = masked_simple / row_sums
print(masked_simple_norm)
```

The result is an attention weight matrix where the attention weights above the diagonal are zeroed-out, and the rows sum to 1:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<DivBackward0>)
```

### Information leakage

When we apply a mask and then renormalize the attention weights, it might initially appear that information from future tokens (which we intend to mask) could still influence the current token because their values are part of the softmax calculation. However, the key insight is that when we renormalize the attention weights after masking,

what we're essentially doing is recalculating the softmax over a smaller subset (since masked positions don't contribute to the softmax value).

The mathematical elegance of softmax is that despite initially including all positions in the denominator, after masking and renormalizing, the effect of the masked positions is nullified—they don't contribute to the softmax score in any meaningful way.

In simpler terms, after masking and renormalization, the distribution of attention weights is as if it was calculated only among the unmasked positions to begin with. This ensures there's no information leakage from future (or otherwise masked) tokens as we intended.

While we could wrap up our implementation of causal attention at this point, we can still improve it. Let's take a mathematical property of the softmax function and implement the computation of the masked attention weights more efficiently in fewer steps, as shown in figure 3.21.

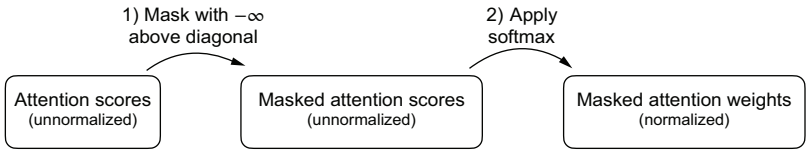



**Figure 3.21 A more efficient way to obtain the masked attention weight matrix in causal attention is to mask the attention scores with negative infinity values before applying the softmax function.**

The softmax function converts its inputs into a probability distribution. When negative infinity values (-∞) are present in a row, the softmax function treats them as zero probability. (Mathematically, this is because $e^{-\infty}$ approaches 0.)

We can implement this more efficient masking "trick" by creating a mask with 1s above the diagonal and then replacing these 1s with negative infinity (`-inf`) values:

```
mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
```

This results in the following mask:

```
tensor([[0.2899,   -inf,   -inf,   -inf,   -inf,   -inf],
        [0.4656, 0.1723,   -inf,   -inf,   -inf,   -inf],
        [0.4594, 0.1703, 0.1731,   -inf,   -inf,   -inf],
        [0.2642, 0.1024, 0.1036, 0.0186,   -inf,   -inf],
        [0.2183, 0.0874, 0.0882, 0.0177, 0.0786,   -inf],
        [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]],
       grad_fn=<MaskedFillBackward0>)
```

Now all we need to do is apply the softmax function to these masked results, and we are done:

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1)
print(attn_weights)
```

As we can see based on the output, the values in each row sum to 1, and no further normalization is necessary:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<SoftmaxBackward0>)
```

We could now use the modified attention weights to compute the context vectors via `context_vec = attn_weights @ values`, as in section 3.4. However, we will first cover another minor tweak to the causal attention mechanism that is useful for reducing overfitting when training LLMs.

### 3.5.2 Masking additional attention weights with dropout

*Dropout* in deep learning is a technique where randomly selected hidden layer units are ignored during training, effectively "dropping" them out. This method helps prevent overfitting by ensuring that a model does not become overly reliant on any specific set of hidden layer units. It's important to emphasize that dropout is only used during training and is disabled afterward.

In the transformer architecture, including models like GPT, dropout in the attention mechanism is typically applied at two specific times: after calculating the attention weights or after applying the attention weights to the value vectors. Here we will apply the dropout mask after computing the attention weights, as illustrated in figure 3.22, because it's the more common variant in practice.

In the following code example, we use a dropout rate of 50%, which means masking out half of the attention weights. (When we train the GPT model in later chapters, we will use a lower dropout rate, such as 0.1 or 0.2.) We apply PyTorch's dropout implementation first to a 6 × 6 tensor consisting of 1s for simplicity:

```
torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5)
example = torch.ones(6, 6)
print(dropout(example))
```

**We choose a dropout rate of 50%.**

**Here, we create a matrix of 1s.**

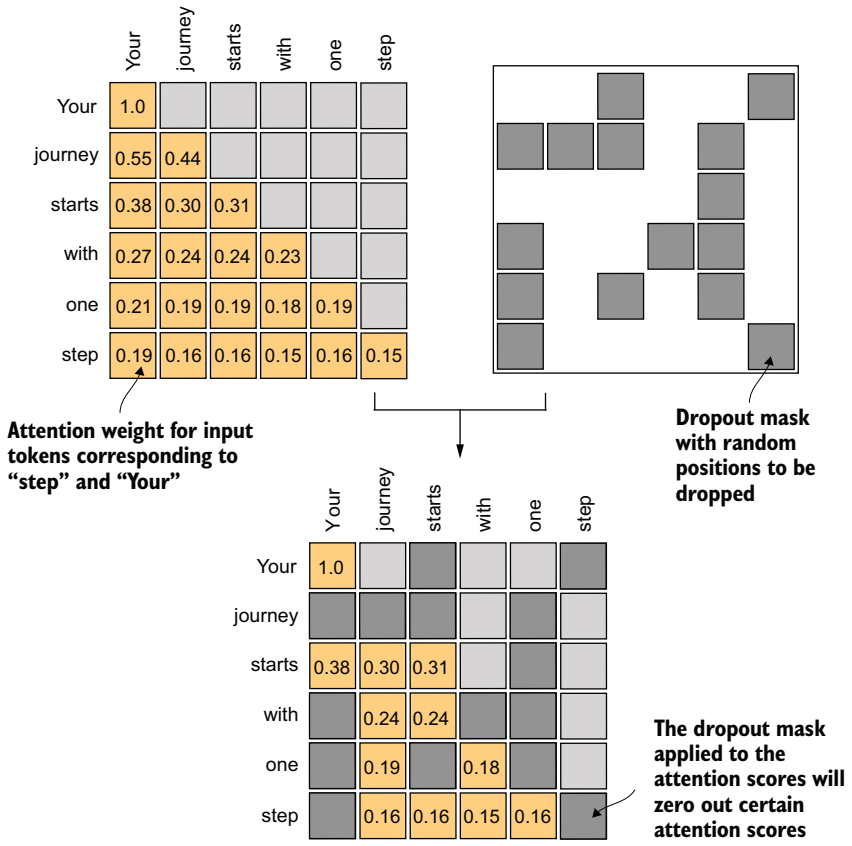



**Figure 3.22 Using the causal attention mask (upper left), we apply an additional dropout mask (upper right) to zero out additional attention weights to reduce overfitting during training.**

As we can see, approximately half of the values are zeroed out:

```
tensor([[2., 2., 0., 2., 2., 0.],
        [0., 0., 0., 2., 0., 2.],
        [2., 2., 2., 2., 0., 2.],
        [0., 2., 2., 0., 0., 2.],
        [0., 2., 0., 2., 0., 2.],
        [0., 2., 2., 2., 2., 0.]])
```

When applying dropout to an attention weight matrix with a rate of 50%, half of the elements in the matrix are randomly set to zero. To compensate for the reduction in active elements, the values of the remaining elements in the matrix are scaled up by a factor of $1/0.5 = 2$. This scaling is crucial to maintain the overall balance of the atten-

tion weights, ensuring that the average influence of the attention mechanism remains consistent during both the training and inference phases.

Now let's apply dropout to the attention weight matrix itself:

```
torch.manual_seed(123)
print(dropout(attn_weights))
```

The resulting attention weight matrix now has additional elements zeroed out and the remaining 1s rescaled:

```
tensor([[2.0000, 0.0000, 0 .0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.7599, 0.6194, 0.6206, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.4921, 0.4925, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.3966, 0.0000, 0.3775, 0.0000, 0.0000],
        [0.0000, 0.3327, 0.3331, 0.3084, 0.3331, 0.0000]],
       grad_fn=<MulBackward0>
```

Note that the resulting dropout outputs may look different depending on your operating system; you can read more about this inconsistency here on the PyTorch issue tracker at https://github.com/pytorch/pytorch/issues/121595.

Having gained an understanding of causal attention and dropout masking, we can now develop a concise Python class. This class is designed to facilitate the efficient application of these two techniques.

### 3.5.3 Implementing a compact causal attention class

We will now incorporate the causal attention and dropout modifications into the `SelfAttention` Python class we developed in section 3.4. This class will then serve as a template for developing *multi-head attention*, which is the final attention class we will implement.

But before we begin, let's ensure that the code can handle batches consisting of more than one input so that the `CausalAttention` class supports the batch outputs produced by the data loader we implemented in chapter 2.

For simplicity, to simulate such batch inputs, we duplicate the input text example:

```
batch = torch.stack((inputs, inputs), dim=0)
print(batch.shape)
```

**Two inputs with six tokens each; each token has embedding dimension 3.**

This results in a three-dimensional tensor consisting of two input texts with six tokens each, where each token is a three-dimensional embedding vector:

```
torch.Size([2, 6, 3])
```

The following `CausalAttention` class is similar to the `SelfAttention` class we implemented earlier, except that we added the dropout and causal mask components.

