9 Transformers

This chapter covers

- Understanding the inner workings of Transformers
- Deriving word embeddings with BERT
- Comparing BERT and Word2Vec
- Working with XLNet

In late 2018, researchers from Google published a paper introducing a deep learning technique that would soon become a major breakthrough: *Bidirectional Encoder Representations from Transformers*, or *BERT* (Devlin et al. 2018). BERT aims to derive word embeddings from raw textual data just like Word2Vec, but does it in a much more clever and powerful manner: it takes into account both the left and right contexts when learning vector representations for words (figure 9.1). In contrast, Word2Vec uses a single piece of context. But this is not the only difference. BERT is grounded in *attention* and, unlike Word2Vec, deploys a *deep* network (recall that Word2Vec uses a *shallow* network with just one hidden layer.)

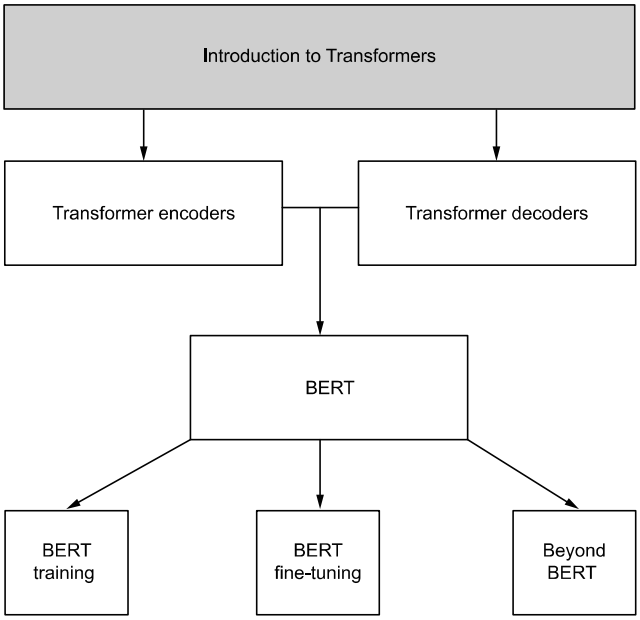

Figure 9.1 Transformers encompass a complex, attention-driven process for deriving word embeddings from raw textual data.

BERT smashed existing performance scores on all tasks it was applied to and (as we see later in this chapter) led to some not-so-trivial insights into deep neural language processing through the analysis of its attention patterns. So, how does BERT do all that? Let's trace BERT back to its roots: Transformers. We will go through the technical background of Transformers in this chapter and defer applications and detailed code to chapter 10.

9.1 BERT up close: Transformers

BERT descends from so-called *Transformers*, which are encoder-decoder models developed by Google (Vaswani et al. 2017). Similar to Word2Vec, BERT models are trained to produce a prediction (a word under a mask) based on an internal representation of context information. Recall from chapter 3 that in Word2Vec, we predict a word given its immediate neighbor words (the *continuous bag-of-words [CBOW]* variant) or, vice versa, a context given a certain word (*skipgram*).

Figure 9.2 repeats these approaches. The CBOW approach aims to infer a missing word w_t from its context based on a single hidden layer representing the projected context words w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}. In contrast, the skipgram approach attempts to predict contexts from the single word w_t.

Word2Vec is centered around a prediction task similar to what BERT addresses: predicting words in contexts, with the aim of representing separate words with vector memories of the contexts they occur in. Since this process can be interpreted as a form of encoding, it is similar to *autoencoders*. Such networks also compress or *encode* input data

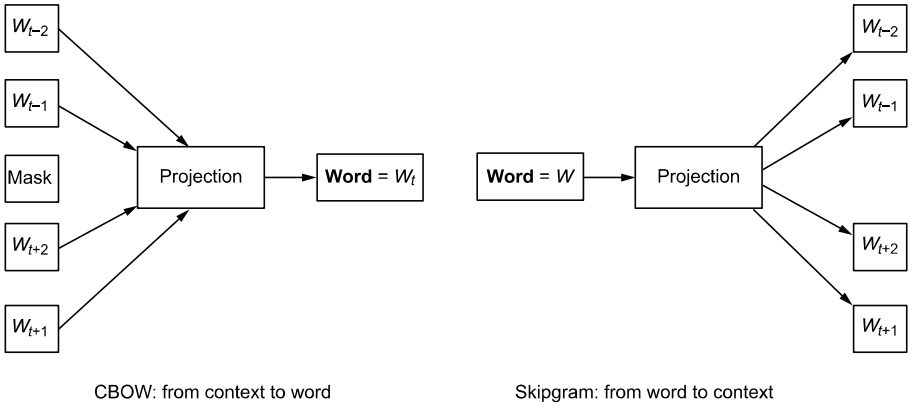

Figure 9.2 Skipgram and CBOW prediction by Word2Vec (Mikolov et al. 2013)

into intermediate representations, and they are trained to minimize *reconstruction loss*: the error the network makes when attempting to reconstruct (*decode*) the original input from the encoded (latent) representation. This type of learning is known as *bottleneck learning*. Figure 9.3 shows the typical structure of an autoencoder; the encoded layer forms a bottleneck, which is usually a lower dimension than the original input, and the network learns to eliminate noise and focus on the important dimensions of the input data, not unlike dimensionality reduction techniques such as principal component analysis (PCA) (https://en.wikipedia.org/wiki/Principal_component_analysis).

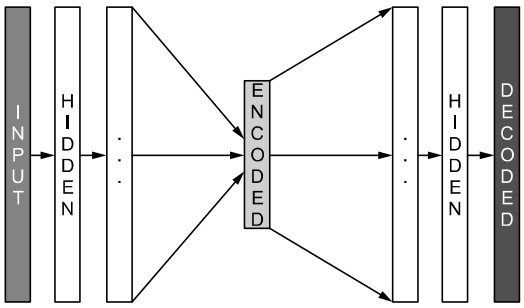

Figure 9.3 An autoencoder. Input is encoded (represented) in one (often lower-dimensional) hidden layer after passing through several other hidden layers. The original input is then reconstructed from this lower-dimensional representation: the model is trained to reproduce the original input. If done properly, the lower-dimensional input captures intrinsic properties of the input and abstracts away irrelevant noise and variation.

Transformers are complex encoder-decoder models. A transformer consists of multiple cross-linked encoder and decoder layers and an extensive self-attention mechanism: words paying attention to themselves and to others. Transformers can be used to map one sequence into another, as in machine translation (translating an English sentence into a German sentence, for instance). Figure 9.4, based on the original Transformer paper (Vaswani et al. 2017), depicts the canonical Transformer architecture.

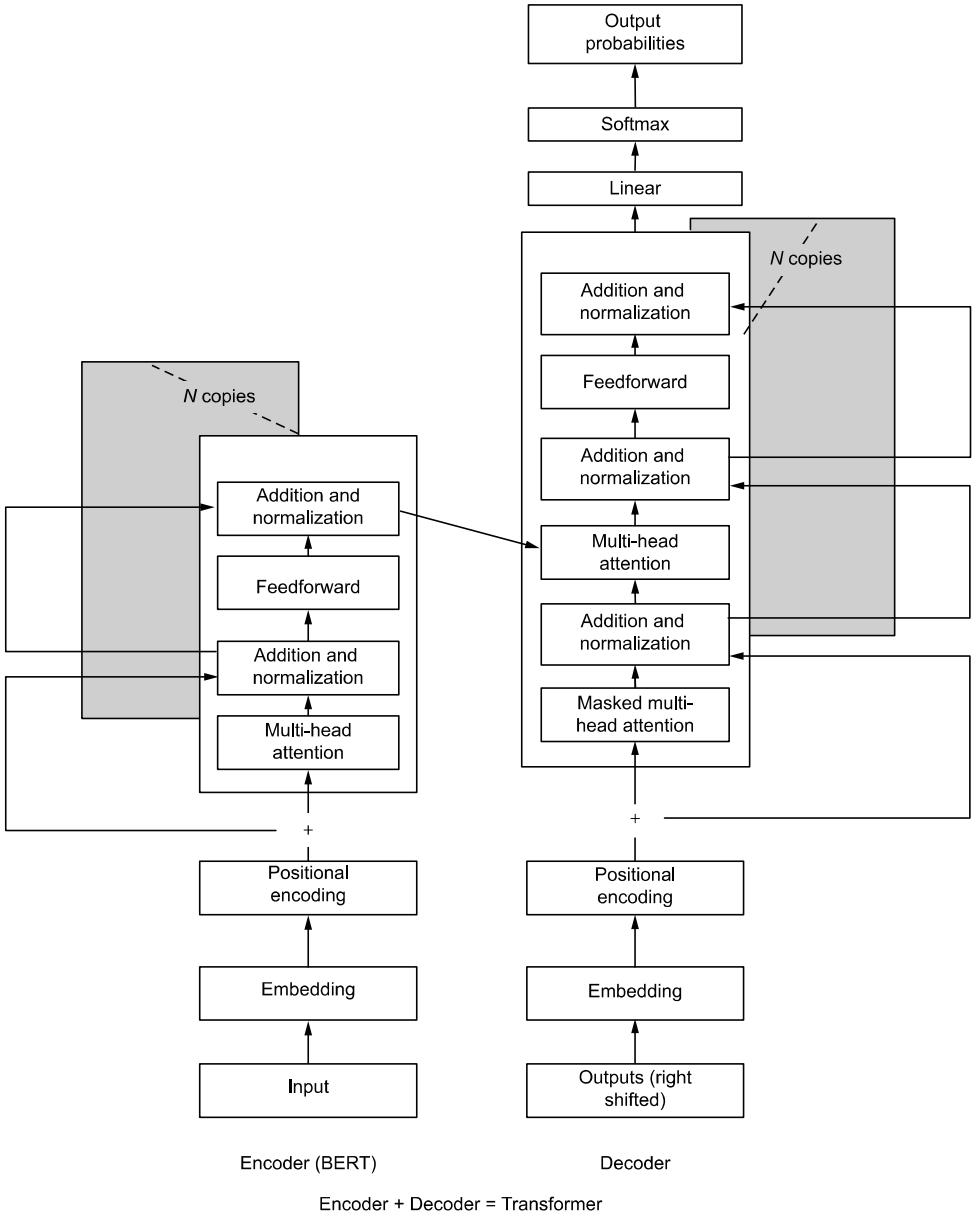

Figure 9.4 The Transformer architecture (see Vaswani et al. 2017). The left part is the encoder, and the right part is the decoder. Both encoder and decoder blocks have multiple copies, and they are connected one by one (every encoder block is connected to a corresponding decoder block).

