16

Transformers – Improving Natural Language Processing with Attention Mechanisms

In the previous chapter, we learned about **recurrent neural networks** (**RNNs**) and their applications in **natural language processing** (**NLP**) through a sentiment analysis project. However, a new architecture has recently emerged that has been shown to outperform the RNN-based **sequence-to-sequence** (**seq2seq**) models in several NLP tasks. This is the so-called **transformer** architecture.

Transformers have revolutionized natural language processing and have been at the forefront of many impressive applications ranging from automated language translation (`https://ai.googleblog.com/2020/06/recent-advances-in-google-translate.html`) and modeling fundamental properties of protein sequences (`https://www.pnas.org/content/118/15/e2016239118.short`) to creating an AI that helps people write code (`https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer`).

In this chapter, you will learn about the basic mechanisms of *attention* and *self-attention* and see how they are used in the original transformer architecture. Then, equipped with an understanding of how transformers work, we will explore some of the most influential NLP models that emerged from this architecture and learn how to use a large-scale language model, the so-called BERT model, in PyTorch.

We will cover the following topics:

- Improving RNNs with an attention mechanism
- Introducing the stand-alone self-attention mechanism
- Understanding the original transformer architecture
- Comparing transformer-based large-scale language models
- Fine-tuning BERT for sentiment classification

Adding an attention mechanism to RNNs

In this section, we discuss the motivation behind developing an **attention mechanism**, which helps predictive models to focus on certain parts of the input sequence more than others, and how it was originally used in the context of RNNs. Note that this section provides a historical perspective explaining why the attention mechanism was developed. If individual mathematical details appear complicated, you can feel free to skip over them as they are not needed for the next section, explaining the self-attention mechanism for transformers, which is the focus of this chapter.

Attention helps RNNs with accessing information

To understand the development of an attention mechanism, consider the traditional RNN model for a **seq2seq task** like language translation, which parses the entire input sequence (for instance, one or more sentences) before producing the translation, as shown in *Figure 16.1*:

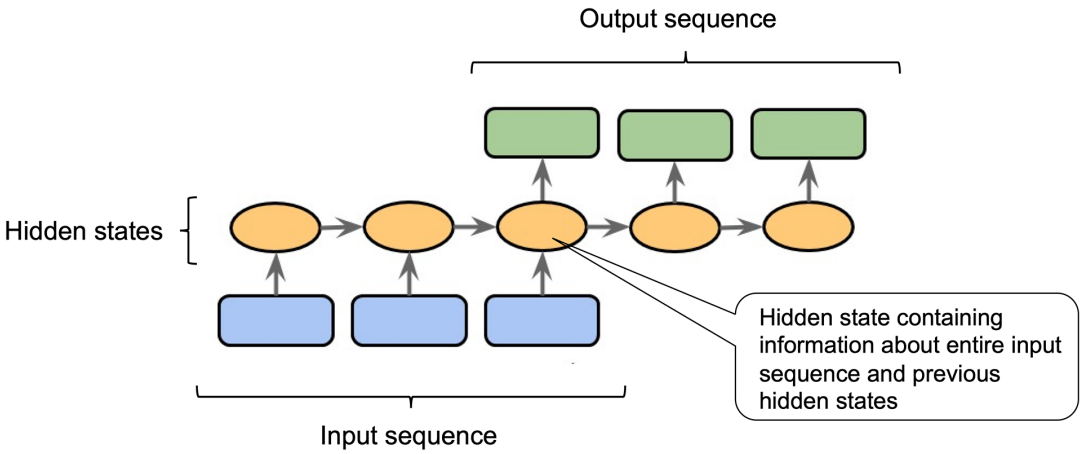

Figure 16.1: A traditional RNN encoder-decoder architecture for a seq2seq modeling task

Why is the RNN parsing the whole input sentence before producing the first output? This is motivated by the fact that translating a sentence word by word would likely result in grammatical errors, as illustrated in *Figure 16.2*:

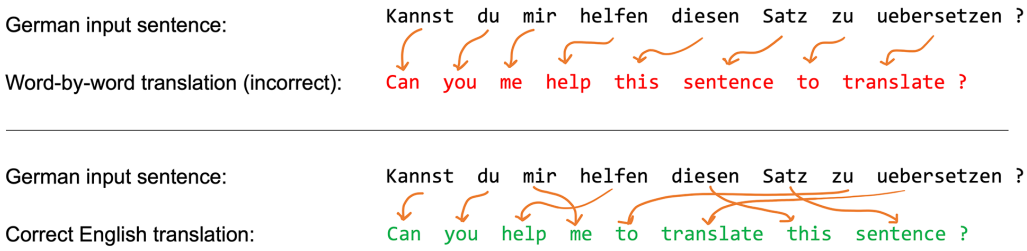

Figure 16.2: Translating a sentence word by word can lead to grammatical errors

However, as illustrated in *Figure 16.2*, one limitation of this seq2seq approach is that the RNN is trying to remember the entire input sequence via one single hidden unit before translating it. Compressing all the information into one hidden unit may cause loss of information, especially for long sequences. Thus, similar to how humans translate sentences, it may be beneficial to have access to the whole input sequence at each time step.

In contrast to a regular RNN, an attention mechanism lets the RNN access all input elements at each given time step. However, having access to all input sequence elements at each time step can be overwhelming. So, to help the RNN focus on the most relevant elements of the input sequence, the attention mechanism assigns different attention weights to each input element. These attention weights designate how important or relevant a given input sequence element is at a given time step. For example, revisiting *Figure 16.2*, the words "mir, helfen, zu" may be more relevant for producing the output word "help" than the words "kannst, du, Satz."

The next subsection introduces an RNN architecture that was outfitted with an attention mechanism to help process long sequences for language translation.

The original attention mechanism for RNNs

In this subsection, we will summarize the mechanics of the attention mechanism that was originally developed for language translation and first appeared in the following paper: *Neural Machine Translation by Jointly Learning to Align and Translate* by *Bahdanau, D., Cho, K., and Bengio, Y.,* 2014, `https://arxiv.org/abs/1409.0473`.

Given an input sequence $x = \left(x^{(1)}, x^{(2)}, \ldots, x^{(T)}\right)$, the attention mechanism assigns a weight to each element $x^{(i)}$ (or, to be more specific, its hidden representation) and helps the model identify which part of the input it should focus on. For example, suppose our input is a sentence, and a word with a larger weight contributes more to our understanding of the whole sentence. The RNN with the attention mechanism shown in *Figure 16.3* (modeled after the previously mentioned paper) illustrates the overall concept of generating the second output word:

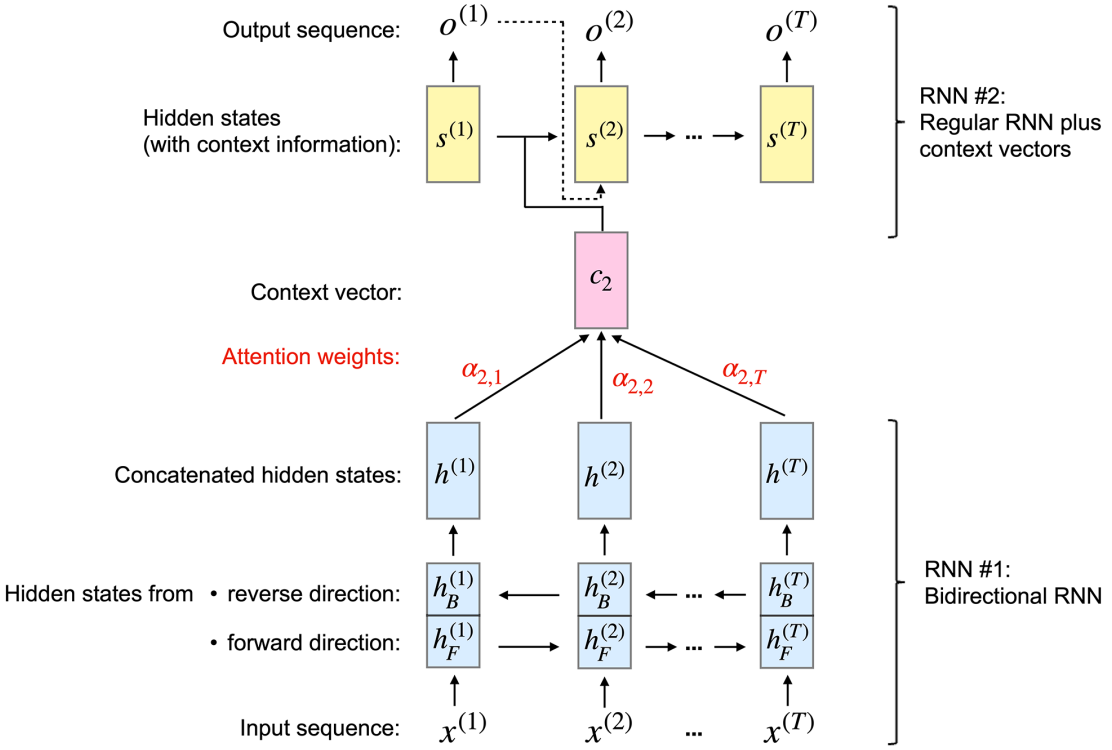

Figure 16.3: RNN with attention mechanism

The attention-based architecture depicted in the figure consists of two RNN models, which we will explain in the next subsections.

Processing the inputs using a bidirectional RNN

The first RNN (RNN #1) of the attention-based RNN in *Figure 16.3* is a bidirectional RNN that generates context vectors, c_i. You can think of a context vector as an augmented version of the input vector, $x^{(i)}$. In other words, the c_i input vector also incorporates information from all other input elements via an attention mechanism. As we can see in *Figure 16.3*, RNN #2 then uses this context vector, prepared by RNN #1, to generate the outputs. In the remainder of this subsection, we will discuss how RNN #1 works, and we will revisit RNN #2 in the next subsection.

The bidirectional RNN #1 processes the input sequence x in the regular forward direction ($1 \dots T$) as well as backward ($T \dots 1$). Parsing a sequence in the backward direction has the same effect as reversing the original input sequence—think of reading a sentence in reverse order. The rationale behind this is to capture additional information since current inputs may have a dependence on sequence elements that came either before or after it in a sentence, or both.

Consequently, from reading the input sequence twice (that is, forward and backward), we have two hidden states for each input sequence element. For instance, for the second input sequence element $x^{(2)}$, we obtain the hidden state $h_F^{(2)}$from the forward pass and the hidden state $h_B^{(2)}$ from the backward pass. These two hidden states are then concatenated to form the hidden state $h^{(2)}$. For example, if both $h_F^{(2)}$ and $h_B^{(2)}$ are 128-dimensional vectors, the concatenated hidden state $h^{(2)}$ will consist of 256 elements. We can consider this concatenated hidden state as the "annotation" of the source word since it contains the information of the *j*th word in both directions.

In the next section, we will see how these concatenated hidden states are further processed and used by the second RNN to generate the outputs.

Generating outputs from context vectors

In *Figure 16.3*, we can consider RNN #2 as the main RNN that is generating the outputs. In addition to the hidden states, it receives so-called context vectors as input. A context vector c_i is a weighted version of the concatenated hidden states, $h^{(1)} \dots h^{(T)}$, which we obtained from RNN #1 in the previous subsection. We can compute the context vector of the *i*th input as a weighted sum:

$$c_i = \sum_{j=1}^{T} \alpha_{ij} h^{(j)}$$

Here, α_{ij} represents the attention weights over the input sequence $j = 1 \dots T$ in the context of the *i*th input sequence element. Note that each *i*th input sequence element has a unique set of attention weights. We will discuss the computation of the attention weights α_{ij} in the next subsection.

For the remainder of this subsection, let us discuss how the context vectors are used via the second RNN in the preceding figure (RNN #2). Just like a vanilla (regular) RNN, RNN #2 also uses hidden states. Considering the hidden layer between the aforementioned "annotation" and final output, let us denote the hidden state at time i as $s^{(i)}$. Now, RNN #2 receives the aforementioned context vector c_i at each time step i as input.

In *Figure 16.3*, we saw that the hidden state $s^{(i)}$ depends on the previous hidden state $s^{(i-1)}$, the previous target word $y^{(i-1)}$, and the context vector $c^{(i)}$, which are used to generate the predicted output $o^{(i)}$ for target word $y^{(i)}$ at time i. Note that the sequence vector $\boldsymbol{y}$ refers to the sequence vector representing the correct translation of input sequence $\boldsymbol{x}$ that is available during training. During training, the true label (word) $y^{(i)}$ is fed into the next state $s^{(i+1)}$; since this true label information is not available for prediction (inference), we feed the predicted output $o^{(i)}$ instead, as depicted in the previous figure.

To summarize what we have just discussed above, the attention-based RNN consists of two RNNs. RNN #1 prepares context vectors from the input sequence elements, and RNN #2 receives the context vectors as input. The context vectors are computed via a weighted sum over the inputs, where the weights are the attention weights α_{ij}.The next subsection discusses how we compute these attention weights.

Computing the attention weights

Finally, let us visit the last missing piece in our puzzle—attention weights. Because these weights pairwise connect the inputs (annotations) and the outputs (contexts), each attention weight α_{ij} has two subscripts: j refers to the index position of the input and i corresponds to the output index position. The attention weight α_{ij} is a normalized version of the alignment score e_{ij}, where the alignment score evaluates how well the input around position j matches with the output at position i. To be more specific, the attention weight is computed by normalizing the alignment scores as follows:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})}$$

Note that this equation is similar to the softmax function, which we discussed in *Chapter 12*, *Parallelizing Neural Network Training with PyTorch*, in the section *Estimating class probabilities in multiclass classification via the softmax function*. Consequently, the attention weights $\alpha_{i1} \ldots \alpha_{iT}$ sum up to 1.

Now, to summarize, we can structure the attention-based RNN model into three parts. The first part computes bidirectional annotations of the input. The second part consists of the recurrent block, which is very much like the original RNN, except that it uses context vectors instead of the original input. The last part concerns the computation of the attention weights and context vectors, which describe the relationship between each pair of input and output elements.

The transformer architecture also utilizes an attention mechanism, but unlike the attention-based RNN, it solely relies on the **self-attention** mechanism and does not include the recurrent process found in the RNN. In other words, a transformer model processes the whole input sequence all at once instead of reading and processing the sequence one element at a time. In the next section, we will introduce a basic form of the self-attention mechanism before we discuss the transformer architecture in more detail in the following section.

