Chapter 12

Transformers

Chapter 10 introduced convolutional networks. These are specialized for processing data that lie on a regular grid. They are particularly suited to processing images, which have a very large number of input variables (precluding the use of fully connected networks) and behave similarly at every position (leading to the idea of parameter sharing).

This chapter introduces transformers. These were originally targeted at natural language processing (NLP) problems, where the network input is a series of high-dimensional embeddings that represent words or word fragments. Language datasets share some of the characteristics of image data. The number of input variables can be very large, and the statistics are similar at every position; it's not sensible to re-learn the meaning of the word dog at every possible position in a body of text. However, language datasets have the complication that input sequences are of variable length, and unlike images, there is no way to easily resize them.

12.1 Processing text data

To motivate the transformer, consider the following passage:

The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and service.

The goal is to design a network that can process this text into a representation that is suitable for downstream tasks. For example, it might be used to classify the review as positive or negative, or answer questions such as "Does the restaurant serve steak?".

We can make three immediate observations. First, the encoded input can be surprisingly large. In this case, each of the 37 words might be represented by an embedding vector of length 1024, so the input would be of length $37 \times 1024 = 37888$ even for this small passage. A more realistically sized input might have hundreds or even thousands of words, so fully connected neural networks are not practical.

Second, one of the defining characteristics of NLP problems is that each input (one

or more sentences) is of a different length; hence, it's not even obvious how to apply a fully connected network. These first two observations both suggest that the network will need to share parameters across the words at different positions in the input in a similar way to the way that a convolutional network shares parameters across different positions in an image.

Third, language is fundamentally ambiguous; it is not clear from the syntax alone that the pronoun it refers to the restaurant and not to the ham sandwich. To understand the text, the word it should somehow be connected to the word restaurant. In the parlance of transformers, the former word should pay *attention* to the latter. This implies that there must be connections between the words and that the strength of these connections will depend on the words themselves. Moreover, these connections need to extend across large spans of the text; the word their in the last sentence also refers to the restaurant.

12.2 Dot-product self-attention

The previous section argued that a model for processing text will (i) use parameter sharing to cope with long input passages of differing lengths, and (ii) contain connections between word representations that depend on the words themselves. The transformer acquires both properties by using *dot-product self-attention*.

A standard neural network layer $\mathbf{f}[\mathbf{x}]$, takes a $D \times 1$ input $\mathbf{x}$, and applies a linear transformation followed by an activation function $\mathbf{a}[\bullet]$:

$$\mathbf{f}[\mathbf{x}] = \mathbf{a}[\boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{x}], \tag{12.1}$$

where $\boldsymbol{\beta}$ contains the biases and $\boldsymbol{\Omega}$ contains the weights.

A self-attention block $\mathbf{sa}[\bullet]$ takes N inputs $\mathbf{x}_n$, each of dimension $D \times 1$, and returns N output vectors of the same size. In the context of NLP, each of the inputs $\mathbf{x}_n$ will represent a word or word fragment. First, a set of *values* are computed for each input:

$$\mathbf{v}_n = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \mathbf{x}_n, \tag{12.2}$$

where $\boldsymbol{\beta}_v$ and $\boldsymbol{\Omega}_v$ represent biases and weights respectively. Then the n^{th} output $\mathbf{sa}[\mathbf{x}_n]$ is a weighted sum of all the values $\mathbf{v}_n$:

$$\mathbf{sa}[\mathbf{x}_n] = \sum_{m=1}^{N} a[\mathbf{x}_m, \mathbf{x}_n]\mathbf{v}_m. \tag{12.3}$$

The scalar weight $a[\mathbf{x}_m, \mathbf{x}_n]$ is the *attention* that output $\mathbf{x}_n$ pays to input $\mathbf{x}_m$. The N weights $a[\bullet, \mathbf{x}_n]$ are non-negative and sum to one. Hence, self-attention can be thought of as *routing* the values in different proportions to create each output (figure 12.1).

The following sections examine dot-product self-attention in more detail by breaking it down into two parts. First, we consider the computation of the values and their subsequent weighting, as described in equation 12.3. Then we describe how to compute the attention weights $a[\mathbf{x}_m, \mathbf{x}_n]$ themselves.