Let's dissect the architecture in detail starting with the part on the left: the encoder layers. As it turns out, BERT, as a member of the Transformer family, only has the encoder part of this architecture.

9.2 Transformer encoders

An encoder layer or block in a Transformer (figure 9.5) has two internal layers: a *self-attention* layer and a fully connected feedforward network layer. The structure of the encoder is laid out in figure 9.6.

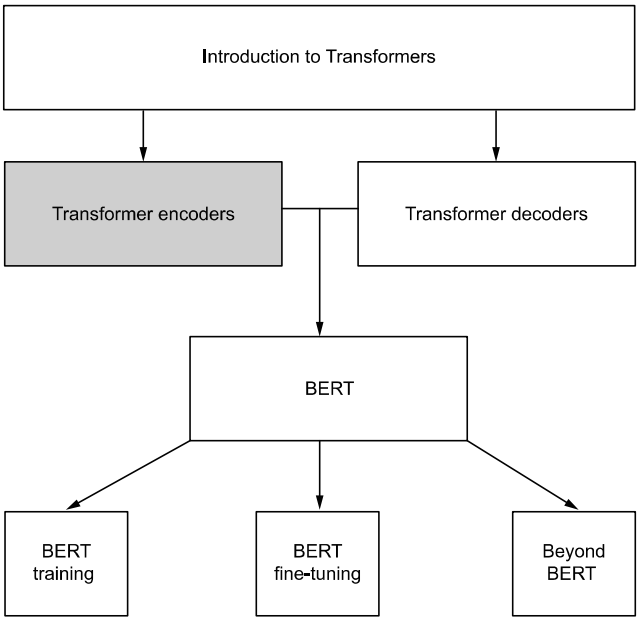

Figure 9.5 Transformer encoders are connected, neighboring constituents of Transformer decoders.

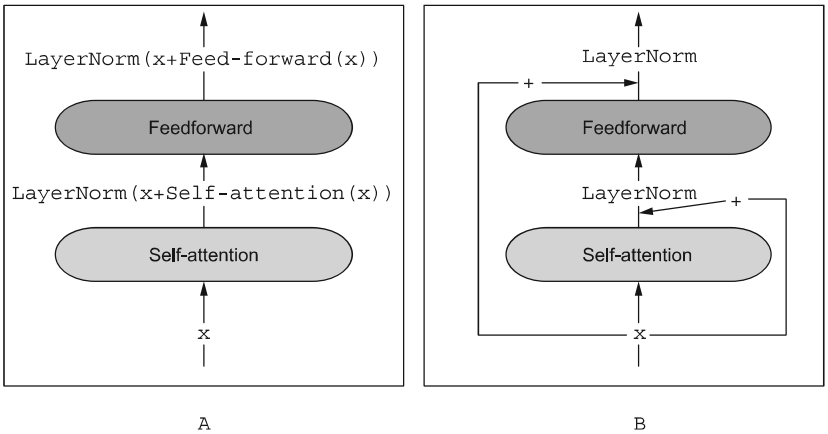

Figure 9.6 Encoder structure for Transformers, shown as two equivalent diagrams. Input `x` is passed through a self-attention layer, then the original input is added to the output of the self-attention layer, and finally the result is normalized. After that, a feed-forward layer is applied, followed again by adding the original input to that result and normalizing it. Adding the unaltered `x` to subsequent layer outputs is called a *residual* or *skip* connection, made explicit in diagram B.

TIP *Self-attention* means a relational attention mechanism that relates different parts of a sequence (like the words in a text). An alternative name for this is *intra-attention.*

Every sublayer has access to the incoming input vector x, which therefore is a form of *residual* or *skip* connection: a connection between two layers bypassing intervening layers. Diagram B in the figure makes this a bit more explicit. The `LayerNorm` operation normalizes the output of the layer based on the mean and standard deviation for all the summed inputs to the neurons in that layer, which has shown measurable effects on training time (see Ba et al. 2016). A standard type of feature normalization in machine learning is *z-score normalization,* where we first compute the mean and standard deviation for every feature in a dataset, subtract each feature's mean from that feature, and divide the result by the standard deviation for that feature. In layer normalization, the mean (mu) and standard deviation (sigma) are defined *per layer (l)* as in figure 9.7. For example, figure 9.8 shows how a three-neuron layer is normalized by subtracting its mean from every neuron activation and then dividing by the standard deviation for the layer.

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$$

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

Figure 9.7 Layer normalization. H is the number of nodes in a (hidden) layer *l*; a_i^l is the summed input to neuron *i* in layer *l*.

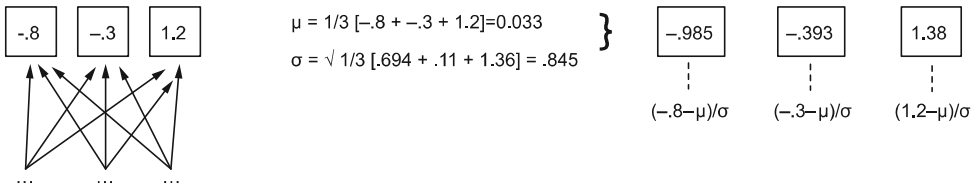

Figure 9.8 Layer normalization example: normalizing three neuron activations based on the mean and standard deviation

TIP In layer normalization, different inputs receive different normalizations and do not depend on the batch statistics for the entire batch they belong to. Since the batch size is a hyperparameter that can be set before training, this eliminates an important source of variability.

All encoder layers are equipped with attention: every input "pays attention" to every other input (but itself). This is done with *attention heads* that derive embeddings of every separate input element in the context of surrounding input elements. The trick here is that within the context window, all context is modeled, and the embeddings for separate input elements contain weighted attention information for every other input element in the context.

Let's dive a bit deeper into attention heads. An attention head is a triple of three matrices W_Q, W_K, W_V containing separate weights. They are initialized with random

values. Given an input $x_1,\ldots,x_n$, with an embedding for every part x_i, we compute three matrix-vector products, resulting in three vectors:

$$k_i = W_K x_i \quad q_i = W_Q x_i \quad v_i = W_V x_i$$

TIP If we multiply a matrix with r rows and c columns by a vector with c components (of size $(c,1)$), we end up with a matrix of shape $(r,1)$. Such a matrix is called a *column vector*.

These values populate three matrices K,Q,V of size (r,c), where c is the fixed length of the input sequences x, and r is an arbitrary number of rows.

The two matrices K and Q seem to encode similar information, but they are distinct. To compute attention weights between all input elements x_i and x_j, we perform the following matrix computation involving the familiar softmax operation we used previously to compute attention

$$\text{ATTENTION}(Q,K,V) = \text{softmax}(QK^T/\sqrt{d_k})V$$

where the V matrix is used as a final weighting factor, and the quantity QK^T is normalized for the dimension of the key vectors k ($\sqrt{d_k}$). K^T is the transpose of K (rows and columns reversed).

Any attention value a_{ij} computed with ATTENTION(Q,K,V) encodes the attention token i pays to token j. Note that this is an asymmetric relation; it does not entail that token j pays the same attention to token i. That's why there are two weight matrices involved: a *query* matrix Q and a *key* matrix K.

NOTE Suppose we have input data that is represented as a batch of N examples, each of which is a 512 × 768 matrix: (padded) sequences of 512 words, with each word represented with a 768-dimensional embedding. Given an attention head, every such matrix (corresponding to one input example) is multiplied by three matrices: the query, key, and value matrices for the attention head. Let these be of size (64,512) for Q and K and (64,768) for V. We end up with a matrix sized (64,64) from the product QK^T. After multiplication with V, the computation of ATTENTION(Q,K,V) produces a matrix of shape (64,768): (64,64) × (64,768) = (64,768).

But this is by no means the end of the story. Why not have multiple attention heads?

TIP Recall that every attention head is a triple of three matrices W_Q, W_K, W_V containing separate weights.

This is where one of the marvels of Transformers comes to light. Having multiple attention heads jointly at work leads to a situation where heads *specialize* in different kinds of attention focus. For instance, when Transformers are fed raw text to produce embeddings, certain attention heads specialize in prepositional phrase patterns, emphasizing

the relation between prepositions and their noun objects (on the table). Other heads specialize in transitive verb-direct object relations (Mary eats an apple) or even reflexive pronoun-antecedent relations (John shaves himself). See Clark et al. (2018) for examples.

The various attention matrices per attention head are then combined into one big matrix, which finally is multiplied by the V matrix. This produces the output for the current encoder layer. In our example, for 8 attention heads per layer, we end up with 512,768-dimensional output. The result of this process is that the original word embeddings assigned to the input words (starting as random embeddings) are incrementally updated with this contextual self-attention information: words paying attention to themselves and each other. In that sense, word embeddings become increasingly tuned for modeling words as a function of their context, similar to Word2Vec, but using much more contextual information.

Exercise

Reflect on the convolutional filters in CNNs (chapter 2). What are the differences from and similarities to attention heads?

Figure 9.4 shows we can have multiple encoder layers stacked onto each other (the N copies), each with its own multiple attention heads. This creates *hierarchies of attention*. The first encoding layer is a bit different from subsequent layers: it encodes information of the embeddings corresponding to the inputs while preserving their word position through something called *positional encoding*. We clearly need this information, since Transformers do not treat texts as bags of words; word order is important to encode contextual information in the word embeddings they produce. It is also important to understand how Transformers, in this respect, differ from sequential models like recurrent neural networks and long short-term memory. Transformers explicitly do not use memory facilities like cell states to encode temporal memory. They oversee more information in their input sequences and look back and forth in those sequences. So, how do they keep track of word positions?

9.2.1 Positional encoding

Positional encoding is a trick to encode word positions into vectors (representing words) by using the cyclic aspects of the sine and cosine functions. Recall from high school algebra that sine and cosine are periodic functions: at fixed increments of their inputs, they produce the same minimum and maximum values. Furthermore, the sin and cos functions, while similar from a periodic point of view, are phase-shifted variants of each other. This is illustrated in figure 9.9.

