12 Transformers

Transformers represent one of the most important developments in deep learning. They are based on a processing concept called *attention*, which allows a network to give different weights to different inputs, with weighting coefficients that themselves depend on the input values, thereby capturing powerful inductive biases related to sequential and other forms of data.

These models are known as transformers because they transform a set of vectors in some representation space into a corresponding set of vectors, having the same dimensionality, in some new space. The goal of the transformation is that the new space will have a richer internal representation that is better suited to solving downstream tasks. Inputs to a transformer can take the form of unstructured sets of vectors, ordered sequences, or more general representations, giving transformers broad applicability.

Transformers were originally introduced in the context of natural language pro-

C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_12

cessing, or NLP (where a 'natural' language is one such as English or Mandarin) and have greatly surpassed the previous state-of-the-art approaches based on recurrent neural networks (RNNs). Transformers have subsequently been found to achieve excellent results in many other domains. For example, vision transformers often outperform CNNs in image processing tasks, whereas multimodal transformers that combine multiple types of data, such as text, images, audio, and video, are amongst the most powerful deep learning models.

One major advantage of transformers is that transfer learning is very effective, so that a transformer model can be trained on a large body of data and then the trained model can be applied to many downstream tasks using some form of fine-tuning. A large-scale model that can subsequently be adapted to solve multiple different tasks is known as a *foundation model*. Furthermore, transformers can be trained in a self-supervised way using unlabelled data, which is especially effective with language models since transformers can exploit vast quantities of text available from the internet and other sources. The *scaling hypothesis* asserts that simply by increasing the scale of the model, as measured by the number of learnable parameters, and training on a commensurately large data set, significant improvements in performance can be achieved, even with no architectural changes. Moreover, the transformer is especially well suited to massively parallel processing hardware such as *graphical processing units*, or GPUs, allowing exceptionally large neural network language models having of the order of a trillion (10^{12}) parameters to be trained in reasonable time. Such models have extraordinary capabilities and show clear indications of emergent properties that have been described as the early signs of artificial general intelligence (Bubeck *et al.*, 2023).

The architecture of a transformer can seem complex, or even daunting, to a newcomer as it involves multiple different components working together, in which the various design choices can seem arbitrary. In this chapter we therefore aim to give a comprehensive step-by-step introduction to all the key ideas behind transformers and to provide clear intuition to motivate the design of the various elements. We first describe the transformer architecture and then focus on natural language processing, before exploring other application domains.

12.1. Attention

Section 12.2.5

The fundamental concept that underpins a transformer is *attention*. This was originally developed as an enhancement to RNNs for machine translation (Bahdanau, Cho, and Bengio, 2014). However, Vaswani *et al.* (2017) later showed that significantly improved performance could be obtained by eliminating the recurrence structure and instead focusing exclusively on the attention mechanism. Today, transformers based on attention have completely superseded RNNs in almost all applications.

We will motivate the use of attention using natural language as an example,

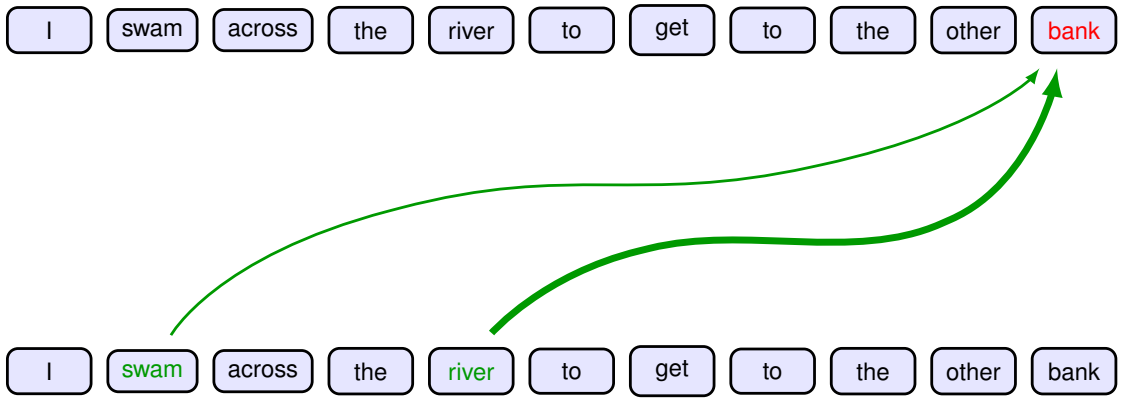

Figure 12.1 Schematic illustration of attention in which the interpretation of the word 'bank' is influenced by the words 'river' and 'swam', with the thickness of each line being indicative of the strength of its influence.

although it has much broader applicability. Consider the following two sentences:

I swam across the river to get to the other bank.

I walked across the road to get cash from the bank.

Here the word 'bank' has different meanings in the two sentences. However, this can be detected only by looking at the context provided by other words in the sequence. We also see that some words are more important than others in determining the interpretation of 'bank'. In the first sentence, the words 'swam' and 'river' most strongly indicate that 'bank' refers to the side of a river, whereas in the second sentence, the word 'cash' is a strong indicator that 'bank' refers to a financial institution. We see that to determine the appropriate interpretation of 'bank', a neural network processing such a sentence should *attend* to, in other words rely more heavily on, specific words from the rest of the sequence. This concept of attention is illustrated in Figure 12.1.

Moreover, we also see that the particular locations that should receive more attention depend on the input sequence itself: in the first sentence it is the second and fifth words that are important whereas in the second sentence it is the eighth word. In a standard neural network, different inputs will influence the output to different extents according to the values of the weights that multiply those inputs. Once the network is trained, however, those weights, and their associated inputs, are fixed. By contrast, attention uses weighting factors whose values depend on the specific input data. Figure 12.2 shows the attention weights from a section of a transformer network trained on natural language.

When we discuss natural language processing, we will see how word embedding can be used to map words into vectors in an embedding space. These vectors can then be used as inputs for subsequent neural network processing. These embeddings capture elementary semantic properties, for example by mapping words with similar meanings to nearby locations in the embedding space. One characteristic of such embeddings is that a given word always maps to the same embedding vector.

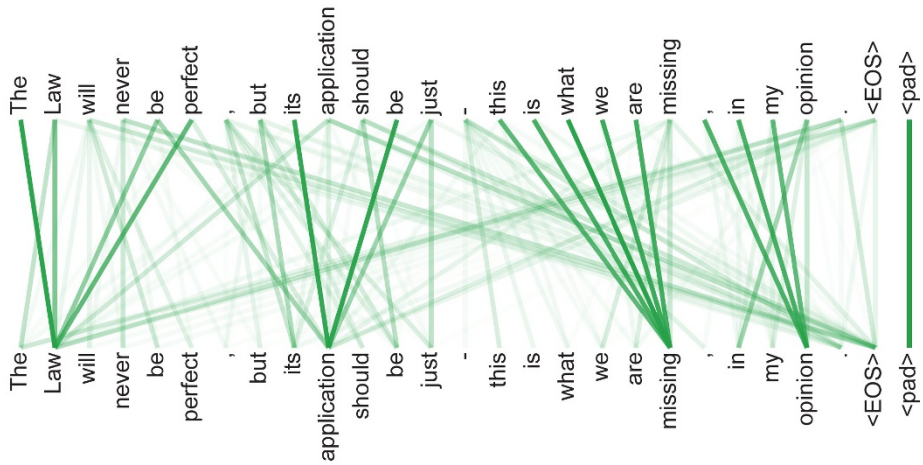

Figure 12.2 An example of learned attention weights. [From Vaswani *et al.* (2017) with permission.]

A transformer can be viewed as a richer form of embedding in which a given vector is mapped to a location that depends on the other vectors in the sequence. Thus, the vector representing 'bank' in our example above could map to different places in a new embedding space for the two different sentences. For example, in the first sentence the transformed representation might put 'bank' close to 'water' in the embedding space, whereas in the second sentence the transformed representation might put it close to 'money'.

As an example of attention, consider the modelling of proteins. We can view a protein as a one-dimensional sequence of molecular units called amino acids. A protein can comprise potentially hundreds or thousands of such units, each of which is given by one of 22 possibilities. In a living cell, a protein folds up into a three-dimensional structure in which amino acids that are widely separated in the one-dimensional sequence can become physically close in three-dimensional space and thereby interact. Transformer models allows these distant amino acids to 'attend' to each other thereby greatly improving the accuracy with which their 3-dimensional structure can be modelled (Vig *et al.*, 2020).

Figure 1.2

12.1.1 Transformer processing

The input data to a transformer is a set of vectors $\{\mathbf{x}_n\}$ of dimensionality D, where $n = 1, \ldots, N$. We refer to these data vectors as *tokens*, where a token might, for example, correspond to a word within a sentence, a patch within an image, or an amino acid within a protein. The elements x_{ni} of the tokens are called *features*. Later we will see how to construct these token vectors for natural language data and for images. A powerful property of transformers is that we do not have to design a new neural network architecture to handle a mix of different data types but instead can simply combine the data variables into a joint set of tokens.

Before we can gain a clear understanding of the operation of a transformer, it

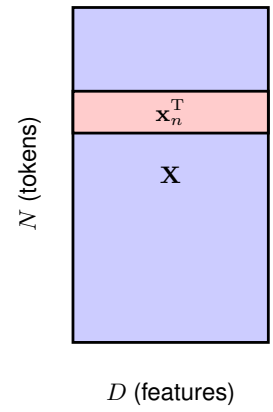

Figure 12.3 The structure of the data matrix $\mathbf{X}$, of dimension $N \times D$, in which row n represents the transposed data vector $\mathbf{x}_n^{\mathrm{T}}$.

is important to be precise about notation. We will follow the standard convention and combine the data vectors into a matrix $\mathbf{X}$ of dimensions $N \times D$ in which the nth row comprises the token vector $\mathbf{x}_n^{\mathrm{T}}$, and where $n = 1, \ldots, N$ labels the rows, as illustrated in Figure 12.3. Note that this matrix represents one set of input tokens, and that for most applications, we will require a data set containing many sets of tokens, such as independent passages of text where each word is represented as one token. The fundamental building block of a transformer is a function that takes a data matrix as input and creates a transformed matrix $\widetilde{\mathbf{X}}$ of the same dimensionality as the output. We can write this function in the form

$$\widetilde{\mathbf{X}} = \mathrm{TransformerLayer}\left[\mathbf{X}\right]. \tag{12.1}$$

We can then apply multiple transformer layers in succession to construct deep networks capable of learning rich internal representations. Each transformer layer contains its own weights and biases, which can be learned using gradient descent using an appropriate cost function, as we will discuss in detail later in the chapter.

Section 12.3

A single transformer layer itself comprises two stages. The first stage, which implements the attention mechanism, mixes together the corresponding features from different token vectors across the columns of the data matrix, whereas the second stage then acts on each row independently and transforms the features within each token vector. We start by looking at the attention mechanism.

12.1.2 Attention coefficients

Suppose that we have a set of input tokens $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in an embedding space and we want to map this to another set $\mathbf{y}_1, \ldots, \mathbf{y}_N$ having the same number of tokens but in a new embedding space that captures a richer semantic structure. Consider a particular output vector $\mathbf{y}_n$. The value of $\mathbf{y}_n$ should depend not just on the corresponding input vector $\mathbf{x}_n$ but on all the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in the set. With attention, this dependence should be stronger for those inputs $\mathbf{x}_m$ that are particularly important for determining the modified representation of $\mathbf{y}_n$. A simple way to achieve this is to define each output vector $\mathbf{y}_n$ to be a linear combination of the input vectors

$\mathbf{x}_1, \ldots, \mathbf{x}_N$ with weighting coefficients a_{nm}:

$$\mathbf{y}_n = \sum_{m=1}^{N} a_{nm}\mathbf{x}_m \tag{12.2}$$

where a_{nm} are called *attention weights*. The coefficients should be close to zero for input tokens that have little influence on the output $\mathbf{y}_n$ and largest for inputs that have most influence. We therefore constrain the coefficients to be non-negative to avoid situations in which one coefficient can become large and positive while another coefficient compensates by becoming large and negative. We also want to ensure that if an output pays more attention to a particular input, this will be at the expense of paying less attention to the other inputs, and so we constrain the coefficients to sum to unity. Thus, the weighting coefficients must satisfy the following two constraints:

$$a_{nm} \geqslant 0 \tag{12.3}$$

$$\sum_{m=1}^{N} a_{nm} = 1. \tag{12.4}$$

Exercise 12.1

Together these imply that each coefficient lies in the range $0 \leqslant a_{nm} \leqslant 1$ and so the coefficients define a 'partition of unity'. For the special case $a_{mm} = 1$, it follows that $a_{nm} = 0$ for $n \neq m$, and therefore $\mathbf{y}_m = \mathbf{x}_m$ so that the input vector is unchanged by the transformation. More generally, the output $\mathbf{y}_m$ is a blend of the input vectors with some inputs given more weight than others.

Note that we have a different set of coefficients for each output vector $\mathbf{y}_n$, and the constraints (12.3) and (12.4) apply separately for each value of n. These coefficients a_{nm} depend on the input data, and we will shortly see how to calculate them.

12.1.3 Self-attention

The next question is how to determine the coefficients a_{nm}. Before we discuss this in detail, it is useful to first introduce some terminology taken from the field of information retrieval. Consider the problem of choosing which movie to watch in an online movie streaming service. One approach would be to associate each movie with a list of attributes describing things such as the genre (comedy, action, etc.), the names of the leading actors, the length of the movie, and so on. The user could then search through a catalogue to find a movie that matches their preferences. We could automate this by encoding the attributes of each movie in a vector called the *key*. The corresponding movie file itself is called a *value*. Similarly, the user could then provide their own personal vector of values for the desired attributes, which we call the *query*. The movie service could then compare the query vector with all the key vectors to find the best match and send the corresponding movie to the user in the form of the value file. We can think of the user 'attending' to the particular movie whose key most closely matches their query. This would be considered a form of *hard attention* in which a single value vector is returned. For the transformer, we generalize this to *soft attention* in which we use continuous variables to measure

the degree of match between queries and keys and we then use these variables to weight the influence of the value vectors on the outputs. This will also ensure that the transformer function is differentiable and can therefore be trained by gradient descent.

Following the analogy with information retrieval, we can view each of the input vectors $\mathbf{x}_n$ as a value vector that will be used to create the output tokens. We also use the vector $\mathbf{x}_n$ directly as the key vector for input token n. That would be analogous to using the movie itself to summarize the characteristics of the movie. Finally, we can use $\mathbf{x}_m$ as the query vector for output $\mathbf{y}_m$, which can then be compared to each of the key vectors. To see how much the token represented by $\mathbf{x}_n$ should attend to the token represented by $\mathbf{x}_m$, we need to work out how similar these vectors are. One simple measure of similarity is to take their dot product $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m$. To impose the constraints (12.3) and (12.4), we can define the weighting coefficients a_{nm} by using the *softmax* function to transform the dot products:

Section 5.3

$$a_{nm} = \frac{\exp(\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m)}{\sum_{m'=1}^{N}\exp(\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_{m'})}. \tag{12.5}$$

Note that in this case there is no probabilistic interpretation of the softmax function and it is simply being used to normalize the attention weights appropriately.

So in summary, each input vector $\mathbf{x}_n$ is transformed to a corresponding output vector $\mathbf{y}_n$ by taking a linear combination of input vectors of the form (12.2) in which the weight a_{nm} applied to input vector $\mathbf{x}_m$ is given by the softmax function (12.5) defined in terms of the dot product $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m$ between the query $\mathbf{x}_n$ for input n and the key $\mathbf{x}_m$ associated with input m. Note that, if all the input vectors are orthogonal, then each output vector is simply equal to the corresponding input vector so that $\mathbf{y}_m = \mathbf{x}_m$ for $m = 1, \ldots, N$.

Exercise 12.3

We can write (12.2) in matrix notation by using the data matrix $\mathbf{X}$, along with the analogous $N \times D$ output matrix $\mathbf{Y}$, whose rows are given by $\mathbf{y}_m$, so that

$$\mathbf{Y} = \mathrm{Softmax}\left[\mathbf{X}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X} \tag{12.6}$$

where $\mathrm{Softmax}[\mathbf{L}]$ is an operator that takes the exponential of every element of a matrix $\mathbf{L}$ and then normalizes each row independently to sum to one. From now on, we will focus on matrix notation for clarity.

This process is called *self-attention* because we are using the same sequence to determine the queries, keys, and values. We will encounter variants of this attention mechanism later in this chapter. Also, because the measure of similarity between query and key vectors is given by a dot product, this is known as *dot-product self-attention*.