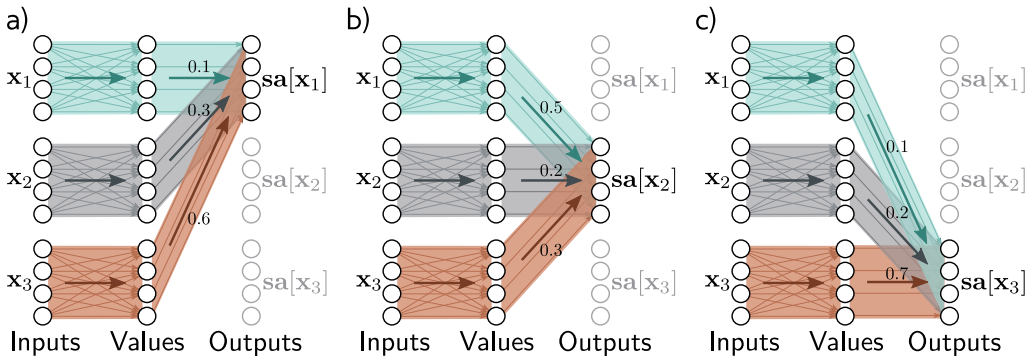

Figure 12.1 Self-attention as routing. The self-attention mechanism takes N inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$, each of size D (here $N = 3$ and $D = 4$) and processes each separately to compute N value vectors. The n^{th} output $\mathbf{sa}[\mathbf{x}_n]$ is then computed as a weighted sum of the N value vectors, where the weights are positive and sum to one. a) Output $\mathbf{sa}[\mathbf{x}_1]$ is computed as $a[1,1] = 0.1$ times the first value vector, $a[1,2] = 0.3$ times the second value vector, and $a[1,3] = 0.6$ times the third value vector. b) Output $\mathbf{sa}[\mathbf{x}_2]$ is computed in the same way, but this time with weights of 0.5, 0.2, and 0.3. c) The weighting for output $\mathbf{sa}[\mathbf{x}_3]$ is different again. Each output can hence be thought of as a different routing of the N values.

12.2.1 Computing and weighting values

Equation 12.2 shows that the same weights $\boldsymbol{\Omega}_v \in \mathbb{R}^{D\times D}$ and biases $\boldsymbol{\beta}_v \in \mathbb{R}^D$ are applied to each input $\mathbf{x}_n \in \mathbb{R}^D$. This computation scales linearly with the sequence length N, and so requires fewer parameters than a fully connected network relating all DN inputs to all DN outputs. The value computation can be viewed as a sparse matrix operation with shared parameters (figure 12.2b).

The attention weights $a[\mathbf{x}_m, \mathbf{x}_n]$ combine the values from different inputs. They are also sparse in a sense, since there is only one weight for each ordered pair of inputs $(\mathbf{x}_m, \mathbf{x}_n)$, regardless of the size of these inputs (figure 12.2c). It follows that the number of attention weights has a quadratic dependence on the sequence length N, but is independent of the length D of each input $\mathbf{x}_n$.

Problem 12.1

12.2.2 Computing attention weights

In the previous section, we saw that the outputs are the result of two chained linear transformations; the value vectors $\boldsymbol{\beta}_v + \boldsymbol{\Omega}_v\mathbf{x}_m$ are computed independently for each input $\mathbf{x}_m$ and these vectors are combined linearly by the attention weights $a[\mathbf{x}_m, \mathbf{x}_n]$. However, the overall self-attention computation is *nonlinear* because the attention weights are themselves nonlinear functions of the input. This is an example of a *hypernetwork*, in which one network is used to compute the weights of another.