Notice how the values of the cosine and sine lie in the interval [–1,1]. For every position, the positional embedding approach generates a vector consisting of sine and

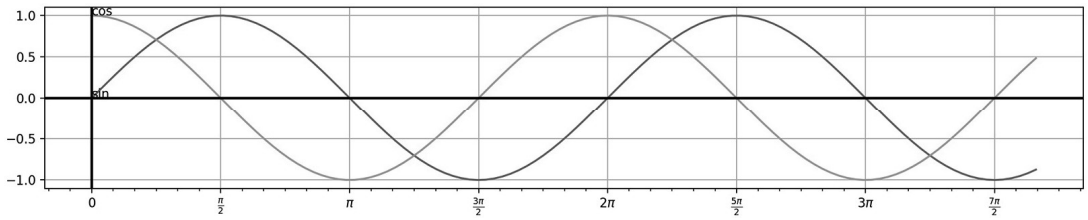

Figure 9.9 Sine and cosine. The y-axis demonstrates the value range of these functions ([–1,1]). On the x-axis, we see that cos x = 0 when x = ±π/2, ±3π/2, ±5π/2, and so on (at uneven multiples of π/2) and sin x = 0 for x = 0,π, 2π, 3π, and so on.

cosine values. These sine and cosine values are computed as a function of both the word position and the current dimension of the vector and uniquely tie words to word positions. These vectors are subsequently added to the original word embedding vector for a word at a given position. The computed values can be interpreted as a continuous alternative to discrete bits, as we see shortly. The computations are as follows:

$$\texttt{positional\_encoding}(p, d, 2i) = \sin(p/(10000^{2i/d}))$$
$$\texttt{positional\_encoding}(p, d, 2i+1) = \cos(p/(10000^{2i/d}))$$

So, we're using `sin` for even word positions (0,2,4,…) and `cos` for uneven word positions (1,3,5,…).

Let's walk through an example. Suppose we have four-dimensional word embeddings for a sequence of 10 words. For every word position 1 to 10, we will generate a positional vector of dimension 4; then we will add these vectors to the original four-dimensional word embeddings, fusing them with positional information.

Listing 9.1 Positional encoding

```
import sys
from math import sin, cos
import numpy as np

dimension=4
max_len=10
pos_enc=np.zeros(shape=(max_len,dimension))
for pos in range(max_len):
   i=0
   while (i<=dimension-2):
      x = pos/(10000**(2.0*i/dimension))
      pos_enc[pos][i] = sin(x)
      pos_enc[pos][i+1] = cos(x)
      i+=2
print(pos_enc)
```

Define a container array consisting of zeros.

Fill the array with the positional sin and cos values.

Print the result.

This produces 10 four-dimensional vectors, one for every word:

```
[[ 0.00000000e+00  1.00000000e+00  0.00000000e+00  1.00000000e+00]
 [ 8.41470985e-01  5.40302306e-01  9.99999998e-05  9.99999995e-01]
 [ 9.09297427e-01 -4.16146837e-01  1.99999999e-04  9.99999980e-01]
 [ 1.41120008e-01 -9.89992497e-01  2.99999995e-04  9.99999955e-01]
 [-7.56802495e-01 -6.53643621e-01  3.99999989e-04  9.99999920e-01]
 [-9.58924275e-01  2.83662185e-01  4.99999979e-04  9.99999875e-01]
 [-2.79415498e-01  9.60170287e-01  5.99999964e-04  9.99999820e-01]
 [ 6.56986599e-01  7.53902254e-01  6.99999943e-04  9.99999755e-01]
 [ 9.89358247e-01 -1.45500034e-01  7.99999915e-04  9.99999680e-01]
 [ 4.12118485e-01 -9.11130262e-01  8.99999879e-04  9.99999595e-01]]
```

Exercise

Compute these values by hand and check the results with those shown here.

For example, for word 1 (with word index 0, as we're using zero-based counting), the four-dimensional vector is built up as follows:

```
[cjn(0/(10000=1))=0, vaz(0/(10000=1))=1, nja(0/(100002/4=100))=0,
aks(0/(100002/4=100))=1]
```

Notice how we shift from $i = 0$ to $i = 2$ in the code. At every position i in the result vector, we generate a value for i itself and $i+1$, and then we skip over to $i+2$ in order not to overwrite the value at $i+1$.

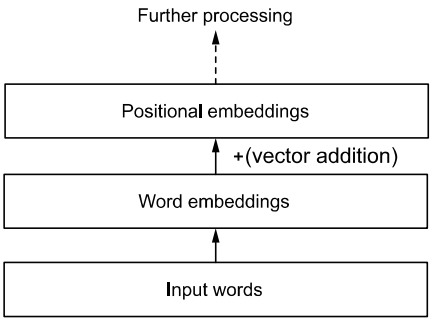

Figure 9.10 Adding position information to word embeddings. Word embeddings are combined through addition with same-sized vectors representing word positions.

Adding these positional vectors v to the original (in this example, four-dimensional) word embeddings w for every word in a sequence $(v + w)$ leads to vectors that represent positional information, as expressed by the diagram in figure 9.10.

Interestingly, only part of the vector space allocated to this positional information is used. Look again at the situation for our toy example: 10 words, with word embeddings

of size 4. We visualize the positional vectors (based on the `sin` and `cos` computations) with `matplotlib` like this (using the `pos_enc` array computed in listing 9.1), which gives us figure 9.11:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

x=range(dimension)
y=range(max_len)
x, y = np.meshgrid(x,y)