Introducing the self-attention mechanism

In the previous section, we saw that attention mechanisms can help RNNs with remembering context when working with long sequences. As we will see in the next section, we can have an architecture entirely based on attention, without the recurrent parts of an RNN. This attention-based architecture is known as **transformer**, and we will discuss it in more detail later.

In fact, transformers can appear a bit complicated at first glance. So, before we discuss transformers in the next section, let us dive into the **self-attention** mechanism used in transformers. In fact, as we will see, this self-attention mechanism is just a different flavor of the attention mechanism that we discussed in the previous section. We can think of the previously discussed attention mechanism as an operation that connects two different modules, that is, the encoder and decoder of the RNN. As we will see, self-attention focuses only on the input and captures only dependencies between the input elements. without connecting two modules.

In the first subsection, we will introduce a basic form of self-attention without any learning parameters, which is very much like a pre-processing step to the input. Then in the second subsection, we will introduce the common version of self-attention that is used in the transformer architecture and involves learnable parameters.

Starting with a basic form of self-attention

To introduce self-attention, let's assume we have an input sequence of length T, $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$, as well as an output sequence, $\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}, \ldots, \boldsymbol{z}^{(T)}$. To avoid confusion, we will use $\boldsymbol{o}$ as the final output of the whole transformer model and $\boldsymbol{z}$ as the output of the self-attention layer because it is an intermediate step in the model.

Each ith element in these sequences, $\boldsymbol{x}^{(i)}$ and $\boldsymbol{z}^{(i)}$, are vectors of size d (that is, $\boldsymbol{x}^{(i)} \in R^d$) representing the feature information for the input at position i, which is similar to RNNs. Then, for a seq2seq task, the goal of self-attention is to model the dependencies of the current input element to all other input elements. To achieve this, self-attention mechanisms are composed of three stages. First, we derive importance weights based on the similarity between the current element and all other elements in the sequence. Second, we normalize the weights, which usually involves the use of the already familiar softmax function. Third, we use these weights in combination with the corresponding sequence elements to compute the attention value.

More formally, the output of self-attention, $\boldsymbol{z}^{(i)}$, is the weighted sum of all T input sequences, $\boldsymbol{x}^{(j)}$ (where $j = 1 \ldots T$). For instance, for the ith input element, the corresponding output value is computed as follows:

$$\boldsymbol{z}^{(i)} = \sum_{j=1}^{T} \alpha_{ij} \boldsymbol{x}^{(j)}$$

Hence, we can think of $\boldsymbol{z}^{(i)}$ as a context-aware embedding vector in input vector $\boldsymbol{x}^{(i)}$ that involves all other input sequence elements weighted by their respective attention weights. Here, the attention weights, α_{ij}, are computed based on the similarity between the current input element, $\boldsymbol{x}^{(i)}$, and all other elements in the input sequence, $\boldsymbol{x}^{(1)} \ldots \boldsymbol{x}^{(T)}$. More concretely, this similarity is computed in two steps explained in the next paragraphs.

First, we compute the dot product between the current input element, $\boldsymbol{x}^{(i)}$, and another element in the input sequence, $\boldsymbol{x}^{(j)}$:

$$\omega_{ij} = \boldsymbol{x}^{(i)\top} \boldsymbol{x}^{(j)}$$

Before we normalize the ω_{ij} values to obtain the attention weights, a_{ij}, let's illustrate how we compute the ω_{ij} values with a code example. Here, let's assume we have an input sentence "can you help me to translate this sentence" that has already been mapped to an integer representation via a dictionary as explained in *Chapter 15*, *Modeling Sequential Data Using Recurrent Neural Networks*:

```
>>> import torch
>>> sentence = torch.tensor(
>>>     [0, # can
>>>      7, # you
>>>      1, # help
>>>      2, # me
>>>      5, # to
>>>      6, # translate
>>>      4, # this
>>>      3] # sentence
>>> )

>>> sentence
tensor([0, 7, 1, 2, 5, 6, 4, 3])
```

Let's also assume that we already encoded this sentence into a real-number vector representation via an embedding layer. Here, our embedding size is 16, and we assume that the dictionary size is 10. The following code will produce the word embeddings of our eight words:

```
>>> torch.manual_seed(123)
>>> embed = torch.nn.Embedding(10, 16)
>>> embedded_sentence = embed(sentence).detach()
>>> embedded_sentence.shape
torch.Size([8, 16])
```

Now, we can compute ω_{ij} as the dot product between the *i*th and *j*th word embeddings. We can do this for all ω_{ij} values as follows:

```
>>> omega = torch.empty(8, 8)
>>> for i, x_i in enumerate(embedded_sentence):
>>>     for j, x_j in enumerate(embedded_sentence):
>>>         omega[i, j] = torch.dot(x_i, x_j)
```

While the preceding code is easy to read and understand, `for` loops can be very inefficient, so let's compute this using matrix multiplication instead:

```
>>> omega_mat = embedded_sentence.matmul(embedded_sentence.T)
```

We can use the `torch.allclose` function to check that this matrix multiplication produces the expected results. If two tensors contain the same values, `torch.allclose` returns `True`, as we can see here:

```
>>> torch.allclose(omega_mat, omega)
True
```

We have learned how to compute the similarity-based weights for the ith input and all inputs in the sequence ($\boldsymbol{x}^{(1)}$ to $\boldsymbol{x}^{(T)}$), the "raw" weights (ω_{i1} to ω_{iT}). We can obtain the attention weights, α_{ij}, by normalizing the ω_{ij} values via the familiar softmax function, as follows:

$$\alpha_{ij} = \frac{\exp(\omega_{ij})}{\sum_{j=1}^{T} \exp(\omega_{ij})} = \text{softmax}\left([\omega_{ij}]_{j=1\ldots T}\right)$$

Notice that the denominator involves a sum over all input elements ($1 \ldots T$). Hence, due to applying this softmax function, the weights will sum to 1 after this normalization, that is,

$$\sum_{j=1}^{T} \alpha_{ij} = 1$$

We can compute the attention weights using PyTorch's softmax function as follows:

```
>>> import torch.nn.functional as F
>>> attention_weights = F.softmax(omega, dim=1)
>>> attention_weights.shape
torch.Size([8, 8])
```

Note that `attention_weights` is an 8×8 matrix, where each element represents an attention weight, α_{ij}. For instance, if we are processing the ith input word, the ith row of this matrix contains the corresponding attention weights for all words in the sentence. These attention weights indicate how relevant each word is to the ith word. Hence, the columns in this attention matrix should sum to 1, which we can confirm via the following code:

```
>>> attention_weights.sum(dim=1)
tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

Now that we have seen how to compute the attention weights, let us recap and summarize the three main steps behind the self-attention operation:

1. For a given input element, $\boldsymbol{x}^{(i)}$, and each jth element in the set {1, ..., T}, compute the dot product, $\boldsymbol{x}^{(i)\top}\boldsymbol{x}^{(j)}$
2. Obtain the attention weight, α_{ij}, by normalizing the dot products using the softmax function
3. Compute the output, $\boldsymbol{z}^{(i)}$, as the weighted sum over the entire input sequence: $\boldsymbol{z}^{(i)} = \sum_{j=1}^{T} \alpha_{ij}\boldsymbol{x}^{(j)}$

These steps are further illustrated in *Figure 16.4*:

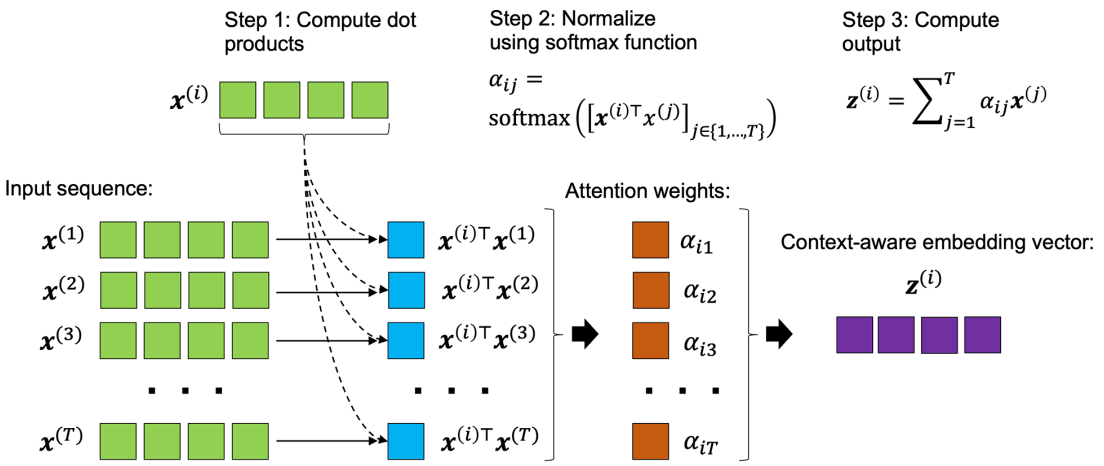

Figure 16.4: A basic self-attention process for illustration purposes

Lastly, let us see a code example for computing the context vectors, $\boldsymbol{z}^{(i)}$, as the attention-weighted sum of the inputs (step 3 in *Figure 16.4*). In particular, let's assume we are computing the context vector for the second input word, that is, $\boldsymbol{z}^{(2)}$:

```
>>> x_2 = embedded_sentence[1, :]
>>> context_vec_2 = torch.zeros(x_2.shape)
>>> for j in range(8):
...     x_j = embedded_sentence[j, :]
...     context_vec_2 += attention_weights[1, j] * x_j
>>> context_vec_2
tensor([-9.3975e-01, -4.6856e-01,  1.0311e+00, -2.8192e-01,  4.9373e-01,
-1.2896e-02, -2.7327e-01, -7.6358e-01,  1.3958e+00, -9.9543e-01,
-7.1288e-04,  1.2449e+00, -7.8077e-02,  1.2765e+00, -1.4589e+00,
-2.1601e+00])
```

Again, we can achieve this more efficiently by using matrix multiplication. Using the following code, we are computing the context vectors for all eight input words:

```
>>> context_vectors = torch.matmul(
...     attention_weights, embedded_sentence)
```

Similar to the input word embeddings stored in `embedded_sentence`, the `context_vectors` matrix has dimensionality 8×16. The second row in this matrix contains the context vector for the second input word, and we can check the implementation using `torch.allclose()` again:

```
>>> torch.allclose(context_vec_2, context_vectors[1])
True
```

As we can see, the manual `for` loop and matrix computations of the second context vector yielded the same results.

This section implemented a basic form of self-attention, and in the next section, we will modify this implementation using learnable parameter matrices that can be optimized during neural network training.

Parameterizing the self-attention mechanism: scaled dot-product attention

Now that you have been introduced to the basic concept behind self-attention, this subsection summarizes the more advanced self-attention mechanism called **scaled dot-product attention** that is used in the transformer architecture. Note that in the previous subsection, we did not involve any learnable parameters when computing the outputs. In other words, using the previously introduced basic self-attention mechanism, the transformer model is rather limited regarding how it can update or change the attention values during model optimization for a given sequence. To make the self-attention mechanism more flexible and amenable to model optimization, we will introduce three additional weight matrices that can be fit as model parameters during model training. We denote these three weight matrices as $\boldsymbol{U}_q$, $\boldsymbol{U}_k$, and $\boldsymbol{U}_v$. They are used to project the inputs into query, key, and value sequence elements, as follows:

- **Query sequence:** $\boldsymbol{q}^{(i)} = \boldsymbol{U}_q \boldsymbol{x}^{(i)}$ for $i \in [1, T]$
- **Key sequence:** $\boldsymbol{k}^{(i)} = \boldsymbol{U}_k \boldsymbol{x}^{(i)}$ for $i \in [1, T]$
- **Value sequence:** $\boldsymbol{v}^{(i)} = \boldsymbol{U}_v \boldsymbol{x}^{(i)}$ for $i \in [1, T]$

Figure 16.5 illustrates how these individual components are used to compute the context-aware embedding vector corresponding to the second input element:

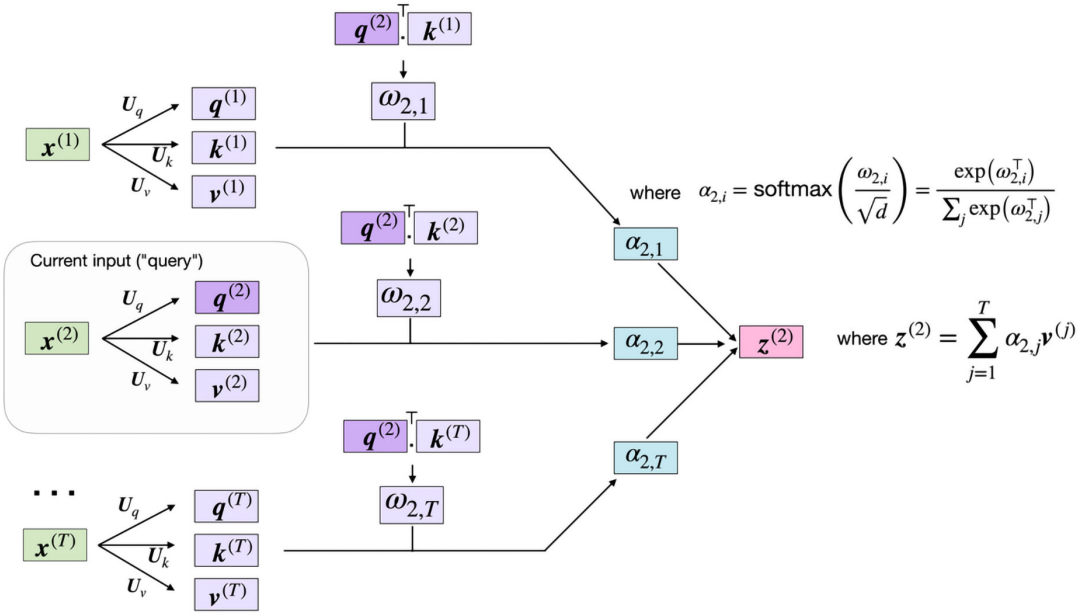

Figure 16.5: Computing the context-aware embedding vector of the second sequence element

Query, key, and value terminology

The terms query, key, and value that were used in the original transformer paper are inspired by information retrieval systems and databases. For example, if we enter a query, it is matched against the key values for which certain values are retrieved.

Here, both $\boldsymbol{q}^{(i)}$ and $\boldsymbol{k}^{(i)}$ are vectors of size d_k. Therefore, the projection matrices $\boldsymbol{U}_q$ and $\boldsymbol{U}_k$ have the shape $d_k \times d$, while $\boldsymbol{U}_v$ has the shape $d_v \times d$. (Note that d is the dimensionality of each word vector, $\boldsymbol{x}^{(i)}$.) For simplicity, we can design these vectors to have the same shape, for example, using $d_k = d_v = d$. To provide additional intuition via code, we can initialize these projection matrices as follows:

```
>>> torch.manual_seed(123)
>>> d = embedded_sentence.shape[1]
>>> U_query = torch.rand(d, d)
>>> U_key = torch.rand(d, d)
>>> U_value = torch.rand(d, d)
```

Using the query projection matrix, we can then compute the query sequence. For this example, consider the second input element, $\boldsymbol{x}^{(i)}$, as our query, as illustrated in *Figure 16.5*:

```
>>> x_2 = embedded_sentence[1]
>>> query_2 = U_query.matmul(x_2)
```

In a similar fashion, we can compute the key and value sequences, $\boldsymbol{k}^{(i)}$and $\boldsymbol{v}^{(i)}$:

```
>>> key_2 = U_key.matmul(x_2)
>>> value_2 = U_value.matmul(x_2)
```

However, as we can see from *Figure 16.5*, we also need the key and value sequences for all other input elements, which we can compute as follows:

```
>>> keys = U_key.matmul(embedded_sentence.T).T
>>> values = U_value.matmul(embedded_sentence.T).T
```

In the key matrix, the *i*th row corresponds to the key sequence of the *i*th input element, and the same applies to the value matrix. We can confirm this by using `torch.allclose()` again, which should return `True`:

```
>>> keys = U_key.matmul(embedded_sentence.T).T
>>> torch.allclose(key_2, keys[1])
>>> values = U_value.matmul(embedded_sentence.T).T
>>> torch.allclose(value_2, values[1])
```

In the previous section, we computed the unnormalized weights, ω_{ij}, as the pairwise dot product between the given input sequence element, $\boldsymbol{x}^{(i)}$, and the *j*th sequence element, $\boldsymbol{x}^{(j)}$. Now, in this parameterized version of self-attention, we compute ω_{ij} as the dot product between the query and key:

$$\omega_{ij} = \boldsymbol{q}^{(i)^\top}\boldsymbol{k}^{(j)}$$

For example, the following code computes the unnormalized attention weight, ω_{23}, that is, the dot product between our query and the third input sequence element:

```
>>> omega_23 = query_2.dot(keys[2])
>>> omega_23
tensor(14.3667)
```

Since we will be needing these later, we can scale up this computation to all keys:

```
>>> omega_2 = query_2.matmul(keys.T)
>>> omega_2
tensor([-25.1623,   9.3602,  14.3667,  32.1482,  53.8976,  46.6626,  -1.2131,
-32.9391])
```

The next step in self-attention is to go from the unnormalized attention weights, ω_{ij}, to the normalized attention weights, α_{ij}, using the softmax function. We can then further use $1/\sqrt{m}$ to scale ω_{ij} before normalizing it via the softmax function, as follows:

$$\alpha_{ij} = \text{softmax}\left(\frac{\omega_{ij}}{\sqrt{m}}\right)$$

Note that scaling ω_{ij} by $1/\sqrt{m}$, where typically $m = d_k$, ensures that the Euclidean length of the weight vectors will be approximately in the same range.

The following code is for implementing this normalization to compute the attention weights for the entire input sequence with respect to the second input element as the query:

```
>>> attention_weights_2 = F.softmax(omega_2 / d**0.5, dim=0)
>>> attention_weights_2
tensor([2.2317e-09, 1.2499e-05, 4.3696e-05, 3.7242e-03, 8.5596e-01, 1.4025e-01,
8.8896e-07, 3.1936e-10])
```

Finally, the output is a weighted average of value sequences: $\boldsymbol{z}^{(i)} = \sum_{j=1}^{T} \alpha_{ij} \boldsymbol{v}^{(j)}$, which can be implemented as follows:

```
>>> context_vector_2 = attention_weights_2.matmul(values)
>>> context_vector_2
tensor([-1.2226, -3.4387, -4.3928, -5.2125, -1.1249, -3.3041,
-1.4316, -3.2765, -2.5114, -2.6105, -1.5793, -2.8433, -2.4142,
-0.3998, -1.9917, -3.3499])
```

In this section, we introduced a self-attention mechanism with trainable parameters that lets us compute context-aware embedding vectors by involving all input elements, which are weighted by their respective attention scores. In the next section, we will learn about the transformer architecture, a neural network architecture centered around the self-attention mechanism introduced in this section.

Attention is all we need: introducing the original transformer architecture

Interestingly, the original transformer architecture is based on an attention mechanism that was first used in an RNN. Originally, the intention behind using an attention mechanism was to improve the text generation capabilities of RNNs when working with long sentences. However, only a few years after experimenting with attention mechanisms for RNNs, researchers found that an attention-based language model was even more powerful when the recurrent layers were deleted. This led to the development of the **transformer architecture**, which is the main topic of this chapter and the remaining sections.

The transformer architecture was first proposed in the NeurIPS 2017 paper *Attention Is All You Need* by *A. Vaswani* and colleagues (`https://arxiv.org/abs/1706.03762`). Thanks to the self-attention mechanism, a transformer model can capture long-range dependencies among the elements in an input sequence—in an NLP context; for example, this helps the model better "understand" the meaning of an input sentence.

Although this transformer architecture was originally designed for language translation, it can be generalized to other tasks such as English constituency parsing, text generation, and text classification. Later, we will discuss popular language models, such as BERT and GPT, which were derived from this original transformer architecture. *Figure 16.6*, which we adapted from the original transformer paper, illustrates the main architecture and components we will be discussing in this section:

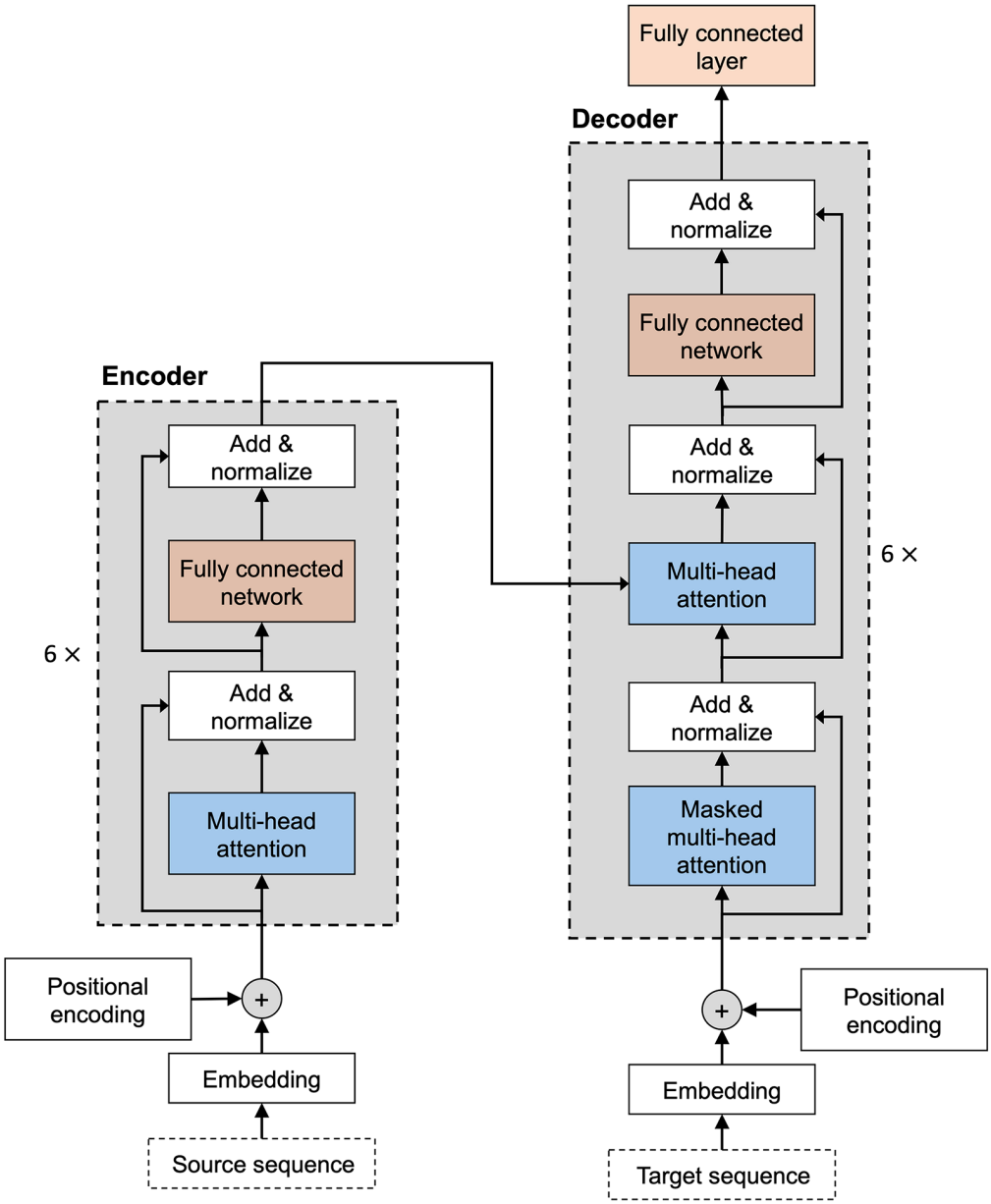

Figure 16.6: The original transformer architecture

In the following subsections, we go over this original transformer model step by step, by decomposing it into two main blocks: an encoder and a decoder. The encoder receives the original sequential input and encodes the embeddings using a multi-head self-attention module. The decoder takes in the processed input and outputs the resulting sequence (for instance, the translated sentence) using a *masked* form of self-attention.

Encoding context embeddings via multi-head attention

The overall goal of the **encoder** block is to take in a sequential input $\boldsymbol{X} = \left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(T)}\right)$ and map it into a continuous representation $\boldsymbol{Z} = \left(\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}, \ldots, \boldsymbol{z}^{(T)}\right)$ that is then passed on to the decoder.

The encoder is a stack of six identical layers. Six is not a magic number here but merely a hyperparameter choice made in the original transformer paper. You can adjust the number of layers according to the model performance. Inside each of these identical layers, there are two sublayers: one computes the multi-head self-attention, which we will discuss below, and the other one is a fully connected layer, which you have already encountered in previous chapters.

Let's first talk about the **multi-head self-attention**, which is a simple modification of scaled dot-product attention covered earlier in this chapter. In the scaled dot-product attention, we used three matrices (corresponding to query, value, and key) to transform the input sequence. In the context of multi-head attention, we can think of this set of three matrices as one attention *head*. As indicated by its name, in multi-head attention, we now have multiple of such heads (sets of query, value, and key matrices) similar to how convolutional neural networks can have multiple kernels.

To explain the concept of multi-head self-attention with h heads in more detail, let's break it down into the following steps.

First, we read in the sequential input $\boldsymbol{X} = \left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(T)}\right)$. Suppose each element is embedded by a vector of length d. Here, the input can be embedded into a $T \times d$ matrix. Then, we create h sets of the query, key, and value learning parameter matrices:

- $\boldsymbol{U}_{q_1}, \boldsymbol{U}_{k_1}, \boldsymbol{U}_{v_1}$
- $\boldsymbol{U}_{q_2}, \boldsymbol{U}_{k_2}, \boldsymbol{U}_{v_2}$
- ...
- $\boldsymbol{U}_{q_h}, \boldsymbol{U}_{k_h}, \boldsymbol{U}_{v_h}$

Because we are using these weight matrices to project each element $\boldsymbol{x}^{(i)}$ for the required dimension-matching in the matrix multiplications, both $\boldsymbol{U}_{q_j}$ and $\boldsymbol{U}_{k_j}$ have the shape $d_k \times d$, and $\boldsymbol{U}_{v_j}$ has the shape $d_v \times d$. As a result, both resulting sequences, query and key, have length d_k, and the resulting value sequence has length d_v. In practice, people often choose $d_k = d_v = m$ for simplicity.

To illustrate the multi-head self-attention stack in code, first consider how we created the single query projection matrix in the previous subsection, *Parameterizing the self-attention mechanism: scaled dot-product attention*:

```
>>> torch.manual_seed(123)
>>> d = embedded_sentence.shape[1]
>>> one_U_query = torch.rand(d, d)
```

Now, assume we have eight attention heads similar to the original transformer, that is, $h = 8$:

```
>>> h = 8
>>> multihead_U_query = torch.rand(h, d, d)
>>> multihead_U_key = torch.rand(h, d, d)
>>> multihead_U_value = torch.rand(h, d, d)
```

As we can see in the code, multiple attention heads can be added by simply adding an additional dimension.

Splitting data across multiple attention heads

In practice, rather than having a separate matrix for each attention head, transformer implementations use a single matrix for all attention heads. The attention heads are then organized into logically separate regions in this matrix, which can be accessed via Boolean masks. This makes it possible to implement multi-head attention more efficiently because multiple matrix multiplications can be implemented as a single matrix multiplication instead. However, for simplicity, we are omitting this implementation detail in this section.

After initializing the projection matrices, we can compute the projected sequences similar to how it's done in scaled dot-product attention. Now, instead of computing one set of query, key, and value sequences, we need to compute h sets of them. More formally, for example, the computation involving the query projection for the ith data point in the jth head can be written as follows:

$$\boldsymbol{q}_j^{(i)} = \boldsymbol{U}_{\boldsymbol{q}_j}\boldsymbol{x}^{(i)}$$

We then repeat this computation for all heads $j \in \{1, \dots h\}$.

In code, this looks like the following for the second input word as the query:

```
>>> multihead_query_2 = multihead_U_query.matmul(x_2)
>>> multihead_query_2.shape
torch.Size([8, 16])
```

The `multihead_query_2` matrix has eight rows, where each row corresponds to the jth attention head.

Similarly, we can compute key and value sequences for each head:

```
>>> multihead_key_2 = multihead_U_key.matmul(x_2)
>>> multihead_value_2 = multihead_U_value.matmul(x_2)
>>> multihead_key_2[2]
tensor([-1.9619, -0.7701, -0.7280, -1.6840, -1.0801, -1.6778,  0.6763,  0.6547,
         1.4445, -2.7016, -1.1364, -1.1204, -2.4430, -0.5982, -0.8292, -1.4401])
```

The code output shows the key vector of the second input element via the third attention head.

However, remember that we need to repeat the key and value computations for all input sequence elements, not just `x_2`—we need this to compute self-attention later. A simple and illustrative way to do this is by expanding the input sequence embeddings to size 8 as the first dimension, which is the number of attention heads. We use the `.repeat()` method for this:

```
>>> stacked_inputs = embedded_sentence.T.repeat(8, 1, 1)
>>> stacked_inputs.shape
torch.Size([8, 16, 8])
```

Then, we can have a batch matrix multiplication, via `torch.bmm()`, with the attention heads to compute all keys:

```
>>> multihead_keys = torch.bmm(multihead_U_key, stacked_inputs)
>>> multihead_keys.shape
torch.Size([8, 16, 8])
```

In this code, we now have a tensor that refers to the eight attention heads in its first dimension. The second and third dimensions refer to the embedding size and the number of words, respectively. Let us swap the second and third dimensions so that the keys have a more intuitive representation, that is, the same dimensionality as the original input sequence `embedded_sentence`:

```
>>> multihead_keys = multihead_keys.permute(0, 2, 1)
>>> multihead_keys.shape
torch.Size([8, 8, 16])
```

After rearranging, we can access the second key value in the second attention head as follows:

```
>>> multihead_keys[2, 1]
tensor([-1.9619, -0.7701, -0.7280, -1.6840, -1.0801, -1.6778,  0.6763,  0.6547,
         1.4445, -2.7016, -1.1364, -1.1204, -2.4430, -0.5982, -0.8292, -1.4401])
```

We can see that this is the same key value that we got via `multihead_key_2[2]` earlier, which indicates that our complex matrix manipulations and computations are correct. So, let's repeat it for the value sequences:

```
>>> multihead_values = torch.matmul(
        multihead_U_value, stacked_inputs)
>>> multihead_values = multihead_values.permute(0, 2, 1)
```

We follow the steps of the single head attention calculation to calculate the context vectors as described in the *Parameterizing the self-attention mechanism: scaled dot-product attention* section. We will skip the intermediate steps for brevity and assume that we have computed the context vectors for the second input element as the query and the eight different attention heads, which we represent as `multihead_z_2` via random data:

```
>>> multihead_z_2 = torch.rand(8, 16)
```

Note that the first dimension indexes over the eight attention heads, and the context vectors, similar to the input sentences, are 16-dimensional vectors. If this appears complicated, think of `multihead_z_2` as eight copies of the $z^{(2)}$ shown in *Figure 16.5*; that is, we have one $z^{(2)}$ for each of the eight attention heads.

Then, we concatenate these vectors into one long vector of length $d_v \times h$ and use a linear projection (via a fully connected layer) to map it back to a vector of length d_v. This process is illustrated in *Figure 16.7*:

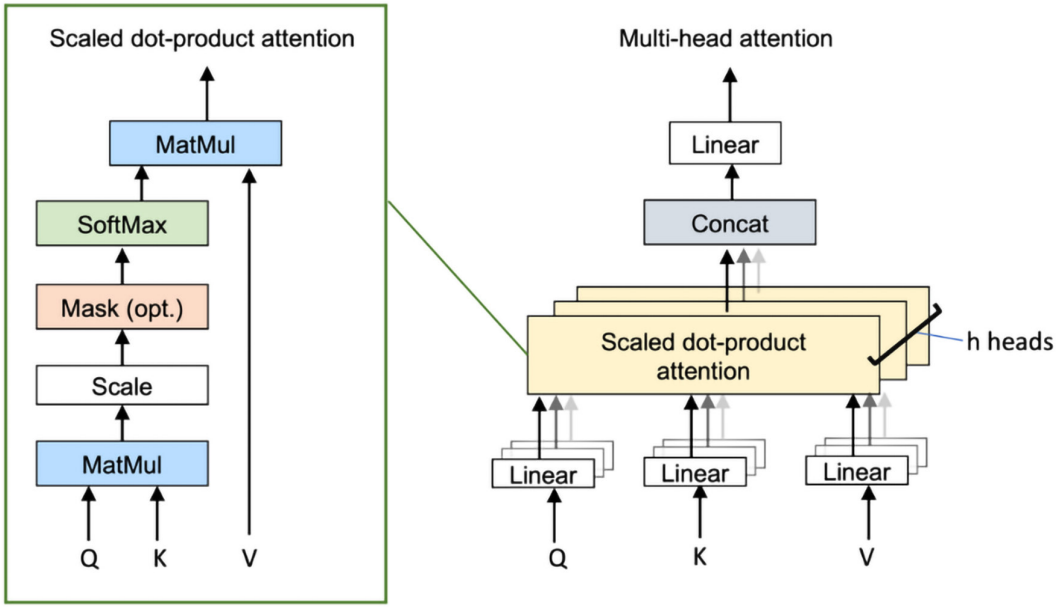

Figure 16.7: Concatenating the scaled dot-product attention vectors into one vector and passing it through a linear projection

In code, we can implement the concatenation and squashing as follows:

```
>>> linear = torch.nn.Linear(8*16, 16)
>>> context_vector_2 = linear(multihead_z_2.flatten())
>>> context_vector_2.shape
torch.Size([16])
```

To summarize, multi-head self-attention is repeating the scaled dot-product attention computation multiple times in parallel and combining the results. It works very well in practice because the multiple heads help the model to capture information from different parts of the input, which is very similar to how the multiple kernels produce multiple channels in a convolutional network, where each channel can capture different feature information. Lastly, while multi-head attention sounds computationally expensive, note that the computation can all be done in parallel because there are no dependencies between the multiple heads.

Learning a language model: decoder and masked multi-head attention

Similar to the encoder, the **decoder** also contains several repeated layers. Besides the two sublayers that we have already introduced in the previous encoder section (the multi-head self-attention layer and fully connected layer), each repeated layer also contains a masked multi-head attention sublayer.

Masked attention is a variation of the original attention mechanism, where masked attention only passes a limited input sequence into the model by "masking" out a certain number of words. For example, if we are building a language translation model with a labeled dataset, at sequence position i during the training procedure, we only feed in the correct output words from positions 1,...,i-1. All other words (for instance, those that come after the current position) are hidden from the model to prevent the model from "cheating." This is also consistent with the nature of text generation: although the true translated words are known during training, we know nothing about the ground truth in practice. Thus, we can only feed the model the solutions to what it has already generated, at position i.

Figure 16.8 illustrates how the layers are arranged in the decoder block:

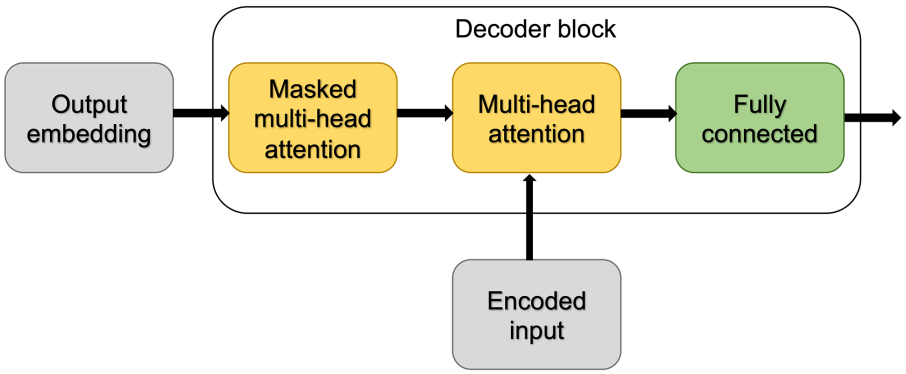

Figure 16.8: Layer arrangement in the decoder part

First, the previous output words (output embeddings) are passed into the masked multi-head attention layer. Then, the second layer receives both the encoded inputs from the encoder block and the output of the masked multi-head attention layer into a multi-head attention layer. Finally, we pass the multi-head attention outputs into a fully connected layer that generates the overall model output: a probability vector corresponding to the output words.

Note that we can use an argmax function to obtain the predicted words from these word probabilities similar to the overall approach we took in the recurrent neural network in *Chapter 15*, *Modeling Sequential Data Using Recurrent Neural Networks*.

Comparing the decoder with the encoder block, the main difference is the range of sequence elements that the model can attend to. In the encoder, for each given word, the attention is calculated across all the words in a sentence, which can be considered as a form of bidirectional input parsing. The decoder also receives the bidirectionally parsed inputs from the encoder. However, when it comes to the output sequence, the decoder only considers those elements that are preceding the current input position, which can be interpreted as a form of unidirectional input parsing.

Implementation details: positional encodings and layer normalization

In this subsection, we will discuss some of the implementation details of transformers that we have glanced over so far but are worth mentioning.

First, let's consider the **positional encodings** that were part of the original transformer architecture from *Figure 16.6*. Positional encodings help with capturing information about the input sequence ordering and are a crucial part of transformers because both scaled dot-product attention layers and fully connected layers are permutation-invariant. This means, without positional encoding, the order of words is ignored and does not make any difference to the attention-based encodings. However, we know that word order is essential for understanding a sentence. For example, consider the following two sentences:

1. Mary gives John a flower
2. John gives Mary a flower

The words occurring in the two sentences are exactly the same; the meanings, however, are very different.

Transformers enable the same words at different positions to have slightly different encodings by adding a vector of small values to the input embeddings at the beginning of the encoder and decoder blocks. In particular, the original transformer architecture uses a so-called sinusoidal encoding:

$$PE_{(i,2k)} = \sin(pos/10000^{2k/d_{\text{model}}})$$

$$PE_{(i,2k+1)} = \cos(pos/10000^{2k/d_{\text{model}}})$$

Here i is the position of the word and k denotes the length of the encoding vector, where we choose k to have the same dimension as the input word embeddings so that the positional encoding and word embeddings can be added together. Sinusoidal functions are used to prevent positional encodings from becoming too large. For instance, if we used absolute position 1,2,3..., n to be positional encodings, they would dominate the word encoding and make the word embedding values negligible.

In general, there are two types of positional encodings, an *absolute* one (as shown in the previous formula) and a *relative* one. The former will record absolute positions of words and is sensitive to word shifts in a sentence. That is to say, absolute positional encodings are fixed vectors for each given position. On the other hand, relative encodings only maintain the relative position of words and are invariant to sentence shift.

Next, let's look at the **layer normalization** mechanism, which was first introduced by J. Ba, J.R. Kiros, and G.E. Hinton in 2016 in the same-named paper *Layer Normalization* (URL: `https://arxiv.org/abs/1607.06450`). While batch normalization, which we will discuss in more detail in *Chapter 17*, *Generative Adversarial Networks for Synthesizing New Data*, is a popular choice in computer vision contexts, layer normalization is the preferred choice in NLP contexts, where sentence lengths can vary. *Figure 16.9* illustrates the main differences of layer and batch normalization side by side:

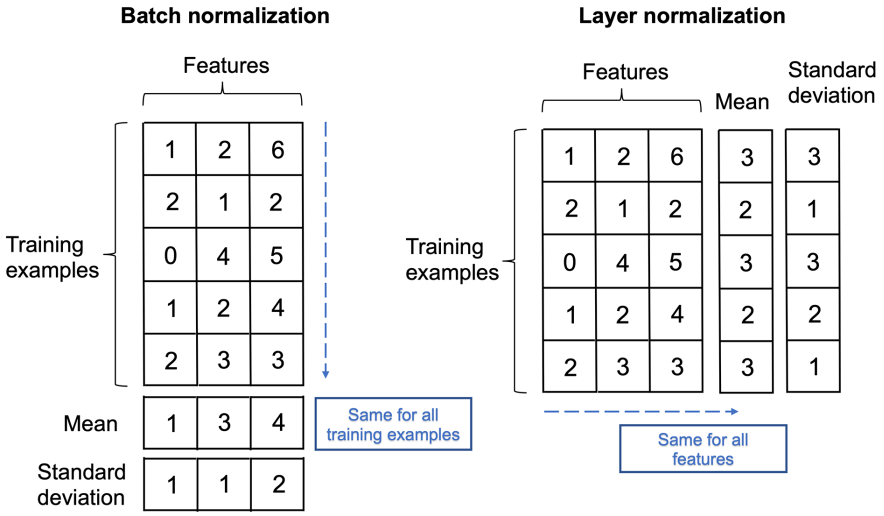

Figure 16.9: A comparison of batch and layer normalization

While layer normalization is traditionally performed across all elements in a given feature for each feature independently, the layer normalization used in transformers extends this concept and computes the normalization statistics across all feature values independently for each training example.

Since layer normalization computes mean and standard deviation for each training example, it relaxes minibatch size constraints or dependencies. In contrast to batch normalization, layer normalization is thus capable of learning from data with small minibatch sizes and varying lengths. However, note that the original transformer architecture does not have varying-length inputs (sentences are padded when needed), and unlike RNNs, there is no recurrence in the model. So, how can we then justify the use of layer normalization over batch normalization? Transformers are usually trained on very large text corpora, which requires parallel computation; this can be challenging to achieve with batch normalization, which has a dependency between training examples. Layer normalization has no such dependency and is thus a more natural choice for transformers.

Building large-scale language models by leveraging unlabeled data

In this section, we will discuss popular large-scale transformer models that emerged from the original transformer. One common theme among these transformers is that they are pre-trained on very large, unlabeled datasets and then fine-tuned for their respective target tasks. First, we will introduce the common training procedure of transformer-based models and explain how it is different from the original transformer. Then, we will focus on popular large-scale language models including **Generative Pre-trained Transformer** (**GPT**), **Bidirectional Encoder Representations from Transformers** (**BERT**), and **Bidirectional and Auto-Regressive Transformers** (**BART**).

Pre-training and fine-tuning transformer models

In an earlier section, *Attention is all we need: introducing the original transformer architecture*, we discussed how the original transformer architecture can be used for language translation. Language translation is a supervised task and requires a labeled dataset, which can be very expensive to obtain. The lack of large, labeled datasets is a long-lasting problem in deep learning, especially for models like the transformer, which are even more data hungry than other deep learning architectures. However, given that large amounts of text (books, websites, and social media posts) are generated every day, an interesting question is how we can use such unlabeled data for improving the model training.

The answer to whether we can leverage unlabeled data in transformers is *yes*, and the trick is a process called **self-supervised learning**: we can generate "labels" from supervised learning from plain text itself. For example, given a large, unlabeled text corpus, we train the model to perform **next-word prediction**, which enables the model to learn the probability distribution of words and can form a strong basis for becoming a powerful language model.

Self-supervised learning is traditionally also referred to as **unsupervised pre-training** and is essential for the success of modern transformer-based models. The "unsupervised" in unsupervised pre-training supposedly refers to the fact that we use unlabeled data; however, since we use the structure of the data to generate labels (for example, the next-word prediction task mentioned previously), it is still a supervised learning process.

To elaborate a bit further on how unsupervised pre-training and next-word prediction works, if we have a sentence containing n words, the pre-training procedure can be decomposed into the following three steps:

1. At time *step 1*, feed in the ground-truth words 1, ..., i-1.
2. Ask the model to predict the word at position i and compare it with the ground-truth word i.
3. Update the model and time step, i:= i+1. Go back to step 1 and repeat until all words are processed.

We should note that in the next iteration, we always feed the model the ground-truth (correct) words instead of what the model has generated in the previous round.

The main idea of pre-training is to make use of plain text and then transfer and fine-tune the model to perform some specific tasks for which a (smaller) labeled dataset is available. Now, there are many different types of pre-training techniques. For example, the previously mentioned next-word prediction task can be considered as a unidirectional pre-training approach. Later, we will introduce additional pre-training techniques that are utilized in different language models to achieve various functionalities.

A complete training procedure of a transformer-based model consists of two parts: (1) pre-training on a large, unlabeled dataset and (2) training (that is, fine-tuning) the model for specific downstream tasks using a labeled dataset. In the first step, the pre-trained model is not designed for any specific task but rather trained as a "general" language model. Afterward, via the second step, it can be generalized to any customized task via regular supervised learning on a labeled dataset.

With the representations that can be obtained from the pre-trained model, there are mainly two strategies for transferring and adopting a model to a specific task: (1) a **feature-based approach** and (2) a **fine-tuning approach**. (Here, we can think of these representations as the hidden layer activations of the last layers of a model.)

The feature-based approach uses the pre-trained representations as additional features to a labeled dataset. This requires us to learn how to extract sentence features from the pre-trained model. An early model that is well-known for this feature extraction approach is **ELMo** (**Embeddings from Language Models**) proposed by Peters and colleagues in 2018 in the paper *Deep Contextualized Word Representations* (URL: `https://arxiv.org/abs/1802.05365`). ELMo is a pre-trained bidirectional language model that masks words at a certain rate. In particular, it randomly masks 15 percent of the input words during pre-training, and the modeling task is to fill in these blanks, that is, predicting the missing (masked) words. This is different from the unidirectional approach we introduced previously, which hides all the future words at time step i. Bidirectional masking enables a model to learn from both ends and can thus capture more holistic information about a sentence. The pre-trained ELMo model can generate high-quality sentence representations that, later on, serve as input features for specific tasks. In other words, we can think of the feature-based approach as a model-based feature extraction technique similar to principal component analysis, which we covered in *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

The fine-tuning approach, on the other hand, updates the pre-trained model parameters in a regular supervised fashion via backpropagation. Unlike the feature-based method, we usually also add another fully connected layer to the pre-trained model, to accomplish certain tasks such as classification, and then update the whole model based on the prediction performance on the labeled training set. One popular model that follows this approach is BERT, a large-scale transformer model pre-trained as a bidirectional language model. We will discuss BERT in more detail in the following subsections. In addition, in the last section of this chapter, we will see a code example showing how to fine-tune a pre-trained BERT model for sentiment classification using the movie review dataset we worked with in *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*, and *Chapter 15*, *Modeling Sequential Data Using Recurrent Neural Networks*.

Before we move on to the next section and start our discussion of popular transformer-based language models, the following figure summarizes the two stages of training transformer models and illustrates the difference between the feature-based and fine-tuning approaches:

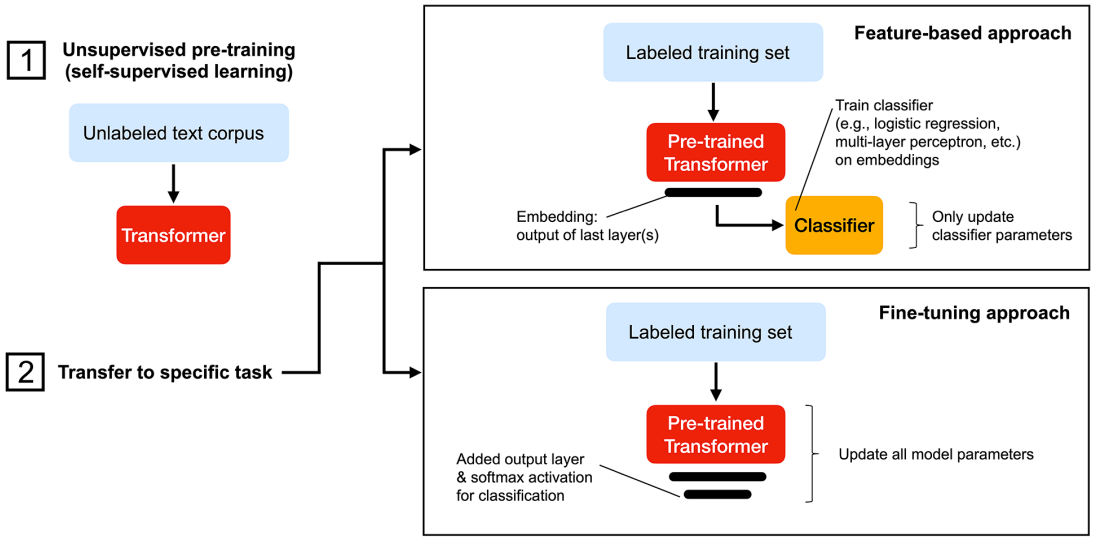

Figure 16.10: The two main ways to adopt a pre-trained transformer for downstream tasks

Leveraging unlabeled data with GPT

The **Generative Pre-trained Transformer** (**GPT**) is a popular series of large-scale language models for generating text developed by OpenAI. The most recent model, GPT-3, which was released in May 2020 (*Language Models are Few-Shot Learners*), is producing astonishing results. The quality of the text generated by GPT-3 is very hard to distinguish from human-generated texts. In this section, we are going to discuss how the GPT model works on a high level, and how it has evolved over the years.

As listed in *Table 16.1*, one obvious evolution within the GPT model series is the number of parameters:

Model	Release year	Number of parameters	Title	Paper link
GPT-1	2018	110 million	Improving Language Understanding by Generative Pre-Training	`https://www.cs.ubc.ca/~amuham01/LING530/papers/radford2018improving.pdf`
GPT-2	2019	1.5 billion	Language Models are Unsupervised Multitask Learners	`https://www.semanticscholar.org/paper/Language-Models-are-Unsupervised-Multitask-Learners-Radford-Wu/9405cc0d6169988371b2755e573cc28650d14dfe`
GPT-3	2020	175 billion	Language Models are Few-Shot Learners	`https://arxiv.org/pdf/2005.14165.pdf`

Table 16.1: Overview of the GPT models

But let's not get ahead of ourselves, and take a closer look at the GPT-1 model first, which was released in 2018. Its training procedure can be decomposed into two stages:

1. Pre-training on a large amount of unlabeled plain text
2. Supervised fine-tuning

As *Figure 16.11* (adapted from the GPT-1 paper) illustrates, we can consider GPT-1 as a transformer consisting of (1) a decoder (and without an encoder block) and (2) an additional layer that is added later for the supervised fine-tuning to accomplish specific tasks:

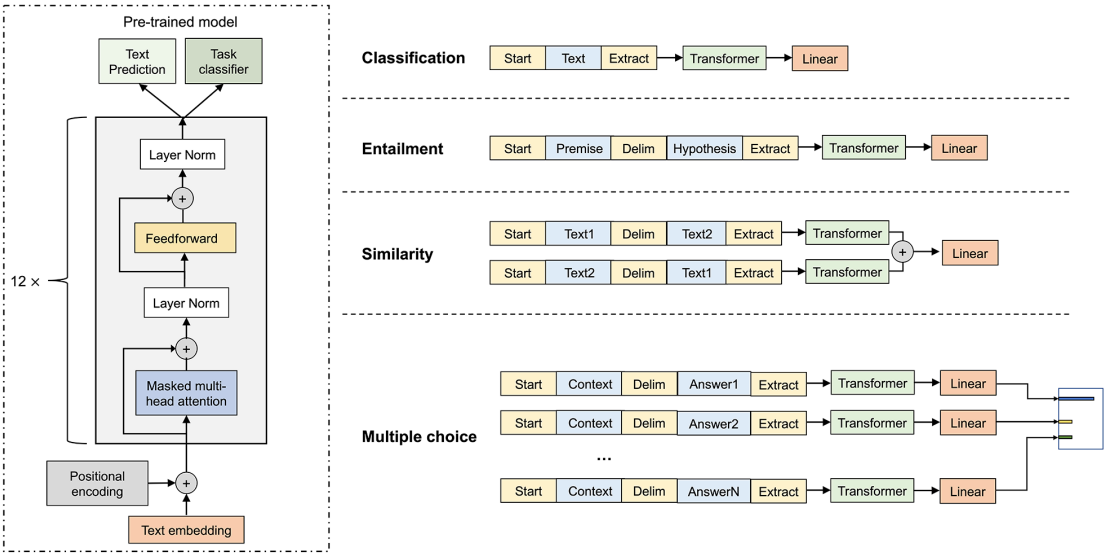

Figure 16.11: The GPT-1 transformer

In the figure, note that if our task is *Text Prediction* (predicting the next word), then the model is ready after the pre-training step. Otherwise, for example, if our task is related to classification or regression, then supervised fine-tuning is required.

During pre-training, GPT-1 utilizes a transformer decoder structure, where, at a given word position, the model only relies on preceding words to predict the next word. GPT-1 utilizes a unidirectional self-attention mechanism, as opposed to a bidirectional one as in BERT (which we will cover later in this chapter), because GPT-1 is focused on text generation rather than classification. During text generation, it produces words one by one with a natural left-to-right direction. There is one other aspect worth highlighting here: during the training procedure, for each position, we always feed the correct words from the previous positions to the model. However, during inference, we just feed the model whatever words it has generated to be able to generate new texts.

After obtaining the pre-trained model (the block in the previous figure labeled as *Transformer*), we then insert it between the input pre-processing block and a linear layer, where the linear layer serves as an output layer (similar to previous deep neural network models we discussed earlier in this book). For classification tasks, fine-tuning is as simple as first tokenizing the input and then feeding it into the pre-trained model and the newly added linear layer, which is followed by a softmax activation function. However, for more complicated tasks such as question answering, inputs are organized in a certain format that is not necessarily matching the pre-trained model, which requires an extra processing step customized for each task. Readers who are interested in specific modifications are encouraged to read the GPT-1 paper for additional details (the link is provided in the previous table).

GPT-1 also performs surprisingly well on **zero-shot tasks**, which proves its ability to be a general language model that can be customized for different types of tasks with minimal task-specific fine-tuning. Zero-shot learning generally describes a special circumstance in machine learning where during testing and inference, the model is required to classify samples from classes that were not observed during training. In the context of GPT, the zero-shot setting refers to unseen tasks.

GPT's adaptability inspired researchers to get rid of the task-specific input and model setup, which led to the development of GPT-2. Unlike its predecessor, GPT-2 does not require any additional modification during the input or fine-tuning stages anymore. Instead of rearranging sequences to match the required format, GPT-2 can distinguish between different types of inputs and perform the corresponding downstream tasks with minor hints, the so-called "contexts." This is achieved by modeling output probabilities conditioned on both input and task type, $p(output|input, task)$, instead of only conditioning on the input. For example, the model is expected to recognize a translation task if the context includes `translate to French, English text, French text`.

This sounds much more "artificially intelligent" than GPT and is indeed the most noticeable improvement besides the model size. Just as the title of its corresponding paper indicates (*Language Models are Unsupervised Multitask Learners*), an unsupervised language model may be key to zero-shot learning, and GPT-2 makes full use of zero-shot task transfer to build this multi-task learner.

Compared with GPT-2, GPT-3 is less "ambitious" in the sense that it shifts the focus from zero- to one-shot and **few-shot learning** via in-context learning. While providing no task-specific training examples seems to be too strict, few-shot learning is not only more realistic but also more human-like: humans usually need to see a few examples to be able to learn a new task. Just as its name suggests, few-shot learning means that the model sees a *few* examples of the task while one-shot learning is restricted to exactly one example.

Figure 16.12 illustrates the difference between zero-shot, one-shot, few-shot, and fine-tuning procedures:

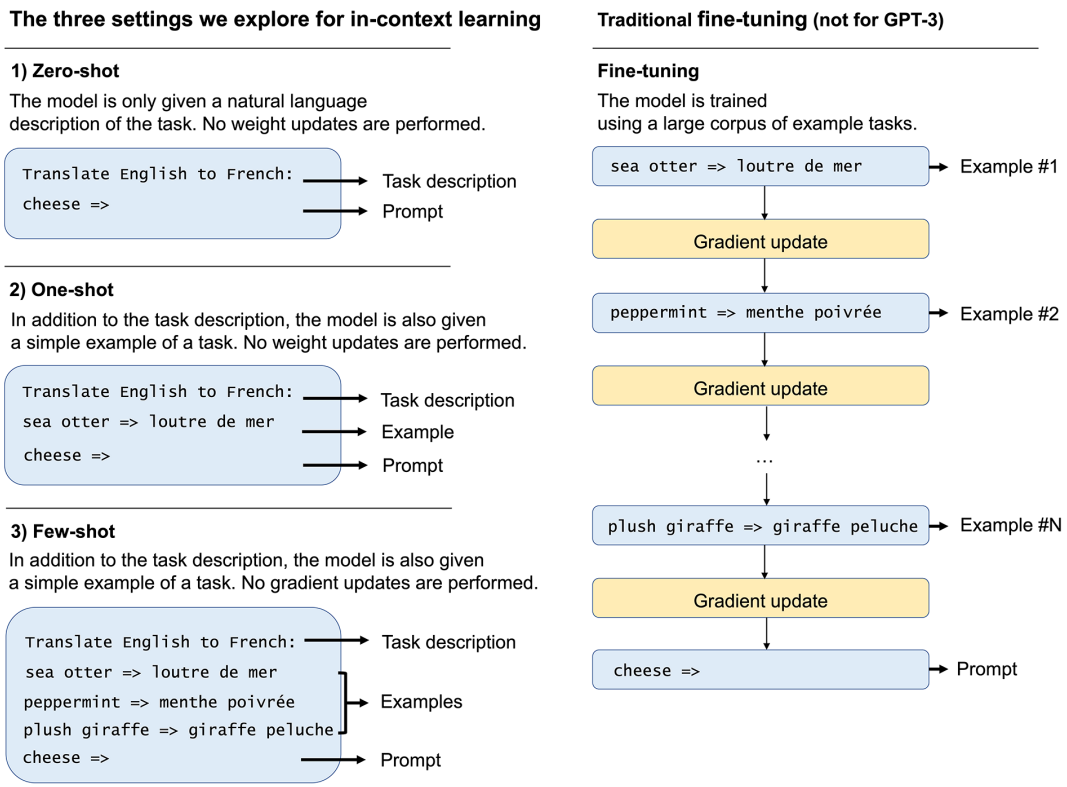

Figure 16.12: A comparison of zero-shot, one-shot, and few-shot learning

The model architecture of GPT-3 is pretty much the same as GPT-2 except for the 100-fold parameter size increase and the use of a sparse transformer. In the original (dense) attention mechanism we discussed earlier, each element attends to all other elements in the input, which scales with $O(n^2)$ complexity. **Sparse attention** improves the efficiency by only attending to a subset of elements with limited size, normally proportional to $n^{1/p}$. Interested readers can learn more about the specific subset selection by visiting the sparse transformer paper: *Generating Long Sequences with Sparse Transformers by Rewon Child et al.* 2019 (URL: `https://arxiv.org/abs/1904.10509`).

Using GPT-2 to generate new text

Before we move on to the next transformer architecture, let us take a look at how we can use the latest GPT models to generate new text. Note that GPT-3 is still relatively new and is currently only available as a beta version via the OpenAI API at `https://openai.com/blog/openai-api/`. However, an implementation of GPT-2 has been made available by Hugging Face (a popular NLP and machine learning company; `http://huggingface.co`), which we will use.

We will be accessing GPT-2 via `transformers`, which is a very comprehensive Python library created by Hugging Face that provides various transformer-based models for pre-training and fine-tuning. Users can also discuss and share their customized models on the forum. Feel free to check out and engage with the community if you are interested: `https://discuss.huggingface.co`.

Installing transformers version 4.9.1

Because this package is evolving rapidly, you may not be able to replicate the results in the following subsections. For reference, this tutorial uses version 4.9.1 released in June 2021. To install the version we used in this book, you can execute the following command in your terminal to install it from PyPI:

```
pip install transformers==4.9.1
```

We also recommend checking the latest instructions on the official installation page:

`https://huggingface.co/transformers/installation.html`

Once we have installed the `transformers` library, we can run the following code to import a pre-trained GPT model that can generate new text:

```
>>> from transformers import pipeline, set_seed
>>> generator = pipeline('text-generation', model='gpt2')
```

Then, we can prompt the model with a text snippet and ask it to generate new text based on that input snippet:

```
>>> set_seed(123)
>>> generator("Hey readers, today is",
...           max_length=20,
...           num_return_sequences=3)

[{'generated_text': "Hey readers, today is not the last time we'll be seeing
one of our favorite indie rock bands"},
 {'generated_text': 'Hey readers, today is Christmas. This is not Christmas,
because Christmas is so long and I hope'},
 {'generated_text': "Hey readers, today is CTA Day!\n\nWe're proud to be
hosting a special event"}]
```

As we can see from the output, the model generated three reasonable sentences based on our text snippet. If you want to explore more examples, please feel free to change the random seed and the maximum sequence length.

Also, as previously illustrated in *Figure 16.10*, we can use a transformer model to generate features for training other models. The following code illustrates how we can use GPT-2 to generate features based on an input text:

```
>>> from transformers import GPT2Tokenizer
>>> tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
>>> text = "Let us encode this sentence"
>>> encoded_input = tokenizer(text, return_tensors='pt')
>>> encoded_input
{'input_ids': tensor([[ 5756,   514, 37773,   428,  6827]]), 'attention_mask':
tensor([[1, 1, 1, 1, 1]])}
```

This code encoded the input sentence text into a tokenized format for the GPT-2 model. As we can see, it mapped the strings to an integer representation, and it set the attention mask to all 1s, which means that all words will be processed when we pass the encoded input to the model, as shown here:

```
>>> from transformers import GPT2Model
>>> model = GPT2Model.from_pretrained('gpt2')
>>> output = model(**encoded_input)
```

The `output` variable stores the last hidden state, that is, our GPT-2-based feature encoding of the input sentence:

```
>>> output['last_hidden_state'].shape
torch.Size([1, 5, 768])
```

To suppress the verbose output, we only showed the shape of the tensor. Its first dimension is the batch size (we only have one input text), which is followed by the sentence length and size of the feature encoding. Here, each of the five words is encoded as a 768-dimensional vector.

Now, we could apply this feature encoding to a given dataset and train a downstream classifier based on the GPT-2-based feature representation instead of using a bag-of-words model as discussed in *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*.

Moreover, an alternative approach to using large pre-trained language models is fine-tuning, as we discussed earlier. We will be seeing a fine-tuning example later in this chapter.

If you are interested in additional details on using GPT-2, we recommend the following documentation pages:

- `https://huggingface.co/gpt2`
- `https://huggingface.co/docs/transformers/model_doc/gpt2`

Bidirectional pre-training with BERT

BERT, its full name being **Bidirectional Encoder Representations from Transformers**, was created by a Google research team in 2018 (*BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* by *J. Devlin, M. Chang, K. Lee,* and *K. Toutanova*, `https://arxiv.org/abs/1810.04805`). For reference, even though we cannot compare GPT and BERT directly as they are different architectures, BERT has 345 million parameters (which makes it only slightly larger than GPT-1, and its size is only 1/5 of GPT-2).

As its name suggests, BERT has a transformer-encoder-based model structure that utilizes a bidirectional training procedure. (Or, more accurately, we can think of BERT as using "nondirectional" training because it reads in all input elements all at once.) Under this setting, the encoding of a certain word depends on both the preceding and the succeeding words. Recall that in GPT, input elements are read in with a natural left-to-right order, which helps to form a powerful generative language model. Bidirectional training disables BERT's ability to generate a sentence word by word but provides input encodings of higher quality for other tasks, such as classification, since the model can now process information in both directions.

Recall that in a transformer's encoder, token encoding is a summation of positional encodings and token embeddings. In the BERT encoder, there is an additional segment embedding indicating which segment this token belongs to. This means that each token representation contains three ingredients, as *Figure 16.13* illustrates:

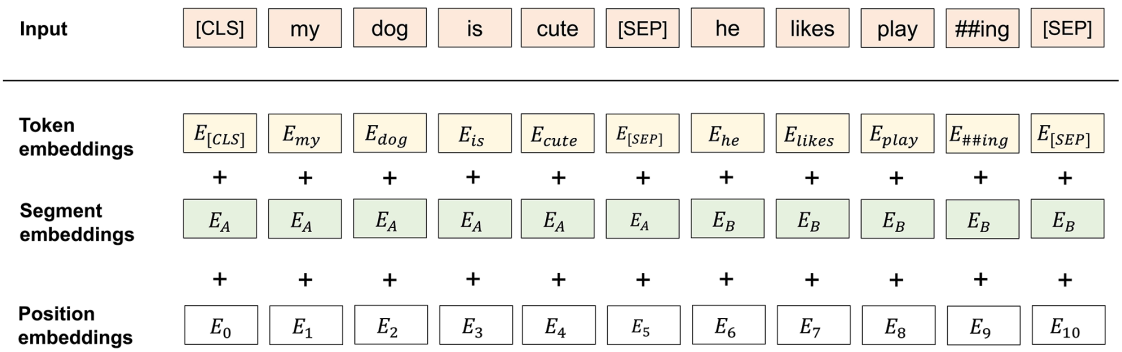

Figure 16.13: Preparing the inputs for the BERT encoder

Why do we need this additional segment information in BERT? The need for this segment information originated from the special pre-training task of BERT called *next-sentence prediction*. In this pre-training task, each training example includes two sentences and thus requires special segment notation to denote whether it belongs to the first or second sentence.

Now, let us look at BERT's pre-training tasks in more detail. Similar to all other transformer-based language models, BERT has two training stages: pre-training and fine-tuning. And pre-training includes two unsupervised tasks: *masked language modeling* and *next-sentence prediction*.

In the **masked language model** (**MLM**), tokens are randomly replaced by so-called *mask tokens*, `[MASK]`, and the model is required to predict these hidden words. Compared with the next-word prediction in GPT, MLM in BERT is more akin to "filling in the blanks" because the model can attend to all tokens in the sentence (except the masked ones). However, simply masking words out can result in inconsistencies between pre-training and fine-tuning since `[MASK]` tokens do not appear in regular texts. To alleviate this, there are further modifications to the words that are selected for masking. For instance, 15 percent of the words in BERT are marked for masking. These 15 percent of randomly selected words are then further treated as follows:

1. Keep the word unchanged 10 percent of the time
2. Replace the original word token with a random word 10 percent of the time
3. Replace the original word token with a mask token, `[MASK]`, 80 percent of the time

Besides avoiding the aforementioned inconsistency between pre-training and fine-tuning when introducing `[MASK]` tokens into the training procedure, these modifications also have other benefits. Firstly, unchanged words include the possibility of maintaining the information of the original token; otherwise, the model can only learn from the context and nothing from the masked words. Secondly, the 10 percent random words prevent the model from becoming lazy, for instance, learning nothing but returning what it is being given. The probabilities for masking, randomizing, and leaving words unchanged were chosen by an ablation study (see the GPT-2 paper); for instance, authors tested different settings and found that this combination worked best.

Figure 16.14 illustrates an example where the word *fox* is masked and, with a certain probability, remains unchanged or is replaced by `[MASK]` or *coffee*. The model is then required to predict what the masked (highlighted) word is as illustrated in *Figure 16.14*:

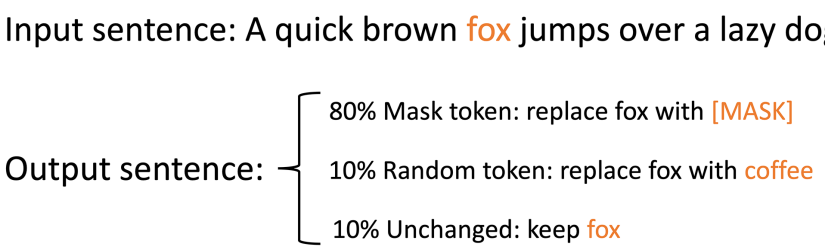

Figure 16.14: An example of MLM

Next-sentence prediction is a natural modification of the next-word prediction task considering the bidirectional encoding of BERT. In fact, many important NLP tasks, such as question answering, depend on the relationship of two sentences in the document. This kind of relationship is hard to capture via regular language models because next-word prediction training usually occurs on a single-sentence level due to input length constraints.

In the next-sentence prediction task, the model is given two sentences, A and B, in the following format:

[CLS] A [SEP] B [SEP]

[CLS] is a classification token, which serves as a placeholder for the predicted label in the decoder output, as well as a token denoting the beginning of the sentences. The [SEP] token, on the other hand, is attached to denote the end of each sentence. The model is then required to classify whether B is the next sentence ("IsNext") of A or not. To provide the model with a balanced dataset, 50 percent of the samples are labeled as "IsNext" while the remaining samples are labeled as "NotNext."

BERT is pre-trained on these two tasks, masked sentences and next-sentence prediction, at the same time. Here, the training objective of BERT is to minimize the combined loss function of both tasks.

Starting from the pre-trained model, specific modifications are required for different downstream tasks in the fine-tuning stage. Each input example needs to match a certain format; for example, it should begin with a [CLS] token and be separated using [SEP] tokens if it consists of more than one sentence.

Roughly speaking, BERT can be fine-tuned on four categories of tasks: (a) sentence pair classification; (b) single-sentence classification; (c) question answering; (d) single-sentence tagging.

Among them, (a) and (b) are sequence-level classification tasks, which only require an additional softmax layer to be added to the output representation of the [CLS] token. (c) and (d), on the other hand, are token-level classification tasks. This means that the model passes output representations of all related tokens to the softmax layer to predict a class label for each individual token.

Question answering

Task (c), question answering, appears to be less often discussed compared to other popular classification tasks such as sentiment classification or speech tagging. In question answering, each input example can be split into two parts, the question and the paragraph that helps to answer the question. The model is required to point out both the start and end token in the paragraph that forms a proper answer to the question. This means that the model needs to generate a tag for every single token in the paragraph, indicating whether this token is a start or end token, or neither. As a side note, it is worth mentioning that the output may contain an end token that appears before the start token, which will lead to a conflict when generating the answer. This kind of output will be recognized as "No Answer" to the question.

As *Figure 16.15* indicates, the model fine-tuning setup has a very simple structure: an input encoder is attached to a pre-trained BERT, and a softmax layer is added for classification. Once the model structure is set up, all the parameters will be adjusted along the learning process.

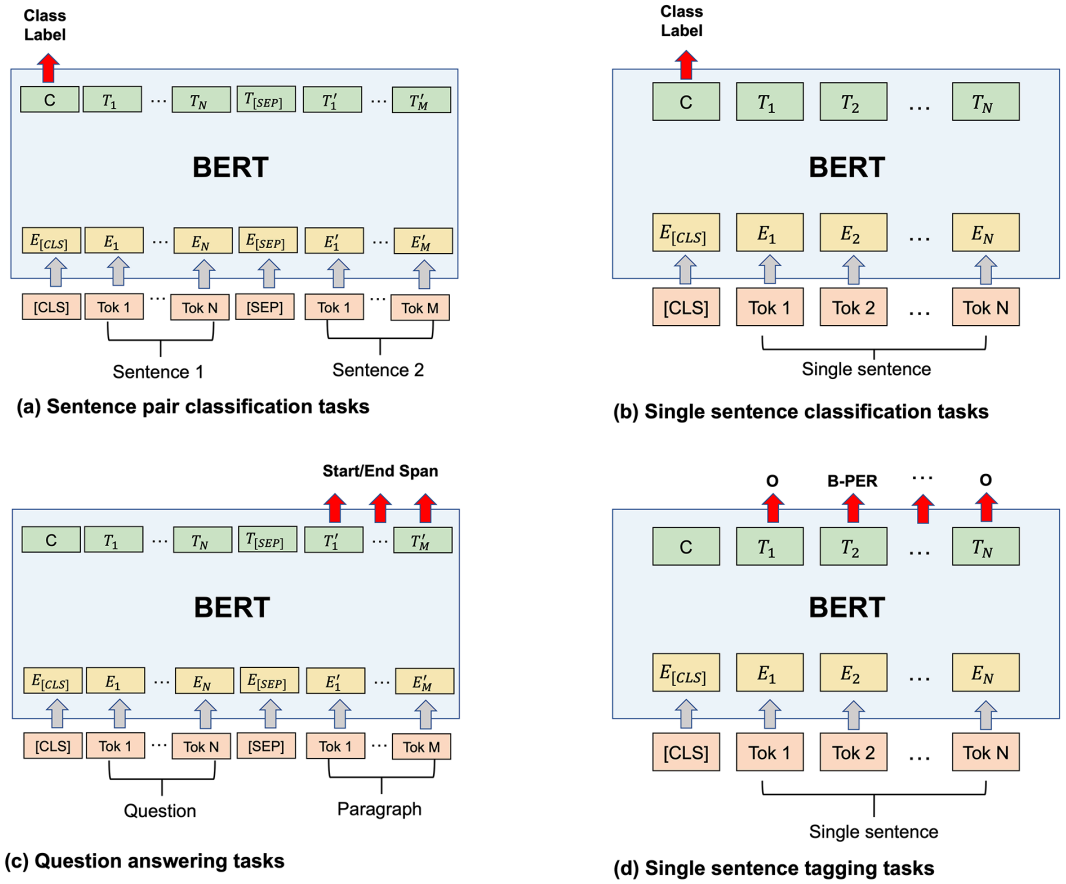

Figure 16.15: Using BERT to fine-tune different language tasks

The best of both worlds: BART

The **Bidirectional and Auto-Regressive Transformer**, abbreviated as **BART**, was developed by researchers at Facebook AI Research in 2019: *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*, *Lewis* and colleagues, `https://arxiv.org/abs/1910.13461`. Recall that in previous sections we argued that GPT utilizes a transformer's decoder structure, whereas BERT utilizes a transformer's encoder structure. Those two models are thus capable of performing different tasks well: GPT's specialty is generating text, whereas BERT performs better on classification tasks. BART can be viewed as a generalization of both GPT and BERT. As the title of this section suggests, BART is able to accomplish both tasks, generating and classifying text. The reason why it can handle both tasks well is that the model comes with a bidirectional encoder as well as a left-to-right autoregressive decoder.

You may wonder how this is different from the original transformer. There are a few changes to the model size along with some minor changes such as activation function choices. However, one of the more interesting changes is that BART works with different model inputs. The original transformer model was designed for language translation so there are two inputs: the text to be translated (source sequence) for the encoder and the translation (target sequence) for the decoder. Additionally, the decoder also receives the encoded source sequence, as illustrated earlier in *Figure 16.6*. However, in BART, the input format was generalized such that it only uses the source sequence as input. BART can perform a wider range of tasks including language translation, where a target sequence is still required to compute the loss and fine-tune the model, but it is not necessary to feed it directly into the decoder.

Now let us take a closer look at the BART's model structure. As previously mentioned, BART is composed of a bidirectional encoder and an autoregressive decoder. Upon receiving a training example as plain text, the input will first be "corrupted" and then encoded by the encoder. These input encodings will then be passed to the decoder, along with the generated tokens. The cross-entropy loss between encoder output and the original text will be calculated and then optimized through the learning process. Think of a transformer where we have two texts in different languages as input to the decoder: the initial text to be translated (source text) and the generated text in the target language. BART can be understood as replacing the former with corrupted text and the latter with the input text itself.

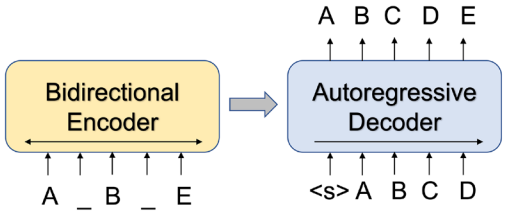

Figure 16.16: BART's model structure

To explain the corruption step in a bit more detail, recall that BERT and GPT are pre-trained by reconstructing masked words: BERT is "filling in the blanks" and GPT is "predicting the next word." These pre-training tasks can also be recognized as reconstructing corrupted sentences because masking words is one way of corrupting a sentence. BART provides the following corruption methods that can be applied to the clean text:

- Token masking
- Token deletion
- Text infilling
- Sentence permutation
- Document rotation

One or more of the techniques listed above can be applied to the same sentence; in the worst scenario, where all the information is contaminated and corrupted, the text becomes useless. Hence, the encoder has limited utility, and with only the decoder module working properly, the model will essentially become more similar to a unidirectional language.

BART can be fine-tuned on a wide range of downstream tasks including (a) sequence classification, (b) token classification, (c) sequence generation, and (d) machine translation. As with BERT, small changes to the inputs need to be made in order to perform different tasks.

In the sequence classification task, an additional token needs to be attached to the input to serve as the generated label token, which is similar to the [CLS] token in BERT. Also, instead of disturbing the input, uncorrupted input is fed into both the encoder and decoder so that the model can make full use of the input.

For token classification, additional tokens become unnecessary, and the model can directly use the generated representation for each token for classification.

Sequence generation in BART differs a bit from GPT because of the existence of the encoder. Instead of generating text from the ground up, sequence generation tasks via BART are more comparable to summarization, where the model is given a corpus of contexts and asked to generate a summary or an abstractive answer to certain questions. To this end, whole input sequences are fed into the encoder while the decoder generates output autoregressively.

Finally, it's natural for BART to perform machine translation considering the similarity between BART and the original transformer. However, instead of following the exact same procedure as for training the original transformer, researchers considered the possibility of incorporating the entire BART model as a pre-trained decoder. To complete the translation model, a new set of randomly initialized parameters is added as a new, additional encoder. Then, the fine-tuning stage can be accomplished in two steps:

1. First, freeze all the parameters except the encoder
2. Then, update all parameters in the model

BART was evaluated on several benchmark datasets for various tasks, and it obtained very competitive results compared to other famous language models such as BERT. In particular, for generation tasks including abstractive question answering, dialogue response, and summarization tasks, BART achieved state-of-the-art results.

Fine-tuning a BERT model in PyTorch

Now that we have introduced and discussed all the necessary concepts and the theory behind the original transformer and popular transformer-based models, it's time to take a look at the more practical part! In this section, you will learn how to fine-tune a BERT model for **sentiment classification** in PyTorch.

Note that although there are many other transformer-based models to choose from, BERT provides a nice balance between model popularity and having a manageable model size so that it can be fine-tuned on a single GPU. Note also that pre-training a BERT from scratch is painful and quite unnecessary considering the availability of the `transformers` Python package provided by Hugging Face, which includes a bunch of pre-trained models that are ready for fine-tuning.

In the following sections, you'll see how to prepare and tokenize the IMDb movie review dataset and fine-tune the distilled BERT model to perform sentiment classification. We deliberately chose sentiment classification as a simple but classic example, though there are many other fascinating applications of language models. Also, by using the familiar IMDb movie review dataset, we can get a good idea of the predictive performance of the BERT model by comparing it to the logistic regression model in *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*, and the RNN in *Chapter 15*, *Modeling Sequential Data Using Recurrent Neural Networks*.

Loading the IMDb movie review dataset

In this subsection, we will begin by loading the required packages and the dataset, split into train, validation, and test sets.

For the BERT-related parts of this tutorial, we will mainly use the open-source `transformers` library (`https://huggingface.co/transformers/`) created by Hugging Face, which we installed in the previous section, *Using GPT-2 to generate new text*.

The **DistilBERT** model we are using in this chapter is a lightweight transformer model created by distilling a pre-trained BERT base model. The original uncased BERT base model contains over 110 million parameters while DistilBERT has 40 percent fewer parameters. Also, DistilBERT runs 60 percent faster and still preserves 95 percent of BERT's performance on the GLUE language understanding benchmark.

The following code imports all the packages we will be using in this chapter to prepare the data and fine-tune the DistilBERT model:

```
>>> import gzip
>>> import shutil
>>> import time

>>> import pandas as pd
>>> import requests
>>> import torch
>>> import torch.nn.functional as F
>>> import torchtext

>>> import transformers
>>> from transformers import DistilBertTokenizerFast
>>> from transformers import DistilBertForSequenceClassification
```

Next, we specify some general settings, including the number of epochs we train the network on, the device specification, and the random seed. To reproduce the results, make sure to set a specific random seed such as `123`:

```
>>> torch.backends.cudnn.deterministic = True
>>> RANDOM_SEED = 123
>>> torch.manual_seed(RANDOM_SEED)
```

```
>>> DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

>>> NUM_EPOCHS = 3
```

We will be working on the IMDb movie review dataset, which you have already seen in *Chapters 8* and *15*. The following code fetches the compressed dataset and unzips it:

```
>>> url = ("https://github.com/rasbt/"
...        "machine-learning-book/raw/"
...        "main/ch08/movie_data.csv.gz")
>>> filename = url.split("/")[-1]

>>> with open(filename, "wb") as f:
...     r = requests.get(url)
...     f.write(r.content)

>>> with gzip.open('movie_data.csv.gz', 'rb') as f_in:
...     with open('movie_data.csv', 'wb') as f_out:
...         shutil.copyfileobj(f_in, f_out)
```

If you have the `movie_data.csv` file from *Chapter 8* still on your hard drive, you can skip this download and unzip procedure.

Next, we load the data into a pandas `DataFrame` and make sure it looks all right:

```
>>> df = pd.read_csv('movie_data.csv')
>>> df.head(3)
```

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

Figure 16.17: The first three rows of the IMDb movie review dataset

The next step is to split the dataset into separate training, validation, and test sets. Here, we use 70 percent of the reviews for the training set, 10 percent for the validation set, and the remaining 20 percent for testing:

```
>>> train_texts = df.iloc[:35000]['review'].values
>>> train_labels = df.iloc[:35000]['sentiment'].values

>>> valid_texts = df.iloc[35000:40000]['review'].values
```

```
>>> valid_labels = df.iloc[35000:40000]['sentiment'].values

>>> test_texts = df.iloc[40000:]['review'].values
>>> test_labels = df.iloc[40000:]['sentiment'].values
```

Tokenizing the dataset

So far, we have obtained the texts and labels for the training, validation, and test sets. Now, we are going to tokenize the texts into individual word tokens using the tokenizer implementation inherited from the pre-trained model class:

```
>>> tokenizer = DistilBertTokenizerFast.from_pretrained(
...     'distilbert-base-uncased'
... )

>>> train_encodings = tokenizer(list(train_texts), truncation=True, padding=True)
>>> valid_encodings = tokenizer(list(valid_texts), truncation=True, padding=True)
>>> test_encodings = tokenizer(list(test_texts), truncation=True, padding=True)
```

Choosing different tokenizers

If you are interested in applying different types of tokenizers, feel free to explore the `tokenizers` package (`https://huggingface.co/docs/tokenizers/python/latest/`), which is also built and maintained by Hugging Face. However, inherited tokenizers maintain the consistency between the pre-trained model and the dataset, which saves us the extra effort of finding the specific tokenizer corresponding to the model. In other words, using an inherited tokenizer is the recommended approach if you want to fine-tune a pre-trained model.

Finally, let's pack everything into a class called `IMDbDataset` and create the corresponding data loaders. Such a self-defined dataset class lets us customize all the related features and functions for our custom movie review dataset in `DataFrame` format:

```
>>> class IMDbDataset(torch.utils.data.Dataset):
...     def __init__(self, encodings, labels):
...         self.encodings = encodings
...         self.labels = labels

>>>     def __getitem__(self, idx):
...         item = {key: torch.tensor(val[idx])
...                 for key, val in self.encodings.items()}
...         item['labels'] = torch.tensor(self.labels[idx])
```

```
...         return item

>>>     def __len__(self):
...         return len(self.labels)


>>> train_dataset = IMDbDataset(train_encodings, train_labels)
>>> valid_dataset = IMDbDataset(valid_encodings, valid_labels)
>>> test_dataset = IMDbDataset(test_encodings, test_labels)

>>> train_loader = torch.utils.data.DataLoader(
...     train_dataset, batch_size=16, shuffle=True)
>>> valid_loader = torch.utils.data.DataLoader(
...     valid_dataset, batch_size=16, shuffle=False)
>>> test_loader = torch.utils.data.DataLoader(
...     test_dataset, batch_size=16, shuffle=False)
```

While the overall data loader setup should be familiar from previous chapters, one noteworthy detail is the `item` variable in the `__getitem__` method. The encodings we produced previously store a lot of information about the tokenized texts. Via the dictionary comprehension that we use to assign the dictionary to the `item` variable, we are only extracting the most relevant information. For instance, the resulting dictionary entries include `input_ids` (unique integers from the vocabulary corresponding to the tokens), `labels` (the class labels), and `attention_mask`. Here, `attention_mask` is a tensor with binary values (0s and 1s) that denotes which tokens the model should attend to. In particular, 0s correspond to tokens used for padding the sequence to equal lengths and are ignored by the model; the 1s correspond to the actual text tokens.

Loading and fine-tuning a pre-trained BERT model

Having taken care of the data preparation, in this subsection, you will see how to load the pre-trained DistilBERT model and fine-tune it using the dataset we just created. The code for loading the pre-trained model is as follows:

```
>>> model = DistilBertForSequenceClassification.from_pretrained(
...     'distilbert-base-uncased')
>>> model.to(DEVICE)
>>> model.train()

>>> optim = torch.optim.Adam(model.parameters(), lr=5e-5)
```

`DistilBertForSequenceClassification` specifies the downstream task we want to fine-tune the model on, which is sequence classification in this case. As mentioned before, `'distilbert-base-uncased'` is a lightweight version of a BERT uncased base model with manageable size and good performance. Note that "uncased" means that the model does not distinguish between upper- and lower-case letters.

Using other pre-trained transformers

The *transformers* package also provides many other pre-trained models and various downstream tasks for fine-tuning. Check them out at `https://huggingface.co/transformers/`.

Now, it's time to train the model. We can break this up into two parts. First, we need to define an accuracy function to evaluate the model performance. Note that this accuracy function computes the conventional classification accuracy. Why is it so verbose? Here, we are loading the dataset batch by batch to work around RAM or GPU memory (VRAM) limitations when working with a large deep learning model:

```
>>> def compute_accuracy(model, data_loader, device):
...         with torch.no_grad():
...             correct_pred, num_examples = 0, 0
...             for batch_idx, batch in enumerate(data_loader):
...                 ### Prepare data
...                 input_ids = batch['input_ids'].to(device)
...                 attention_mask = \
...                     batch['attention_mask'].to(device)
...                 labels = batch['labels'].to(device)