12.1.4 Network parameters

As it stands, the transformation from input vectors $\{\mathbf{x}_n\}$ to output vectors $\{\mathbf{y}_n\}$ is fixed and has no capacity to learn from data because it has no adjustable parameters. Furthermore, each of the feature values within a token vector $\mathbf{x}_n$ plays an equal role in determining the attention coefficients, whereas we would like the network to

have the flexibility to focus more on some features than others when determining token similarity. We can address both issues if we define modified feature vectors given by a linear transformation of the original vectors in the form

$$\widetilde{\mathbf{X}} = \mathbf{X}\mathbf{U} \tag{12.7}$$

where $\mathbf{U}$ is a $D \times D$ matrix of learnable weight parameters, analogous to a 'layer' in a standard neural network. This gives a modified transformation of the form

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{X}\mathbf{U}\mathbf{U}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X}\mathbf{U}. \tag{12.8}$$

Although this has much more flexibility, it has the property that the matrix

$$\mathbf{X}\mathbf{U}\mathbf{U}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}} \tag{12.9}$$

is symmetric, whereas we would like the attention mechanism to support significant asymmetry. For example, we might expect that 'chisel' should be strongly associated with 'tool' since every chisel is a tool, whereas 'tool' should only be weakly associated with 'chisel' because there are many other kinds of tools besides chisels. Although the softmax function means the resulting matrix of attention weights is not itself symmetric, we can create a much more flexible model by allowing the queries and the keys to have independent parameters. Furthermore, the form (12.8) uses the same parameter matrix $\mathbf{U}$ to define both the value vectors and the attention coefficients, which again seems like an undesirable restriction.

We can overcome these limitations by defining separate query, key, and value matrices each having their own independent linear transformations:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(\mathrm{q})} \tag{12.10}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}^{(\mathrm{k})} \tag{12.11}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}^{(\mathrm{v})} \tag{12.12}$$

where the weight matrices $\mathbf{W}^{(\mathrm{q})}$, $\mathbf{W}^{(\mathrm{k})}$, and $\mathbf{W}^{(\mathrm{v})}$ represent parameters that will be learned during the training of the final transformer architecture. Here the matrix $\mathbf{W}^{(\mathrm{k})}$ has dimensionality $D \times D_{\mathrm{k}}$ where D_{k} is the length of the key vector. The matrix $\mathbf{W}^{(\mathrm{q})}$ must have the same dimensionality $D \times D_{\mathrm{k}}$ as $\mathbf{W}^{(\mathrm{k})}$ so that we can form dot products between the query and key vectors. A typical choice is $D_{\mathrm{k}} = D$. Similarly, $\mathbf{W}^{(\mathrm{v})}$ is a matrix of size $D \times D_{\mathrm{v}}$, where D_{v} governs the dimensionality of the output vectors. If we set $D_{\mathrm{v}} = D$, so that the output representation has the same dimensionality as the input, this will facilitate the inclusion of residual connections, which we discuss later. Also, multiple transformer layers can be stacked on top of each other if each layer has the same dimensionality. We can then generalize (12.6) to give

Section 12.1.7

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{Q}\mathbf{K}^{\mathrm{T}}\right]\mathbf{V} \tag{12.13}$$

where $\mathbf{Q}\mathbf{K}^{\mathrm{T}}$ has dimension $N \times N$, and the matrix $\mathbf{Y}$ has dimension $N \times D_{\mathrm{v}}$. The calculation of the matrix $\mathbf{Q}\mathbf{K}^{\mathrm{T}}$ is illustrated in Figure 12.4, whereas the evaluation of the matrix $\mathbf{Y}$ is illustrated in Figure 12.5.

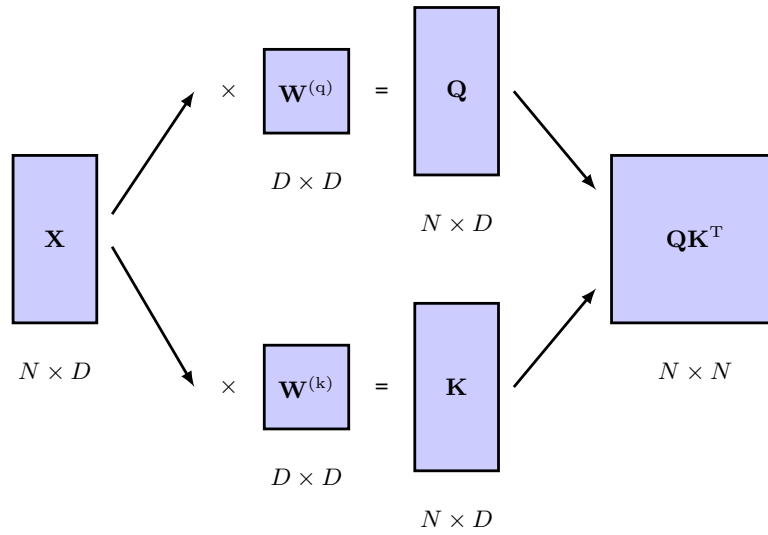

Figure 12.4 Illustration of the evaluation of the matrix $\mathbf{QK}^{\mathrm{T}}$, which determines the attention coefficients in a transformer. The input $\mathbf{X}$ is separately transformed using (12.10) and (12.11) to give the query matrix $\mathbf{Q}$ and key matrix $\mathbf{K}$, respectively, which are then multiplied together.

Section 6.2.1

In practice we can also include bias parameters in these linear transformations. However, the bias parameters can be absorbed into the weight matrices, as we did with standard neural networks, by augmenting the data matrix $\mathbf{X}$ with an additional column of 1's and by augmenting the weight matrices with an additional row of parameters to represent the biases. From now on we will treat the bias parameters as implicit to avoid cluttering the notation.

Compared to a conventional neural network, the signal paths have multiplicative relations between activation values. Whereas standard networks multiply activations by fixed weights, here the activations are multiplied by the data-dependent attention coefficients. This means, for example, that if one of the attention coefficients is close to zero for a particular choice of input vector, the resulting signal path will ignore the corresponding incoming signal, which will therefore have no influence

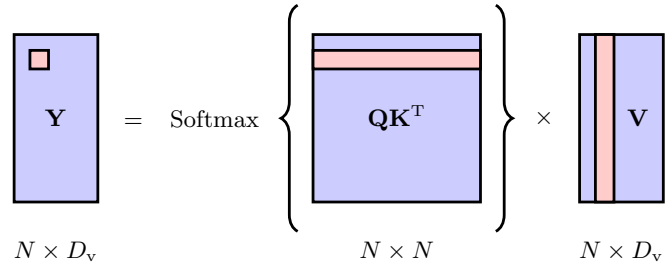

Figure 12.5 Illustration of the evaluation of the output from an attention layer given the query, key, and value matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, respectively. The entry at the position highlighted in the output matrix $\mathbf{Y}$ is obtained from the dot product of the highlighted row and column of the $\mathrm{Softmax}\left[\mathbf{QK}^{\mathrm{T}}\right]$ and $\mathbf{V}$ matrices, respectively.

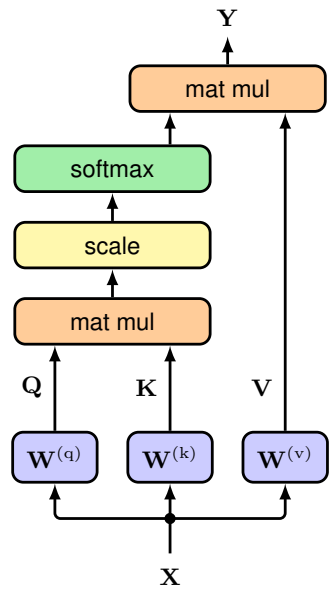

Figure 12.6 Information flow in a scaled dot-product self-attention neural network layer. Here 'mat mul' denotes matrix multiplication, and 'scale' refers to the normalization of the argument to the softmax using $\sqrt{D_{\mathrm{k}}}$. This structure constitutes a single attention 'head'.

on the network outputs. By contrast, if a standard neural network learns to ignore a particular input or hidden-unit variable, it does so for all input vectors.

12.1.5 Scaled self-attention

There is one final refinement we can make to the self-attention layer. Recall that the gradients of the softmax function become exponentially small for inputs of high magnitude, just as happens with $\tanh$ or logistic-sigmoid activation functions. To help prevent this from happening, we can re-scale the product of the query and key vectors before applying the softmax function. To derive a suitable scaling, note that if the elements of the query and key vectors were all independent random numbers with zero mean and unit variance, then the variance of the dot product would be D_{k}. We therefore normalize the argument to the softmax using the standard deviation given by the square root of D_{k}, so that the output of the attention layer takes the form

Exercise 12.4

$$\mathbf{Y} = \mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \equiv \mathrm{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{D_{\mathrm{k}}}}\right]\mathbf{V}. \tag{12.14}$$

This is called *scaled dot-product self-attention*, and is the final form of our self-attention neural network layer. The structure of this layer is summarized in Figure 12.6 and in Algorithm 12.1.

12.1.6 Multi-head attention

The attention layer described so far allows the output vectors to attend to data-dependent patterns of input vectors and is called an *attention head*. However, there

Algorithm 12.1: Scaled dot-product self-attention

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Weight matrices $\{\mathbf{W}^{(\mathrm{q})}, \mathbf{W}^{(\mathrm{k})}\} \in \mathbb{R}^{D \times D_\mathrm{k}}$ and $\mathbf{W}^{(\mathrm{v})} \in \mathbb{R}^{D \times D_\mathrm{v}}$
Output: $\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \in \mathbb{R}^{N \times D_\mathrm{v}} : \{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$

$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(\mathrm{q})}$ // compute queries $\mathbf{Q} \in \mathbb{R}^{N \times D_\mathrm{k}}$
$\mathbf{K} = \mathbf{X}\mathbf{W}^{(\mathrm{k})}$ // compute keys $\mathbf{K} \in \mathbb{R}^{N \times D_\mathrm{k}}$
$\mathbf{V} = \mathbf{X}\mathbf{W}^{(\mathrm{v})}$ // compute values $\mathbf{V} \in \mathbb{R}^{N \times D}$
return $\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{D_\mathrm{k}}}\right]\mathbf{V}$

might be multiple patterns of attention that are relevant at the same time. In natural language, for example, some patterns might be relevant to tense whereas others might be associated with vocabulary. Using a single attention head can lead to averaging over these effects. Instead we can use multiple attention heads in parallel. These consist of identically structured copies of the single head, with independent learnable parameters that govern the calculation of the query, key, and value matrices. This is analogous to using multiple different filters in each layer of a convolutional network.

Suppose we have H heads indexed by $h = 1, \ldots, H$ of the form

$$\mathbf{H}_h = \mathrm{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h) \tag{12.15}$$

where $\mathrm{Attention}(\cdot, \cdot, \cdot)$ is given by (12.14), and we have defined separate query, key, and value matrices for each head using

$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{q})} \tag{12.16}$$

$$\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{k})} \tag{12.17}$$

$$\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{v})}. \tag{12.18}$$

The heads are first concatenated into a single matrix, and the result is then linearly transformed using a matrix $\mathbf{W}^{(\mathrm{o})}$ to give a combined output in the form

$$\mathbf{Y}(\mathbf{X}) = \mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right]\mathbf{W}^{(\mathrm{o})}. \tag{12.19}$$

This is illustrated in Figure 12.7.

Each matrix $\mathbf{H}_h$ has dimension $N \times D_\mathrm{v}$, and so the concatenated matrix has dimension $N \times HD_\mathrm{v}$. This is transformed by the linear matrix $\mathbf{W}^{(\mathrm{o})}$ of dimension $HD_\mathrm{v} \times D$ to give the final output matrix $\mathbf{Y}$ of dimension $N \times D$, which is the same as the original input matrix $\mathbf{X}$. The elements of the matrix $\mathbf{W}^{(\mathrm{o})}$ are learned during the training phase along with the query, key, and value matrices. Typically D_v is

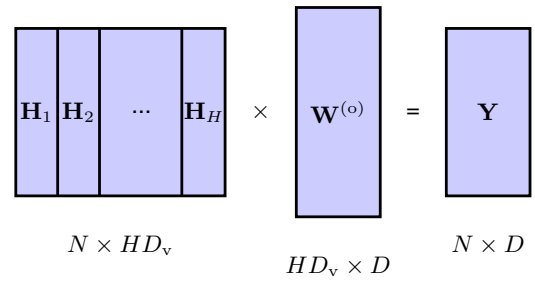

Figure 12.7 Network architecture for multi-head attention. Each head comprises the structure shown in Figure 12.6, and has its own key, query, and value parameters. The outputs of the heads are concatenated and then linearly projected back to the input data dimensionality.

chosen to be equal to D/H so that the resulting concatenated matrix has dimension $N \times D$. Multi-head attention is summarized in Algorithm 12.2, and the information flow in a multi-head attention layer is illustrated in Figure 12.8.

Note that the formulation of multi-head attention given above, which follows that used in the research literature, includes some redundancy in the successive multiplication of the $\mathbf{W}^{(\mathrm{v})}$ matrix for each head and the output matrix $\mathbf{W}^{(\mathrm{o})}$. Removing this redundancy allows a multi-head self-attention layer to be written as a sum over contributions from each of the heads separately.

Exercise 12.5

12.1.7 Transformer layers

Multi-head self-attention forms the core architectural element in a transformer network. We know that neural networks benefit greatly from depth, and so we would like to stack multiple self-attention layers on top of each other. To improve training

Algorithm 12.2: Multi-head attention

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Query weight matrices $\{\mathbf{W}_1^{(\mathrm{q})}, \ldots, \mathbf{W}_H^{(\mathrm{q})}\} \in \mathbb{R}^{D \times D}$
Key weight matrices $\{\mathbf{W}_1^{(\mathrm{k})}, \ldots, \mathbf{W}_H^{(\mathrm{k})}\} \in \mathbb{R}^{D \times D}$
Value weight matrices $\{\mathbf{W}_1^{(\mathrm{v})}, \ldots, \mathbf{W}_H^{(\mathrm{v})}\} \in \mathbb{R}^{D \times D_{\mathrm{v}}}$
Output weight matrix $\mathbf{W}^{(\mathrm{o})} \in \mathbb{R}^{HD_{\mathrm{v}} \times D}$
Output: $\mathbf{Y} \in \mathbb{R}^{N \times D} : \{\mathbf{y}_1, \ldots, \mathbf{x}_N\}$

```
// compute self-attention for each head (Algorithm 12.1)
```
for $h = 1, \ldots, H$ **do**
  $\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{q})}, \quad \mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{k})}, \quad \mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{v})}$
  $\mathbf{H}_h = \mathrm{Attention}\left(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h\right)$ // $\mathbf{H}_h \in \mathbb{R}^{N \times D_{\mathrm{v}}}$
end for
$\mathbf{H} = \mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_N\right]$ // concatenate heads
return $\mathbf{Y}(\mathbf{X}) = \mathbf{H}\mathbf{W}^{(\mathrm{o})}$

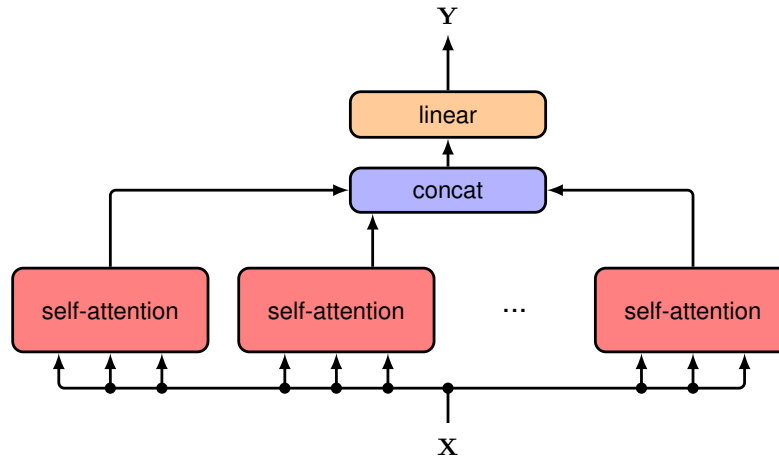

Figure 12.8 Information flow in a multi-head attention layer. The associated computation, given by Algorithm 12.2, is illustrated in Figure 12.7.