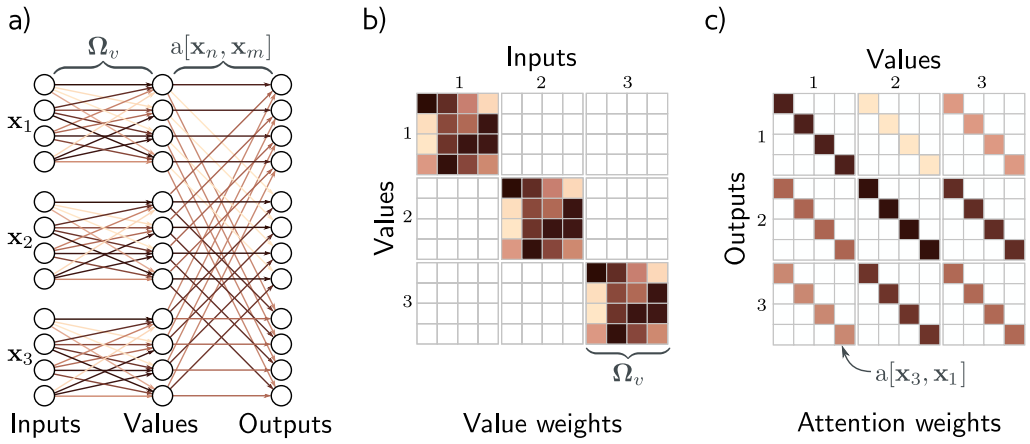

Figure 12.2 Self-attention for $N = 3$ inputs $\mathbf{x}_n$, each with dimension $D = 4$. a) Each input $\mathbf{x}_n$ is operated on independently by the same weights $\boldsymbol{\Omega}_v$ (same color equals same weight) and biases $\boldsymbol{\beta}_v$ (not shown) to form the values $\boldsymbol{\beta}_v + \boldsymbol{\Omega}_v\mathbf{x}_n$. Each output is a linear combination of the values, where there is a shared attention weight $\mathrm{a}[\mathbf{x}_m, \mathbf{x}_n]$ that defines the contribution of the m^{th} value to the n^{th} output. b) Matrix showing block sparsity of linear transformation $\boldsymbol{\Omega}_v$ between inputs and values. c) Matrix showing sparsity of attention weights relating values and outputs (transposed as it will ultimately post-multiply the values).

To compute the attention, we apply two more linear transformations to the inputs:

$$\begin{aligned}\mathbf{q}_n &= \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q\mathbf{x}_n \\ \mathbf{k}_n &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{x}_n,\end{aligned} \tag{12.4}$$

where $\mathbf{q}_n$ and $\mathbf{k}_k$ are referred to as queries and keys, respectively. Then we compute dot products between the queries and keys and pass the results through a softmax function:

$$\begin{aligned}a[\mathbf{x}_m, \mathbf{x}_n] &= \mathrm{softmax}_m\left[\mathbf{k}_\bullet^T\mathbf{q}_n\right] \\ &= \frac{\exp\left[\mathbf{k}_m^T\mathbf{q}_n\right]}{\sum_{m'=1}^{N}\exp\left[\mathbf{k}_{m'}^T\mathbf{q}_n\right]},\end{aligned} \tag{12.5}$$

so for each $\mathbf{x}_n$ they are positive and sum to one (figure 12.3). For obvious reasons, this is known as *dot-product self-attention*.

The names "queries" and "keys" were inherited from the field of information retrieval and have the following interpretation: the dot product operation returns a measure of similarity between its inputs, and so the weights $a[\mathbf{x}_\bullet, \mathbf{x}_n]$ depend on the relative similarities between each query and the keys. The softmax function means that we can