**Listing 3.3 A compact causal attention class**

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
           'mask',
           torch.triu(torch.ones(context_length, context_length),
           diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.transpose(1, 2)
        attn_scores.masked_fill_(
            self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        attn_weights = self.dropout(attn_weights)

        context_vec = attn_weights @ values
        return context_vec
```

**Compared to the previous SelfAttention_v1 class, we added a dropout layer.**

**The register_buffer call is also a new addition (more information is provided in the following text).**

**We transpose dimensions 1 and 2, keeping the batch dimension at the first position (0).**

**In PyTorch, operations with a trailing underscore are performed in-place, avoiding unnecessary memory copies.**

While all added code lines should be familiar at this point, we now added a `self.register_buffer()` call in the `__init__` method. The use of `register_buffer` in PyTorch is not strictly necessary for all use cases but offers several advantages here. For instance, when we use the `CausalAttention` class in our LLM, buffers are automatically moved to the appropriate device (CPU or GPU) along with our model, which will be relevant when training our LLM. This means we don't need to manually ensure these tensors are on the same device as your model parameters, avoiding device mismatch errors.

We can use the `CausalAttention` class as follows, similar to `SelfAttention` previously:

```
torch.manual_seed(123)
context_length = batch.shape[1]
ca = CausalAttention(d_in, d_out, context_length, 0.0)
context_vecs = ca(batch)
print("context_vecs.shape:", context_vecs.shape)
```

The resulting context vector is a three-dimensional tensor where each token is now represented by a two-dimensional embedding:

```
context_vecs.shape: torch.Size([2, 6, 2])
```

Figure 3.23 summarizes what we have accomplished so far. We have focused on the concept and implementation of causal attention in neural networks. Next, we will expand on this concept and implement a multi-head attention module that implements several causal attention mechanisms in parallel.

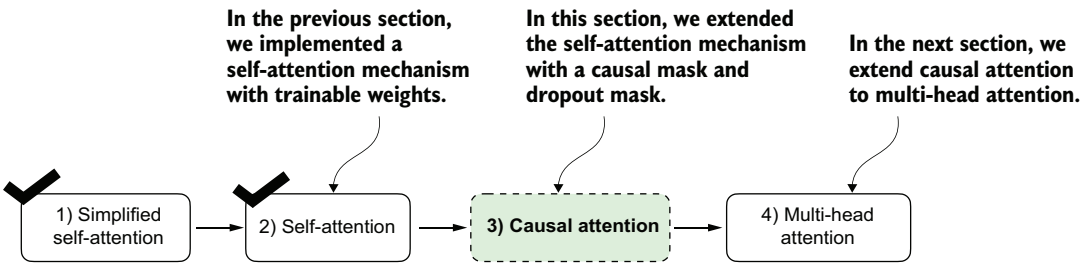



**Figure 3.23 Here's what we've done so far. We began with a simplified attention mechanism, added trainable weights, and then added a causal attention mask. Next, we will extend the causal attention mechanism and code multi-head attention, which we will use in our LLM.**

## 3.6 Extending single-head attention to multi-head attention

Our final step will be to extend the previously implemented causal attention class over multiple heads. This is also called *multi-head attention*.

The term "multi-head" refers to dividing the attention mechanism into multiple "heads," each operating independently. In this context, a single causal attention module can be considered single-head attention, where there is only one set of attention weights processing the input sequentially.

We will tackle this expansion from causal attention to multi-head attention. First, we will intuitively build a multi-head attention module by stacking multiple `CausalAttention` modules. Then we will then implement the same multi-head attention module in a more complicated but more computationally efficient way.

### 3.6.1 Stacking multiple single-head attention layers

In practical terms, implementing multi-head attention involves creating multiple instances of the self-attention mechanism (see figure 3.18), each with its own weights, and then combining their outputs. Using multiple instances of the self-attention mechanism can be computationally intensive, but it's crucial for the kind of complex pattern recognition that models like transformer-based LLMs are known for.

Figure 3.24 illustrates the structure of a multi-head attention module, which consists of multiple single-head attention modules, as previously depicted in figure 3.18, stacked on top of each other.

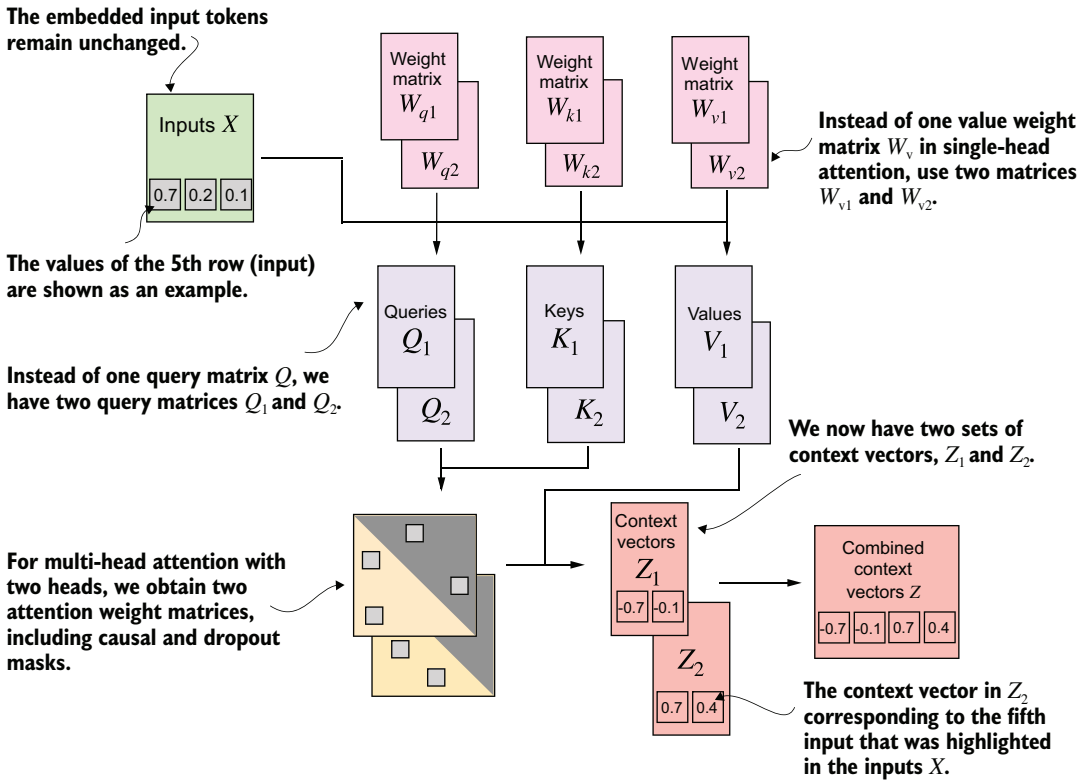



**Figure 3.24 The multi-head attention module includes two single-head attention modules stacked on top of each other. So, instead of using a single matrix $W_v$ for computing the value matrices, in a multi-head attention module with two heads, we now have two value weight matrices: $W_{v1}$ and $W_{v2}$. The same applies to the other weight matrices, $W_Q$ and $W_k$. We obtain two sets of context vectors $Z_1$ and $Z_2$ that we can combine into a single context vector matrix $Z$.**

As mentioned before, the main idea behind multi-head attention is to run the attention mechanism multiple times (in parallel) with different, learned linear projections—the results of multiplying the input data (like the query, key, and value vectors in attention mechanisms) by a weight matrix. In code, we can achieve this by implementing a simple `MultiHeadAttentionWrapper` class that stacks multiple instances of our previously implemented `CausalAttention` module.

**Listing 3.4 A wrapper class to implement multi-head attention**

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(
                 d_in, d_out, context_length, dropout, qkv_bias
             )
             for _ in range(num_heads)]
        )

    def forward(self, x):
        return torch.cat([head(x) for head in self.heads], dim=-1)
```

For example, if we use this `MultiHeadAttentionWrapper` class with two attention heads (via `num_heads=2`) and `CausalAttention` output dimension `d_out=2`, we get a four-dimensional context vector (`d_out*num_heads=4`), as depicted in figure 3.25.

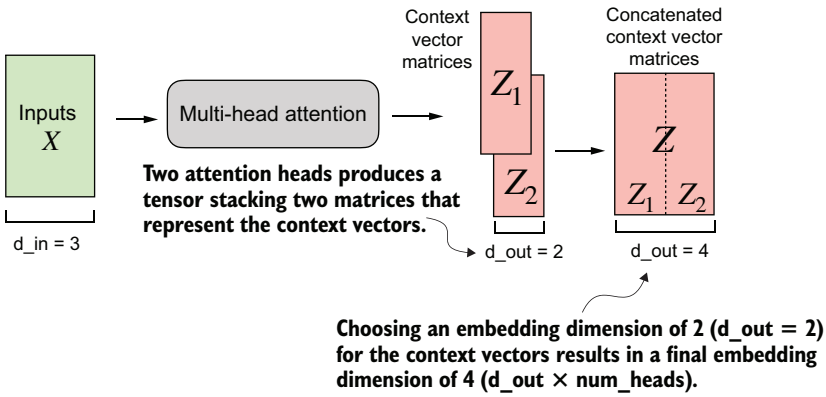



**Figure 3.25 Using the `MultiHeadAttentionWrapper`, we specified the number of attention heads (`num_heads`). If we set `num_heads=2`, as in this example, we obtain a tensor with two sets of context vector matrices. In each context vector matrix, the rows represent the context vectors corresponding to the tokens, and the columns correspond to the embedding dimension specified via `d_out=4`. We concatenate these context vector matrices along the column dimension. Since we have two attention heads and an embedding dimension of 2, the final embedding dimension is 2 × 2 = 4.**