Section 9.5

Section 7.4.3

efficiency, we can introduce residual connections that bypass the multi-head structure. To do this we require that the output dimensionality is the same as the input dimensionality, namely $N \times D$. This is then followed by *layer normalization* (Ba, Kiros, and Hinton, 2016), which improves training efficiency. The resulting transformation can be written as

$$\mathbf{Z} = \text{LayerNorm}\left[\mathbf{Y}(\mathbf{X}) + \mathbf{X}\right] \tag{12.20}$$

where $\mathbf{Y}$ is defined by (12.19). Sometimes the layer normalization is replaced by *pre-norm* in which the normalization layer is applied before the multi-head self-attention instead of after, as this can result in more effective optimization, in which case we have

$$\mathbf{Z} = \mathbf{Y}(\mathbf{X}') + \mathbf{X}, \quad \text{where} \quad \mathbf{X}' = \text{LayerNorm}\left[\mathbf{X}\right]. \tag{12.21}$$

In each case, $\mathbf{Z}$ again has the same dimensionality $N \times D$ as the input matrix $\mathbf{X}$.

We have seen that the attention mechanism creates linear combinations of the value vectors, which are then linearly combined to produce the output vectors. Also, the values are linear functions of the input vectors, and so we see that the outputs of an attention layer are constrained to be linear combinations of the inputs. Nonlinearity does enter through the attention weights, and so the outputs will depend nonlinearly on the inputs via the softmax function, but the output vectors are still constrained to lie in the subspace spanned by the input vectors and this limits the expressive capabilities of the attention layer. We can enhance the flexibility of the transformer by post-processing the output of each layer using a standard nonlinear neural network with D inputs and D outputs, denoted $\text{MLP}[\cdot]$ for 'multilayer perceptron'. For example, this might consist of a two-layer fully connected network with ReLU hidden units. This needs to be done in a way that preserves the ability

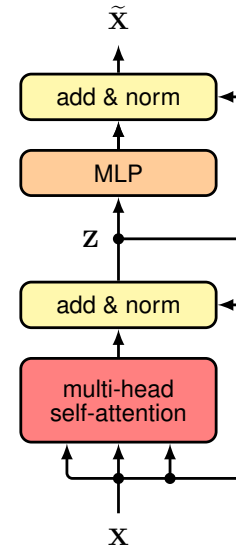

Figure 12.9 One layer of the transformer architecture that implements the transformation (12.1). Here 'MLP' stands for multilayer perceptron, while 'add and norm' denotes a residual connection followed by layer normalization.

of the transformer to process sequences of variable length. To achieve this, the same shared network is applied to each of the output vectors, corresponding to the rows of $\mathbf{Z}$. Again, this neural network layer can be improved by using a residual connection. It also includes layer normalization so that the final output from the transformer layer has the form

$$\widetilde{\mathbf{X}} = \text{LayerNorm}\left[\text{MLP}\left[\mathbf{Z}\right] + \mathbf{Z}\right]. \tag{12.22}$$

This leads to an overall architecture for a transformer layer shown in Figure 12.9 and summarized in Algorithm 12.3. Again, we can use a pre-norm instead, in which case the final output is given by

$$\widetilde{\mathbf{X}} = \text{MLP}(\mathbf{Z}') + \mathbf{Z}, \quad \text{where} \quad \mathbf{Z}' = \text{LayerNorm}\left[\mathbf{Z}\right]. \tag{12.23}$$

In a typical transformer there are multiple such layers stacked on top of each other. The layers generally have identical structures, although there is no sharing of weights and biases between different layers.

12.1.8 Computational complexity

The attention layer discussed so far takes a set of N vectors each of length D and maps them into another set of N vectors having the same dimensionality. Thus, the inputs and outputs each have overall dimensionality ND. If we had used a standard fully connected neural network to map the input values to the output values, it would have $\mathcal{O}(N^2D^2)$ independent parameters. Likewise the computational cost of evaluating one forward pass through such a network would also be $\mathcal{O}(N^2D^2)$.

In the attention layer, the matrices $\mathbf{W}^{(\text{q})}$, $\mathbf{W}^{(\text{k})}$, and $\mathbf{W}^{(\text{v})}$ are shared across input tokens, and therefore the number of independent parameters is $\mathcal{O}(D^2)$, assuming $D_{\text{k}} \simeq D_{\text{v}} \simeq D$. Since there are N input tokens, the number of computational steps

Algorithm 12.3: Transformer layer

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Multi-head self-attention layer parameters
Feed-forward network parameters
Output: $\widetilde{\mathbf{X}} \in \mathbb{R}^{N \times D} : \{\widetilde{\mathbf{x}}_1, \ldots, \widetilde{\mathbf{x}}_N\}$

$\mathbf{Z} = \text{LayerNorm}\left[\mathbf{Y}(\mathbf{X}) + \mathbf{X}\right]$ `// Y(X) from Algorithm 12.2`
$\widetilde{\mathbf{X}} = \text{LayerNorm}\left[\text{MLP}\left[\mathbf{Z}\right] + \mathbf{Z}\right]$ `// shared neural network`
return $\widetilde{\mathbf{X}}$

in evaluating the dot products in a self-attention layer is $\mathcal{O}(N^2D)$. We can think of a self-attention layer as a sparse matrix in which parameters are shared between specific blocks of the matrix. The subsequent neural network layer, which has D inputs and D outputs, has a cost that is $\mathcal{O}(D^2)$. Since it is shared across tokens, it has a complexity that is linear in N, and therefore overall this layer has a cost that is $\mathcal{O}(ND^2)$. Depending on the relative sizes of N and D, either the transformer layer or the MLP layer may dominate the computational cost. Compared to a fully connected network, a transformer layer is computationally more efficient. Many variants of the transformer architecture have been proposed (Lin *et al.*, 2021; Phuong and Hutter, 2022) including modifications aimed at improving efficiency (Tay *et al.*, 2020).

Exercise 12.6

12.1.9 Positional encoding

In the transformer architecture, the matrices $\mathbf{W}_h^{(\text{q})}$, $\mathbf{W}_h^{(\text{k})}$, and $\mathbf{W}_h^{(\text{v})}$ are shared across the input tokens, as is the subsequent neural network. As a consequence, the transformer has the property that permuting the order of the input tokens, i.e., the rows of $\mathbf{X}$, results in the same permutation of the rows of the output matrix $\widetilde{\mathbf{X}}$. In other words a transformer is *equivariant* with respect to input permutations. The sharing of parameters in the network architecture facilitates the massively parallel processing of the transformer, and also allows the network to learn long-range dependencies just as effectively as short-range dependencies. However, the lack of dependence on token order becomes a major limitation when we consider sequential data, such as the words in a natural language, because the representation learned by a transformer will be independent of the input token ordering. The two sentences '*The food was bad, not good at all.*' and '*The food was good, not bad at all.*' contain the same tokens but they have very different meanings because of the different token ordering. Clearly token order is crucial for most sequential processing tasks including natural language processing, and so we need to find a way to inject token order information into the network.

Exercise 12.7
Section 10.2

Since we wish to retain the powerful properties of the attention layers that we have carefully constructed, we aim to encode the token order in the data itself in-

stead of having to be represented in the network architecture. We will therefore construct a position encoding vector $\mathbf{r}_n$ associated with each input position n and then combine this with the associated input token embedding $\mathbf{x}_n$. One obvious way to combine these vectors would be to concatenate them, but this would increase the dimensionality of the input space and hence of all subsequent attention spaces, creating a significant increase in computational cost. Instead, we can simply add the position vectors onto the token vectors to give

$$\widetilde{\mathbf{x}}_n = \mathbf{x}_n + \mathbf{r}_n. \tag{12.24}$$

This requires that the positional encoding vectors have the same dimensionality as the token-embedding vectors.

At first it might seem that adding position information onto the token vector would corrupt the input vectors and make the task of the network much more difficult. However, some intuition as to why this can work well comes from noting that two randomly chosen uncorrelated vectors tend to be nearly orthogonal in spaces of high dimensionality, indicating that the network is able to process the token identity information and the position information relatively separately. Note also that, because of the residual connections across every layer, the position information does not get lost in going from one transformer layer to the next. Moreover, due to the linear processing layers in the transformer, a concatenated representation has similar properties to an additive one.

Exercise 12.8

Exercise 12.9

The next task is to construct the embedding vectors $\{\mathbf{r}_n\}$. A simple approach would be to associate an integer $1, 2, 3, \ldots$ with each position. However, this has the problem that the magnitude of the value increases without bound and therefore may start to corrupt the embedding vector significantly. Also it may not generalize well to new input sequences that are longer than those used in training, since these will involve coding values that lie outside the range of those used in training. Alternatively we could assign a number in the range $(0, 1)$ to each token in the sequence, which keeps the representation bounded. However, this representation is not unique for a given position as it depends on the overall sequence length.

An ideal positional encoding should provide a unique representation for each position, it should be bounded, it should generalize to longer sequences, and it should have a consistent way to express the number of steps between any two input vectors irrespective of their absolute position because the relative position of tokens is often more important than the absolute position.

There are many approaches to positional encoding (Dufter, Schmitt, and Schütze, 2021). Here we describe a technique based on sinusoidal functions introduced by Vaswani *et al.* (2017). For a given position n the associated position-encoding vector has components r_{ni} given by

$$r_{ni} = \begin{cases} \sin\left(\dfrac{n}{L^{i/D}}\right), & \text{if } i \text{ is even,} \\ \cos\left(\dfrac{n}{L^{(i-1)/D}}\right), & \text{if } i \text{ is odd.} \end{cases} \tag{12.25}$$

We see that the elements of the embedding vector $\mathbf{r}_n$ are given by a series of sine and cosine functions of steadily increasing wavelength, as illustrated in Figure 12.10(a).

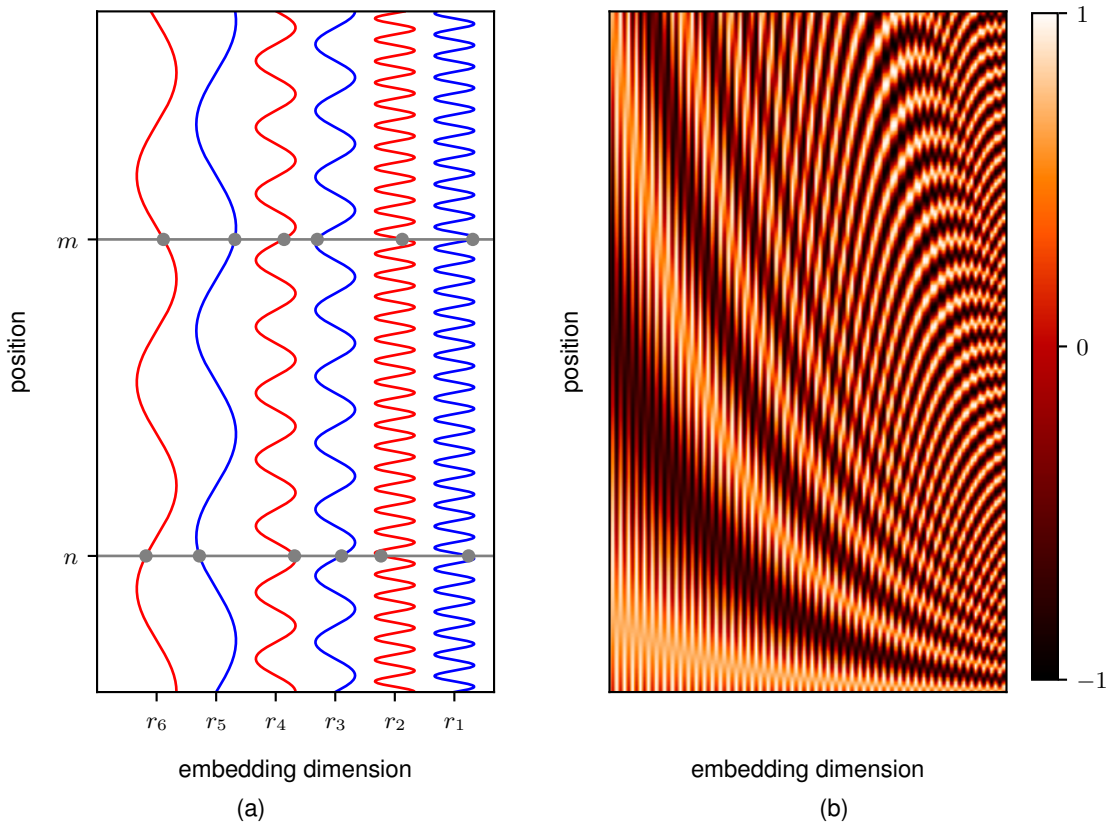

Figure 12.10 Illustrations of the functions defined by (12.25) and used to construct position-encoding vectors. (a) A plot in which the horizontal axis shows the different components of the embedding vector $\mathbf{r}$ whereas the vertical axis shows the position in the sequence. The values of the vector elements for two positions n and m are shown by the intersections of the sine and cosine curves with the horizontal grey lines. (b) A heat map illustration of the position-encoding vectors defined by (12.25) for dimension $D = 100$ with $L = 30$ for the first $N = 200$ positions.

This encoding has the property that the elements of the vector $\mathbf{r}_n$ all lie in the range $(-1, 1)$. It is reminiscent of the way binary numbers are represented, with the lowest order bit alternating with high frequency, and subsequent bits alternating with steadily decreasing frequencies:

$$\begin{array}{ll} \mathbf{1}: & 0\ 0\ 0\ 1 \\ \mathbf{2}: & 0\ 0\ 1\ 0 \\ \mathbf{3}: & 0\ 0\ 1\ 1 \\ \mathbf{4}: & 0\ 1\ 0\ 0 \\ \mathbf{5}: & 0\ 1\ 0\ 1 \\ \mathbf{6}: & 0\ 1\ 1\ 0 \\ \mathbf{7}: & 0\ 1\ 1\ 1 \\ \mathbf{8}: & 1\ 0\ 0\ 0 \\ \mathbf{9}: & 1\ 0\ 0\ 1 \end{array}$$

For the encoding given by (12.25), however, the vector elements are continuous variables rather than binary. A plot of the position-encoding vectors is shown in Figure 12.10(b).

One nice property of the sinusoidal representation given by (12.25) is that, for any fixed offset k, the encoding at position $n + k$ can be represented as a linear combination of the encoding at position n, in which the coefficients do not depend on the absolute position but only on the value of k. The network should therefore be able to learn to attend to relative positions. Note that this property requires that the encoding makes use of both sine and cosine functions.

Exercise 12.10

Another popular approach to positional representation is to use learned position encodings. This is done by having a vector of weights at each token position that can be learned jointly with the rest of the model parameters during training, and avoids using hand-crafted representations. Because the parameters are not shared between the token positions, the tokens are no longer invariant under a permutation, which is the purpose of a positional encoding. However, this approach does not meet the criteria we mentioned earlier of generalizing to longer input sequences, as the encoding will be untrained for positional encodings not seen during training. Therefore, this approach is generally most suitable when the input length is relatively constant during both training and inference.

12.2. Natural Language

Now that we have studied the architecture of the transformer, we will explore how this can be used to process language data consisting of words, sentences, and paragraphs. Although this is the modality that transformers were originally developed to operate on, they have proved to be a very general class of models and have become the state-of-the-art for most input data types. Later in this chapter we will look at their use in other domains.

Section 12.4

Many languages, including English, comprise a series of words separated by white space, along with punctuation symbols, and therefore represent an example of

Section 11.3
Section 12.2.2

sequential data. For the moment we will focus on the words, and we will return to punctuation later.

The first challenge is to convert the words into a numerical representation that is suitable for use as the input to a deep neural network. One simple approach is to define a fixed dictionary of words and then introduce vectors of length equal to the size of the dictionary along with a 'one hot' representation for each word, in which the kth word in the dictionary is encoded with a vector having a 1 in position k and 0 in all other positions. For example if 'aardwolf' is the third word in our dictionary then its vector representation would be $(0, 0, 1, 0, \ldots, 0)$.

An obvious problem with a one-hot representation is that a realistic dictionary might have several hundred thousand entries leading to vectors of very high dimensionality. Also, it does not capture any similarities or relationships that might exist between words. Both issues can be addressed by mapping the words into a lower-dimensional space through a process called *word embedding* in which each word is represented as a dense vector in a space of typically a few hundred dimensions.