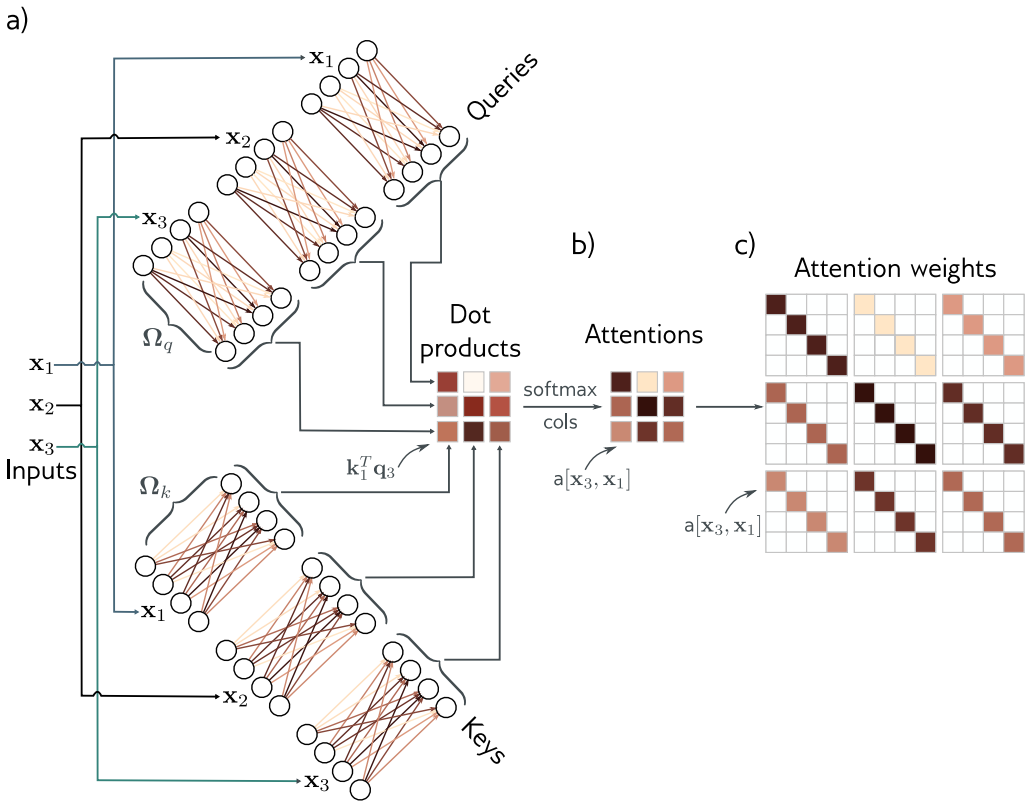

Figure 12.3 Computing attention weights. a) Query vectors $\mathbf{q}_n = \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q\mathbf{x}_n$ and key vectors $\mathbf{k}_n = \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{x}_n$ are computed for each input $\mathbf{x}_n$. b) The dot products between each query and the three keys are passed through a softmax function to form non-negative attentions that sum to one. c) These are used to route the value vectors (figure 12.1) via the sparse matrix from figure 12.2c.

think of the key vectors as "competing" with one another to contribute to the final result. The queries and keys must have the same dimensions. However, these can differ from the dimensions of the values, which are usually the same size as the input so that the representation does not change size.

Problem 12.2

12.2.3 Self-attention summary

The n^{th} output is a weighted sum of the same linear transformation $\mathbf{v}_\bullet = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v\mathbf{x}_\bullet$ applied to all of the inputs, where these attention weights are positive and sum to one. The weights depend on a measure of similarity between input $\mathbf{x}_n$ and the other inputs.

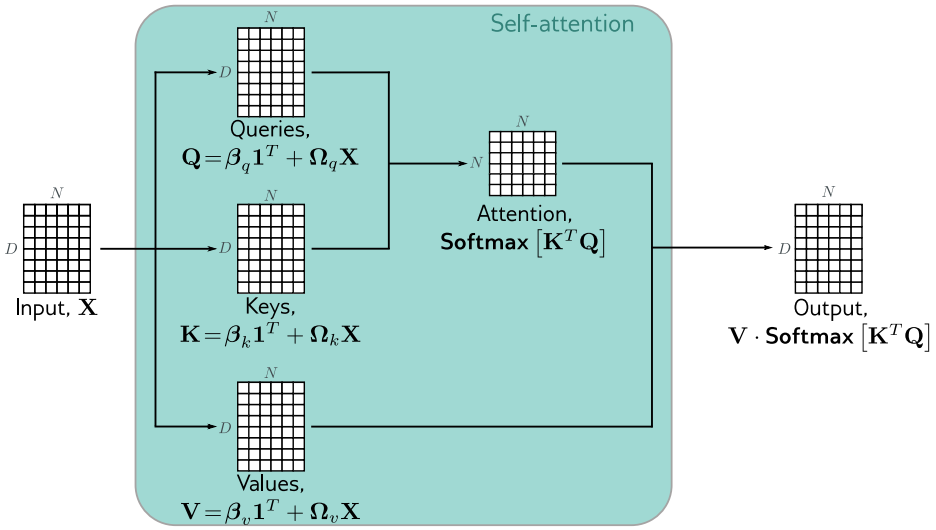

Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication and a softmax operation is applied independently to each column of the resulting matrix to compute the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