To illustrate this further with a concrete example, we can use the `MultiHeadAttentionWrapper` class similar to the `CausalAttention` class before:

```
torch.manual_seed(123)
context_length = batch.shape[1] # This is the number of tokens
d_in, d_out = 3, 2
```

```
mha = MultiHeadAttentionWrapper(
    d_in, d_out, context_length, 0.0, num_heads=2
)
context_vecs = mha(batch)

print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

This results in the following tensor representing the context vectors:

```
tensor([[[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]],

        [[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]]], grad_fn=<CatBackward0>)
context_vecs.shape: torch.Size([2, 6, 4])
```

The first dimension of the resulting `context_vecs` tensor is 2 since we have two input texts (the input texts are duplicated, which is why the context vectors are exactly the same for those). The second dimension refers to the 6 tokens in each input. The third dimension refers to the four-dimensional embedding of each token.

**Exercise 3.2 Returning two-dimensional embedding vectors**

Change the input arguments for the `MultiHeadAttentionWrapper(..., num_heads=2)` call such that the output context vectors are two-dimensional instead of four dimensional while keeping the setting `num_heads=2`. Hint: You don't have to modify the class implementation; you just have to change one of the other input arguments.

Up to this point, we have implemented a `MultiHeadAttentionWrapper` that combined multiple single-head attention modules. However, these are processed sequentially via `[head(x) for head in self.heads]` in the forward method. We can improve this implementation by processing the heads in parallel. One way to achieve this is by computing the outputs for all attention heads simultaneously via matrix multiplication.

### 3.6.2 Implementing multi-head attention with weight splits

So far, we have created a `MultiHeadAttentionWrapper` to implement multi-head attention by stacking multiple single-head attention modules. This was done by instantiating and combining several `CausalAttention` objects.

Instead of maintaining two separate classes, `MultiHeadAttentionWrapper` and `CausalAttention`, we can combine these concepts into a single `MultiHeadAttention` class. Also, in addition to merging the `MultiHeadAttentionWrapper` with the `CausalAttention` code, we will make some other modifications to implement multi-head attention more efficiently.

In the `MultiHeadAttentionWrapper`, multiple heads are implemented by creating a list of `CausalAttention` objects (`self.heads`), each representing a separate attention head. The `CausalAttention` class independently performs the attention mechanism, and the results from each head are concatenated. In contrast, the following `MultiHeadAttention` class integrates the multi-head functionality within a single class. It splits the input into multiple heads by reshaping the projected query, key, and value tensors and then combines the results from these heads after computing attention.

Let's take a look at the `MultiHeadAttention` class before we discuss it further.

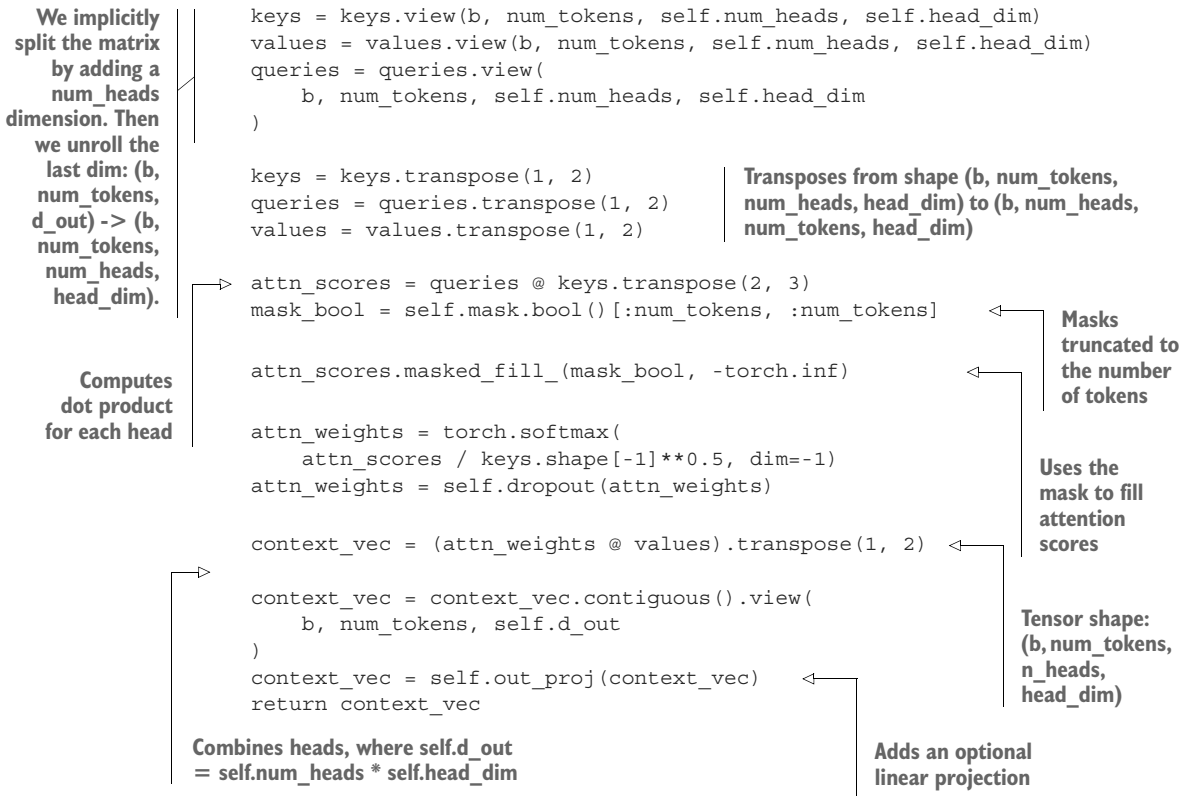
**Listing 3.5 An efficient multi-head attention class**

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out,
                 context_length, dropout, num_heads, qkv_bias=False):
        super().__init__()
        assert (d_out % num_heads == 0), \
            "d_out must be divisible by num_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            "mask",
            torch.triu(torch.ones(context_length, context_length),
                       diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
```

**Reduces the projection dim to match the desired output dim**

**Uses a Linear layer to combine head outputs**

**Tensor shape: (b, num_tokens, d_out)**




Even though the reshaping (`.view`) and transposing (`.transpose`) of tensors inside the `MultiHeadAttention` class looks very mathematically complicated, the `MultiHeadAttention` class implements the same concept as the `MultiHeadAttentionWrapper` earlier.

On a big-picture level, in the previous `MultiHeadAttentionWrapper`, we stacked multiple single-head attention layers that we combined into a multi-head attention layer. The `MultiHeadAttention` class takes an integrated approach. It starts with a multi-head layer and then internally splits this layer into individual attention heads, as illustrated in figure 3.26.

The splitting of the query, key, and value tensors is achieved through tensor reshaping and transposing operations using PyTorch's `.view` and `.transpose` methods. The input is first transformed (via linear layers for queries, keys, and values) and then reshaped to represent multiple heads.

The key operation is to split the `d_out` dimension into `num_heads` and `head_dim`, where `head_dim = d_out / num_heads`. This splitting is then achieved using the `.view` method: a tensor of dimensions `(b, num_tokens, d_out)` is reshaped to dimension `(b, num_tokens, num_heads, head_dim)`.

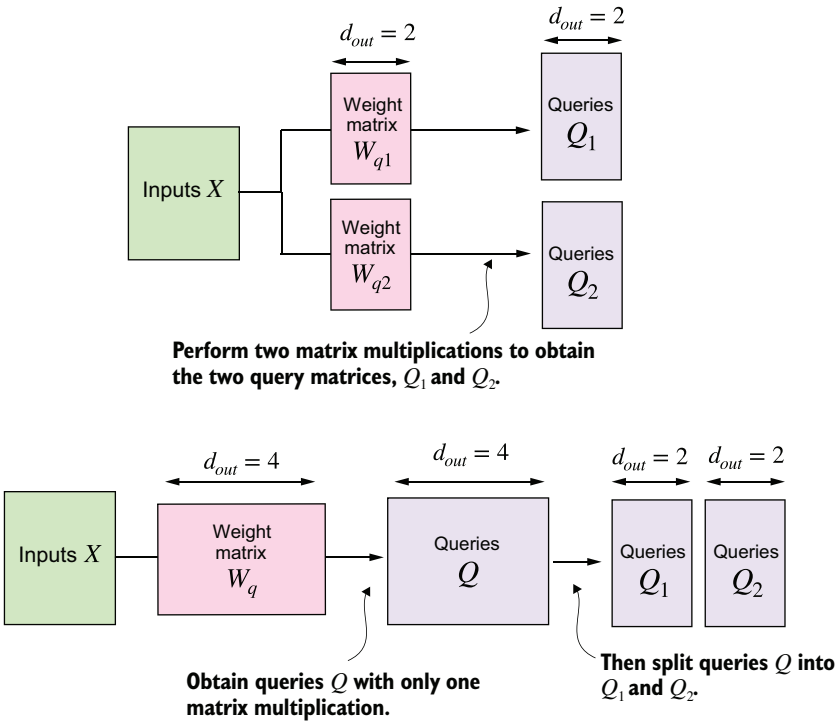



**Figure 3.26 In the `MultiHeadAttentionWrapper` class with two attention heads, we initialized two weight matrices, $W_{q1}$ and $W_{q2}$, and computed two query matrices, $Q_1$ and $Q_2$ (top). In the `MultiheadAttention` class, we initialize one larger weight matrix $W_q$, only perform one matrix multiplication with the inputs to obtain a query matrix $Q$, and then split the query matrix into $Q_1$ and $Q_2$ (bottom). We do the same for the keys and values, which are not shown to reduce visual clutter.**

The tensors are then transposed to bring the `num_heads` dimension before the `num_tokens` dimension, resulting in a shape of `(b, num_heads, num_tokens, head_dim)`. This transposition is crucial for correctly aligning the queries, keys, and values across the different heads and performing batched matrix multiplications efficiently.