...                 outputs = model(input_ids,
...                    attention_mask=attention_mask)
...                 logits = outputs['logits']
...                 predicted_labels = torch.argmax(logits, 1)
...                 num_examples += labels.size(0)
...                 correct_pred += \
...                     (predicted_labels == labels).sum()
...         return correct_pred.float()/num_examples * 100
```

In the `compute_accuracy` function, we load a given batch and then obtain the predicted labels from the outputs. While doing this, we keep track of the total number of examples via `num_examples`. Similarly, we keep track of the number of correct predictions via the `correct_pred` variable. Finally, after we iterate over the complete dataset, we compute the accuracy as the proportion of correctly predicted labels.

Overall, via the `compute_accuracy` function, you can already get a glimpse at how we can use the transformer model to obtain the class labels. That is, we feed the model the `input_ids` along with the `attention_mask` information that, here, denotes whether a token is an actual text token or a token for padding the sequences to equal length. The `model` call then returns the outputs, which is a transformer library-specific `SequenceClassifierOutput` object. From this object, we then obtain the logits that we convert into class labels via the `argmax` function as we have done in previous chapters.

Finally, let us get to the main part: the training (or rather, fine-tuning) loop. As you will notice, fine-tuning a model from the *transformers* library is very similar to training a model in pure PyTorch from scratch:

```
>>> start_time = time.time()

>>> for epoch in range(NUM_EPOCHS):

...     model.train()

...     for batch_idx, batch in enumerate(train_loader):

...         ### Prepare data
...         input_ids = batch['input_ids'].to(DEVICE)
...         attention_mask = batch['attention_mask'].to(DEVICE)
...         labels = batch['labels'].to(DEVICE)

...         ### Forward pass
...         outputs = model(input_ids,
...                         attention_mask=attention_mask,
...                         labels=labels)
...         loss, logits = outputs['loss'], outputs['logits']

...         ### Backward pass
...         optim.zero_grad()
...         loss.backward()
...         optim.step()

...         ### Logging
...         if not batch_idx % 250:
...             print(f'Epoch: {epoch+1:04d}/{NUM_EPOCHS:04d}'
...                   f' | Batch'
...                   f'{batch_idx:04d}/'
...                   f'{len(train_loader):04d} | '
...                   f'Loss: {loss:.4f}')

...     model.eval()

...     with torch.set_grad_enabled(False):
...         print(f'Training accuracy: '
...               f'{compute_accuracy(model, train_loader, DEVICE):.2f}%'
```

```
...              f'\nValid accuracy: '
...              f'{compute_accuracy(model, valid_loader, DEVICE):.2f}%')

...     print(f'Time elapsed: {(time.time() - start_time)/60:.2f} min')

... print(f'Total Training Time: {(time.time() - start_time)/60:.2f} min')
... print(f'Test accuracy: {compute_accuracy(model, test_loader, DEVICE):.2f}%')
```

The output produced by the preceding code is as follows (note that the code is not fully deterministic, which is why the results you are getting may be slightly different):

```
Epoch: 0001/0003 | Batch 0000/2188 | Loss: 0.6771
Epoch: 0001/0003 | Batch 0250/2188 | Loss: 0.3006
Epoch: 0001/0003 | Batch 0500/2188 | Loss: 0.3678
Epoch: 0001/0003 | Batch 0750/2188 | Loss: 0.1487
Epoch: 0001/0003 | Batch 1000/2188 | Loss: 0.6674
Epoch: 0001/0003 | Batch 1250/2188 | Loss: 0.3264
Epoch: 0001/0003 | Batch 1500/2188 | Loss: 0.4358
Epoch: 0001/0003 | Batch 1750/2188 | Loss: 0.2579
Epoch: 0001/0003 | Batch 2000/2188 | Loss: 0.2474
Training accuracy: 96.32%
Valid accuracy: 92.34%
Time elapsed: 20.67 min
Epoch: 0002/0003 | Batch 0000/2188 | Loss: 0.0850
Epoch: 0002/0003 | Batch 0250/2188 | Loss: 0.3433
Epoch: 0002/0003 | Batch 0500/2188 | Loss: 0.0793
Epoch: 0002/0003 | Batch 0750/2188 | Loss: 0.0061
Epoch: 0002/0003 | Batch 1000/2188 | Loss: 0.1536
Epoch: 0002/0003 | Batch 1250/2188 | Loss: 0.0816
Epoch: 0002/0003 | Batch 1500/2188 | Loss: 0.0786
Epoch: 0002/0003 | Batch 1750/2188 | Loss: 0.1395
Epoch: 0002/0003 | Batch 2000/2188 | Loss: 0.0344
Training accuracy: 98.35%
Valid accuracy: 92.46%
Time elapsed: 41.41 min
Epoch: 0003/0003 | Batch 0000/2188 | Loss: 0.0403
Epoch: 0003/0003 | Batch 0250/2188 | Loss: 0.0036
Epoch: 0003/0003 | Batch 0500/2188 | Loss: 0.0156
Epoch: 0003/0003 | Batch 0750/2188 | Loss: 0.0114
Epoch: 0003/0003 | Batch 1000/2188 | Loss: 0.1227
Epoch: 0003/0003 | Batch 1250/2188 | Loss: 0.0125
```

```
Epoch: 0003/0003 | Batch 1500/2188 | Loss: 0.0074
Epoch: 0003/0003 | Batch 1750/2188 | Loss: 0.0202
Epoch: 0003/0003 | Batch 2000/2188 | Loss: 0.0746
Training accuracy: 99.08%
Valid accuracy: 91.84%
Time elapsed: 62.15 min
Total Training Time: 62.15 min
Test accuracy: 92.50%
```

In this code, we iterate over multiple epochs. In each epoch we perform the following steps:

1. Load the input into the device we are working on (GPU or CPU)
2. Compute the model output and loss
3. Adjust the weight parameters by backpropagating the loss
4. Evaluate the model performance on both the training and validation set

Note that the training time may vary on different devices. After three epochs, accuracy on the test dataset reaches around 93 percent, which is a substantial improvement compared to the 85 percent test accuracy that the RNN achieved in *Chapter 15*.

Fine-tuning a transformer more conveniently using the Trainer API

In the previous subsection, we implemented the training loop in PyTorch manually to illustrate that fine-tuning a transformer model is really not that much different from training an RNN or CNN model from scratch. However, note that the `transformers` library contains several nice extra features for additional convenience, like the Trainer API, which we will introduce in this subsection.

The Trainer API provided by Hugging Face is optimized for transformer models with a wide range of training options and various built-in features. When using the Trainer API, we can skip the effort of writing training loops on our own, and training or fine-tuning a transformer model is as simple as a function (or method) call. Let's see how this works in practice.

After loading the pre-trained model via

```
>>> model = DistilBertForSequenceClassification.from_pretrained(
...     'distilbert-base-uncased')
>>> model.to(DEVICE)
>>> model.train();
```

The training loop from the previous section can then be replaced by the following code:

```
>>> optim = torch.optim.Adam(model.parameters(), lr=5e-5)

>>> from transformers import Trainer, TrainingArguments
```

```
>>> training_args = TrainingArguments(
...     output_dir='./results',
...     num_train_epochs=3,
...     per_device_train_batch_size=16,
...     per_device_eval_batch_size=16,
...     logging_dir='./logs',
...     logging_steps=10,
... )

>>> trainer = Trainer(
...    model=model,
...    args=training_args,
...    train_dataset=train_dataset,
...    optimizers=(optim, None) # optim and learning rate scheduler
... )
```

In the preceding code snippets, we first defined the training arguments, which are relatively self-explanatory settings regarding the input and output locations, number of epochs, and batch sizes. We tried to keep the settings as simple as possible; however, there are many additional settings available, and we recommend consulting the `TrainingArguments` documentation page for additional details: `https://huggingface.co/transformers/main_classes/trainer.html#trainingarguments`.

We then passed these `TrainingArguments` settings to the `Trainer` class to instantiate a new `trainer` object. After initiating the `trainer` with the settings, the model to be fine-tuned, and the training and evaluation sets, we can train the model by calling the `trainer.train()` method (we will use this method further shortly). That's it, using the Trainer API is as simple as shown in the preceding code, and no further boilerplate code is required.

However, you may have noticed that the test dataset was not involved in these code snippets, and we haven't specified any evaluation metrics in this subsection. This is because the Trainer API only shows the training loss and does not provide model evaluation along the training process by default. There are two ways to display the final model performance, which we will illustrate next.

The first method for evaluating the final model is to define an evaluation function as the `compute_metrics` argument for another `Trainer` instance. The `compute_metrics` function operates on the models' test predictions as logits (which is the default output of the model) and the test labels. To instantiate this function, we recommend installing Hugging Face's `datasets` library via `pip install datasets` and use it as follows:

```
>>> from datasets import load_metric
>>> import numpy as np

>>> metric = load_metric("accuracy")

>>> def compute_metrics(eval_pred):
```

```
...       logits, labels = eval_pred
...       # note: logits are a numpy array, not a pytorch tensor
...       predictions = np.argmax(logits, axis=-1)
...       return metric.compute(
...           predictions=predictions, references=labels)
```

The updated `Trainer` instantiation (now including `compute_metrics`) is then as follows:

```
>>> trainer=Trainer(
...     model=model,
...     args=training_args,
...     train_dataset=train_dataset,
...     eval_dataset=test_dataset,
...     compute_metrics=compute_metrics,
...     optimizers=(optim, None) # optim and learning rate scheduler
... )
```

Now, let's train the model (again, note that the code is not fully deterministic, which is why you might be getting slightly different results):

```
>>> start_time = time.time()
>>> trainer.train()

***** Running training *****
  Num examples = 35000
  Num Epochs = 3
  Instantaneous batch size per device = 16
  Total train batch size (w. parallel, distributed & accumulation) = 16
  Gradient Accumulation steps = 1
  Total optimization steps = 6564

Step  Training Loss
10    0.705800
20    0.684100
30    0.681500
40    0.591600
50    0.328600
60    0.478300
...


>>> print(f'Total Training Time: '
...       f'{(time.time() - start_time)/60:.2f} min')
Total Training Time: 45.36 min
```

After the training has completed, which can take up to an hour depending on your GPU, we can call `trainer.evaluate()` to obtain the model performance on the test set:

```
>>> print(trainer.evaluate())

***** Running Evaluation *****
Num examples = 10000
Batch size = 16
100%|████████████████████████████████████| 625/625 [10:59<00:00,  1.06s/
it]
{'eval_loss': 0.30534815788269043,
 'eval_accuracy': 0.9327,
 'eval_runtime': 87.1161,
 'eval_samples_per_second': 114.789,
 'eval_steps_per_second': 7.174,
 'epoch': 3.0}
```

As we can see, the evaluation accuracy is around 94 percent, similar to our own previously used PyTorch training loop. (Note that we have skipped the training step, because the `model` is already fine-tuned after the previous `trainer.train()` call.) There is a small discrepancy between our manual training approach and using the `Trainer` class, because the `Trainer` class uses some different and some additional settings.

The second method we could employ to compute the final test set accuracy is re-using our `compute_accuracy` function that we defined in the previous section. We can directly evaluate the performance of the fine-tuned model on the test dataset by running the following code:

```
>>> model.eval()
>>> model.to(DEVICE)

>>> print(f'Test accuracy: {compute_accuracy(model, test_loader,
DEVICE):.2f}%')

Test accuracy: 93.27%
```

In fact, if you want to check the model's performance regularly during training, you can require the trainer to print the model evaluation after each epoch by defining the training arguments as follows:

```
>>> from transformers import TrainingArguments

>>> training_args = TrainingArguments("test_trainer",
...     evaluation_strategy="epoch", ...)
```

However, if you are planning to change or optimize hyperparameters and repeat the fine-tuning procedure several times, we recommend using the validation set for this purpose, in order to keep the test set independent. We can achieve this by instantiating the `Trainer` using `valid_dataset`:

```
>>> trainer=Trainer(
...     model=model,
...     args=training_args,
...     train_dataset=train_dataset,
...     eval_dataset=valid_dataset,
...     compute_metrics=compute_metrics,
... )
```

In this section, we saw how we can fine-tune a BERT model for classification. This is different from using other deep learning architectures like RNNs, which we usually train from scratch. However, unless we are doing research and are trying to develop new transformer architectures—a very expensive endeavor—pre-training transformer models is not necessary. Since transformer models are trained on general, unlabeled dataset resources, pre-training them ourselves may not be a good use of our time and resources; fine-tuning is the way to go.

Summary

In this chapter, we introduced a whole new model architecture for natural language processing, the transformer architecture. The transformer architecture is built on a concept called self-attention, and we started introducing this concept step by step. First, we looked at an RNN outfitted with attention in order to improve its translation capabilities for long sentences. Then, we gently introduced the concept of self-attention and explained how it is used in the multi-head attention module within the transformer.

Many different derivatives of the transformer architecture have emerged and evolved since the original transformer was published in 2017. In this chapter, we focused on a selection of some of the most popular ones: the GPT model family, BERT, and BART. GPT is a unidirectional model that is particularly good at generating new text. BERT takes a bidirectional approach, which is better suited for other types of tasks, for example, classification. Lastly, BART combines both the bidirectional encoder from BERT and the unidirectional decoder from GPT. Interested readers can find out about additional transformer-based architectures via the following two survey articles:

1. *Pre-trained Models for Natural Language Processing: A Survey* by *Qiu* and colleagues, 2020. Available at `https://arxiv.org/abs/2003.08271`
2. *AMMUS : A Survey of Transformer-based Pretrained Models in Natural Language Processing* by *Kayan* and colleagues, 2021. Available at `https://arxiv.org/abs/2108.05542`

Transformer models are generally more data hungry than RNNs and require large amounts of data for pre-training. The pre-training leverages large amounts of unlabeled data to build a general language model that can then be specialized to specific tasks by fine-tuning it on smaller labeled datasets.

To see how this works in practice, we downloaded a pre-trained BERT model from the Hugging Face `transformers` library and fine-tuned it for sentiment classification on the IMDb movie review dataset.

In the next chapter, we will discuss generative adversarial networks. As the name suggests, generative adversarial networks are models that can be used for generating new data, similar to the GPT models we discussed in this chapter. However, we are now leaving the natural language modeling topic behind us and will look at generative adversarial networks in the context of computer vision and generating new images, the task that these networks were originally designed for.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`