plt.xticks(range(dimension))
plt.pcolormesh(x, y, pos_enc ,cmap=cm.gray)
plt.colorbar()
plt.xlabel('Embedding dimension')
plt.ylabel('Word position in sequence')
plt.show()
```

Define a grid of size (x,y).

Plot pos_enc with the position encodings on the grid in grayscale.

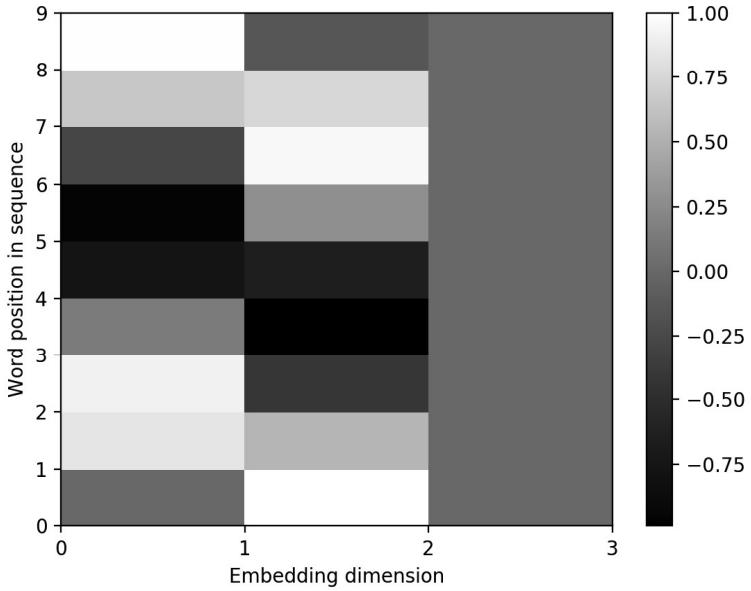

Figure 9.11 Positional embedding for sequences of maximally 10 words, embedding size = 4. Each row is a vector to be added to the word embedding vector for the word in the corresponding position (the y-axis values).

Here you see that the first two dimensions of the positional vector are discriminating between the various word positions in a nonbinary, continuous manner: the gray shades (to which the computed `cos` and `sin` values, all in the interval [−1,1], have

been converted by `matplotlib`) demonstrate this. Consider trying to discriminate between these positions with binary values, on the other hand: that would demand 4 bits ($2^4 = 16$).

Compare this to a more realistic situation: sequences of at most 100 words, 300 dimensions. This produces the picture in figure 9.12. Again, notice how the first, say, 50 dimensions appear to dominate the positional encoding.

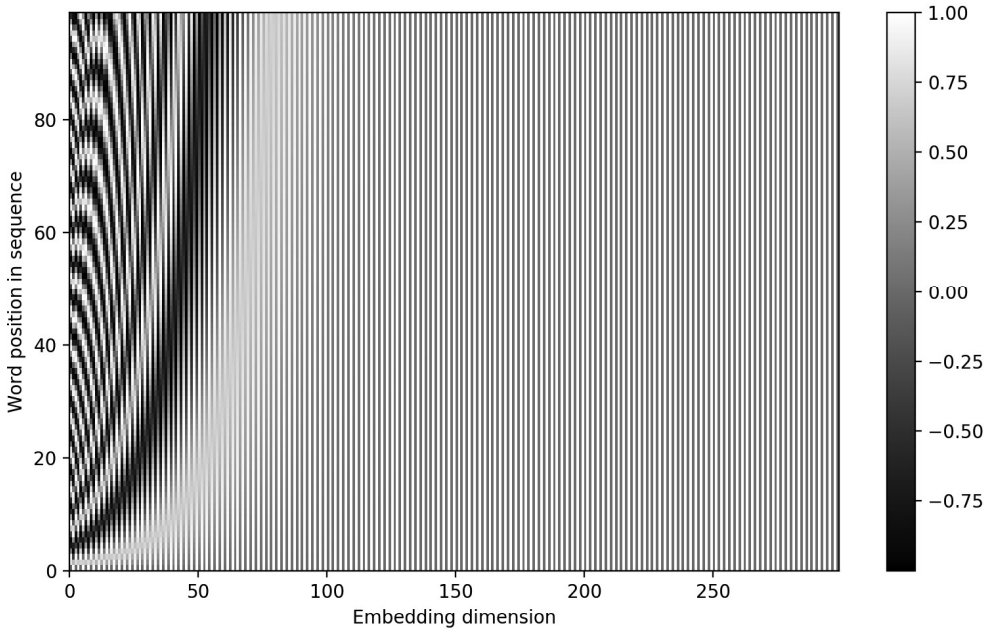

Figure 9.12 Positional embedding for sequences with a maximum of 100 words, embedding size = 300. Each row is a vector to be added to the word embedding vector for the word in the corresponding position (the y-axis values).

So, how is this raw text processed to enter the encoding layers? BERT uses a tokenizer called *WordPiece* (Schuster and Nakajima 2012) that chunks words into subwords. For example, WordPiece chops *echoscopy* into *echo*, *sco*, and *py*. This significantly reduces the chance of encountering out-of-vocabulary words: just as only 26 characters make up English words, the subword combinations create rare words. With this trick, BERT manages to keep its tokenizer vocabulary to just over 30,000 for English.

For example, figure 9.13 shows how raw text enters the first encoder layer. All layers pass on the attention weights their attention heads compute (concatenated) and feed these weights into a `Dense` layer that is the input for the subsequent layer.

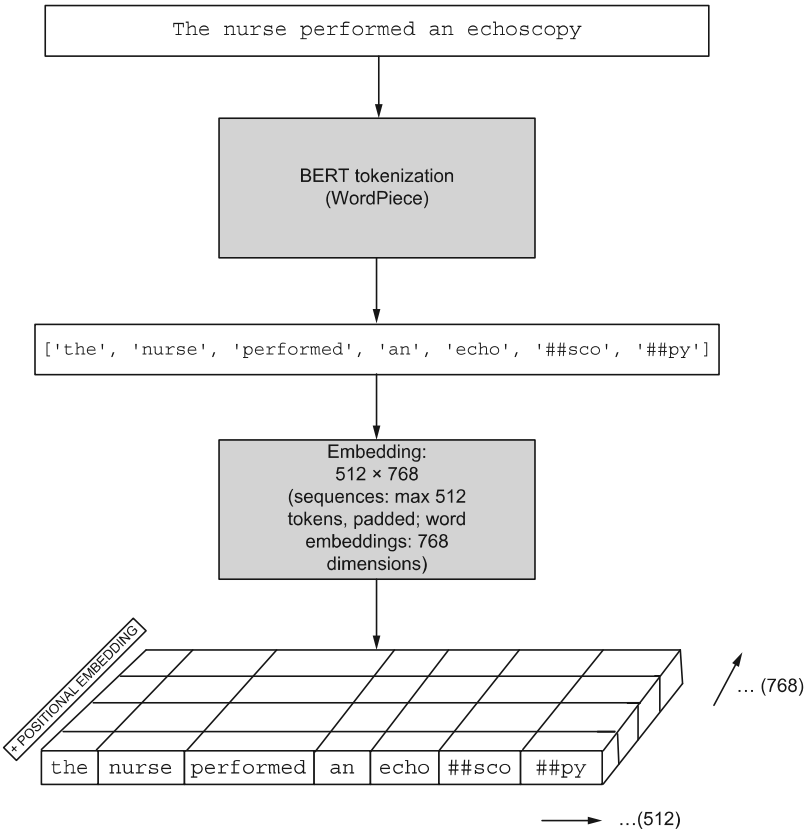

Figure 9.13 Token embedding for BERT. The input sentence first is tokenized. The ## prefixes indicate word pieces: subwords split off from a word by the BERT tokenizer. After that, the sentence is embedded in a 512 × 768 embedding (padded up to 512 token positions) to which the positional vectors are added.

9.3 Transformer decoders

Recall that a Transformer is meant to *transform* sequences (figure 9.14): it can be used for any sequence-to-sequence task like machine translation. The decoder essentially attempts to generate an output sequence word by word based on the encoder representation of the input data. However, feeding some extra information to the decoder helps greatly. Take a look at the Transformer decoder architecture shown in figure 9.15. The first thing to notice is that the decoder also has access to the desired output symbols, excluding the word it should currently predict. That is, it cannot peek into the future: it should generate a current word based on the previous words it has generated. Such a decoder is called *autoregressive*.

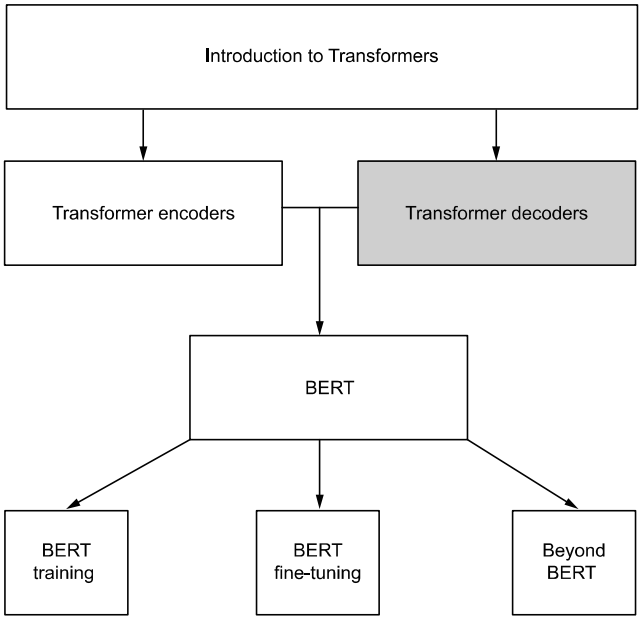

Figure 9.14 Transformer decoders are connected, neighboring constituents of Transformer encoders, decoding encodings to desired output symbols (such as words, when translating language A to language B).

Let's now turn to the right side of the Transformer architecture: the decoders. We repeat the Transformer architecture picture in figure 9.15.

During training, feeding the ground truth data to the decoder mimics this generation process: we act like the decoder has generated the ground truth. One important caveat is that the decoder starts at a *shifted* position compared to the encoder: one word to the right. This is implemented by prefixing the desired output sequence to the encoder with a designated symbol. An extra symbol is added to the desired output to denote the end of the sequence (the ground truth) for the decoder, and decoding stops when the decoder generates this symbol. If the decoder were working on a *non-shifted* version of the desired output sequence, it would be trained to copy that sequence rather than infer the next words.

As an example, suppose we are teaching a Transformer to translate "The nurse performed an echoscopy" to German:

The nurse performed an echoscopy => Die Krankenschwester führte eine Echoskopie durch

During training, the decoder receives the following (assuming a windowed presentation of training data) for predicting "führte"

The latent encoded representation for the entire sentence 'The nurse performed an echoscopy' [START] 'Die Krankenschwester'

and therefore never sees "führte" in its ground truth input when predicting "führte." It cannot see the word it should be predicting.

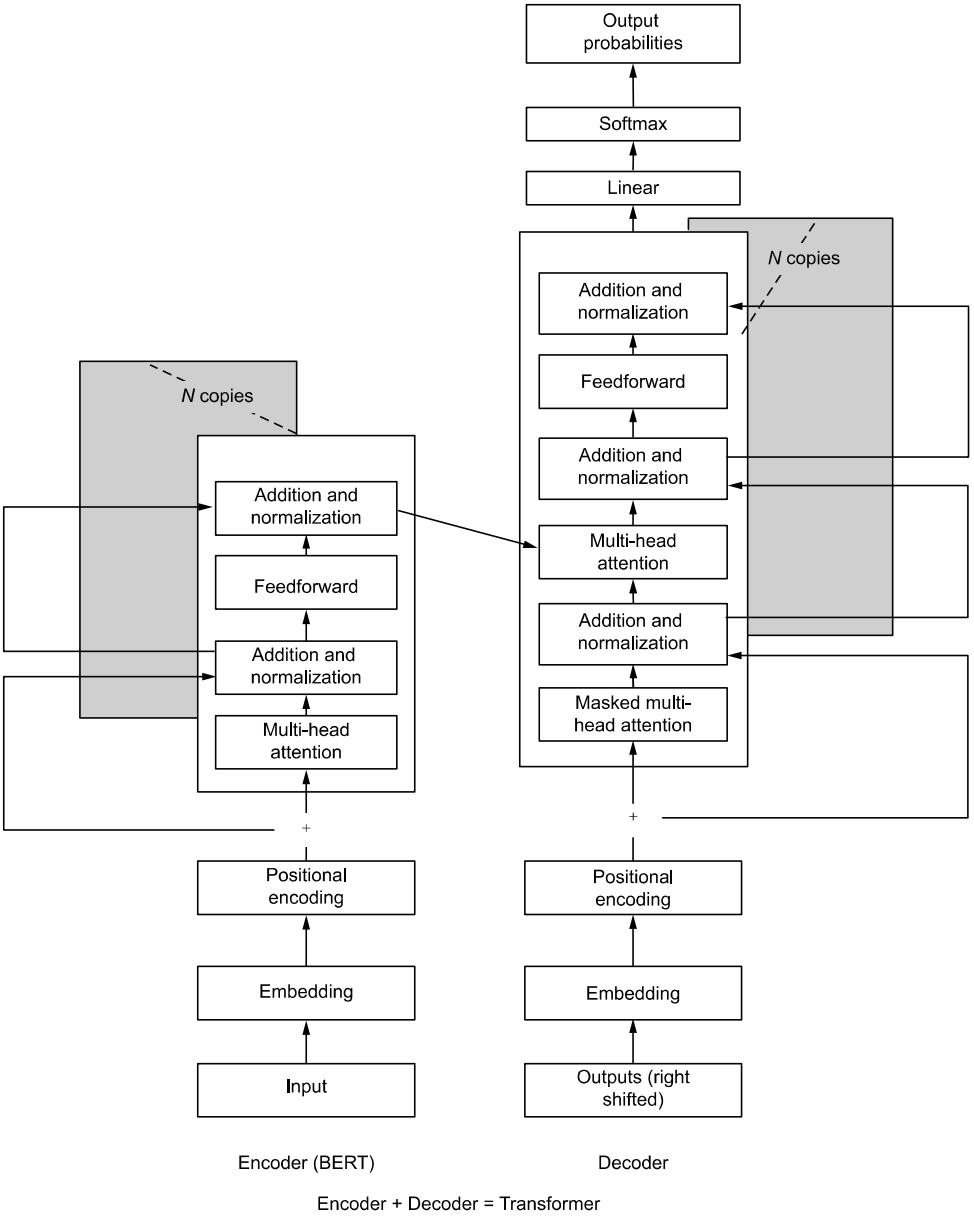

Figure 9.15 The Transformer architecture (from Vaswani et al. 2017). The left part is the encoder, and the right part is the decoder.

TIP Giving the decoder access to the output ground truth data is called *teacher forcing*. It allows us to safeguard the decoder from derailing when making its predictions, since its loss function will take this ground truth into account. Errors made by the decoder are fed all the way back down to the encoder.

In addition to applying the decoder to shifted data, self-attention for a word at position i in the decoder is limited to the words preceding that position: words $w_1,\ldots,w_{i-1}$. This is implemented by masking out those words with a binary filter (a *mask*).

Once trained, the decoder operates in this stepwise fashion:

1. The decoder receives an empty sequence prefixed with the designated start symbol. We refer to that symbol with `[START]`. Similarly, we denote the end symbol with `[END]`.
2. The decoder generates the first word based on the encoder representation that the trained encoder uses to encode its input.
3. In an iterative manner, the decoder completes its decoded sequence word by word. In the second step, it produces the second word based on (again) the encoded representation and `[START] <first word>`, and so on, until it generates the `[END]` symbol.

This implements the `autoregressive_property` for Transformers: the decoder bases its next step (prediction) on its own previously produced output; as said, this is mimicked during training by pretending the input data sent to the decoder is generated by the decoder itself. Now, how does BERT relate to the Transformer architecture?

9.4 BERT: Masked language modeling

BERT is not a full Transformer: it only has the encoder part. Recall, for that matter, its full name: Bidirectional Encoder Representations from Transformers. This makes sense: BERT is not meant to transform sequences into other sequences; it was created to produce word representations (vector embeddings), just like Word2Vec (figure 9.16). Toward this end, BERT uses *masked language models*, which model word distribution patterns by masking out certain words, and has the models predict the words under the masks. Technically, BERT performs *denoising autoencoding*. It attempts to reconstruct a sequence of words from a corrupted, noisy signal: a sequence of words containing masks, indicated with a specific masking symbol. Such masking symbols are not used in Word2Vec. On the fly, BERT infers an optimal encoding of words.

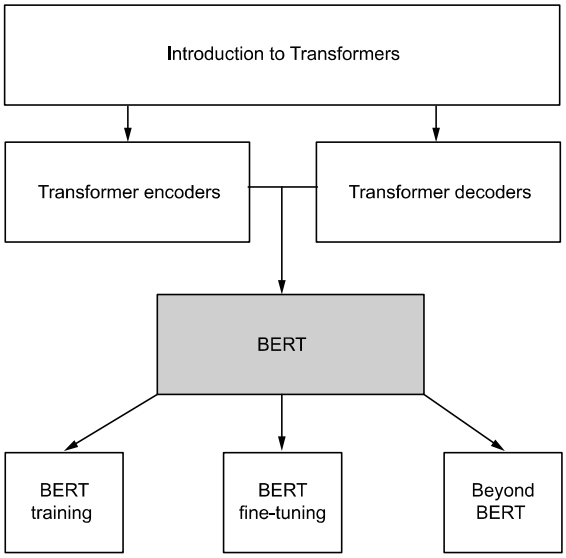

Figure 9.16 BERT represents masked language models, and its task is to pack word context information into word embeddings.

Masked language modeling draws inspiration from a well-established test for assessing the mastery of a native or secondary language and the readability of texts: the *Cloze* test, developed in the 1950s (Taylor 1953). In this test, participants are asked to fill in the blanks in sentences like these:

- The _ performed an echoscopy ('nurse','doctor')
- The _ recovered quickly from the procedure ('patient')

Again, recall from Word2Vec that a limited, unidirectional form of the Cloze test was at the heart of context prediction (predicting, for a given word, a neighbor word or an immediate context). BERT takes this idea further by exploiting *remote* and *non-adjacent* left and right (bidirectional) contexts for predicting a blanked-out, masked word.

9.4.1 Training BERT

As mentioned earlier, BERT consists of the encoder part of a Transformer. BERT is trained on two objectives simultaneously (figure 9.17):

- Predicting words hidden under masks as a bidirectional, masked language model.
- Predicting, for an arbitrary pair of two sentences, whether the second sentence is a natural progression of the first sentence. This task establishes additional relations between words across sentences, and it was added to BERT with question-answering applications in mind, where these relations between sentences often play a role.

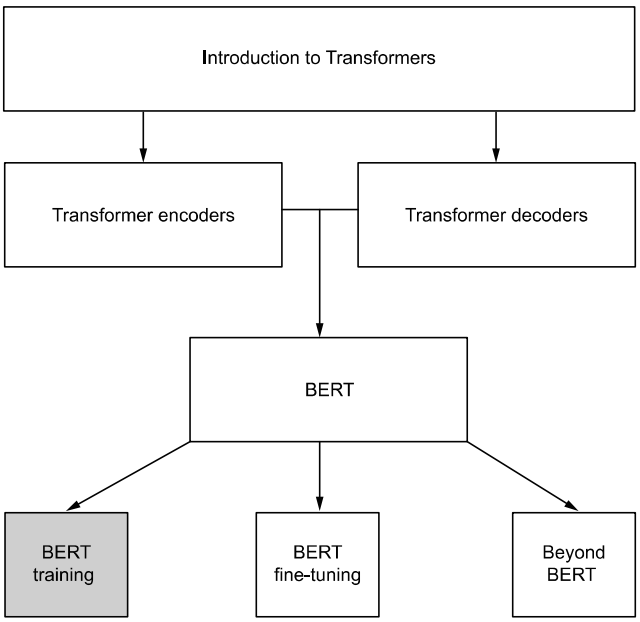

Figure 9.17 Training BERT models

The first objective is implemented in BERT as follows. BERT needs to work on input text in this format:

```
[CLS] The nurse performed an echoscopy [SEP] The patient recovered quickly [SEP]
```

After internal preprocessing, BERT receives a large set of such sentence pairs. The `CLS` tag at the beginning of each pair denotes whether the two sentences form a natural pair for the second objective (it is a binary class label). BERT input data consists of a set of sentences in the specified format, which is partially permuted: it contains real sentence pairs and artificial pairs. The `SEP` tag separates sentences.

TIP Refer to our implementation of Word2Vec (see chapter 3), where we did something similar when generating context samples.

After we apply the WordPiece tokenizer to the tokens in each sentence in a pair, tokens are embedded with a (1,768)-sized vector. The sentence positions are added as vectors, followed by the positional vectors. The maximum sequence length in BERT is 512, so we end up with a matrix of (N,512,768) for N sentence pairs. See figure 9.18.

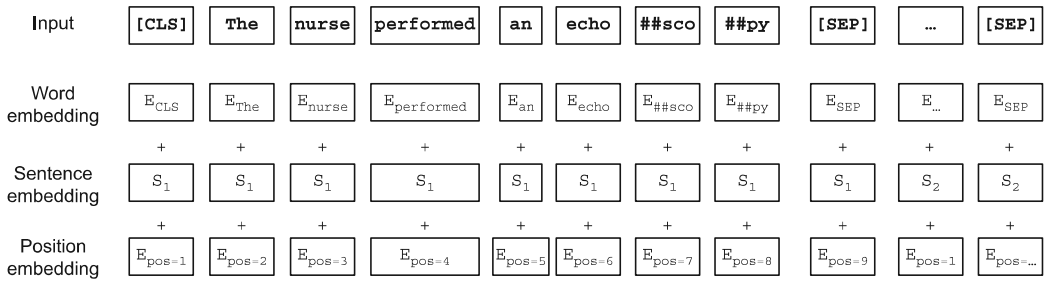

Figure 9.18 BERT input data. Position embedding and sentence flags (indicating the sentence in which a word piece occurs) are combined into word embeddings. These word embeddings are fine-tuned with attention during training BERT.

Internally, BERT generates its training data as follows:

1. From a raw, sentence-based corpus, create sentence pairs (50% natural pairs, 50% random pairs). Create `[CLS] … [SEP] … [SEP]` sequences.
2. Choose 15% random tokens from the WordPiece-processed corpus.
3. For every such token, 80% of the time, mask the token in its sequence with a designated `MASK` token; 12% of the tokens become masked. In 10% (1.5% of the tokens), replace the token with a random token from the BERT vocabulary (which has 30,000 tokens total). In the remaining 10% (1.5% of the tokens), leave the selected token unchanged.

The rationale behind this process is that BERT wants to limit the mismatch of corrupted data (as a result of inserting `MASK` symbols) with subsequent uncorrupted input

data in applications or uncorrupted *downstream* data during fine-tuning (more about that later). The random tokens infuse some random noise into this process, which is a familiar machine learning technique to prevent overfitting models.

The following code implements the masking procedure. It is typically tucked away in BERT; there is no need to explicitly preprocess input data in this manner.

Listing 9.2 Masking