To illustrate this batched matrix multiplication, suppose we have the following tensor:

```
a = torch.tensor([[[[0.2745, 0.6584, 0.2775, 0.8573],
                    [0.8993, 0.0390, 0.9268, 0.7388],
                    [0.7179, 0.7058, 0.9156, 0.4340]],

                   [[0.0772, 0.3565, 0.1479, 0.5331],
                    [0.4066, 0.2318, 0.4545, 0.9737],
                    [0.4606, 0.5159, 0.4220, 0.5786]]]])
```

**The shape of this tensor is (b, num_heads, num_tokens, head_dim) = (1, 2, 3, 4).**

Now we perform a batched matrix multiplication between the tensor itself and a view of the tensor where we transposed the last two dimensions, `num_tokens` and `head_dim`:

```
print(a @ a.transpose(2, 3))
```

The result is

```
tensor([[[[1.3208, 1.1631, 1.2879],
          [1.1631, 2.2150, 1.8424],
          [1.2879, 1.8424, 2.0402]],

         [[0.4391, 0.7003, 0.5903],
          [0.7003, 1.3737, 1.0620],
          [0.5903, 1.0620, 0.9912]]]])
```

In this case, the matrix multiplication implementation in PyTorch handles the four-dimensional input tensor so that the matrix multiplication is carried out between the two last dimensions `(num_tokens, head_dim)` and then repeated for the individual heads.

For instance, the preceding becomes a more compact way to compute the matrix multiplication for each head separately:

```
first_head = a[0, 0, :, :]
first_res = first_head @ first_head.T
print("First head:\n", first_res)

second_head = a[0, 1, :, :]
second_res = second_head @ second_head.T
print("\nSecond head:\n", second_res)
```

The results are exactly the same results as those we obtained when using the batched matrix multiplication `print(a @ a.transpose(2, 3))`:

```
First head:
 tensor([[1.3208, 1.1631, 1.2879],
        [1.1631, 2.2150, 1.8424],
        [1.2879, 1.8424, 2.0402]])

Second head:
 tensor([[0.4391, 0.7003, 0.5903],
        [0.7003, 1.3737, 1.0620],
        [0.5903, 1.0620, 0.9912]])
```

Continuing with `MultiHeadAttention`, after computing the attention weights and context vectors, the context vectors from all heads are transposed back to the shape `(b, num_tokens, num_heads, head_dim)`. These vectors are then reshaped (flattened) into the shape `(b, num_tokens, d_out)`, effectively combining the outputs from all heads.

Additionally, we added an output projection layer (`self.out_proj`) to `MultiHeadAttention` after combining the heads, which is not present in the `CausalAttention` class. This output projection layer is not strictly necessary (see appendix B for

more details), but it is commonly used in many LLM architectures, which is why I added it here for completeness.

Even though the `MultiHeadAttention` class looks more complicated than the `MultiHeadAttentionWrapper` due to the additional reshaping and transposition of tensors, it is more efficient. The reason is that we only need one matrix multiplication to compute the keys, for instance, `keys = self.W_key(x)` (the same is true for the queries and values). In the `MultiHeadAttentionWrapper`, we needed to repeat this matrix multiplication, which is computationally one of the most expensive steps, for each attention head.

The `MultiHeadAttention` class can be used similar to the `SelfAttention` and `CausalAttention` classes we implemented earlier:

```
torch.manual_seed(123)
batch_size, context_length, d_in = batch.shape
d_out = 2
mha = MultiHeadAttention(d_in, d_out, context_length, 0.0, num_heads=2)
context_vecs = mha(batch)
print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

The results show that the output dimension is directly controlled by the `d_out` argument:

```
tensor([[[0.3190, 0.4858],
         [0.2943, 0.3897],
         [0.2856, 0.3593],
         [0.2693, 0.3873],
         [0.2639, 0.3928],
         [0.2575, 0.4028]],

        [[0.3190, 0.4858],
         [0.2943, 0.3897],
         [0.2856, 0.3593],
         [0.2693, 0.3873],
         [0.2639, 0.3928],
         [0.2575, 0.4028]]], grad_fn=<ViewBackward0>)
context_vecs.shape: torch.Size([2, 6, 2])
```

We have now implemented the `MultiHeadAttention` class that we will use when we implement and train the LLM. Note that while the code is fully functional, I used relatively small embedding sizes and numbers of attention heads to keep the outputs readable.

For comparison, the smallest GPT-2 model (117 million parameters) has 12 attention heads and a context vector embedding size of 768. The largest GPT-2 model (1.5 billion parameters) has 25 attention heads and a context vector embedding size of 1,600. The embedding sizes of the token inputs and context embeddings are the same in GPT models (`d_in = d_out`).

### Exercise 3.3 Initializing GPT-2 size attention modules

Using the `MultiHeadAttention` class, initialize a multi-head attention module that has the same number of attention heads as the smallest GPT-2 model (12 attention heads). Also ensure that you use the respective input and output embedding sizes similar to GPT-2 (768 dimensions). Note that the smallest GPT-2 model supports a context length of 1,024 tokens.

## Summary

- Attention mechanisms transform input elements into enhanced context vector representations that incorporate information about all inputs.
- A self-attention mechanism computes the context vector representation as a weighted sum over the inputs.
- In a simplified attention mechanism, the attention weights are computed via dot products.
- A dot product is a concise way of multiplying two vectors element-wise and then summing the products.
- Matrix multiplications, while not strictly required, help us implement computations more efficiently and compactly by replacing nested `for` loops.
- In self-attention mechanisms used in LLMs, also called scaled-dot product attention, we include trainable weight matrices to compute intermediate transformations of the inputs: queries, values, and keys.
- When working with LLMs that read and generate text from left to right, we add a causal attention mask to prevent the LLM from accessing future tokens.
- In addition to causal attention masks to zero-out attention weights, we can add a dropout mask to reduce overfitting in LLMs.
- The attention modules in transformer-based LLMs involve multiple instances of causal attention, which is called multi-head attention.
- We can create a multi-head attention module by stacking multiple instances of causal attention modules.
- A more efficient way of creating multi-head attention modules involves batched matrix multiplications.

# 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.

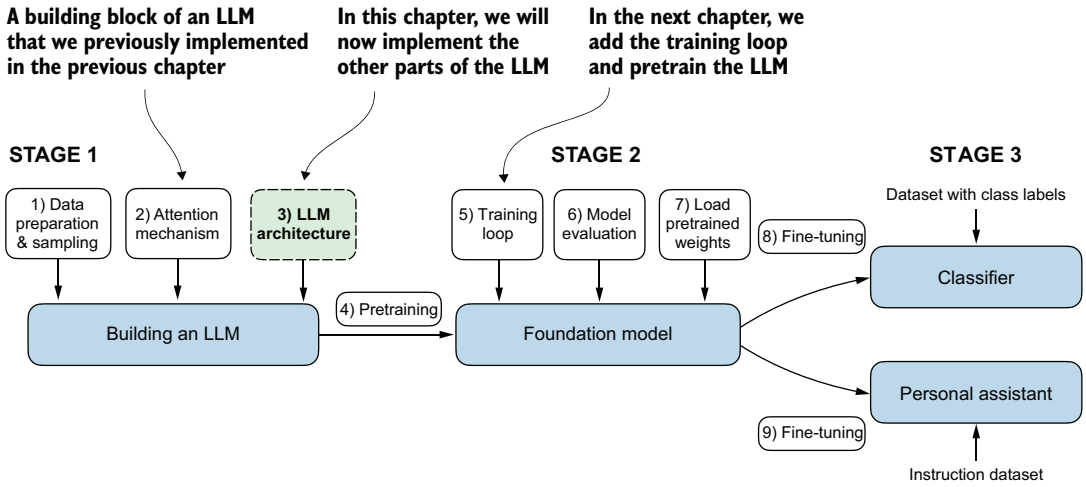



**Figure 4.1 The three main stages of coding an LLM. This chapter focuses on step 3 of stage 1: implementing the LLM architecture.**

## 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.

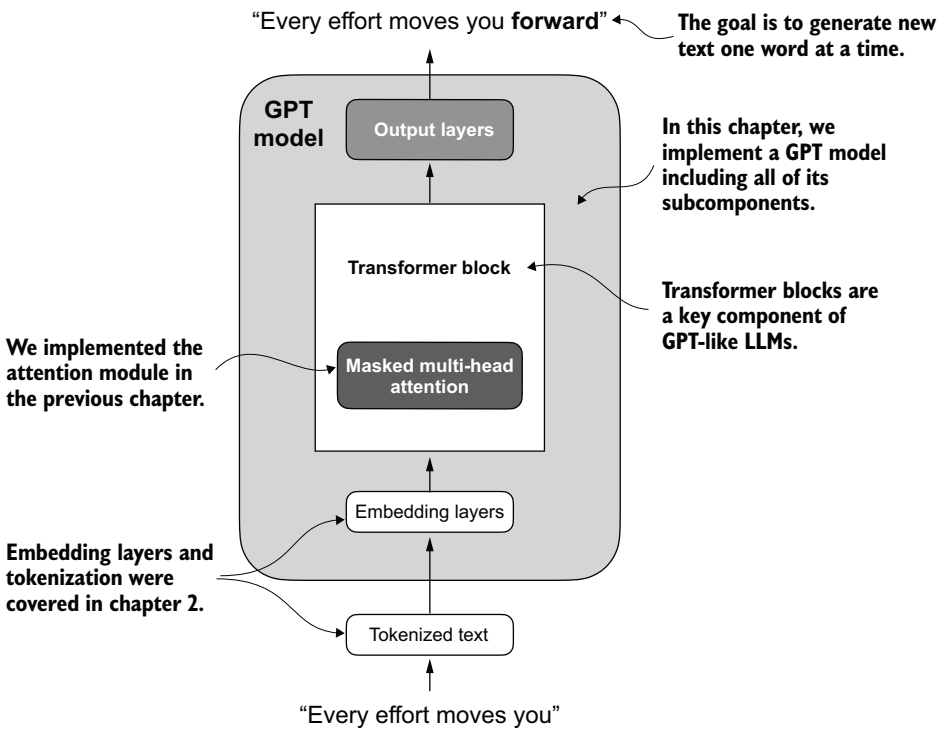



**Figure 4.2 A GPT model. In addition to the embedding layers, it consists of one or more transformer blocks containing the masked multi-head attention module we previously implemented.**

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`.

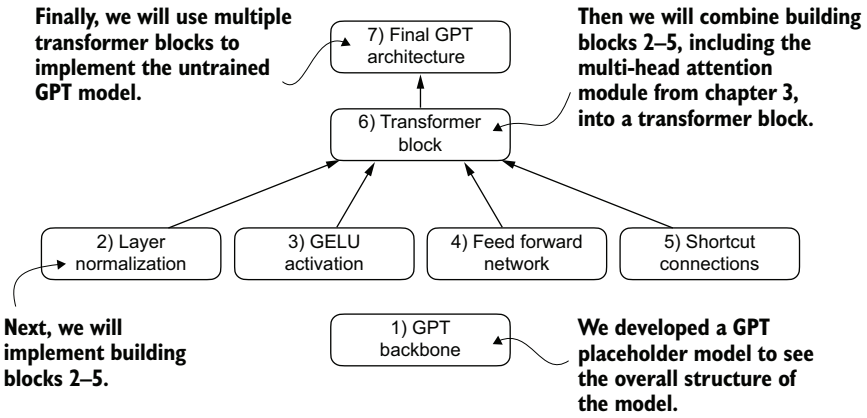



**Figure 4.3 The order in which we code the GPT architecture. We start with the GPT backbone, a placeholder architecture, before getting to the individual core pieces and eventually assembling them in a transformer block for the final GPT architecture.**

**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)
```