12.2.1 Word embedding

The embedding process can be defined by a matrix $\mathbf{E}$ of size $D \times K$ where D is the dimensionality of the embedding space and K is the dimensionality of the dictionary. For each one-hot encoded input vector $\mathbf{x}_n$ we can then calculate the corresponding embedding vector using

$$\mathbf{v}_n = \mathbf{E}\mathbf{x}_n. \tag{12.26}$$

Because $\mathbf{x}_n$ has a one-hot encoding, the vector $\mathbf{v}_n$ is simply given by the corresponding column of the matrix $\mathbf{E}$.

We can learn the matrix $\mathbf{E}$ from a corpus (i.e., a large data set) of text, and there are many approaches to doing this. Here we look at a popular technique called *word2vec* (Mikolov *et al.*, 2013), which can be viewed as a simple two-layer neural network. A training set is constructed in which each sample is obtained by considering a 'window' of M adjacent words in the text, where a typical value might be $M = 5$. The samples are considered to be independent, and the error function is defined as the sum of the error functions for each sample. There are two variants of this approach. In *continuous bag of words*, the target variable for network training is the middle word, and the remaining *context* words form the inputs, so that the network is being trained to 'fill in the blank'. A closely related approach, called *skip-grams*, reverses the inputs and outputs, so that the centre word is presented as the input and the target values are the context words. These models are illustrated in Figure 12.11.

This training procedure can be viewed as a form of *self-supervised* learning since the data consists simply of a large corpus of unlabelled text from which many small windows of word sequences are drawn at random. Labels are obtained from the text itself by 'masking' out those words whose values the network is trying to predict.

Once the model is trained, the embedding matrix $\mathbf{E}$ is given by the transpose of the second-layer weight matrix for the continuous bag-of-words approach and by the first-layer weight matrix for skip-grams. Words that are semantically related are mapped to nearby positions in the embedding space. This is to be expected

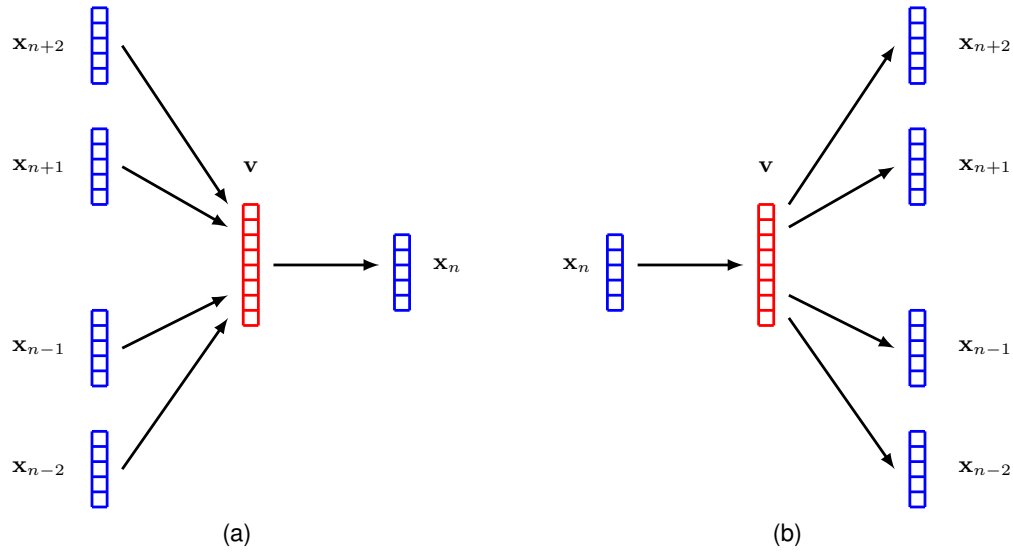

Figure 12.11 Two-layer neural networks used to learn word embeddings, where (a) shows the continuous bag-of-words approach, and (b) shows the skip-grams approach.

since related words are more likely to occur with similar context words compared to unrelated words. For example, the words 'city' and 'capital' might occur with higher frequency as context for target words such as 'Paris' or 'London' and less frequently as context for 'orange' or 'polynomial'. The network can more easily predict the probability of the missing words if 'Paris' and 'London' are mapped to nearby embedding vectors.

It turns out that the learned embedding space often has an even richer semantic structure than just the proximity of related words, and that this allows for simple vector arithmetic. For example, the concept that 'Paris is to France as Rome is to Italy' can be expressed through operations on the embedding vectors. If we use $\mathbf{v}(\text{word})$ to denote the embedding vector for 'word', then we find

$$\mathbf{v}(\text{Paris}) - \mathbf{v}(\text{France}) + \mathbf{v}(\text{Italy}) \simeq \mathbf{v}(\text{Rome}). \tag{12.27}$$

Word embeddings were originally developed as natural language processing tools in their own right. Today, they are more likely to be used as pre-processing steps for deep neural networks. In this regard they can be viewed as the first layer in a deep neural network. They can be fixed using some standard pre-trained embedding matrix, or they can be treated as an adaptive layer that is learned as part of the overall end-to-end training of the system. In the latter case the embedding layer can be initialized either using random weight values or using a standard embedding matrix.

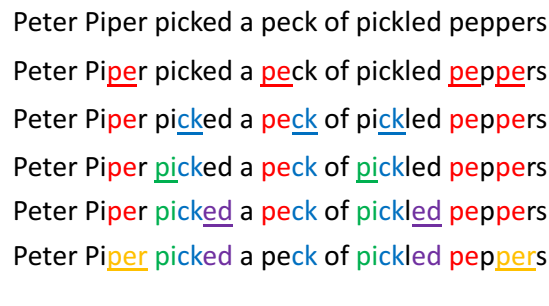

Figure 12.12 An illustration of the process of tokenizing natural language by analogy with byte pair encoding. In this example, the most frequently occurring pair of characters is 'pe', which occurs four times, and so these form a new token that replaces all the occurrences of 'pe'. Note that 'Pe' is not included in this since upper-case 'P' and lower-case 'p' are distinct characters. Next the pair 'ck' is added since this occurs three times. This is followed by tokens such as 'pi', 'ed', and 'per', all of which occur twice, and so on.

12.2.2 Tokenization

One problem with using a fixed dictionary of words is that it cannot cope with words not in the dictionary or which are misspelled. It also does not take account of punctuation symbols or other character sequences such as computer code. An alternative approach that addresses these problems would be to work at the level of characters instead of using words, so that our dictionary comprises upper-case and lower-case letters, numbers, punctuation, and white-space symbols such as spaces and tabs. A disadvantage of this approach, however, is that it discards the semantically important word structure of language, and the subsequent neural network would have to learn to reassemble words from elementary characters. It would also require a much larger number of sequential steps for a given body of text, thereby increasing the computational cost of processing the sequence.

We can combine the benefits of character-level and word-level representations by using a pre-processing step that converts a string of words and punctuation symbols into a string of *tokens*, which are generally small groups of characters and might include common words in their entirety, along with fragments of longer words as well as individual characters that can be assembled into less common words (Schuster and Nakajima, 2012). This tokenization also allows the system to process other kinds of sequences such as computer code or even other modalities such as images. It also means that variations of the same word can have related representations. For example, 'cook', 'cooks', 'cooked', 'cooking', and 'cooker' are all related and share the common element 'cook', which itself could be represented as one of the tokens.

Section 12.4.1

There are many approaches to tokenization. As an example, a technique called *byte pair encoding* that is used for data compression, can be adapted to text tokenization by merging characters instead of bytes (Sennrich, Haddow, and Birch, 2015). The process starts with the individual characters and iteratively merges them into longer strings. The list of tokens is first initialized with the list of individual characters. Then a body of text is searched for the most frequently occurring adjacent pairs of tokens and these are replaced with a new token. To ensure that words are not merged, a new token is not formed from two tokens if the second token starts with a white space. The process is repeated iteratively as illustrated in Figure 12.12.

Initially the number of tokens is equal to the number of characters, which is relatively small. As tokens are formed, the total number of tokens increases, and

if this is continued long enough, the tokens will eventually correspond to the set of words in the text. The total number of tokens is generally fixed in advance, as a compromise between character-level and word-level representations. The algorithm is stopped when this number of tokens is reached.

In practical applications of deep learning to natural language, the input text is typically first mapped into a tokenized representation. However, for the remainder of this chapter, we will use word-level representations as this makes it easier to illustrate and motivate key concepts.

12.2.3 Bag of words

We now turn to the task of modelling the joint distribution $p(\mathbf{x}_1, \ldots, \mathbf{x}_N)$ of an ordered sequence of vectors, such as words (or tokens) in a natural language. The simplest approach is to assume that the words are drawn independently from the same distribution and hence that the joint distribution is fully factorized in the form

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n). \tag{12.28}$$

This can be expressed as a probabilistic graphical model in which the nodes are isolated with no interconnecting links.

Figure 11.28

The distribution $p(\mathbf{x})$ is shared across the variables and can be represented, without loss of generality, as a simple table listing the probabilities of each of the possible states of $\mathbf{x}$ (corresponding to the dictionary of words or tokens). The maximum likelihood solution for this model is obtained simply by setting each of these probabilities to the fraction of times that the word occurs in the training set. This is known as a *bag-of-words* model because it completely ignores the ordering of the words.

Exercise 12.11

We can use the bag-of-words approach to construct a simple text classifier. This could be used for example in sentiment analysis in which a passage of text representing a restaurant review is to be classified as positive or negative. The *naive Bayes* classifier assumes that the words are independent within each class $\mathcal{C}_k$, but with a different distribution for each class, so that

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N | \mathcal{C}_k) = \prod_{n=1}^{N} p(\mathbf{x}_n | \mathcal{C}_k). \tag{12.29}$$

Given prior class probabilities $p(\mathcal{C}_k)$, the posterior class probabilities for a new sequence are given by:

$$p(\mathcal{C}_k | \mathbf{x}_1, \ldots, \mathbf{x}_N) \propto p(\mathcal{C}_k) \prod_{n=1}^{N} p(\mathbf{x}_n | \mathcal{C}_k). \tag{12.30}$$

Both the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$ and the prior probabilities $p(\mathcal{C}_k)$ can be estimated using frequencies from the training data set. For a new sequence, the table entries are multiplied together to get the desired posterior probabilities. Note that if a word occurs in the test set that was not present in the training set then the

corresponding probability estimate will be zero, and so these estimates are typically 'smoothed' after training by reassigning a small level of probability uniformly across all entries to avoid zero values.

12.2.4 Autoregressive models

One obvious major limitation of the bag-of-words model is that it completely ignores word order. To address this we can take an autoregressive approach. Without loss of generality we can decompose the distribution over the sequence of words into a product of conditional distributions in the form

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n | \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}). \tag{12.31}$$

Figure 11.27

This can be represented as a probabilistic graphical model in which each node in the sequence receives a link from every previous node. We could represent each term on the right-hand side of (12.31) by a table whose entries are once again estimated using simple frequency counts from the training set. However, the size of these tables grows exponentially with the length of the sequence, and so this approach would become prohibitively expensive.

Exercise 12.12

We can simplify the model dramatically by assuming that each of the conditional distributions on the right-hand side of (12.31) is independent of all previous observations except the L most recent words. For example, if $L = 2$ then the joint distribution for a sequence of N observations under this model is given by

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1) \prod_{n=3}^{N} p(\mathbf{x}_n | \mathbf{x}_{n-1}, \mathbf{x}_{n-2}). \tag{12.32}$$

Figure 11.30

In the corresponding graphical model each node has links from the two previous nodes. Here we assume that the conditional distributions $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ are shared across all variables. Again each of the distributions on the right-hand side of (12.32) can be represented as tables whose values are estimated from the statistics of triplets of successive words drawn from a training corpus.

The case with $L = 1$ is known as a *bi-gram* model because it depends on pairs of adjacent words. Similarly $L = 2$, which involves triplets of adjacent words, is called a *tri-gram* model, and in general these are called *n-gram* models.

All the models discussed so far in this section can be run *generatively* to synthesize novel text. For example, if we provide the first and second words in a sequence, then we can sample from the tri-gram statistics $p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{x}_{n-2})$ to generate the third word, and then we can use the second and third words to sample the fourth word, and so on. The resulting text, however, will be incoherent because each word is predicted only on the basis of the two previous words. High-quality text models must take account of the long-range dependencies in language. On the other hand, we cannot simply increase the value of L because the size of the probability tables grows exponentially in L so that it is prohibitively expensive to go much beyond tri-gram models. However, the autoregressive representation will play a central role

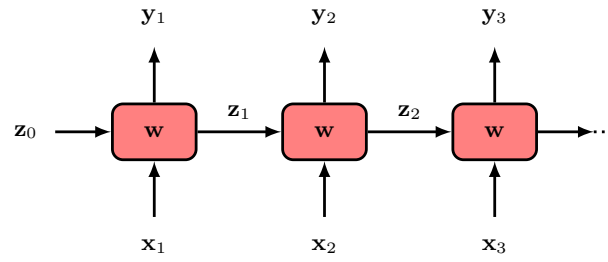

Figure 12.13 A general RNN with parameters $\mathbf{w}$. It takes a sequence $\mathbf{x}_1, \ldots, \mathbf{x}_N$ as input and generates a sequence $\mathbf{y}_1, \ldots, \mathbf{y}_N$ as output. Each of the boxes corresponds to a multilayer network with nonlinear hidden units.

when we consider modern language models based not on probability tables but on deep neural networks configured as transformers.

One way to allow longer-range dependencies, while avoiding the exponential growth in the number of parameters of an n-gram model, is to use a *hidden Markov model* whose graphical structure is shown in Figure 11.31. The number of learnable parameters is governed by the dimensionality of the latent variables whereas the distribution over a given observation $\mathbf{x}_n$ depends, in principle, on all previous observations. However, the influence of more distant observations is still very limited since their effect must be carried through the chain of latent states which are themselves being updated by more recent observations.

Section 11.3.1

12.2.5 Recurrent neural networks

Techniques such as n-grams have very poor scaling with sequence length because they store completely general tables of conditional distributions. We can achieve much better scaling by using parameterized models based on neural networks. Suppose we simply apply a standard feed-forward neural network to sequences of words in natural language. One problem that arises is that the network has a fixed number of inputs and outputs, whereas we need to be able to handle sequences in the training and test sets that have variable length. Furthermore, if a word, or group of words, at a particular location in a sequence represents some concept then the same word, or group of words, at a different location is likely to represent the same concept at that new location. This is reminiscent of the equivariance property we encountered in processing image data. If we can construct a network architecture that is able to share parameters across the sequence then not only can we capture this equivariance property but we can greatly reduce the number of free parameters in the model as well as handle sequences having different lengths.

Chapter 10

To address this we can borrow inspiration from the hidden Markov model and introduce an explicit hidden variable $\mathbf{z}_n$ associated with each step n in the sequence. The neural network takes as input both the current word $\mathbf{x}_n$ and the current hidden state $\mathbf{z}_{n-1}$ and produces an output word $\mathbf{y}_n$ as well as the next state $\mathbf{z}_n$ of the hidden variable. We can then chain together copies of this network, in which the weight values are shared across the copies. The resulting architecture is called a *recurrent neural network* (RNN) and is illustrated in Figure 12.13. Here the initial value of the hidden state may be initialized for example to some default value such as $\mathbf{z}_0 =$

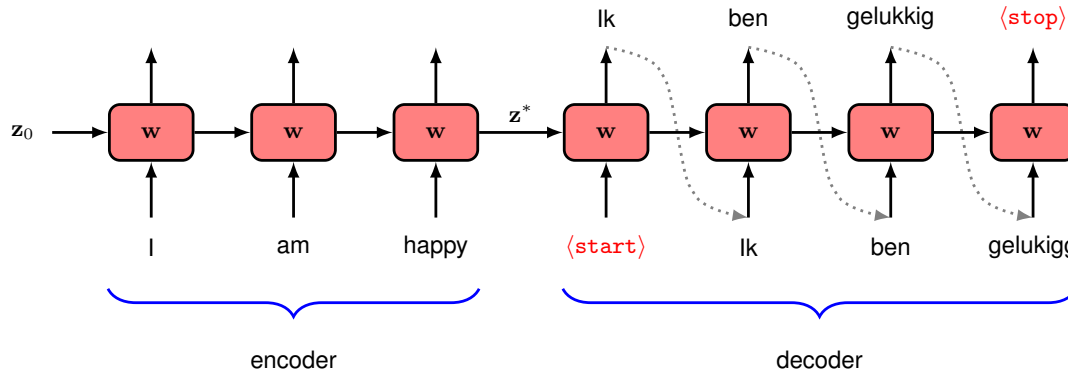

Figure 12.14 An example of a recurrent neural network used for language translation. See the text for details.

$(0, 0, \ldots, 0)^{\mathrm{T}}$.