There is no activation function, but the mechanism is nonlinear due to the dot-product and a softmax operation used to compute the attention weights.

Note that this mechanism fulfills the initial requirements. First, there is a single shared set of parameters $\boldsymbol{\phi} = \{\boldsymbol{\beta}_v, \boldsymbol{\Omega}_v, \boldsymbol{\beta}_q, \boldsymbol{\Omega}_q, \boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}$. This is independent of the number of inputs N, and so the network can be applied to different sequence lengths. Second, the connections between the inputs (words) depend on the input representations themselves via the computed attention values.

12.2.4 Matrix form

Problem 12.3

The above computation can be written in a compact form if the N inputs $\mathbf{x}_n$ form the columns of the $D \times N$ matrix $\mathbf{X}$. The values, queries, and keys can be computed as:

$$\begin{aligned}\mathbf{V}[\mathbf{X}] &= \boldsymbol{\beta}_v\mathbf{1}^\mathbf{T}+\boldsymbol{\Omega}_\mathbf{v}\mathbf{X}\\ \mathbf{Q}[\mathbf{X}] &= \boldsymbol{\beta}_q\mathbf{1}^\mathbf{T}+\boldsymbol{\Omega}_\mathbf{q}\mathbf{X}\\ \mathbf{K}[\mathbf{X}] &= \boldsymbol{\beta}_k\mathbf{1}^\mathbf{T}+\boldsymbol{\Omega}_\mathbf{k}\mathbf{X},\end{aligned} \tag{12.6}$$

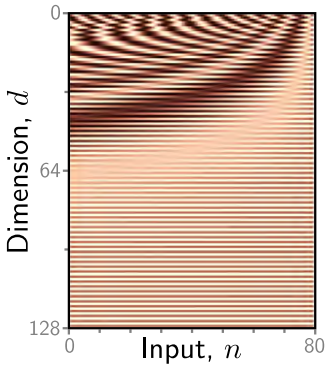

Figure 12.5 Position encodings. The self-attention architecture is equivariant to permutations of the inputs. To ensure that inputs at different positions are treated differently, a position embedding matrix $\boldsymbol{\pi}$ can be added to the data matrix. Each column is different and so the positions can be distinguished. In this case, the position embeddings were predefined to be sinusoidal, but in other cases, they are learned.

where $\mathbf{1}$ is a $D \times 1$ vector containing ones. The self-attention computation is then:

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V}[\mathbf{X}] \cdot \mathbf{Softmax}\Big[\mathbf{K}[\mathbf{X}]^T\mathbf{Q}[\mathbf{X}]\Big], \tag{12.7}$$

where the function $\mathbf{Softmax}[\bullet]$ takes a matrix and performs the softmax operation independently on each of its columns (figure 12.4). In this formulation, we have explicitly included the dependence of the values, queries, and keys on the input $\mathbf{X}$ to emphasize the self-attention computes a kind of triple product based on the inputs. However, from now on we will drop this dependence and just write:

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V} \cdot \mathbf{Softmax}[\mathbf{K}^T\mathbf{Q}]. \tag{12.8}$$

12.3 Extensions to dot-product self-attention

In the previous section, we described the dot-product self-attention mechanism. Here, we introduce three extensions that are almost always used in practice.

12.3.1 Positional encoding

Observant readers will have noticed that the above mechanism loses some important information; the computation is the same regardless of the order of the inputs $\mathbf{x}_n$. More precisely, the self-attention mechanism is equivariant with respect to permutations of the inputs. However, when the inputs correspond to the words in a sentence, the order is important. The sentence The woman ate the raccoon has a quite different meaning to The raccoon ate the woman. There are two main approaches to incorporating position information.