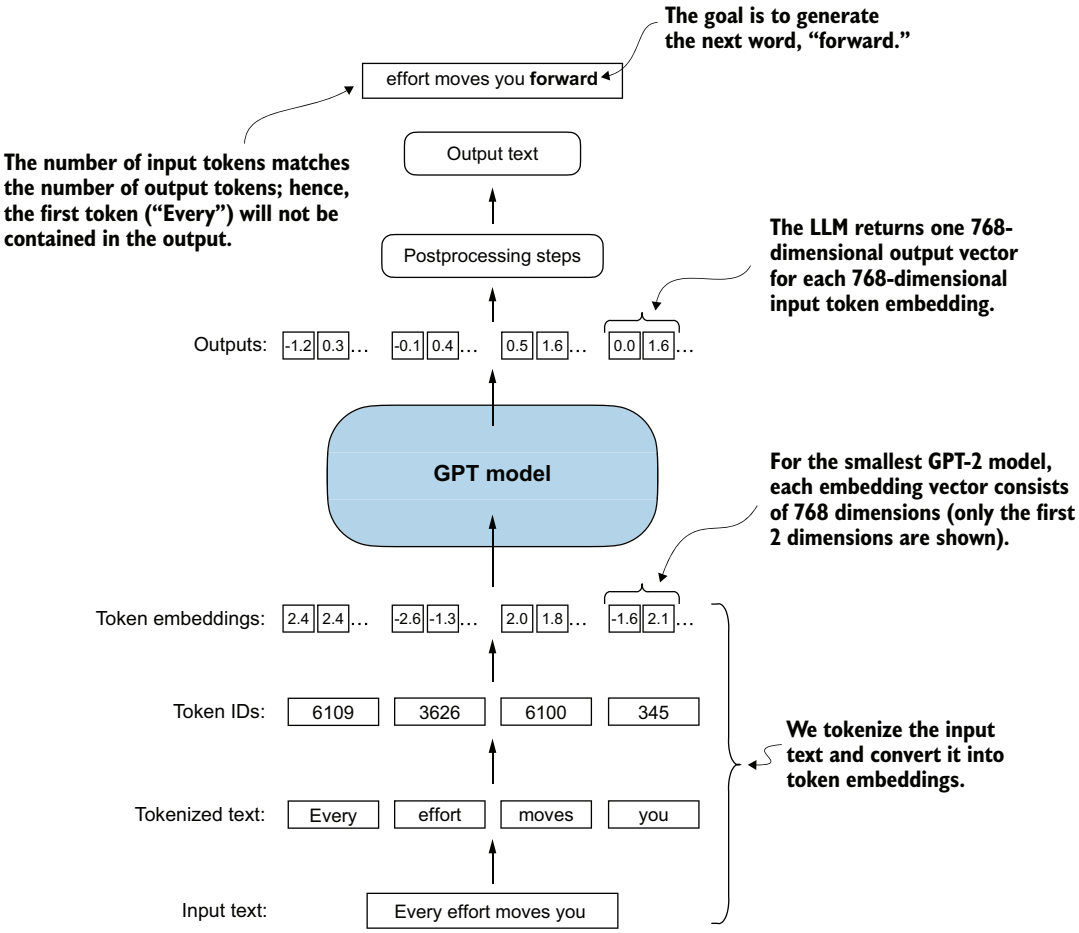



**Figure 4.4 A big-picture overview showing how the input data is tokenized, embedded, and fed to the GPT model. Note that in our `DummyGPTClass` coded earlier, the token embedding is handled inside the GPT model. In LLMs, the embedded input token dimension typically matches the output dimension. The 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.

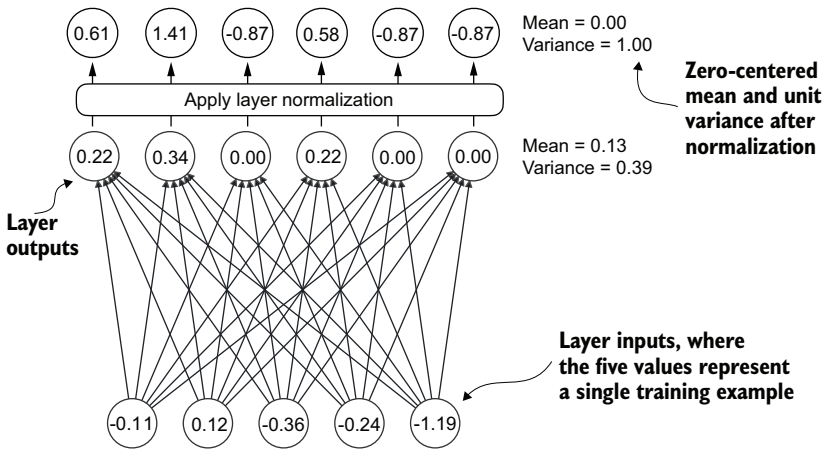



**Figure 4.5 An illustration of layer normalization where the six outputs of the layer, also called activations, are normalized such that they have a 0 mean and a variance of 1.**

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 nonlinear 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

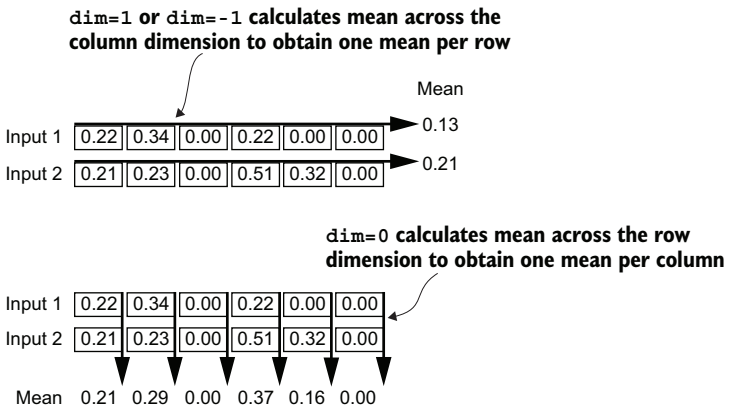



**Figure 4.6 An illustration of the dim parameter when calculating the mean of a tensor. For instance, if we have a two-dimensional tensor (matrix) with dimensions `[rows, columns]`, using `dim=0` will perform the operation across rows (vertically, as shown at the bottom), resulting in an output that 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 $-5.9605 \times 10^{-8}$, 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.