As an example of how an RNN might be used in practice, consider the specific task of translating sentences from English into Dutch. The sentences can have variable length, and each output sentence might have a different length from the corresponding input sentence. Furthermore, the network may need to see the whole of the input sentence before it can even start to generate the output sentence. We can address this using an RNN by feeding in the complete English sentence followed by a special input token, which we denote by ⟨start⟩, to trigger the start of translation. During training the network learns to associate ⟨start⟩ with the beginning of the output sentence. We also take each successively generated word and feed it into the input at the next time step, as shown in Figure 12.14. The network can be trained to generate a specific ⟨stop⟩ token to signify the completion of the translation. The first few stages of the network are used to absorb the input sequence, and the associated output vectors are simply ignored. This part of the network can be viewed as an 'encoder' in which the entire input sentence has been compressed into the state $\mathbf{z}^\star$ of the hidden variable. The remaining network stages function as the 'decoder', which generates the translated sentence as output one word at a time. Notice that each output word is fed as input to the next stage of the network, and so this approach has an autoregressive structure analogous to (12.31).

12.2.6 Backpropagation through time

RNNs can be trained by stochastic gradient descent using gradients calculated by backpropagation and evaluated through automatic differentiation, just as with regular neural networks. The error function consists of a sum over all output units of the error for each unit, in which each output unit has a softmax activation function along with an associated cross-entropy error function. During forward propagation, the activation values are propagated all the way from the first input in the sequence through to all the output nodes in the sequence, and error signals are then backpropagated along the same paths. This process is called *backpropagation through time*

Section 5.4.4

and in principle is straightforward. However, in practice, for very long sequences, training can be difficult due to the problems of *vanishing gradients* or *exploding gradients* that arise with very deep network architectures.

Section 7.4.2

Another problem with standard RNNs is that they deal poorly with long-range dependencies. This is especially problematic for natural language where such dependencies are widespread. In a long passage of text, a concept might be introduced that plays an important role in predicting words occurring much later in the text. In the architecture shown in Figure 12.14, the entire concept of the English sentence must be captured in the single hidden vector $\mathbf{z}^\star$ of fixed length, and this becomes increasingly problematic with longer sequences. This is known as the *bottleneck problem* because a sequence of arbitrary length has to be summarized in a single hidden vector of activations and the network can start to generate the output translation only once the full input sequence has been processed.

One approach for addressing both the vanishing and exploding gradients problems and the limited long-range dependencies is to modify the architecture of the neural network to allow additional signal paths that bypass many of the processing steps within each stage of the network and hence allow information to be remembered over a larger number of time steps. *Long short-term memory* (LSTM) models (Hochreiter and Schmidhuber, 1997) and *gated recurrent unit* (GRU) models (Cho *et al.*, 2014) are the most widely known examples. Although they improve performance compared to standard RNNs, they still have a limited ability to model long-range dependencies. Also, the additional complexity of each cell means that LSTMs are even slower to train than standard RNNs. Furthermore, all recurrent networks have signal paths that grow linearly with the number of steps in the sequence. Moreover, they do not support parallel computation within a single training example due to the sequential nature of the processing. In particular, this means that RNNs struggle to make efficient use of modern highly parallel hardware based on GPUs. These problems are addressed by replacing RNNs with transformers.

12.3. Transformer Language Models

The transformer processing layer is a highly flexible component for building powerful neural network models with broad applicability. In this section we explore the application of transformers to natural language. This has given rise to the development of massive neural networks known as *large language models* (LLMs), which have proven to be exceptionally capable (Zhao *et al.*, 2023).

Transformers can be applied to many different kinds of language processing task, and can be grouped into three categories according to the form of the input and output data. In a problem such as sentiment analysis, we take a sequence of words as input and provide a single variable representing the sentiment of the text, for example happy or sad, as output. Here a transformer is acting as an 'encoder' of the sequence. Other problems might take a single vector as input and generate a word sequence as output, for example if we wish to generate a text caption given an input image. In such cases the transformer functions as a 'decoder', generating

a sequence as output. Finally, in sequence-to-sequence processing tasks, both the input and the output comprise a sequence of words, for example if our goal is to translate from one language to another. In this case, transformers are used in both encoder and decoder roles. We discuss each of these classes of language model in turn, using illustrative examples of model architectures.

12.3.1 Decoder transformers

We start by considering decoder-only transformer models. These can be used as *generative models* that create output sequences of tokens. As an illustrative example, we will focus on a class of models called *GPT* which stands for *generative pre-trained transformer* (Radford *et al.*, 2019; Brown *et al.*, 2020; OpenAI, 2023). The goal is to use the transformer architecture to construct an autoregressive model of the form defined by (12.31) in which the conditional distributions $p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$ are expressed using a transformer neural network that is learned from data.

The model takes as input a sequence consisting of the first $n-1$ tokens, and its corresponding output represents the conditional distribution for token n. If we draw a sample from this distribution then we have extended the sequence to n tokens and this new sequence can be fed back through the model to give a distribution over token $n+1$, and so on. The process can be repeated to generate sequences up to a maximum length determined by the number of inputs to the transformer. We will shortly discuss strategies for sampling from the conditional distributions, but for the moment we focus on how to construct and train the network.

Section 12.3.2

The architecture of a GPT model consists of a stack of transformer layers that take a sequence $\mathbf{x}_1, \ldots, \mathbf{x}_N$ of tokens, each of dimensionality D, as input and produce a sequence $\widetilde{\mathbf{x}}_1, \ldots, \widetilde{\mathbf{x}}_N$ of tokens, again of dimensionality D, as output. Each output needs to represent a probability distribution over the dictionary of tokens at that time step, and this dictionary has dimensionality K whereas the tokens have a dimensionality of D. We therefore make a linear transformation of each output token using a matrix $\mathbf{W}^{(\mathrm{p})}$ of dimensionality $D \times K$ followed by a softmax activation function in the form

$$\mathbf{Y} = \mathrm{Softmax}\left(\widetilde{\mathbf{X}}\mathbf{W}^{(\mathrm{p})}\right) \tag{12.33}$$

where $\mathbf{Y}$ is a matrix whose nth row is $\mathbf{y}_n^{\mathrm{T}}$, and $\widetilde{\mathbf{X}}$ is a matrix whose nth row is $\widetilde{\mathbf{x}}_n^{\mathrm{T}}$. Each softmax output unit has an associated cross-entropy error function. The architecture of the model is shown in Figure 12.15.

Section 5.4.4

The model can be trained using a large corpus of unlabelled natural language by taking a self-supervised approach. Each training sample consists of a sequence of tokens $\mathbf{x}_1, \ldots, \mathbf{x}_n$, which form the input to the network, along with an associated target value $\mathbf{x}_{n+1}$ consisting of the next token in the sequence. The sequences are considered to be independent and identically distributed so that the error function used for training is the sum of the cross-entropy error values summed over the training set, grouped into appropriate mini-batches. Naively we could process each such training sample independently using a forward pass through the model. However, we can achieve much greater efficiency by processing an entire sequence at once so that each token acts both as a target value for the sequence of previous tokens and as

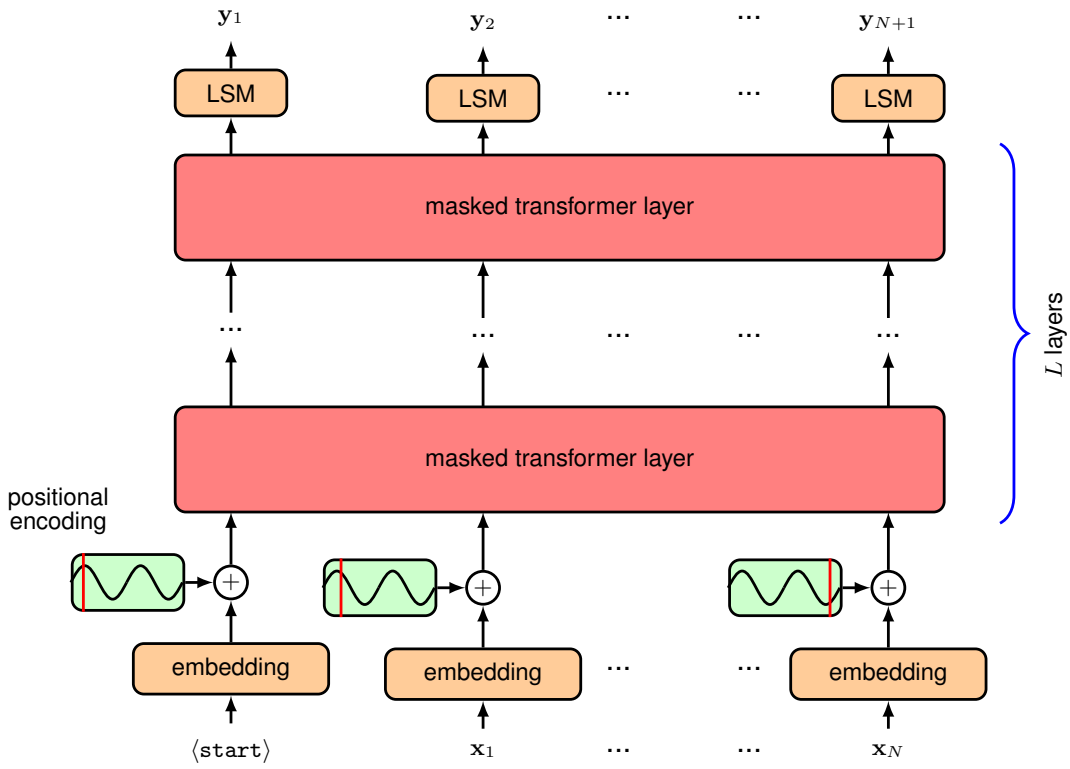

Figure 12.15 Architecture of a GPT decoder transformer network. Here 'LSM' stands for linear-softmax and denotes a linear transformation whose learnable parameters are shared across the token positions, followed by a softmax activation function. Masking is explained in the text.

an input value for subsequent tokens. For example, consider the word sequence

I swam across the river to get to the other bank.

We can use 'I swam across' as an input sequence with an associated target of 'the', and also use 'I swam across the' as an input sequence with an associated target of 'river', and so on. However, to process these in parallel we have to ensure that the network is not able to 'cheat' by looking ahead in the sequence, otherwise it will simply learn to copy the next input directly to the output. If it did this, it would then be unable to generate new sequences since the subsequent token by definition is not available at test time. To address this problem we do two things. First, we shift the input sequence to the right by one step, so that input $\mathbf{x}_n$ corresponds to output $\mathbf{y}_{n+1}$, with target $\mathbf{x}_{n+1}$, and an additional special token denoted $\langle \texttt{start} \rangle$ is prepended in the first position of the input sequence. Second, note that the tokens in a transformer are processed independently, except when they are used to compute the attention weights, when they interact in pairs through the dot product. We therefore introduce *masked attention*, sometimes called *causal attention*, into each of the at-

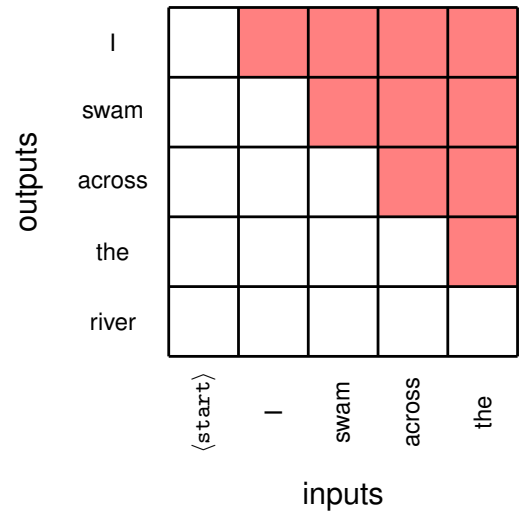

Figure 12.16 An illustration of the mask matrix for masked self-attention. Attention weights corresponding to the red elements are set to zero. Thus, in predicting the token 'across', the output can depend only on the input tokens '⟨start⟩' 'I' and 'swam'.

tention layers, in which we set to zero all of the attention weights that correspond to a token attending to any later token in the sequence. This simply involves setting to zero all the corresponding elements of the attention matrix $\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ defined by (12.14) and then normalizing the remaining elements so that each row once again sums to one. In practice, this can be achieved by setting the corresponding pre-activation values to $-\infty$ so that the softmax evaluates to zero for the associated outputs and also takes care of the normalization across the non-zero outputs. The structure of the masked attention matrix is illustrated in Figure 12.16.

In practice, we wish to make efficient use of the massive parallelism of GPUs, and hence multiple sequences may be stacked together into an input tensor for parallel processing in a single batch. However, this requires the sequences to be of the same length, whereas text sequences naturally have variable length. This can be addressed by introducing a specific token, which we denote by ⟨pad⟩, that is used to fill unused positions to bring all sequences up to the same length so that they can be combined into a single tensor. An additional mask is then used in the attention weights to ensure that the output vectors do not pay attention to any inputs occupied by the ⟨pad⟩ token. Note that the form of this mask depends on the particular input sequence.

The output of the trained model is a probability distribution over the space of tokens, given by the softmax output activation function, which represents the probability of the next token given the current token sequence. Once this next word is chosen, the token sequence with the new token included can then be fed through the model again to generate the subsequent token in the sequence, and this process can be repeated indefinitely or until an end-of-sequence token is generated. This may appear to be quite inefficient since data must be fed through the whole model for each new generated token. However, note that due to the masked attention, the embedding learned for a particular token depends only on that token itself and on earlier tokens

and hence does not change when a new, later token is generated. Consequently, much of the computation can be recycled when processing a new token.

12.3.2 Sampling strategies

We have seen that the output of a decoder transformer is a probability distribution over values for the next token in the sequence, from which a particular value for that token must be chosen to extend the sequence. There are several options for selecting the value of the token based on the computed probabilities (Holtzman *et al.*, 2019). One obvious approach, called greedy search, is simply to select the token with the highest probability. This has the effect of making the model deterministic, in that a given input sequence always generates the same output sequence. Note that simply choosing the highest probability token at each stage is not the same as selecting the highest probability sequence of tokens. To find the most probable sequence, we would need to maximize the joint distribution over all tokens, which is given by

Exercise 12.15

$$p(\mathbf{y}_1, \ldots, \mathbf{y}_N) = \prod_{n=1}^{N} p(\mathbf{y}_n | \mathbf{y}_1, \ldots, \mathbf{y}_{n-1}). \tag{12.34}$$

If there are N steps in the sequence and the number of token values in the dictionary is K then the total number of sequences is $\mathcal{O}(K^N)$, which grows exponentially with the length of the sequence, and hence finding the single most probable sequence is infeasible. By comparison, greedy search has cost $\mathcal{O}(KN)$, which is linear in the sequence length.

One technique that has the potential to generate higher probability sequences than greedy search is called *beam search*. Instead of choosing the single most probable token value at each step, we maintain a set of B hypotheses, where B is called the *beam width*, each consisting of a sequence of token values up to step n. We then feed all these sequences through the network, and for each sequence we find the B most probable token values, thereby creating B^2 possible hypotheses for the extended sequence. This list is then pruned by selecting the most probable B hypotheses according to the total probability of the extended sequence. Thus, the beam search algorithm maintains B alternative sequences and keeps track of their probabilities, finally selecting the most probable sequence amongst those considered. Because the probability of a sequence is obtained by multiplying the probabilities at each step of the sequence and since these probability are always less than or equal to one, a long sequence will generally have a lower probability than a short one, biasing the results towards short sequences. For this reason the sequence probabilities are generally normalized by the corresponding lengths of the sequence before making comparisons. Beam search has cost $\mathcal{O}(BKN)$, which is again linear in the sequence length. However, the cost of generating a sequence is increased by a factor of B, and so for very large language models, where the cost of inference can become significant, this makes beam search much less attractive.