Problem 12.4

Absolute position embeddings: A matrix $\boldsymbol{\Pi}$ is added to the input $\mathbf{X}$ that encodes positional information (figure 12.5). Each column of $\boldsymbol{\Pi}$ is unique, and so contains in-

formation about the position in the input sequence. This matrix may either be chosen by hand or learned. It may be added to the network inputs or added at every network layer. Sometimes it is added to $\mathbf{X}$ in the computation of the queries and keys, but not for the values.

Relative position embeddings: The input to a self-attention mechanism may be an entire sentence, many sentences, or just a fragment of a sentence, and the absolute position of a word is much less important than the relative position between two inputs. Of course, this can be recovered if the system knows the absolute position of both, but relative position embeddings encode this information directly. Each element of the attention matrix corresponds to a particular offset between query position a and key position b. Relative position embeddings learn a parameter $\pi_{a,b}$ for each offset and use this to modify the attention matrix by adding these values, multiplying by them, or using them to modify the attention matrix in some other way.

12.3.2 Scaled dot product self-attention

The dot products in the attention computation can have very large magnitudes and move the arguments to the softmax function into a region where the largest value completely dominates. Now small changes to the inputs to the softmax function have little effect on the output (i.e., the gradients are very small) and the model becomes hard to train. To prevent this, the dot products are scaled by the square root of the dimension D_q of the queries and keys (i.e., the number of rows in $\boldsymbol{\Omega}_q$ and $\boldsymbol{\Omega}_k$, which must be the same):

Problem 12.5

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V} \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{D_q}}\right]. \tag{12.9}$$

This is known as *scaled dot product self-attention.*

12.3.3 Multiple heads

Multiple self-attention mechanisms are usually applied in parallel, and this is known as *multi-head self-attention.* Now H different sets of values, keys, and queries are computed:

$$\begin{aligned}\mathbf{V}_h &= \boldsymbol{\beta}_{vh}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{vh}\mathbf{X} \\ \mathbf{Q}_h &= \boldsymbol{\beta}_{qh}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{qh}\mathbf{X} \\ \mathbf{K}_h &= \boldsymbol{\beta}_{kh}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{kh}\mathbf{X}.\end{aligned} \tag{12.10}$$

The h^{th} self-attention mechanism or *head* can be written as:

$$\mathbf{Sa}_h[\mathbf{X}] = \mathbf{V}_h \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}_h^T\mathbf{Q}_h}{\sqrt{D_q}}\right], \tag{12.11}$$