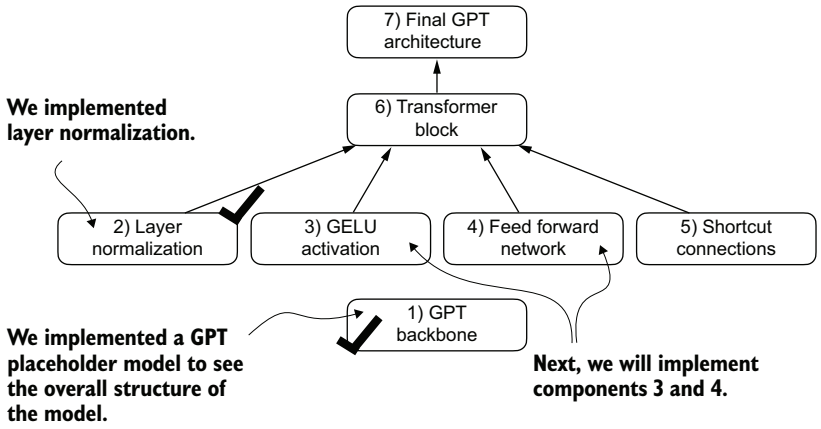



**Figure 4.7 The building blocks necessary to build the GPT architecture. So far, we have completed the GPT backbone and layer normalization. Next, we will focus on GELU activation and the feed forward network.**

### 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 $\text{GELU}(x) = x \cdot \Phi(x)$, where $\Phi(x)$ 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)
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()
```

**Creates 100 sample data points in the range –3 to 3**

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 $x = -0.75$).

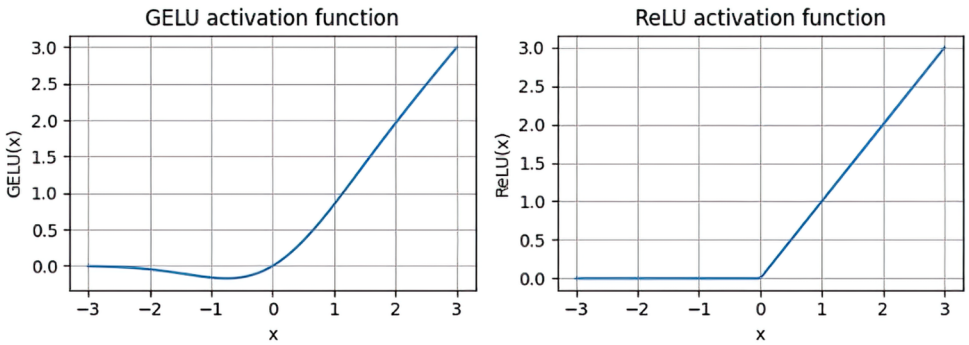



**Figure 4.8 The output of the GELU and ReLU plots using matplotlib. The x-axis shows the function inputs and the y-axis shows the function outputs.**

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.

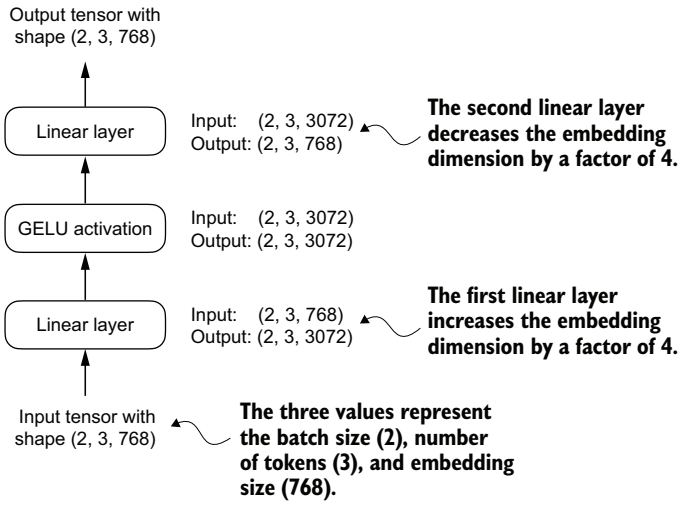



**Figure 4.9 An overview of the connections between the layers of the feed forward neural network. This neural network can accommodate variable batch sizes and numbers of tokens in the input. However, the embedding size for each token is determined and fixed when initializing the weights.**

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)
out = ffn(x)
print(out.shape)
```

**Creates sample input with batch dimension 2**

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.

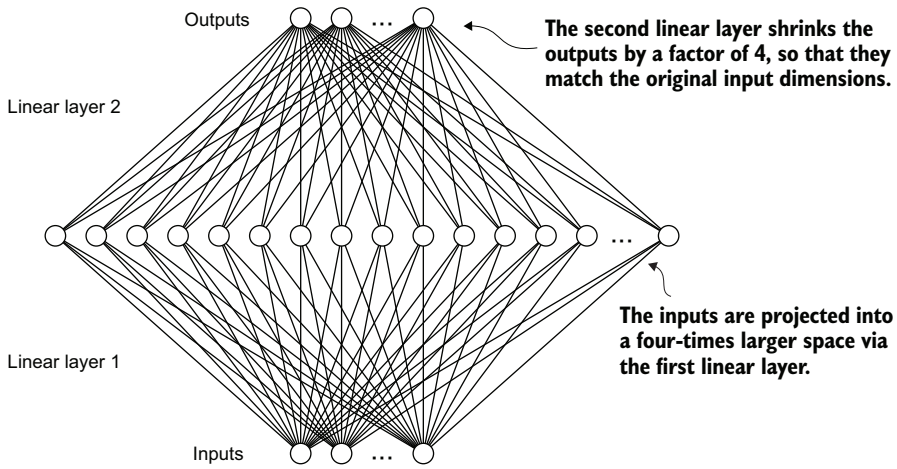



**Figure 4.10 An illustration of the expansion and contraction of the layer outputs in the feed forward neural network. First, the inputs expand by a factor of 4 from 768 to 3,072 values. Then, the second layer compresses the 3,072 values back into a 768-dimensional representation.**

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.

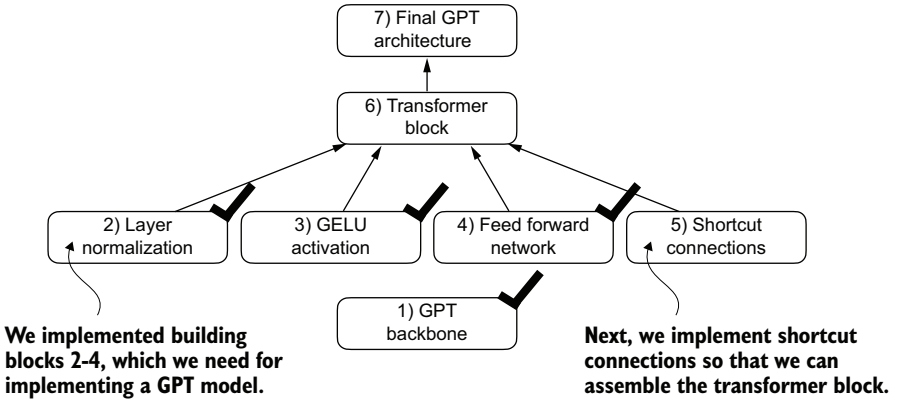



**Figure 4.11 The building blocks necessary to build the GPT architecture. The black checkmarks indicating those we have already covered.**