```
def mask_tokens(tokens, vocab):
    masked_tokens=tokens
    predictions=[]
    size=int(0.15 * len(tokens))
    tokens_indices=enumerate(tokens)
    dictionary=dict(tokens_indices)
    sample_tokens=np.random.choice(len(tokens),
        size=size, replace=False)
    for token in sample_tokens:
        if random.random() < 0.8:
            mask = '[MASK]'
        else:
            if random.random() < 0.5:
                mask = tokens[token]
            else:
                mask = random.randint(0, len(vocab) - 1)
        masked_tokens[dictionary[token]] = mask
        predictions.append(
            (token, masked_tokens[token]))
    return masked_tokens, predictions
```

masked_tokens is set to tokens. Random elements of masked_tokens will be masked.

Input to this function consists of a list of tokens (word identifiers—that is, numbers) and a vocabulary (the BERT vocabulary).

The predictions list consists of word identifiers and their eventual masking. This will generate the targets for BERT.

Compute the 15% size of the input tokens. Typically, in BERT, this is 0.15 × 512 = 76 tokens.

token_indices is a list [(word_position, token id),…].

Build a dictionary from the token indices, mapping a position to a word identifier.

Sample 15% random tokens from the input tokens.

In 80% of all cases (based on drawing random numbers), set the mask.

In the remaining 20% of cases, flip a coin: either keep the word unmasked or …

… draw a random word from the BERT vocabulary and set it as a mask.

Store the mask decision (consisting of the mask symbol, the word itself, or a random word).

Store the mask decision for each word identifier.

Return the results.

BERT optimizes both the loss function for sentence pair prediction ("Do these two sentences form a natural pair?") and the loss function for unveiling the masked tokens.

BERT comes in two ready-made flavors:

- *BERT-BASE*—12 encoder layers, 768-size embeddings, 12 multi-attention heads per layer, 110 million parameters
- *BERT-LARGE*—24 encoder layers, 1,024-size embeddings, 16 multi-attention heads per layer, 340 million parameters

Empirically, it was shown in Vaswani et al. (2017) that summing the last four hidden layers of BERT to produce the final embedding for a word produced optimal results,

with an additional slight improvement by not summing but concatenating those four layers (producing an unwieldy $4 \times 768 = 3{,}072$-dimensional vector).

It is important to notice here that BERT, unlike Word2Vec, produces a *contextual embedding*: we typically feed a pretrained BERT model a sentence and extract the embeddings for every word from the attention patterns the input evokes in the model. So, if we feed BERT the following two sentences

I took my money to the bank. I took my guitar to the river bank.

the word *bank* will receive two different embedding vectors (768-dimensional, for BERT). The encoder layers redo their computation of key-query attention values, deploying the pretrained weight matrices for queries, keys, and values and generating contextual embeddings for the words in the input. This contrasts with Word2Vec, where we build a lexicon of words with generic vectors, packing (hopefully) all different contexts. Word2Vec only generates one vector for *bank*. In the next chapter, we will take a closer look at that.

9.4.2 Fine-tuning BERT

Once pretrained and similar to Word2Vec, a BERT embedding can be *fine-tuned* by connecting it to a classification problem, a so-called *downstream task* (figure 9.19). The loss on the classification predictions is fed back end to end into the pretrained BERT model. Fine-tuning BERT is no more complex than switching on a subset of the layers of a pretrained BERT model for fine-tuning (making them trainable, just as we did for Word2Vec). Importing a BERT model and adding a softmax layer plus labeled data is all we need to fine-tune the model.

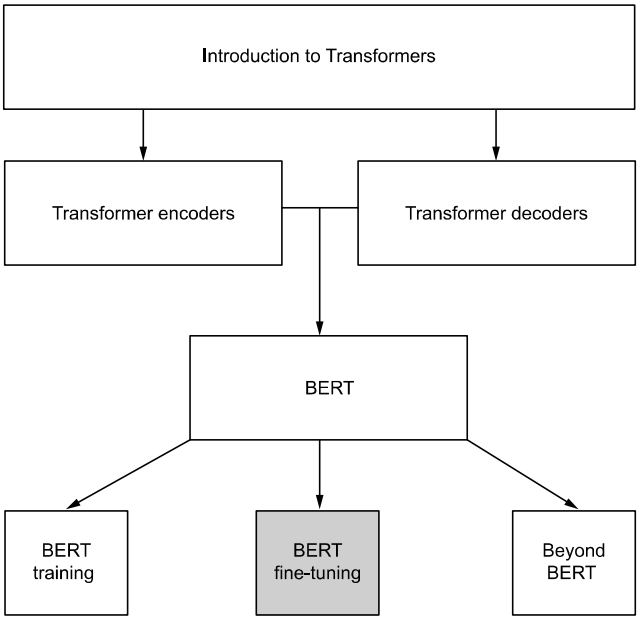

Figure 9.19 Pretrained BERT models can be fine-tuned with downstream tasks.

For instance, we can fine-tune a pretrained BERT model for sentiment analysis, leading to changes in the vector space spanned by BERT that reflect word similarities relevant for, for example, polarity classification. In the next chapter, we will see how to fine-tune BERT for such a task.

BERT-style language models have produced very competitive results across all NLP tasks (Devlin et al. 2018). As mentioned, BERT is an autoencoding model. It corrupts (masks) its input data during training and tries to find an optimal encoding to reconstruct the corrupted (masked) input data. Autoregressive models pair such an encoder with an explicit decoder, allowing for text generation. The GPT-3 model by OpenAI (Brown et al. 2020) has an additional decoder and a whopping 175 billion parameters. It can generate new texts word by word with its decoder, which is very hard to discern from human-produced text, causing quite a stir (and a lot of concern) in the scientific world (see, for example, McGuffie and Newhouse 2020).

9.4.3 Beyond BERT

One recognized shortcoming of BERT is its implementation of masking (figure 9.20). During pretraining, where BERT is trained to predict the words under masks and predict sentence pairs, the masks make perfect sense. But fine-tuning is a different story. The `[MASK]` tag is irrelevant for the data used for fine-tuning BERT, but it is still inserted during BERT's preprocessing stage. Furthermore, if multiple tokens are masked in the pretraining stage of BERT, the interactions between those tokens (and their possible contributions to each other) are lost: BERT makes an *independence* assumption and reconstructs every mask in isolation from neighboring masks. Unlike

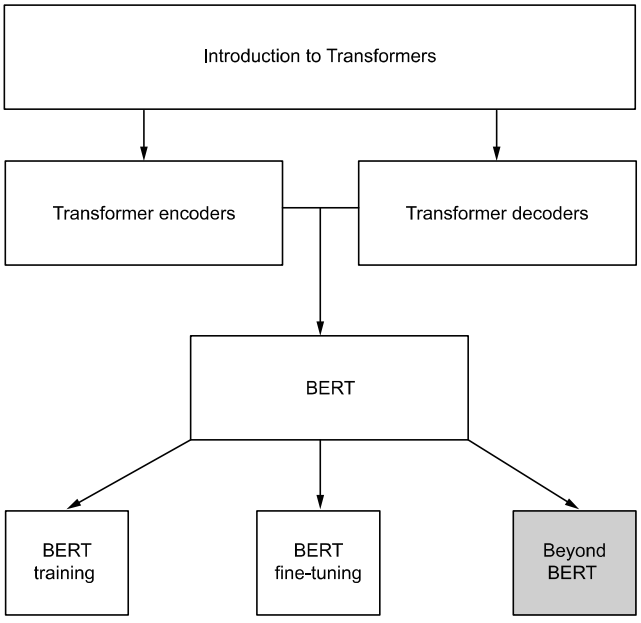

Figure 9.20 BERT has its shortcomings; other approaches attempt to alleviate them.

autoregressive language models, where probabilities of word sequences are based on products of all probabilities for words in the context, BERT addresses every masked word in turn, and its unmasking of one mask will not influence the unmasking of another masked word.

So, for example, if BERT receives the following sequence

```
[MASK] [MASK] and a Happy New Year
```

it will maximize the following sum of probabilities, since for the two masks, `Merry` and `Christmas` are the prediction targets:

```
log(P(Merry|and a happy New year))+log(P(Christmas|and a happy New Year))
```

but none of the probabilities is conditioned on the other, although they clearly are related.

For these reasons, an alternative Transformer-based encoder for creating word embeddings saw the light in 2019: XLNet (Yang et al. 2019). XLNet takes a different approach and optimizes the following in our example:

```
log(P(Merry|and a happy New year))+log(P(Christmas|Merry, and a happy New Year))
```

How does it do that? It deploys *permutation language modeling*: words are predicted for different *factorizations* (permutations). Let's re-address our example

The nurse performed an echoscopy

This sequence has 5! = 120 different permutations. XLNet uses a subset of these permutations (and addresses a subset of the positions in these permutations based on a cutoff estimated hyperparameter). XLNet is autoregressive like Transformers: it does not peek into the future, and every word w at position i is predicted based only on words preceding i. But this is implemented differently: XLNet is autoregressive over the permutations it generates. It uses a permutation language model. Let's look at the following permutation of our example:

nurse, The, an, echoscopy, performed

We can represent this permutation with numbers indicating the original word positions:

[2,1,4,3,5]

Being autoregressive over a permutation means we can only look to the left (the past) when predicting a token in the context of a permutation; this leaves open the possibility that words from the right of a current token will emerge in the left (permuted) context.

To predict word 2, there is no left context, so XLNet needs to optimize as follows:

```
P(nurse|_)
```

For the next token, word 1, there is a left context: word 2. So, we have

```
P(The|nurse)
```

For the next token, word 4 (*echoscopy*), the available context is words 1 and 2. XLNet encodes this context with a mask applied to the permutation at hand.

For word 2 (*nurse*), the mask is

```
[0,0,0,0,0]
```

meaning none of the words in the permutations serves as a context for predicting *nurse*.

For word 1 (*The*), we have

```
[0,1,0,0,0]
```

meaning the original token at position 2 (*nurse*) serves as context for predicting *The*.

For word 4 (*an*), we have as context (1,2), which is *The nurse*, leading to the mask

```
[1,1,0,0,0]
```

and so on. These masks are applied one by one in the model. Notice how the original permutation can be fully reconstructed from these masks. Since XLNet is autoregressive and always *left-looking*, we can directly read the precedence relations between the words in the permuted sequence from the binary masks:

- Word 1 has mask [0,1,0,0,0], implying {2} < 1.
- Word 2 has mask [0,0,0,0,0], implying {}<2: nothing precedes 2. Therefore, 2 must start the permutation sequence.
- Word 3 has mask [1,1,0,1,0], implying {1,2,4} < 3. We can conclude 2,1,4,3 or 2,4,1,3 at this point.
- Word 4 has mask [1,1,0,0,0,0], implying {1,2} < 4. We know that {2}<1, so we can decide for 2,1,4,3.
- Word 5 has mask [1,1,1,1,0], implying 5 closes off the permutation sequence. We have reconstructed 2,1,4,3,5.

With these ingredients, the story is not complete yet. XLNet has no real sense of word order at this point, and it cannot learn that in

```
P(The|nurse)
```

The is in the first word position and *nurse* in the second. We would like XLNet to condition this probability as follows:

```
P(The|pos=1, nurse~2~)
```

In BERT, we solved this by adding positional information vectors to the embeddings. In this case, doing so would lead to an anomaly like

