

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.

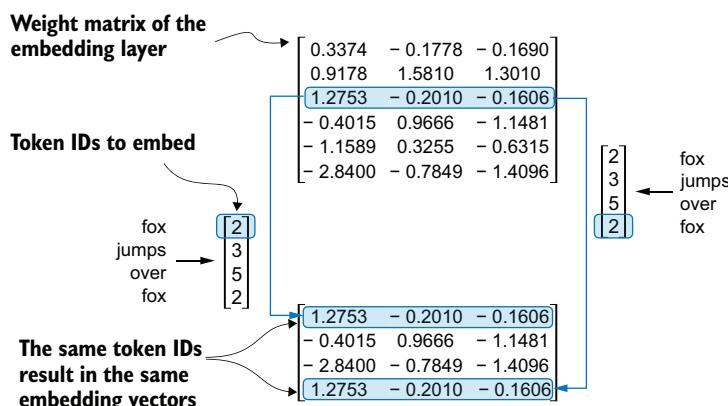


Figure 2.17 The embedding layer converts a token ID into the same vector representation regardless of where it is located in the input sequence. For example, the token ID 5, whether it’s in the first or fourth position in the token ID input vector, will result in the same embedding vector.

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.

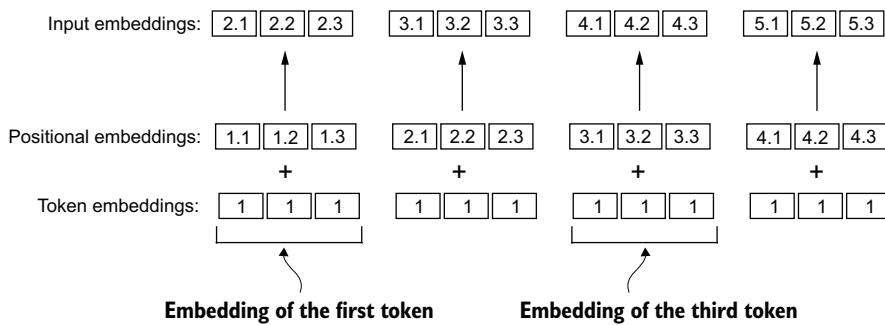


Figure 2.18 Positional embeddings are added to the token embedding vector to create the input embeddings for an LLM. The positional vectors have the same dimension as the original token embeddings. The token embeddings are shown with value 1 for simplicity.

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 8×4 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)
```