## 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.

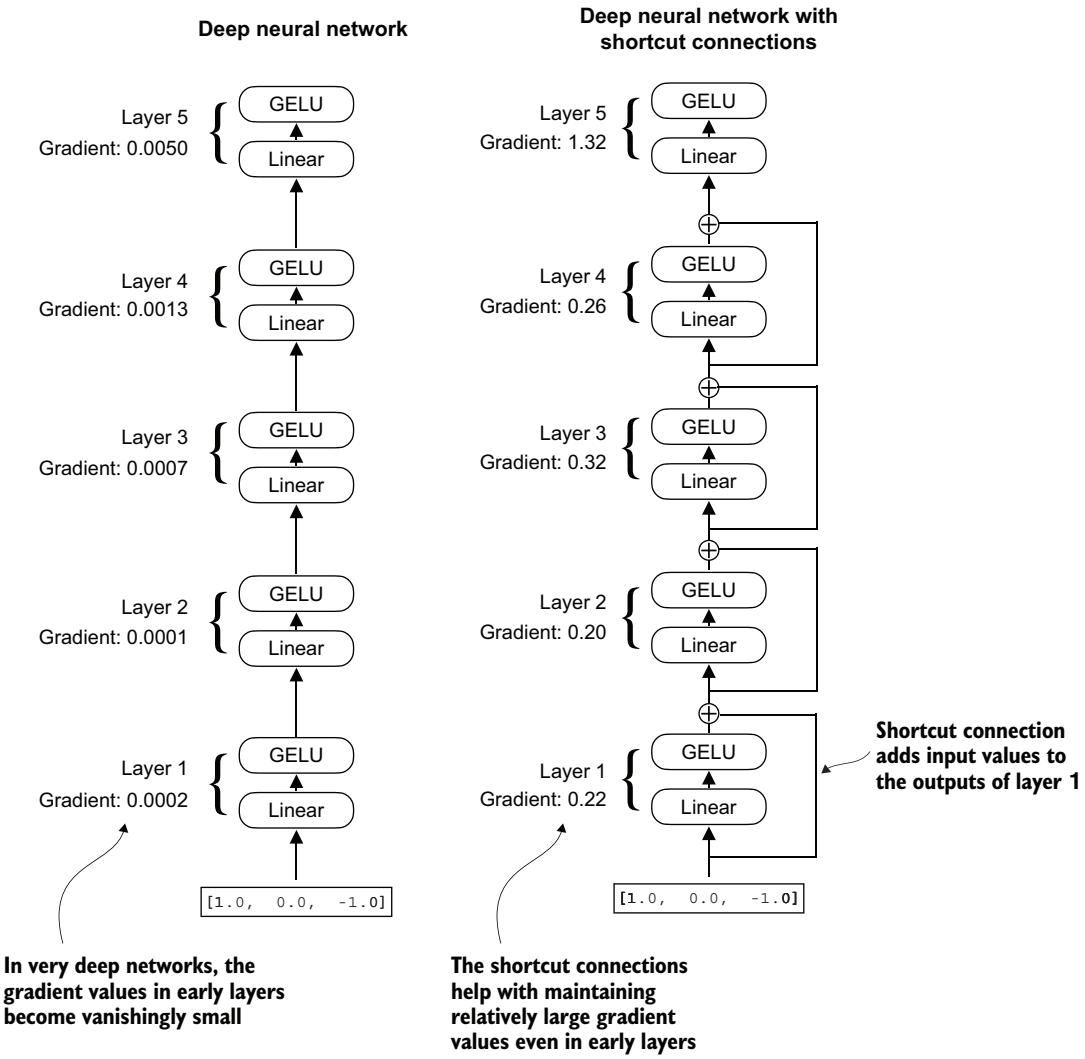



**Figure 4.12** A comparison between a deep neural network consisting of five layers without (left) and with shortcut connections (right). Shortcut connections involve adding the inputs of a layer to its outputs, effectively creating an alternate path that bypasses certain layers. The gradients denote 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.

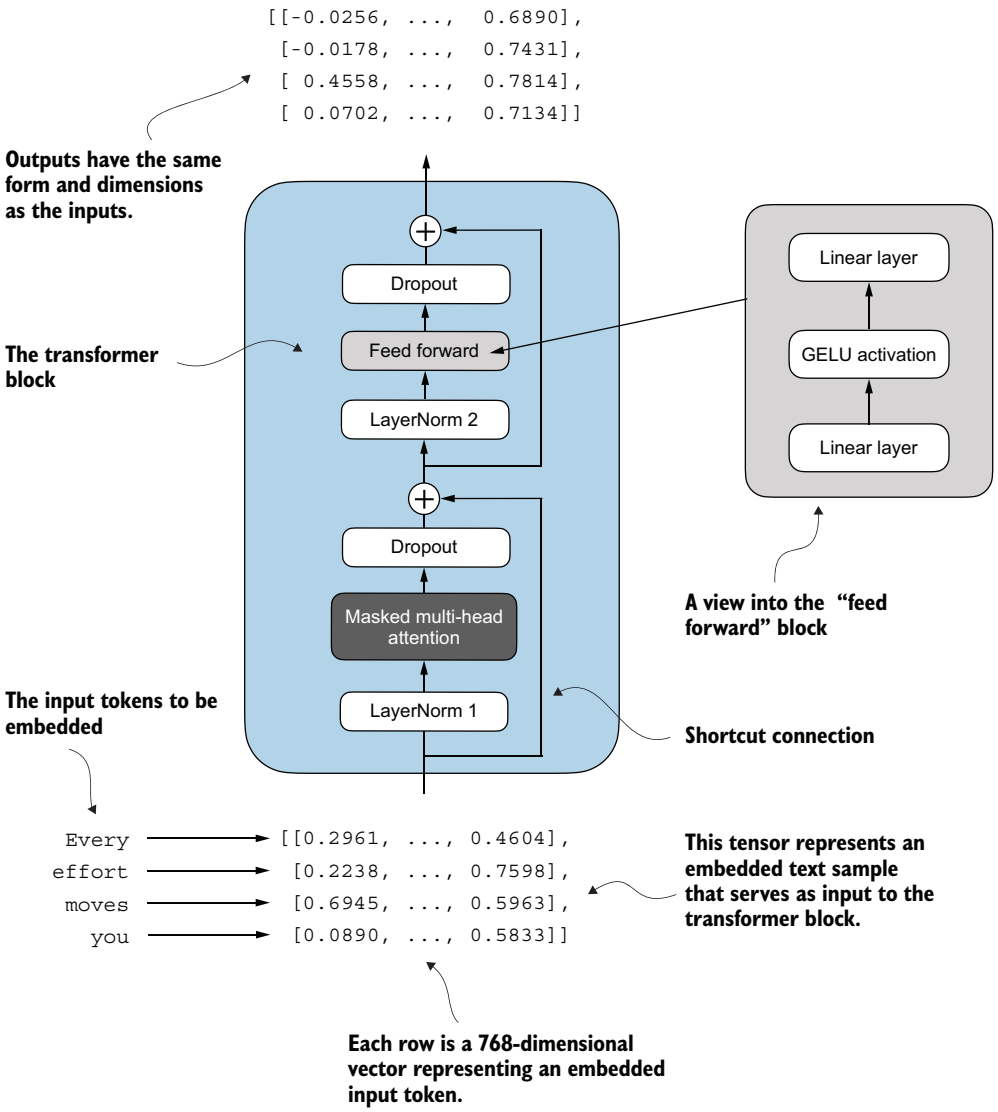



**Figure 4.13** An illustration of a transformer block. Input tokens have been embedded into 768-dimensional vectors. Each row corresponds to one token's vector representation. The outputs of the transformer block are vectors of the same dimension as the input, which can then be fed into subsequent layers 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 connection for feed forward block
        shortcut = x
        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)                          # Creates sample input of shape [batch_size, num_tokens, emb_dim]
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)

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

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.

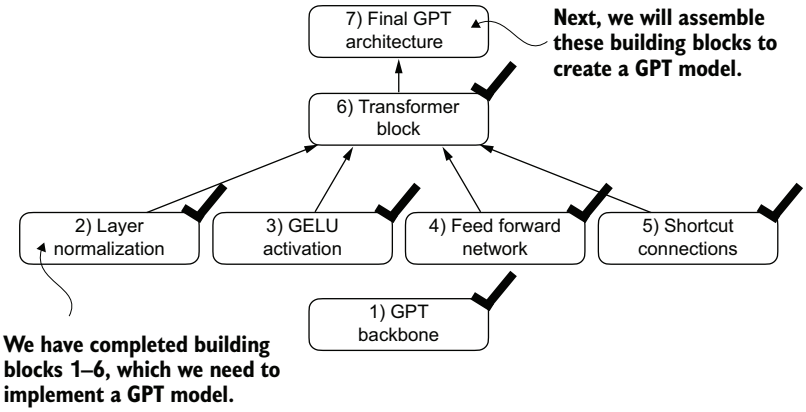



**Figure 4.14 The building blocks necessary to build the GPT architecture. The black checks indicate the blocks we have completed.**

## 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.

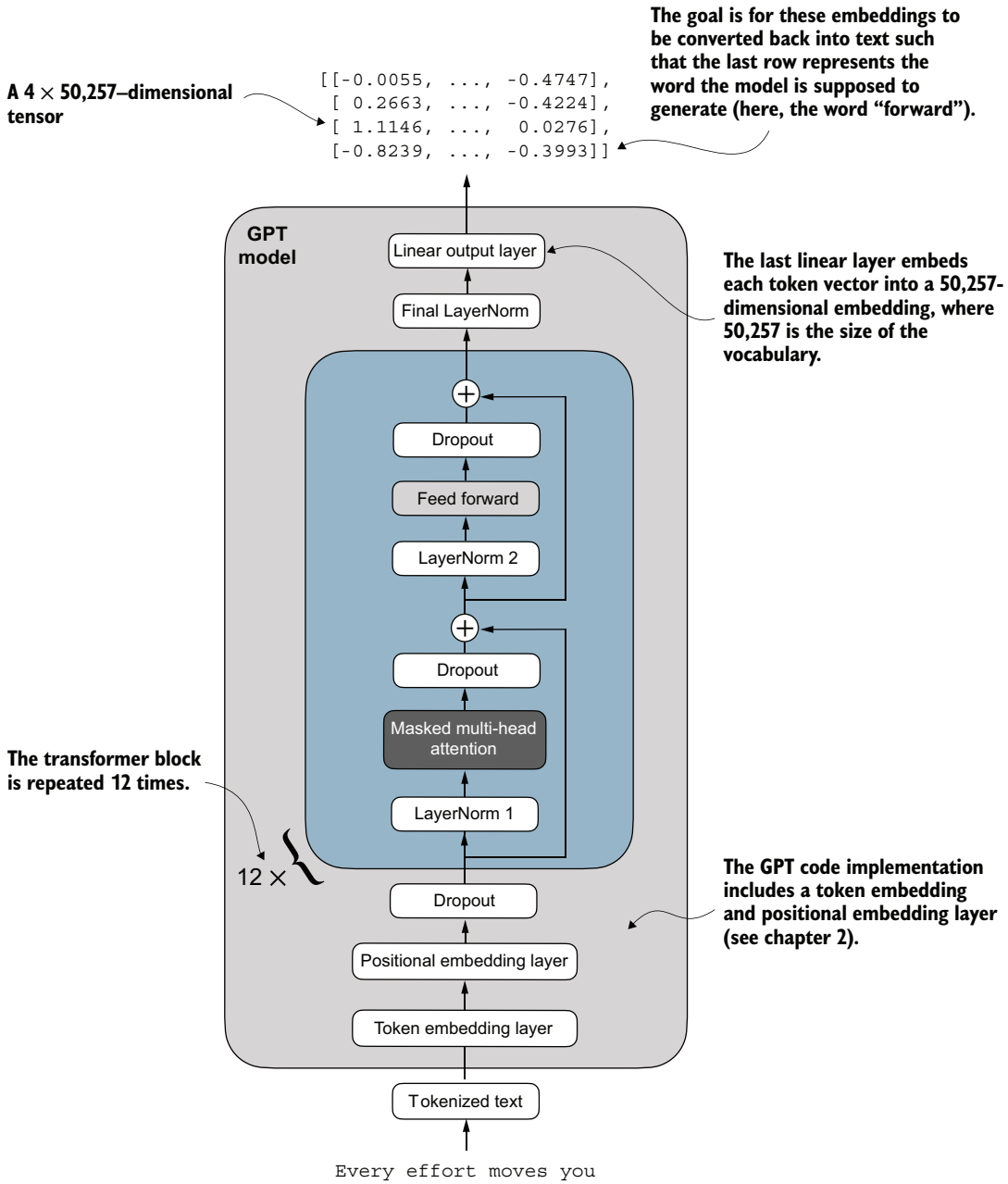



**Figure 4.15** An overview of the GPT model architecture showing the flow of data through the GPT model. Starting from the bottom, tokenized text is first converted into token embeddings, which are then augmented with positional embeddings. This combined information forms a tensor that is passed through 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],
         [-0.1076,  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