```
P(The|The+pv~1~, nurse+pv~2~)
```

where pv_i is the positional vector for position i. This makes the prediction task trivial. XLNet puts forward a dual self-attention mechanism to solve this conundrum;

basically, it carefully separates the token information from the positional information to omit this situation and inserts an extra 1 for every token in their permutation mask to be able to inspect itself. We will not go any deeper into this; you're invited to look at Yang et al. (2019) for further details.

Performance-wise: what has this given us? XLNet appears to outperform BERT by sizable margins throughout (refer to Yang et al. [2019] for experimental results).

To recap, XLNet does away with explicit masking, which creates a disadvantage for BERT during fine-tuning. It uses a smart trick: a form of permutation language modeling. XLNet also overcomes the independence assumption BERT makes for multiple masks in one sentence. It turns out XLNet performs better than BERT across a wide range of applications. This comes—as always—at a price: permutations are costly to compute and process, and XLNet consequently makes a number of simplifying assumptions for practical reasons.

It is time we turn to practical implementations. In the next chapter, we will implement the necessary methods for working with BERT on actual problems.

Summary

- Transformers are complex encoder-decoder networks based on self-attention.
- BERT is a Transformer encoder.
- BERT derives attention-weighted word embeddings using masked language modeling—a complex attention mechanism—and positional encoding.
- BERT differs from Word2Vec by creating dynamic embeddings that discriminate between different contexts and is similar in its downstream fine-tuning facilities.
- XLNet differs from BERT by omitting the masking of words using a permutation language model, and it may be a better option in some circumstances while being more costly from a computational point of view.

10 Applications of Transformers: Hands-on with BERT

This chapter covers

- Creating a BERT layer for importing existing BERT models
- Training BERT on data
- Fine-tuning BERT
- Extracting embeddings from BERT and inspecting them

This chapter addresses the practicalities of working with the BERT Transformer in your implementations. We will not implement BERT ourselves—that would be a daunting job and unnecessary since BERT has been implemented efficiently in various frameworks, including Keras. But we will get close to the inner workings of BERT code. We saw in chapter 9 that BERT has been reported to improve NLP applications significantly. While we do not carry out an extensive comparison in this chapter, you are encouraged to revisit the applications in the previous chapters and swap, for instance, Word2Vec embeddings with BERT embeddings. With the material in chapters 9 and 10, you should be able to do so.

10.1 Introduction: Working with BERT in practice

The financial costs of pretraining BERT and related models like XLNet from scratch on large amounts of data can be prohibitive (figure 10.1). The original BERT paper (Devlin et al. 2018; see chapter 9) mentions that

> *[The] training of BERT – Large was performed on 16 Cloud TPUs (64 TPU chips total) [with several pretraining phases]. Each pretraining [phase] took 4 days to complete.*

If the authors used Google Cloud TPUs, which are GPUs optimized for TensorFlow computations (TensorFlow is Google's native deep learning formalism), at the current TPU price per hour of $4.50 to $8.00, this would amount to a total pretraining price of $6,912 to $12,288 (16 TPUs × 96 hours × $4.50—$8.00 per hour).

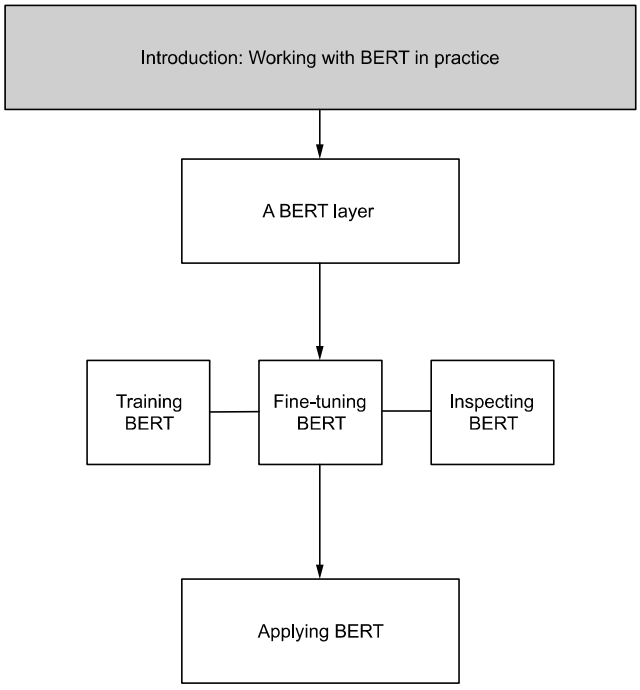

Figure 10.1 Practicalities of working with BERT

For XLNet, with its more complex permutation language modeling approach, non-official estimates amount to a steep pretraining cost of $61,440 (Bradbury 2019). Likewise, the cost of training a Transformer for the GPT-3 language model, which was trained on 1 billion words, has been estimated at a whopping $4.6 million (Li 2020), and such models are only available through commercial licenses.

Luckily, smaller pretrained BERT or XLNet models are becoming increasingly available for free, and they may serve as stepping stones for fine-tuning. For example,

see https://huggingface.co/transformers/pretrained_models.html for an overview of many Transformer models for a variety of languages.

In practice, you can download a pretrained BERT or XLNet model, incorporate it into your network, and fine-tune it with much more manageable, smaller datasets. In this chapter, we see how that works. We begin by incorporating existing BERT models into our models. For this to work, we need a dedicated BERT layer: a landing hub for BERT models.

10.2 A BERT layer

In deep learning networks, BERT layers (figure 10.2), like any other embedding layers, are usually positioned right on top of an input layer, as figure 10.3 shows. They serve a purpose similar to any other embedding layer: they encode words in the input layer to embedding vectors. To be able to work with BERT, we need two things:

- A pretrained BERT model
- A facility for importing such a model and exposing it to the rest of our code

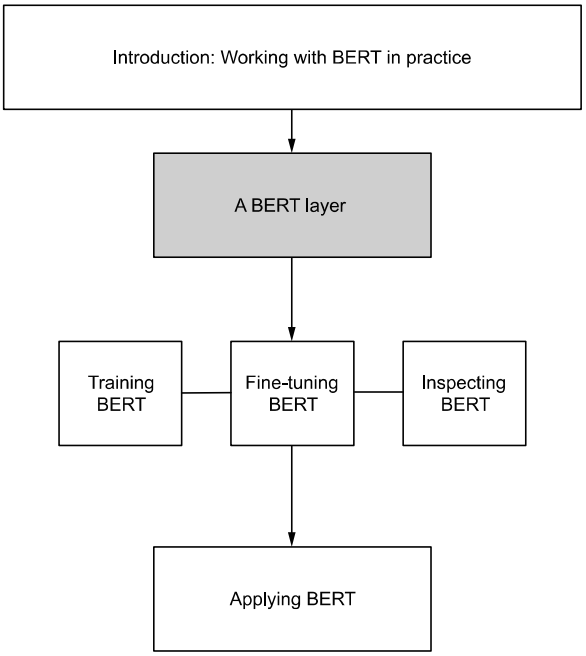

Figure 10.2 Creating a BERT layer in which we can load BERT models

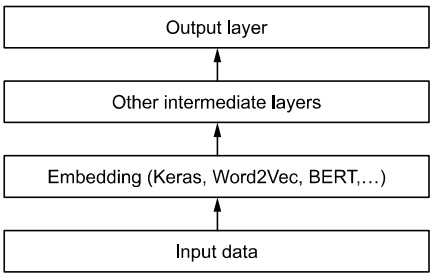

Figure 10.3 The position of a BERT layer in a deep learning network. As with any other embedding layer, data sent through a BERT layer produces embeddings of that data.

Google has made available to a general audience a very valuable platform for obtaining pretrained BERT models: TensorFlow Hub (www.tensorflow.org/hub). This is a platform for downloading not only BERT models and the like but also functional

parts of preconstructed deep learning networks, which, in Google's idiom, are subgraphs of TensorFlow graphs. Recall that TensorFlow is Google's native deep learning formalism.

> **NOTE** Starting with Keras version 2.3 (2019), Keras stopped supporting Theano as a backend and now uses TensorFlow exclusively. It is advisable to use the TensorFlow-embedded version of Keras. The BERT libraries we use in this chapter depend on TensorFlow 2.0 and above. You can read more in Aj_MLstater (2020) and Rosebrock (2019).

This means we can download both models and other useful code from TensorFlow Hub. At https://tfhub.dev/google/collections/bert/1, you will find a large list of all kinds of BERT models. One set of models is based on the official implementation of Devlin et al. (2018); although this implementation has been superseded by others, we will use one of its models.

Let's first define a special-purpose layer to which to attach downloaded models. Recall from chapter 5 that we can define our own layers in Keras as classes. Such class definitions need only three obligatory methods: `init()`, `build()` and `call()`.

Scenario: Working with BERT
You would like to use existing BERT models in your applications. How should you attach such models to your code? You decide to implement a dedicated Keras layer for harboring BERT models.

Listing 10.1 shows how to implement such a Keras BERT layer. Calling this layer entails downloading a BERT model and optimizing it for fine-tuning if desired. The fine-tuning consists of specifying the number of BERT attention layers to be fine-tuned. This is a crucial parameter that determines both the time complexity of the operation and the quality of the final model: fine-tuning more layers generally leads to higher quality, but it comes with a time trade-off.

> **NOTE** The layer used in this example is currently found in many implementations, including those at https://freecontent.manning.com/getting-started-with-baselines and https://gist.github.com/jacobzweig/31aa9b2abed395195d50bf8ad98a3eb9. Its origins are unclear.

Listing 10.1 A dedicated Keras layer for BERT models

```
class BertLayer(tf.keras.layers.Layer):

    def __init__(
        self,
        n_fine_tune_layers=12,
```

The first obligatory method to implement when defining a Python class: initializing the class object

Set the number of layers that need to be fine-tuned if we choose to fine-tune an imported BERT model.