One problem with approaches such as greedy search and beam search is that they limit the diversity of potential outputs and can even cause the generation process to become stuck in a loop, where the same sub-sequence of words is repeated over and

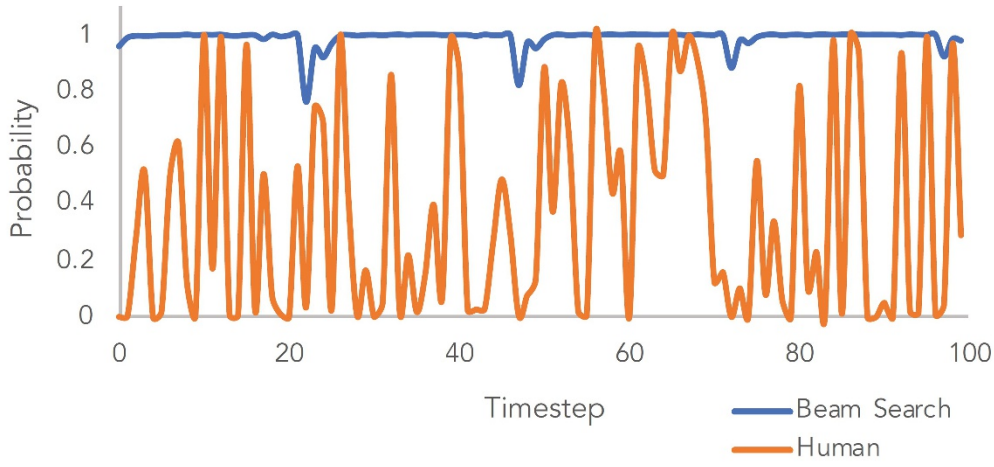

Figure 12.17 A comparison of the token probabilities from beam search and human text for a given trained transformer language model and a given initial input sequence, showing how the human sequence has much lower token probabilities. [From Holtzman *et al.* (2019) with permission.]

over. As can be seen in Figure 12.17, human-generated text may have lower probability and hence be more surprising with respect to a given model than automatically generated text.

Instead of trying to find a sequence with the highest probability, we can instead generate successive tokens simply by sampling from the softmax distribution at each step. However, this can lead to sequences that are nonsensical. This arises from the typically very large size of the token dictionary, in which there is a long tail of many token states each of which has a very small probability but which in aggregate account for a significant fraction of the total probability mass. This leads to the problem in which there is a significant chance that the system will make a bad choice for the next token.

As a balance between these extremes, we can consider only the states having the top K probabilities, for some choice of K, and then sample from these according to their renormalized probabilities. A variant of this approach, called top-p sampling or *nucleus sampling*, calculates the cumulative probability of the top outputs until a threshold is reached and then samples from this restricted set of token states.

A 'softer' version of top-K sampling is to introduce a parameter T called *temperature* into the definition of the softmax function (Hinton, Vinyals, and Dean, 2015) so that

$$y_i = \frac{\exp(a_i/T)}{\sum_j \exp(a_j/T)} \tag{12.35}$$

and then sample the next token from this modified distribution. When $T = 0$, the probability mass is concentrated on the most probable state, with all other states having zero probability, and hence this becomes greedy selection. For $T = 1$, we

recover the unmodified softmax distribution, and as $T \to \infty$, the distribution becomes uniform across all states. By choosing a value in the range $0 < T < 1$, the probability is concentrated towards the higher values.

One challenge with sequence generation is that during the learning phase, the model is trained on a human-generated input sequence, whereas when it is running generatively, the input sequence is itself generated from the model. This means that the model can drift away from the distribution of sequences seen during training.

12.3.3 Encoder transformers

We next consider transformer language models based on encoders, which are models that take sequences as input and produce fixed-length vectors, such as class labels, as output. An example of such a model is *BERT*, which stands for *bidirectional encoder representations from transformers* (Devlin *et al.*, 2018). The goal is to pre-train a language model using a large corpus of text and then to fine-tune the model using *transfer learning* for a broad range of downstream tasks each of which requires a smaller application-specific training data set. The architecture of an encoder transformer is illustrated in Figure 12.18. This approach is a straightforward application of the transformer layers discussed previously.

Section 12.1.7

The first token of every input string is given by a special token ⟨`class`⟩, and the corresponding output of the model is ignored during pre-training. Its role will become apparent when we discuss fine-tuning. The model is pre-trained by presenting token sequences at the input. A randomly chosen subset of the tokens, say 15%, are replaced with a special token denoted ⟨`mask`⟩. The model is trained to predict the missing tokens at the corresponding output nodes. This is analogous to the masking used in word2vec to learn word embeddings. For example, an input sequence might be

Section 12.2.1

I ⟨`mask`⟩ across the river to get to the ⟨`mask`⟩ bank.

and the network should predict ‘swam’ at output node 2 and ‘other’ at output node 10. In this case only two of the outputs contribute to the error function and the other outputs are ignored.

The term ‘bidirectional’ refers to the fact that the network sees words both before and after the masked word and can use both sources of information to make a prediction. As a consequence, unlike decoder models, there is no need to shift the inputs to the right by one place, and there is no need to mask the outputs of each layer from seeing input tokens occurring later in the sequence. Compared to the decoder model, an encoder is less efficient since only a fraction of the sequence tokens are used as training labels. Moreover, an encoder model is unable to generate sequences.

The procedure of replacing randomly selected tokens with ⟨`mask`⟩ means the training set has a mismatch compared to subsequent fine-tuning sets in that the latter will not contain any ⟨`mask`⟩ tokens. To mitigate any problems this might cause, Devlin *et al.* (2018) modified the procedure slightly, so that of the 15% of randomly selected tokens, 80% are replaced with ⟨`mask`⟩, 10% are replaced with a word selected at random from the vocabulary, and in 10% of the cases, the original words are retained at the input, but they still have to be correctly predicted at the output.

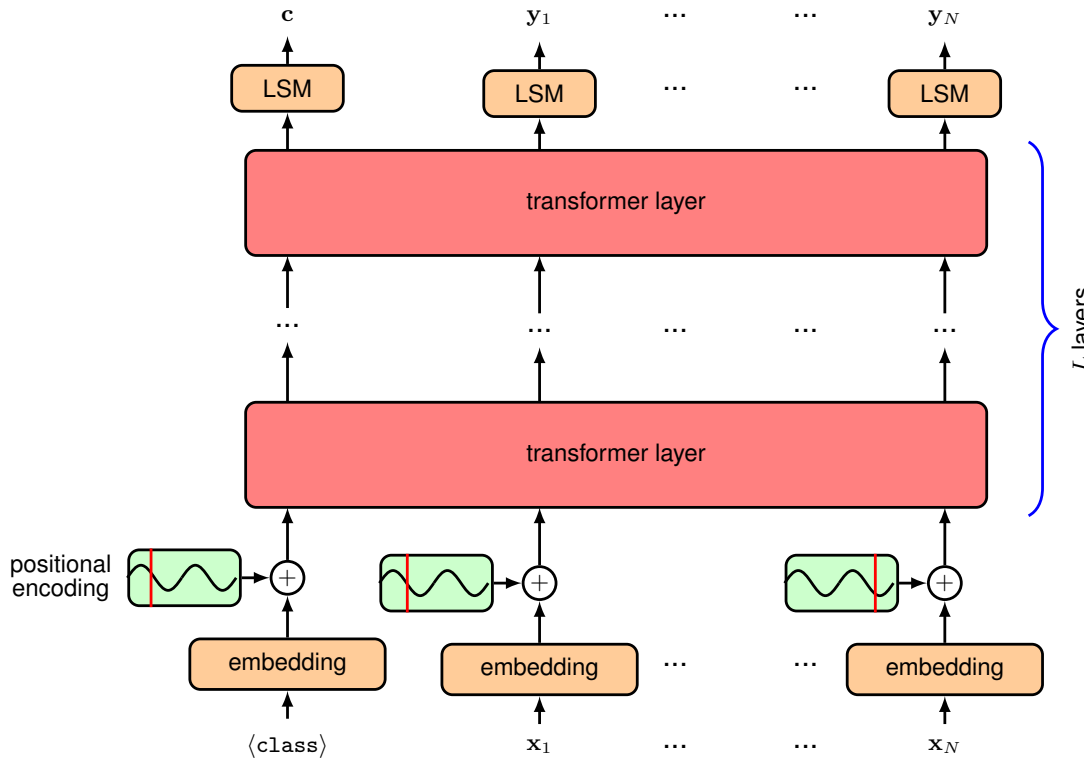

Figure 12.18 Architecture of an encoder transformer model. The boxes labelled 'LSM' denote a linear transformation whose learnable parameters are shared across the token positions, followed by a softmax activation function. The main differences compared to the decoder model are that the input sequence is not shifted to the right, and the 'look ahead' masking matrix is omitted and therefore, within each self-attention layer, every output token can attend to any of the input tokens.

Once the encoder model is trained it can then be fine-tuned for a variety of different tasks. To do this a new output layer is constructed whose form is specific to the task being solved. For a text classification task, only the first output position is used, which corresponds to the ⟨class⟩ token that always appears in the first position of the input sequence. If this output has dimension D then a matrix of parameters of dimension $D \times K$, where K is the number of classes, is appended to the first output node and this in turn feeds into a K-dimensional softmax function or a vector of dimension $D \times 1$ followed by a logistic sigmoid for $K = 2$. The linear output transformation could alternatively be replaced with a more complex differentiable model such as an MLP. If the goal is to classify each token of the input string, for example to assign each token to a category (such as person, place, colour, etc) then the first output is ignored and the subsequent outputs have a shared linear-plus-softmax layer. During fine-tuning all model parameters including the new output matrix are learned by stochastic gradient descent using the log probability

of the correct label. Alternatively the output of a pre-trained model might feed into a sophisticated generative deep learning model for applications such as text-to-image synthesis.

Chapter 20

12.3.4 Sequence-to-sequence transformers

For completeness, we discuss briefly the third category of transformer model, which combines an encoder with a decoder, as discussed in the original transformer paper of Vaswani *et al.* (2017). Consider the task of translating an English sentence into a Dutch sentence. We can use a decoder model to generate the token sequence corresponding to the Dutch output, token by token, as discussed previously. The main difference is that this output needs to be conditioned on the entire input sequence corresponding to the English sentence. An encoder transformer can be used to map the input token sequence into a suitable internal representation, which we denote by $\mathbf{Z}$. To incorporate $\mathbf{Z}$ into the generative process for the output sequence, we use a modified form of the attention mechanism called *cross attention*. This is the same as self-attention except that although the query vectors come from the sequence being generated, in this case the Dutch output sequence, the key and value vectors come from the sequence represented by $\mathbf{Z}$, as illustrated in Figure 12.19. Returning to our analogy with a video streaming service, this would be like the user sending their query vector to a different streaming company who then compares it with their own set of key vectors to find the best match and then returns the associated value vector in the form of a movie.

Section 12.3.1

When we combine the encoder and decoder modules, we obtain the architecture of the model shown in Figure 12.20. The model can be trained using paired input and output sentences.

12.3.5 Large language models

The most important recent development in the field of machine learning has been the creation of very large transformer-based neural networks for natural language processing, known as *large language models* or LLMs. Here 'large' refers to the number of weight and bias parameters in the network, which can number up to around one trillion (10^{12}) at the time of writing. Such models are expensive to train, and the motivation for building them comes from their extraordinary capabilities.

In addition to the availability of large data sets, the training of ever larger models has been facilitated by the advent of massively parallel training hardware based on GPUs (graphics processing units) and similar processors tightly coupled in large clusters equipped fast interconnect and lots of onboard memory. The transformer architecture has played a key role in the development of these models because it is able to make very efficient use of such hardware. Very often, increasing the size of the training data set, along with a commensurate increase in the number of model parameters, leads to improvements in performance that outpace architectural improvements or other ways to incorporate more domain knowledge (Sutton, 2019; Kaplan *et al.*, 2020). For example, the impressive increase in performance of the GPT series of models (Radford *et al.*, 2019; Brown *et al.*, 2020; OpenAI, 2023) through successive generations has come primarily from an increase in scale. These kinds

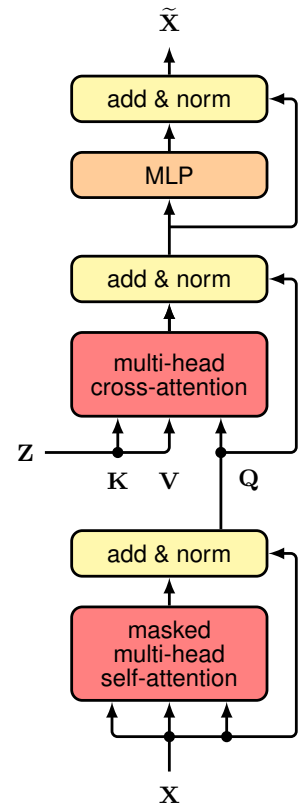

Figure 12.19 Schematic illustration of one cross-attention layer as used in the decoder section of a sequence-to-sequence transformer. Here $\mathbf{Z}$ denotes the output from the encoder section. $\mathbf{Z}$ determines the key and value vectors for the cross-attention layer, whereas the query vectors are determined within the decoder section.

of performance improvements have driven a new kind of Moore's law in which the number of compute operations required to train a state-of-the-art machine learning model has grown exponentially since about 2012 with a doubling time of around 3.4 months.

Figure 1.16

Early language models were trained using supervised learning. For example, to build a translation system, the training set would consist of matched pairs of sentences in two languages. A major limitation of supervised learning, however, is that the data typically has to be human-curated to provide labelled examples, and this severely limits the quantity of data available, thereby requiring heavy use of inductive biases such as feature engineering and architecture constraints to achieve reasonable performance.

Large language models are trained instead by self-supervised learning on very large data sets of text, along with potentially other token sequences such as computer code. We have seen how a decoder transformer can be trained on token sequences in which each token acts as a labelled target example, with the preceding sequence as input, to learn a conditional probability distribution. This 'self-labelling' hugely expands the quantity of training data available and therefore allows exploitation of deep neural networks having large numbers of parameters.

Section 12.3.1

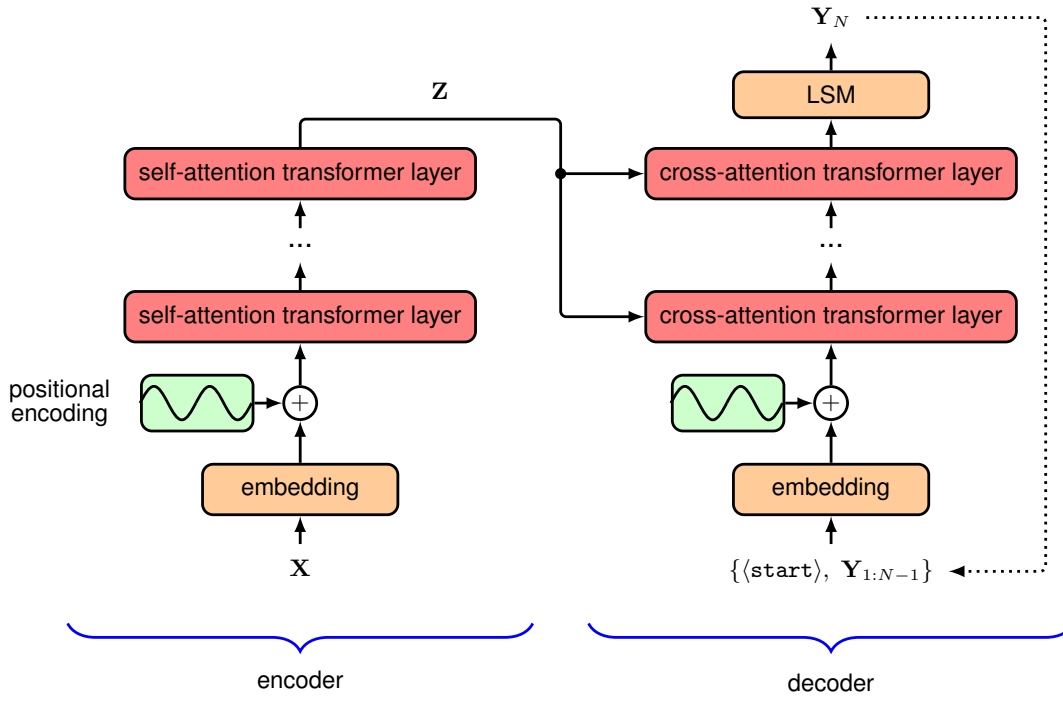

Figure 12.20 Schematic illustration of a sequence-to-sequence transformer. To keep the diagram uncluttered the input tokens are collectively shown as a single box, and likewise for the output tokens. Positional-encoding vectors are added to the input tokens for both the encoder and decoder sections. Each layer in the encoder corresponds to the structure shown in Figure 12.9, and each cross-attention layer is of the form shown in Figure 12.19.

This use of self-supervised learning led to a paradigm shift in which a large model is first *pre-trained* using unlabelled data and then subsequently *fine-tuned* using supervised learning based on a much smaller set of labelled data. This is effectively a form of transfer learning, and the same pre-trained model can be used for multiple 'downstream' applications. A model with broad capabilities that can be subsequently fine-tuned for specific tasks is called a *foundation model* (Bommasani *et al.*, 2021).