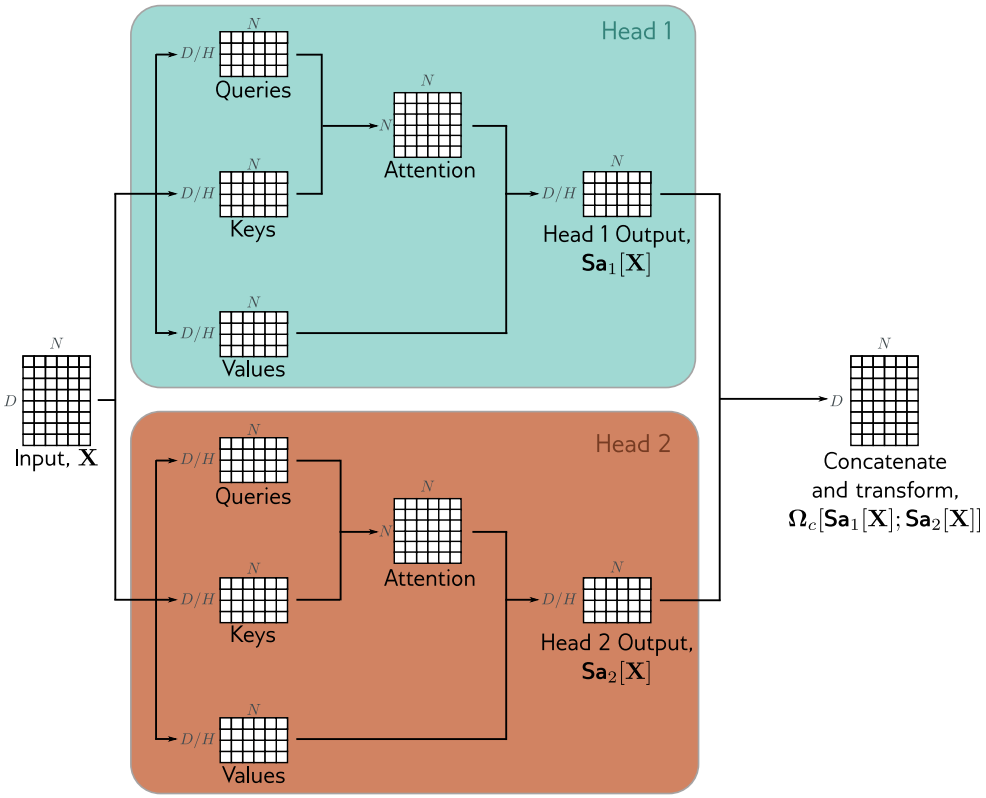

Figure 12.6 Multi-head self-attention. Self-attention occurs in parallel across multiple "heads". Each has its own queries, keys, and values. Here two heads are depicted, in the cyan and orange boxes, respectively. The outputs are vertically concatenated and another linear transformation $\boldsymbol{\Omega}_c$ is used to recombine them.

where we have different parameters $\{\boldsymbol{\beta}_{vh}, \boldsymbol{\Omega}_{vh}\}$, $\{\boldsymbol{\beta}_{qh}, \boldsymbol{\Omega}_{qh}\}$, and $\{\boldsymbol{\beta}_{kh}, \boldsymbol{\Omega}_{kh}\}$ for each head. Typically, if the dimension of the inputs $\mathbf{x}_m$ is D and there are H heads, then the values, queries, and keys will all be of size D/H, as this allows for an efficient implementation. The outputs of these self-attention mechanisms are vertically concatenated and another linear transform $\boldsymbol{\Omega}_c$ is applied to combine them (figure 12.6):

Problem 12.6

$$\mathbf{MhSa}[\mathbf{X}] = \boldsymbol{\Omega}_c\left[\mathbf{Sa}_1[\mathbf{X}]; \mathbf{Sa}_2[\mathbf{X}]; \ldots; \mathbf{Sa}_H[\mathbf{X}]\right]. \tag{12.12}$$

Multiple heads seem to be necessary to make the transformer work well. It has been speculated that they make the self-attention network more robust to bad initializations.

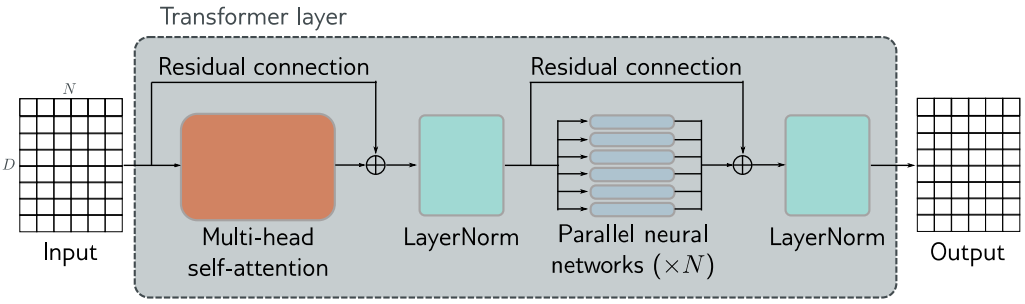

Figure 12.7 The transformer layer. The input consists of a $D \times N$ matrix containing the D dimensional word embeddings for each of the N input tokens. The output is a matrix of the same size. The transformer layer consists of a series of operations. First, there is a multi-head attention block, which allows the word embeddings to interact with one another. This forms the processing of a residual block, so the inputs are added back to the output. Second, a LayerNorm operation is applied. Third, there is a second residual layer where the same two-layer fully connected neural network is applied to each word representation separately. Finally, LayerNorm is applied again.