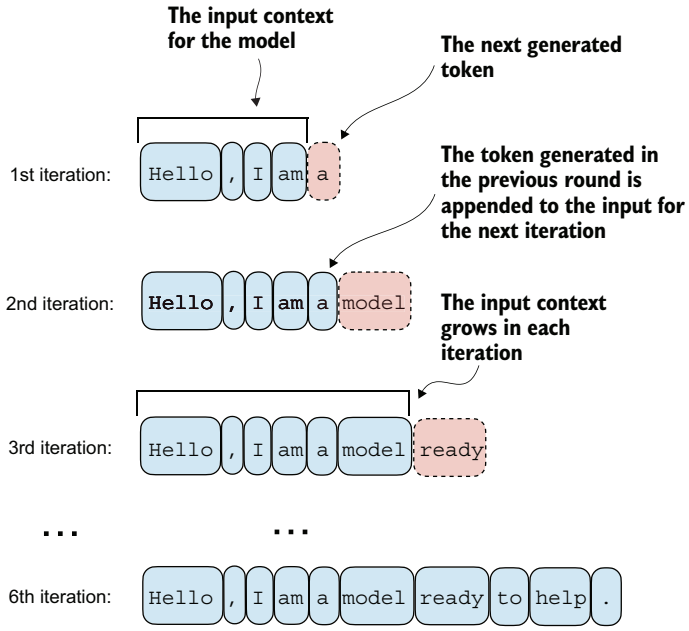



**Figure 4.16 The step-by-step process by which an LLM generates text, one token at a time. Starting with an initial input context ("Hello, I am"), the model predicts a subsequent token during each iteration, appending it to the input context for the next round of prediction. As shown, the first iteration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

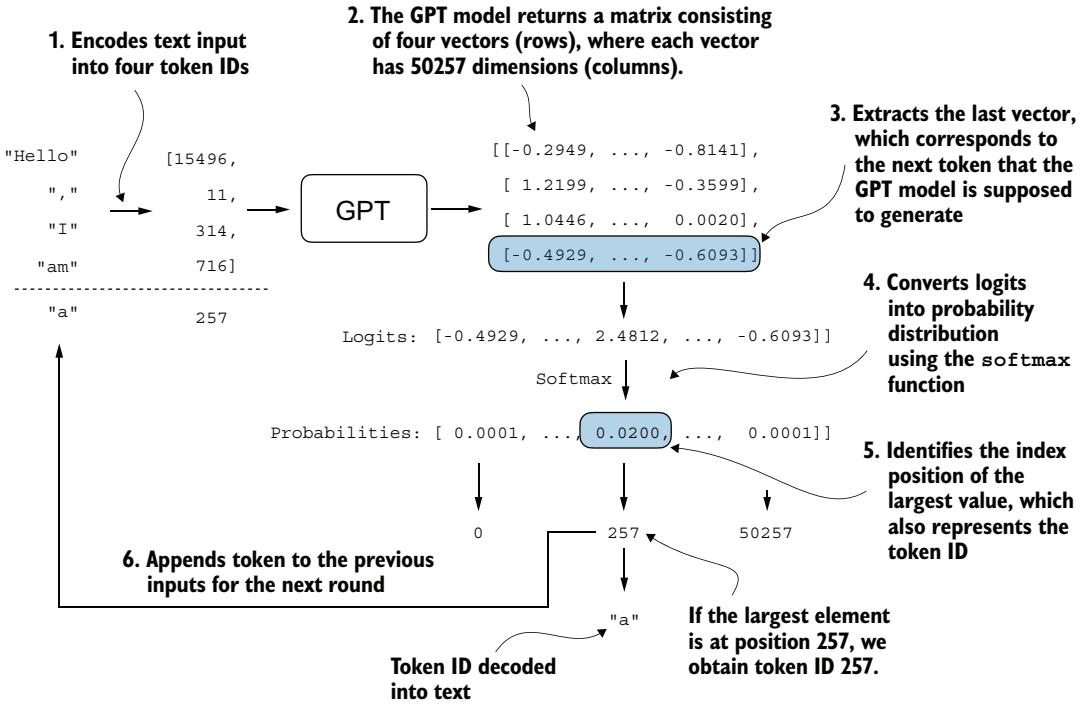



**Figure 4.17** The mechanics of text generation in a GPT model by showing a single iteration in the token generation process. The process begins by encoding the input text into token IDs, which are then fed into the GPT model. The outputs of the model are then converted back into text and appended to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

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

idx_next has shape (batch, 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)
print("encoded_tensor.shape:", encoded_tensor.shape)
```

**Adds batch dimension**

The encoded IDs are

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

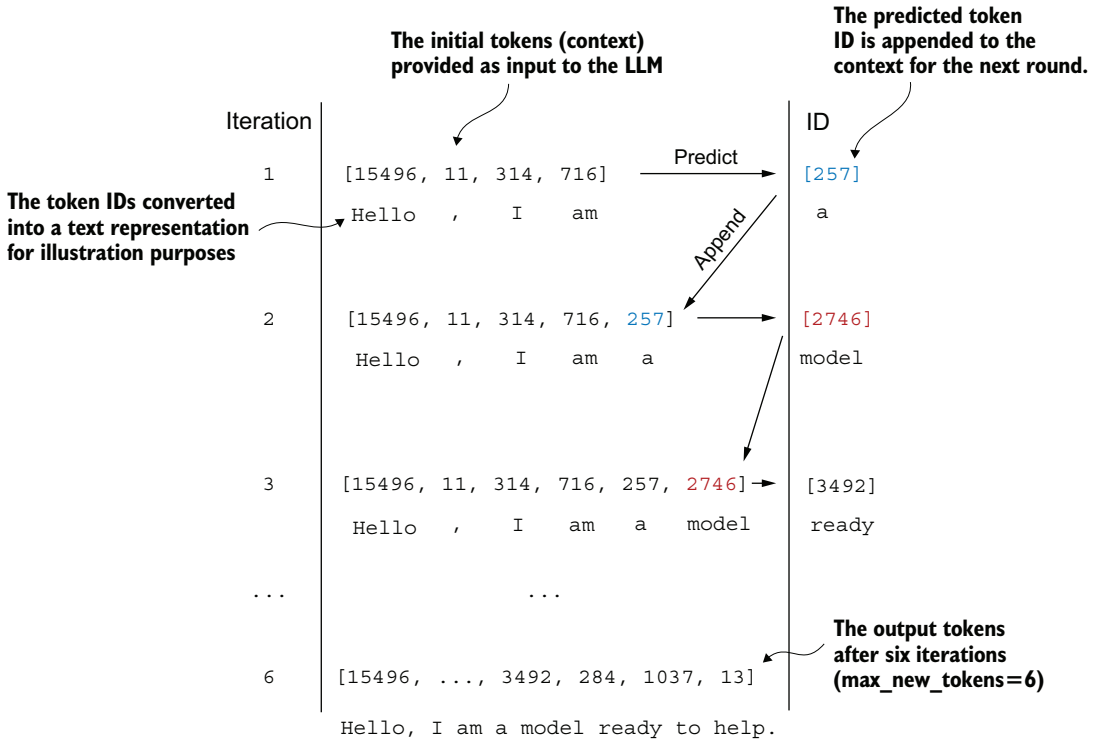



**Figure 4.18 The six iterations of a token prediction cycle, where the model takes a sequence of initial token IDs as input, predicts the next token, and appends this token to the input sequence for the next iteration. (The token IDs are also translated into their corresponding text for better understanding.)**

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 we are not training the model
out = generate_text_simple(
    model=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.

# 5 Pretraining on unlabeled data

***This chapter covers***

- Computing the training and validation set losses to assess the quality of LLM-generated text during training
- Implementing a training function and pretraining the LLM
- Saving and loading model weights to continue training an LLM
- Loading pretrained weights from OpenAI

Thus far, we have implemented the data sampling and attention mechanism and coded the LLM architecture. It is now time to implement a training function and pretrain the LLM. We will learn about basic model evaluation techniques to measure the quality of the generated text, which is a requirement for optimizing the LLM during the training process. Moreover, we will discuss how to load pretrained weights, giving our LLM a solid starting point for fine-tuning. Figure 5.1 lays out our overall plan, highlighting what we will discuss in this chapter.