The fine-tuning can be done by adding extra layers to the outputs of the network or by replacing the last few layers with fresh parameters and then using the labelled data to train these final layers. During the fine-tuning stage, the weights and biases in the main model can either be left unchanged or be allowed to undergo small levels of adaptation. Typically the cost of the fine-tuning is small compared to that of pre-training.

One very efficient approach to fine-tuning is called *low-rank adaptation* or LoRA (Hu *et al.*, 2021). This approach is inspired by results which show that a trained over-parameterized model has a low intrinsic dimensionality with respect to fine-tuning, meaning that changes in the model parameters during fine-tuning lie on a manifold

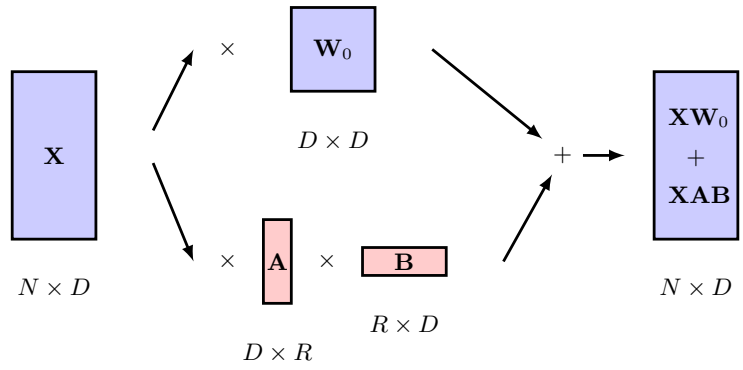

Figure 12.21 Schematic illustration low-rank adaptation showing a weight matrix $\mathbf{W}_0$ from one of the attention layers in a pre-trained transformer. Additional weights given by matrices $\mathbf{A}$ and $\mathbf{B}$ are adapted during fine-tuning and their product $\mathbf{AB}$ is then added to the original matrix for subsequent inference.

whose dimensionality is much smaller than the total number of learnable parameters in the model (Aghajanyan, Zettlemoyer, and Gupta, 2020). LoRa exploits this by freezing the weights of the original model and adding additional learnable weight matrices into each layer of the transformer in the form of low-rank products. Typically only attention-layer weights are modified, whereas MLP-layer weights are kept fixed. Consider a weight matrix $\mathbf{W}_0$ having dimension $D \times D$, which might represent a query, key, or value matrix in which the matrices from multiple attention heads are treated together as a single matrix. We introduce a parallel set of weights defined by the product of two matrices $\mathbf{A}$ and $\mathbf{B}$ with dimensions $D \times R$ and $R \times D$, respectively, as shown schematically in Figure 12.21. This layer then generates an output given by $\mathbf{XW}_0 + \mathbf{XAB}$. The number of parameters in the additional weight matrix $\mathbf{AB}$ is $2RD$ compared to the D^2 parameters in the original weight matrix $\mathbf{W}_0$, and so if $R \ll D$ then the number of parameters that need to be adapted during fine-tuning is much smaller than the number in the original transformer. In practice, this can reduce the number of parameters that need to be trained by a factor of 10,000. Once the fine-tuning is complete, the additional weights can be added to the original weight matrices to give a new weight matrix

$$\widehat{\mathbf{W}} = \mathbf{W}_0 + \mathbf{AB} \tag{12.36}$$

so that during inference there is no additional computational overhead compared to running the original model since the updated model has the same size as the original.

As language models have become larger and more powerful, the need for fine-tuning has diminished, with generative language models now able to solve a broad range of tasks simply through text-based interaction. For example, if a text string

English: the cat sat on the mat. French:

is given as the input sequence, an autoregressive language model can continue to generate subsequent tokens until a ⟨stop⟩ token is generated, in which the newly gen-

erated tokens represent the French translation. Note that the model was not trained specifically to do translation but has learned to do so as a result of being trained on a vast corpus of data that includes multiple languages.

A user can interact with such models using a natural language dialogue, making them very accessible to broad audiences. To improve the user experience and the quality of the generated outputs, techniques have been developed for fine-tuning large language models through human evaluation of generated output, using methods such as *reinforcement learning through human feedback* or RLHF (Christiano *et al.*, 2017). Such techniques have helped to create large language models with impressively easy-to-use conversational interfaces, most notably the system from OpenAI called *ChatGPT*.

The sequence of input tokens given by the user is called a *prompt*. For example, it might consist of the opening words of a story, which the model is required to complete. Or it might comprise a question, and the model should provide the answer. By using different prompts, the same trained neural network may be capable of solving a broad range of tasks such as generating computer code from a simple text request or writing rhyming poetry on demand. The performance of the model now depends on the form of the prompt, leading to a new field called *prompt engineering* (Liu *et al.*, 2021), which aims to design a good form for a prompt that results in high-quality output for the downstream task. The behaviour of the model can also be modified by adapting the user's prompt before feeding it into the language model by pre-pending an additional token sequence called a *prefix prompt* to the user prompt to modify the form of the output. For example, the pre-prompt might consist of instructions, expressed in standard English, to tell the network not to include offensive language in its output.

This allows the model to solve new tasks simply by providing some examples within the prompt, without needing to adapt the parameters of the model. This is an example of *few-shot learning*.

Current state-of-the-art models such as GPT-4 have become so powerful that they are exhibiting remarkable properties which have been described as the first indications of artificial general intelligence (Bubeck *et al.*, 2023) and are driving a new wave of technological innovation. Moreover, the capabilities of these models continue to improve at an impressive pace.

12.4. Multimodal Transformers

Although transformers were initially developed as an alternative to recurrent networks for processing sequential language data, they have become prevalent in nearly all areas of deep learning. They have proved to be general-purpose models, as they make very few assumptions about the input data, in contrast, for example, to convolutional networks, which make strong assumptions about equivariances and locality. Due to their generality, transformers have become the state-of-the-art for many different modalities, including text, image, video, point cloud, and audio data, and have been used for both discriminative and generative applications within each of these

Chapter 10

domains. The core architecture of the transformer layer has remained relatively constant, both over time and across applications. Therefore, the key innovations that enabled the use of transformers in areas other than natural language have largely focused on the representation and encoding of the inputs and outputs.

One big advantage of a single architecture that is capable of processing many different kinds of data is that it makes *multimodal* computation relatively straightforward. In this context, multimodal refers to applications that combine two or more different types of of data, either in the inputs or outputs or both. For example, we may wish to generate an image from a text prompt or design a robot that can combine information from multiple sensors such as cameras, radar, and microphones. The important thing to note is that if we can tokenize the inputs and decode the output tokens, then it is likely that we can use a transformer.

12.4.1 Vision transformers

Transformers have been applied with great success to computer vision and have achieved state-of-the-art performance on many tasks. The most common choice for discriminative tasks is a standard transformer encoder, and this approach in the vision domain is known as a *vision transformer*, or ViT (Dosovitskiy *et al.*, 2020). When using a transformer, we need to decide how to convert an input image into tokens, and the simplest choice is to use each pixel as a token, following a linear projection. However, the memory required by a standard transformer implementation grows quadratically with the number of input tokens, and so this approach is generally infeasible. Instead, the most common approach to tokenization is to split the image into a set of patches of the same size. Suppose the images have dimension $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ where H and W are the height and width of the image in pixels and C is the number of channels (where typically $C = 3$ for R, G, and B colours). Each image is split into non-overlapping patches of size $P \times P$ (where $P = 16$ is a common choice) and then 'flattened' into a one-dimensional vector, which gives a representation $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 C)}$ where $N = HW/P^2$ is the total number of patches for one image. The ViT architecture is shown in Figure 12.22.

Another approach to tokenization is to feed the image through a small convolutional neural network (CNN). This can down-sample the image to give a manageable number of tokens each represented by one of the network outputs. For example a typical ResNet18 encoder architecture down-samples an image by a factor of 8 in both the height and width dimensions, giving 64 times fewer tokens than pixels.

Chapter 10

We also need a way to encode positional information in the tokens. It is possible to construct explicit positional embeddings that encode the two-dimensional positional information of the image patches, but in practice this does not generally improve performance, and so it is most common to just use learned positional embeddings. In contrast to the transformers used for natural language, vision transformers generally take a fixed number of tokens as input, which avoids the problem of learned positional encodings not generalizing to inputs of a different size.

A vision transformer has a very different architectural design compared to a CNN. Although strong inductive biases are baked into a CNN model, the only two-dimensional inductive bias in a vision transformer is due to the patches used to tok-

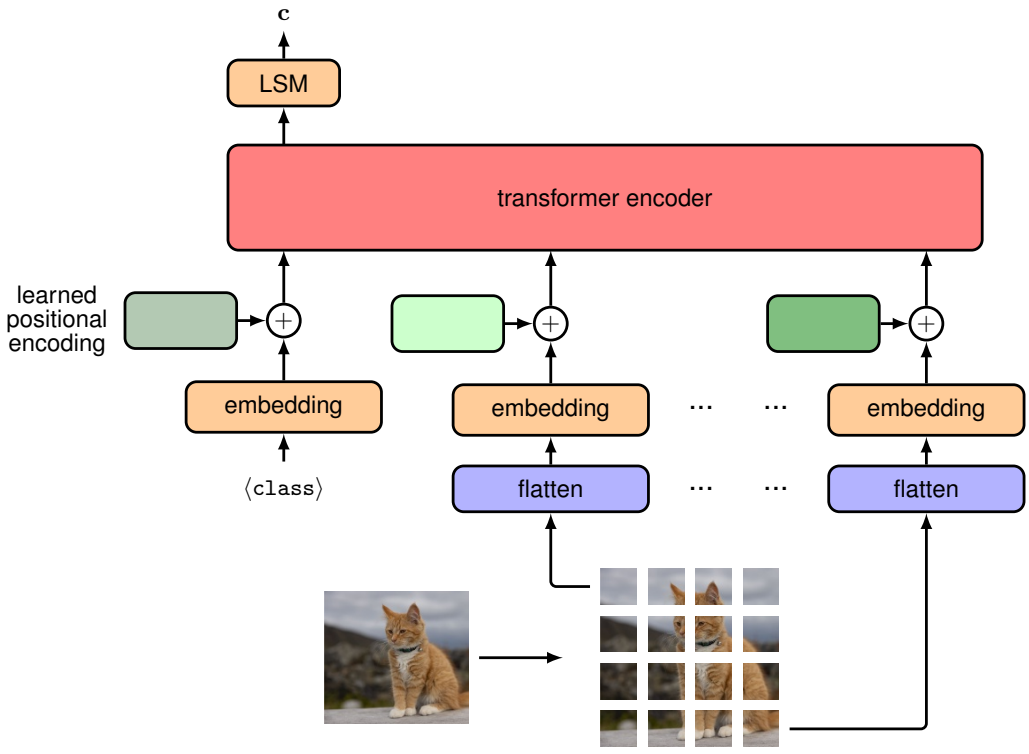

Figure 12.22 Illustration of the vision transformer architecture for a classification task. Here a learnable ⟨class⟩ token is included as an additional input, and the associated output is transformed by a linear layer with a softmax activation, denoted by LSM, to give the final class-vector output $\mathbf{c}$.

enize the input. A transformer therefore generally requires more training data than a comparable CNN as it has to learn the geometrical properties of images from scratch. However, because there are no strong assumptions about the structure of the inputs, transformers are often able to converge to a higher accuracy. This provides another illustration of the trade-off between inductive bias and the scale of the training data (Sutton, 2019).

12.4.2 Generative image transformers

In the language domain, the most impressive results have come when transformers are used as an autoregressive generative model for synthesizing text. It is therefore natural to ask whether we can also use transformers to synthesize realistic images. Since natural language is inherently sequential, it fits neatly into the autoregressive framework, whereas images have no natural ordering of their pixels so that it is not as intuitive that decoding them autoregressively would be useful. However, any distribution can be decomposed into a product of conditionals, provided we first define some ordering of the variables. Thus, the joint distribution over ordered

Section 11.1.2

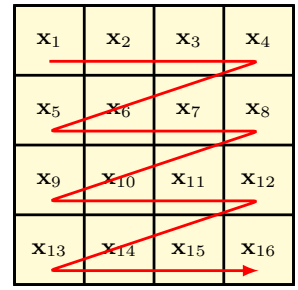

Figure 12.23 Illustration of a *raster scan* that defines a specific linear ordering of the pixels in a two-dimensional image.

variables $\mathbf{x}_1, \ldots, \mathbf{x}_N$ can be written

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n | \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}). \tag{12.37}$$

This factorization is completely general and makes no restrictions on the form of the individual conditional distributions $p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$.

For an image we can choose $\mathbf{x}_n$ to represent the nth pixel as a three-dimensional vector of the RGB values. We now need to decide on an ordering for the pixels, and one widely used choice is called a *raster scan* as illustrated in Figure 12.23. A schematic illustration of an image being generated using an autoregressive model, based on a raster-scan ordering, is shown in Figure 12.24.

Note that the use of autoregressive generative models of images predates the introduction of transformers. For example, PixelCNN (Oord *et al.*, 2016) and PixelRNN (Oord, Kalchbrenner, and Kavukcuoglu, 2016) used bespoke masked convolution layers that preserve the conditional independence defined for each pixel by the corresponding term on the right-hand side of 12.37.

Representations of an image using continuous values can work well in discriminative tasks. However, much better results are obtained for image generation by using discrete representations. Continuous conditional distributions learned by maximum likelihood, such as Gaussians for which the negative log likelihood function is a sum-of-squares error function, tend to learn averages of the training data, leading to blurry images. Conversely, discrete distributions can handle multimodality with ease. For example, one of the conditional distributions $p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$ in

Section 4.2

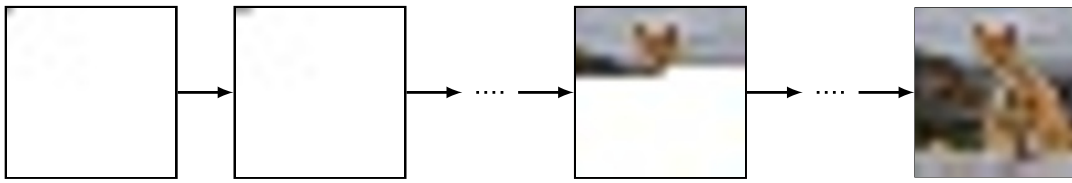

Figure 12.24 An illustration of how an image can be sampled from an autoregressive model. The first pixel is sampled from the marginal distribution $p(\mathbf{x}_{11})$, the second pixel from the conditional distribution $p(\mathbf{x}_{12}|\mathbf{x}_{11})$, and so on in raster scan order until we have a complete image.

(12.37) might learn that a pixel could be either black or white, whereas a regression model might learn that the pixel should be grey.

However, working with discrete spaces also brings its challenges. The R, G, and B values of image pixels are typically represented with at least 8 bits of precision, so that each pixel has $2^{24} \simeq 16\text{M}$ possible values. Learning a conditional softmax distribution over a such a high-dimensional space is infeasible.

One way to address the problem of the high dimensionality is to use the technique of *vector quantization*, which can be viewed as a form of data compression. Suppose we have a set of data vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ each of dimensionality D, which might, for example, represent image pixels, and we then introduce a set of K *codebook vectors* $\mathcal{C} = \mathbf{c}_1, \ldots, \mathbf{c}_K$ also of dimensionality D, where typically $K \ll D$. We now approximate each data vector by its nearest codebook vector according to some similarity metric, usually Euclidean distance, so that

Section 15.1.1

$$\mathbf{x}_n \rightarrow \arg\min_{\mathbf{c}_k \in \mathcal{C}} ||\mathbf{x}_n - \mathbf{c}_k||^2. \tag{12.38}$$

Since there are K codebook vectors, we can represent each $\mathbf{x}_n$ by a one-hot encoded K-dimensional vector, and since we can choose the value of K, we can control the trade-off between more accurate representation of the data, by using a larger value of K, or greater compression, by using a smaller value of K.

We can therefore take the original image pixels and map them into the lower-dimensional codebook space. An autoregressive transformer can then be trained to generate a sequence of codebook vectors, and this sequence can be mapped back into the original image space by replacing each codebook index k with the corresponding D-dimensional codebook vector $\mathbf{c}_k$.