```
    bert_path=
    "https://tfhub.dev/google/bert_uncased_L-12_H-768_A-12/1",
    **kwargs
):
    self.n_fine_tune_layers = n_fine_tune_layers
    self.trainable = True
    self.output_size = 768
    self.bert_path = bert_path

    super(BertLayer, self).__init__(**kwargs)

def build(self, input_shape):
    self.bert = hub.Module(
        self.bert_path,
        trainable=self.trainable,
        name=f"{self.name}_module"
    )
    trainable_vars = self.bert.variables
    trainable_vars =
    [var for var in trainable_vars if not "/cls/" in var.name]
    trainable_vars = trainable_vars[-self.n_fine_tune_layers :]

    for var in trainable_vars:
        self._trainable_weights.append(var)

    for var in self.bert.variables:
        if var not in self._trainable_weights:
            self._non_trainable_weights.append(var)

    super(BertLayer, self).build(input_shape)

def call(self, inputs):
    inputs = [K.cast(x, dtype="int32") for x in inputs]

    input_ids, input_mask, segment_ids = inputs

    bert_inputs = dict(
        input_ids=input_ids, input_mask=input_mask,
        segment_ids=segment_ids
    )
    result = self.bert(inputs=bert_inputs, signature="tokens",
    as_dict=True)[
        "sequence_output"
    ]
    return result

def compute_output_shape(self, input_shape):
    return (input_shape[0], self.output_size)
```

The path to a downloadable (or locally installed) BERT model. This particular URL leads to an uncased (lowercase) ready-made BERT model with 12 hidden layers and a standard output dimension of 768 (see chapter 9).

Set the output size (again, standard 768 for BERT).

Switch the trainable flag to True so the standard setting will be to fine-tune the imported BERT model.

The trainable variables refer to layers in the BERT model we're using. Since BERT models are complex and have many layers, we can opt to limit training (fine-tuning) to a subset of these layers. The BERT model has been loaded into self.bert; it has a folder structure in which paths not containing the string "/cls/" lead to trainable layers (for idiosyncratic reasons).

The build method does administrative work on the weights of the layer. This method will be called automatically by call() on its first invocation.

Create the subset of layers to be fine-tuned.

Append the trainable layer variables to the (initially empty) list of trainable weights.

Store the non-trainable variables (the remaining variables).

The call method defines what happens if we call the layer: that is, apply it as a function to input data.

Base the input data to 32-bit integers to avoid unexpected numerical values.

Decompose the inputs into token IDs, an input mask (defining which tokens the model should attend to), and a list of segment IDs. See listing 10.7 for details.

Store these ingredients as key/value pairs in a dictionary.

Define the result as the application of a self.bert function (referring back to the TensorFlow Hub model) to the input.

An obligatory function that computes the shapes of input and output

As mentioned, the trade-off we make between trainable and non-trainable variables allows us to directly balance time complexity and quality. Fine-tuning fewer attention layers leads to producing lower-quality embeddings but also guarantees a shorter training time. It's up to the developer to balance this trade-off. Before we dive into fine-tuning an existing BERT model, let's take a quick look at the alternative: building your own BERT model and training it on your data from the start.

10.3 Training BERT on your data

If you have enough resources available (data, GPU quota, and patience), it is possible to bootstrap a BERT model from your data (figure 10.4).

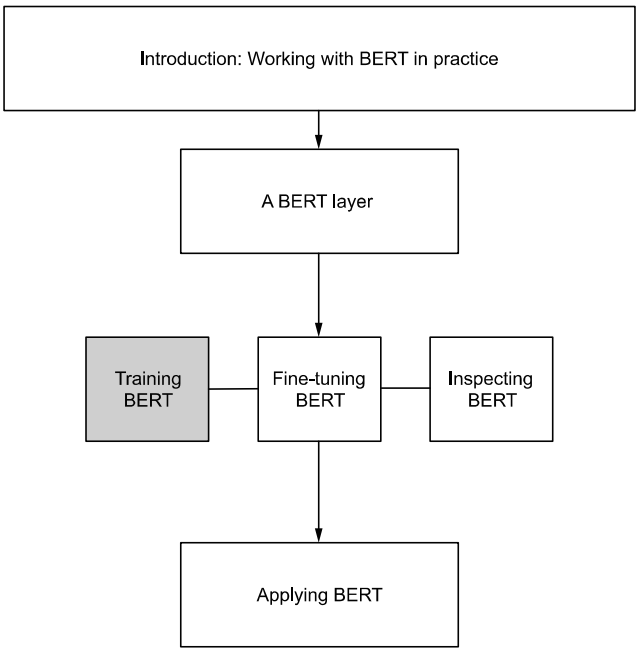

Figure 10.4 The process of training BERT on your data

Scenario: Training BERT on your data

You choose to build a BERT model entirely from scratch. Maybe you want full control over the data you base your BERT model on. This is possible—at the cost of lots of GPU cycles if your dataset is large.

Currently, many Python frameworks based on Keras allow us to work smoothly with BERT, such as `fast-bert` (https://pypi.org/project/fast-bert) and `keras-bert` (https://pypi.org/project/keras-bert). For other options, https://pypi.org/search/?q=bert. `keras-bert` offers simple methods for directly creating a proprietary BERT

model, and we use it in our examples in this chapter. Needless to say, training BERT starts with prepping our data in the right format. Recall from chapter 9 that data comes in the form of paired sentences, because BERT is also trained on the next-sentence-prediction task.

Let's assume we process a set of documents beforehand into a newline-separated list of sentences, like this fragment of Edgar Allen Poe's *The Cask of Amontillado*:

> *He had a weak point — this Fortunato — although in other regards he was a man to be respected and even feared. He prided himself on his connoisseurship in wine. Few Italians have the true virtuoso spirit. For the most part their enthusiasm is adopted to suit the time and opportunity—to practise imposture upon the British and Austrian millionaires.*

Using a simple sentence splitter will get you to this point. Optionally, you can split not just on closing punctuation like periods but also on commas, dashes, and so on; doing so would create additional (pseudo-)sentences like

> *He had a weak point this Fortunato although in other regards he was a man to be respected and even feared*

and would lead to more sentence pairs.

The following function turns such a list into the paired data we need for training a BERT model.

Listing 10.2 Processing input data for BERT

```
def readSentencePairs(fn):
    with open(fn) as f:
        lines = f.readlines()

    pairs=zip(lines, lines[1:])
    paired_sentences=[[a.rstrip().split(),b.rstrip().split()]
    ➥for (a,b) in pairs]

    tokenD = get_base_dict()

    for pairs in paired_sentences:
        for token in pairs[0] + pairs[1]:
            if token not in tokenD:
                tokenD[token] = len(tokenD)
    tokenL = list(tokenD.keys())
    return (paired_sentences,tokenD,tokenL)
```

Invoke the function with a filename.

All lines in the files are read into one list.

From this list, pairs are created with the Python built-in zip().

All sentence pairs in this list are split into words, and newlines are removed.

keras_bert has a base dictionary containing a few special symbols like UNK, CLS, and SEP. This dictionary will be expanded with the words in our data.

For every pair of sentences, any words in the sentences that are not already in the dictionary are added with a new index number.

Gather all tokens in the dictionary into a list.

Return the paired sentences, token dictionary, and token list.

For our example, this produces a nested list of paired sentences split into words:

```
[
  [['he','had','a', 'weak', 'point','—','this', 'Fortunato',...],
```

```
  [['He','prided', 'himself', 'on', 'his', 'connoisseurship', 'in', 'wine']
  ...,
]
```

Next, leaning on the precooked methods in `keras_bert`, we can build and train a BERT model quite swiftly (see https://pypi.org/project/keras-bert). We start by defining a generator function that produces an iterable object with pointers to the next data points. So, instead of generating all the BERT data in one go (which can be prohibitive for large datasets), this generator creates an object for working in an effective and memory-friendly way through large amounts of data.

Listing 10.3 Generating batch data for BERT

```
from keras_bert gen_batch_inputs

def BertGenerator(paired_sentences, tokenD, tokenL):
    while True:
        yield gen_batch_inputs(
            paired_sentences,
            tokenD,
            tokenL,
            seq_len=200,
            mask_rate=0.3,
            swap_sentence_rate=0.5,
        )
```

The generator uses paired-sentence data generated by readSentencePairs().

It enters a perpetual loop (ended by an external control facility that does not bother us here).

Use the keras_bert routine gen_batch_inputs(), and specify the probability for masking out words (mask_rate) and a parameter for swapping sentences that controls using one sentence as the continuation of the next or vice versa; the model has to determine the correct order.

Here is how this generator will be used.

Listing 10.4 Training a proprietary BERT model on data

```
from tensorflow import keras
from keras_bert import get_model, compile_model

def buildBertModel(paired_sentences,tokenD,tokenL, model_path):
    model = get_model(
        token_num=len(tokenD),
        head_num=5,
        transformer_num=12,
        embed_dim=256,
        feed_forward_dim=100,
        seq_len=200,
        pos_num=200,
        dropout_rate=0.05
    )
    compile_model(model)

    model.fit_generator(
        generator=BertGenerator(paired_sentences,tokenD,tokenL),
        steps_per_epoch=100,
        epochs=10
    )
```

The model-building function takes the paired sentence data and a model path as input parameters.

The keras_bert method get_model instantiates a model structure with values for the number of attention heads per layer, the number of transformer layers, the embedding size, the size of the feedforward layers, the length of the token sequences, the corresponding number of positions (for positional encoding), and a dropout rate.

The model is fitted on the data produced by the generator for the number of epochs specified and for a specified number of steps within every epoch.

Save the model after training.

```
    model.save(model_path)

sentences="./my-sentences.txt"
(paired_sentences,tokenD,tokenL)=readSentencePairs(sentences)
model_path="./bert.model"
buildBertModel(paired_sentences,tokenD,tokenL,model_path)
```

This is how everything comes together.

Under the hood, `keras_bert` inserts the `CLS` and `SEP` delimiters in the paired-sentence data and tokenizes the words in the inputs into subwords using the `WordPiece` algorithm (see chapter 9).

Let's use a manual approach to clarify what's happening. We first define a simple class called `InputExample`.

Listing 10.5 `InputExample` class

```
class InputExample(object):
    def __init__(self, text, label=None):
        self.text = text
        self.label = label
```

NOTE This code is based on and extends a few existing repositories: https://towardsdatascience.com/bert-in-keras-with-tensorflow-hub-76bcbc9417b, https://github.com/strongio/keras-bert/blob/master/keras-bert.py, https://www.kaggle.com/igetii/bert-keras, and https://github.com/huggingface/transformers/pull/2891/files.

Instances of the class are just containers holding labeled text items. We need them to store our labeled BERT sentences.

Next, we need a tokenizer to tokenize our input text. We will use another handy BERT Python library: `bert-for-tf2` (BERT for TensorFlow version 2 and above). We install this library under Python3 as follows:

```
sudo pip3 install bert-for-tf2
```

After this, it can be loaded with

```
import bert
```

The following listing shows how to obtain a tokenizer from TensorFlow Hub.

Listing 10.6 Obtaining a tokenizer from TensorFlow Hub

```
import tensorflow_hub as hub
import tensorflow as tf
from bert import bert_tokenization

def create_tokenizer_from_hub_module(bert_hub_path):
  with tf.Graph().as_default():
    bert_module = hub.Module(bert_hub_path)
    tokenization_info = bert_module(signature="tokenization_info",
    ➥as_dict=True)
```

We are operating on a TensorFlow graph (https://www.tensorflow.org/api_docs/python/tf/Graph).

The path to our BERT model on TensorFlow Hub