Autoregressive transformers were first applied to images in ImageGPT (Chen, Radford, *et al.*, 2020). Here each pixel is treated as one of a discrete set of three-dimensional colour codebook vectors, each corresponding to a cluster in a K-means clustering of the colour space. A one-hot encoding therefore gives discrete tokens, analogous to language tokens, and allows the transformer to be trained in the same way as language models, with a next-token classification objective. This is a powerful objective for representation learning for subsequent fine-tuning, again in a similar way to language modelling.

Section 15.1

Section 6.3.3

Using the individual pixels as tokens directly, however, can lead to high computational cost since a forward pass is required per pixel, which means that both training and inference scale poorly with image resolution. Also, using individual pixels as inputs means that low-resolution images have to be used to give a reasonable context length when decoding the pixels later in the raster scan. As we saw with the ViT model, it is preferable to use patches of the image as tokens instead of pixels, as this can result in dramatically fewer tokens and therefore facilitates working with higher-resolution images. As before, we need to work with a discrete space of token values due to the potential multimodality of the conditional distributions. Again, this raises the challenge of dimensionality, which is now much more severe with patches than with individual pixels since the dimensionality is exponential with respect to the number of pixels in the patch. For example, even with just two possible pixel

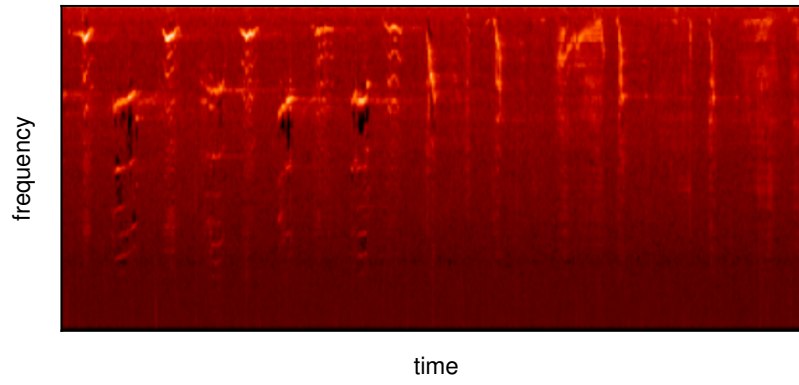

Figure 12.25 An example mel spectrogram of a humpback whale song. [Source data copyright ©2013–2023, librosa development team.]

tokens, representing black and white, and patches of size 16×16, we would have a dictionary of patch tokens of size $2^{256} \simeq 10^{77}$.

Once again we turn to vector quantization to address the challenge of dimensionality. The codebook vectors can be learned from a data set of image patches using simple clustering algorithms such as K-means or with more sophisticated methods such as fully convolutional networks (Oord, Vinyals, and Kavukcuoglu, 2017; Esser, Rombach, and Ommer, 2020) or even vision transformers (Yu *et al.*, 2021). One problem with learning to map each patch to a discrete set of codes and back again, is that vector quantization is a non-differentiable operation. Fortunately we can use a technique called straight-through gradient estimation (Bengio, Léonard, and Courville, 2013), which is a simple approximation that just copies the gradients through the non-differentiable function during backpropagation.

The use of autoregressive transformers to generate images can be extended to videos by treating a video as one long sequence of these vector-quantized tokens (Rakhimov *et al.*, 2020; Yan *et al.*, 2021; Hu *et al.*, 2023).

12.4.3 Audio data

We next look at the application of transformers to audio data. Sound is generally stored as a waveform obtained by measuring the amplitude of the air pressure at regular time intervals. Although this waveform could be used directly as input to a deep learning model, in practice it is more effective to pre-process it into a *mel spectrogram*. This is a matrix whose columns represent time steps and whose rows correspond to frequencies. The frequency bands follow a standard convention that was chosen through subjective assessment to give equal perceptual differences between successive frequencies (the word 'mel' comes from melody). An example of a mel spectrogram is shown in Figure 12.25.

One application for transformers in the audio domain is classification in which segments of audio are assigned to one of a number of predefined categories. For example, the *AudioSet* data set (Gemmeke *et al.*, 2017) is a widely used benchmark.

It contains classes such as 'car', 'animal', and 'laughter'. Until the development of the transformer, the state-of-the-art approach for audio classification was based on mel spectrograms treated as images and used as the input to a convolutional neural network (CNN). However, although a CNN is good at understanding local relationships, one drawback is that it struggles with longer-range dependencies, which can be important in processing audio.

Chapter 10

Just as transformers replaced RNNs as the state-of-the-art in natural language processing, they have also come to replace CNNs for tasks such as audio classification. For example, a transformer encoder model of identical structure to that used for both language and vision, as shown in Figure 12.18, can be used to predict the class of audio inputs (Gong, Chung, and Glass, 2021). Here the mel spectrogram is viewed as an image which is then tokenized. This is done by splitting the image into patches in a similar way to vision transformers, possibly with some overlap so as not to lose any important neighbourhood relations. Each patch is then flattened, meaning it is converted to a one-dimensional array, in this case of length 256. A unique positional encoding is then added to each token, a specific ⟨`class`⟩ token is appended, and the tokens are then fed through the transformer encoder. The output token corresponding to the ⟨`class`⟩ input token from the last transformer layer can then be decoded using a linear layer followed by a softmax activation function, and the whole model can be trained end-to-end using a cross-entropy loss.

12.4.4 Text-to-speech

Classification is not the only task that deep learning, and more specifically the transformer architecture, has revolutionized in the audio domain. The success of transformers at synthesizing speech that imitates the voice of a given speaker is another demonstration of their versatility, and their application to this task is an informative case study in how to apply transformers in a new context.

Generating speech corresponding to a given passage of text is known as *text-to-speech synthesis*. A more traditional approach would be to collect recordings of speech from a given speaker and train a supervised regression model to predict the speech output, possibly in the form of a mel spectrogram, from corresponding transcribed text. During inference, the text for which we would like to synthesize speech is presented as input and the resulting mel spectrogram output can then be decoded back to an audio waveform since this is a fixed mapping.

This approach has a few major drawbacks, however. First, if we predict speech at a low level, for example using sub-word components known as *phonemes*, a larger context is needed to make the resulting sentences sound fluid. However, if we predict longer segments, then the space of possible inputs grows significantly, and an infeasible amount of training data might be required to achieve good generalization. Second, this approach does not transfer knowledge across speakers, and so a lot of data will be required for each new speaker. Finally, the problem is really a generative modelling task, as there are multiple correct speech outputs for a given speaker and text pair, so regression may not be suitable since it tends to average over target values.

Section 4.2

If instead we treat audio data in the same way as natural language and frame text-to-speech as a conditional language modelling task, then we should be able to

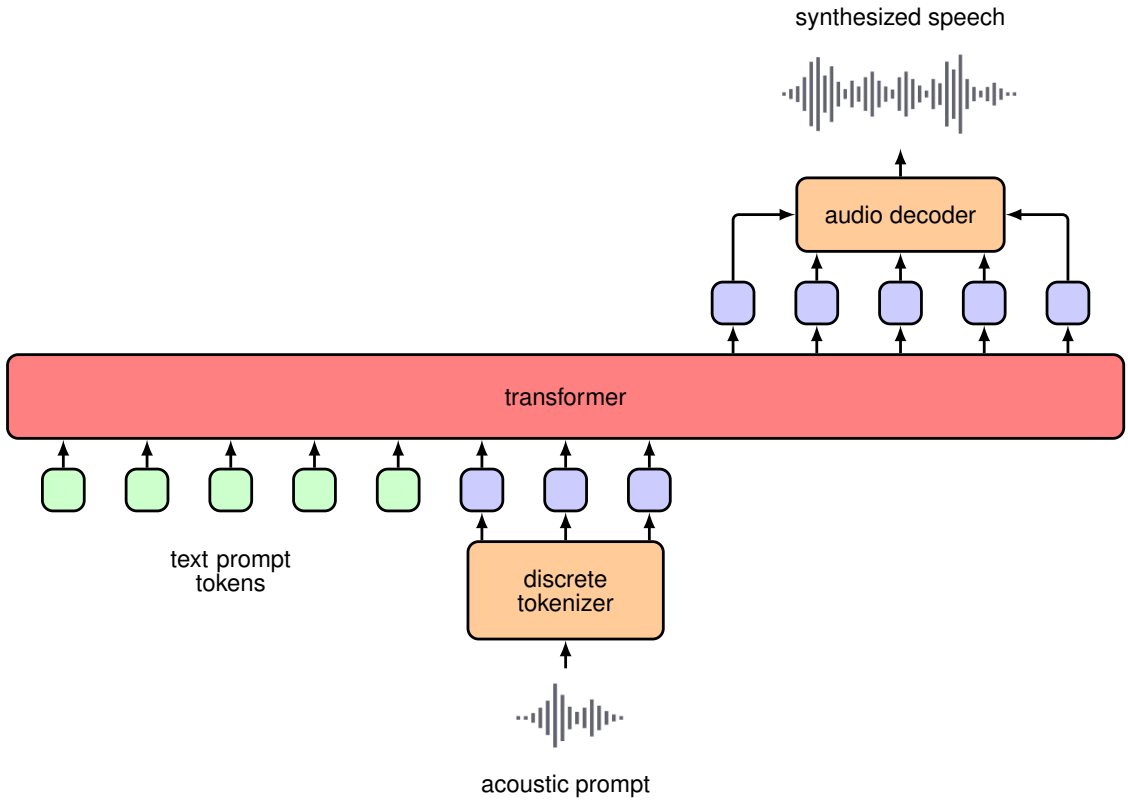

Figure 12.26 A diagram showing the high-level architecture of Vall-E. The input to the transformer model consists of standard text tokens, which prompt the model as to what words the synthesized speech should contain, together with acoustic prompt tokens that determine the speaker style and tone information. The sampled model output tokens are decoded back to speech with the learned decoder. For simplicity, the positional encodings and linear projections are not shown.

train the model in much the same way as with text-based large language models. There are two main implementation details that need to be addressed. The first is how to tokenize the training data and decode the predictions, and the second is how to condition the model on the speaker's voice.

One approach to text-to-speech synthesis that makes use of transformers and language modelling techniques is *Vall-E* (Wang *et al.*, 2023). New text can be mapped into speech in the voice of a new speaker using only a few seconds of sample speech from that person. Speech data is converted into a sequence of discrete tokens from a learned dictionary or *codebook* obtained using vector quantization, and we can think of these tokens as analogous to the one-hot encoded tokens from the natural language domain. The input consists of text tokens from a passage of text whereas the target outputs for training consist of the corresponding speech tokens. Additional speech tokens from a short segment of unrelated speech from the same speaker are

Section 12.4.2

appended to the input text tokens, as illustrated in Figure 12.26. By including examples from many different speakers, the system can learn to read out a passage of text while imitating the voice represented by the additional speech input tokens. Once trained the system can be presented with new text, along with audio tokens from a brief segment of speech captured from a new speaker, and the resulting output tokens can be decoded, using the same codebook used during training, to create a speech waveform. This allows the system to synthesize speech corresponding to the input text in the voice of the new speaker.

12.4.5 Vision and language transformers

We have seen how to generate discrete tokens for text, audio, and images, and so it is a natural next step to ask if we can train a model with input tokens of one modality and output tokens of another, or whether we can have a combination of different modalities for either inputs or outputs or both. We will focus on the combination of text and vision data as this is the most widely studied example, but in principle the approaches discussed here could be applied to other combinations of input and output modalities.

The first requirement is that we have a large data set for training. The LAION-400M data set (Schuhmann *et al.*, 2021) has greatly accelerated research in text-to-image generation and image-to-text captioning in much the same way that ImageNet was critical in the development of deep image classification models. Text-to-image generation is actually much like the unconditional image generation we have looked at so far, except that we also allow the model to take as input the text information to condition the generation process. This is straightforward when using transformers as we can simply provide the text tokens as additional input when decoding each image token.

This approach can also be viewed as treating the text-to-image problem as a sequence-to-sequence language modelling problem, such as machine translation, except that the target tokens are discrete image tokens rather than language tokens. It therefore makes sense to choose a full encoder-decoder transformer model, as shown in Figure 12.20, in which $\mathbf{X}$ corresponds to the input text tokens and $\mathbf{Y}$ corresponds to the output image tokens. This is the approach taken in a model called *Parti* (Yu *et al.*, 2022) in which the transformer is scaled to 20 billion parameters while showing consistent performance improvements with increasing model size.

A lot of research has also been done on using pre-trained language models, and modifying or fine-tuning them so that they can also accept visual data as input (Alayrac *et al.*, 2022; Li *et al.*, 2022). These approaches largely use bespoke architectures, along with continuous-valued image tokens, and therefore are not natural fits for also generating visual data. Moreover, they cannot be used directly if we wish to include new modalities such as audio tokens. Although this is a step towards multimodality, we would ideally like to use both text and image tokens as both input and output. The simplest approach is to treat everything as a sequence of tokens as if this were natural language but with a dictionary that is the concatenation of a language token dictionary and the image token codebook. We can then treat any stream of audio and visual data as simply a sequence of tokens.

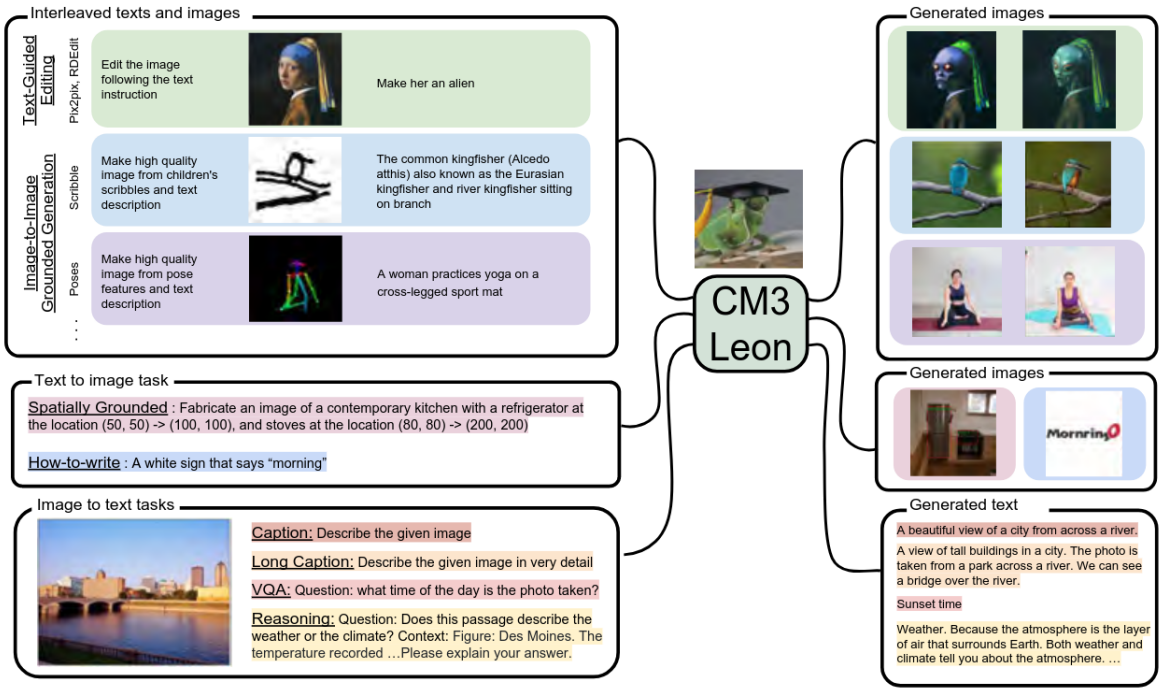

Figure 12.27 Examples of the CM3Leon model performing a variety of different tasks in the joint space of text and images. [From (Yu *et al.*, 2023) with permission.]

In CM3 (Aghajanyan *et al.*, 2022) and CM3Leon (Yu *et al.*, 2023), a variation of language modelling is used to train on HTML documents containing both image and text data taken from online sources. When this large quantity of training data was combined with a scalable architecture, the models became very powerful. Moreover, the multimodal nature of the training means that the models are very flexible. Such models are capable of completing many tasks that otherwise might require task-specific model architectures and training regimes, such as text-to-image generation, image-to-text captioning, image editing, text completion, and many more, including anything a regular language model is capable of. Examples of the CM3Leon model completing instances of a few different tasks are shown in Figure 12.27.

Exercises

12.1 ($\star\star$) Consider a set of coefficients a_{nm}, for $m = 1, \ldots, N$, with the properties that

$$a_{nm} \geqslant 0 \tag{12.39}$$

$$\sum_m a_{nm} = 1. \tag{12.40}$$