12.4 Transformer layers

Self-attention is just one part of a larger *transformer layer*. This consists of a multi-head self-attention unit (which allows the word representations to interact with each other) followed by a fully connected network $\mathbf{mlp}[\mathbf{x}_\bullet]$ (that operates separately on each word). Both units are residual networks (i.e., their output is added back to the original input). In addition, it is typical to add a LayerNorm operation after both the self-attention and fully connected networks. This is similar to BatchNorm but uses statistics across the tokens within a single input sequence to perform the normalization (section 11.4 and figure 11.14). The complete layer can be described by the following series of operations:

$$\begin{aligned}
\mathbf{X} &\leftarrow \mathbf{X} + \mathbf{MhSa}[\mathbf{X}] \\
\mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}] \\
\mathbf{x}_n &\leftarrow \mathbf{x}_n + \mathbf{mlp}[\mathbf{x}_n] \qquad \forall\, n \in \{1, \ldots, N\} \\
\mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}], \qquad (12.13)
\end{aligned}$$

where the column vectors $\mathbf{x}_n$ are separately taken from the full data matrix $\mathbf{X}$. In a real network, the data would pass through a series of these layers.

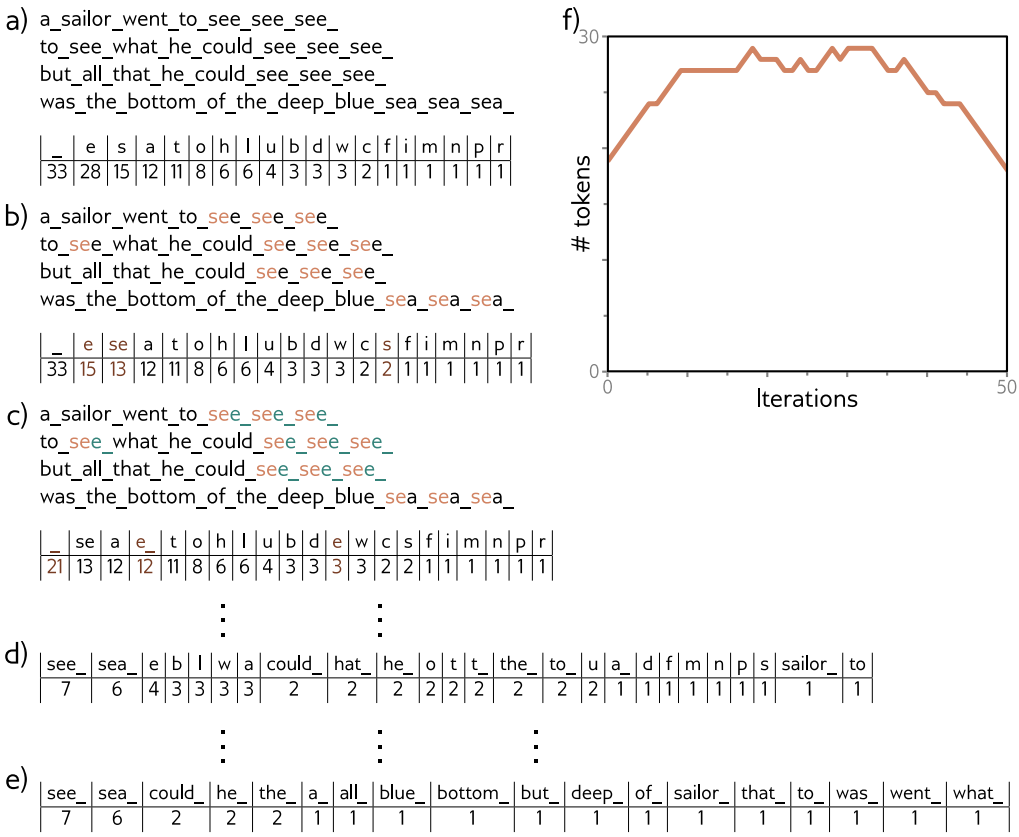

Figure 12.8 Sub-word tokenization. a) A passage of text from a nursery rhyme. The tokens are initially just the characters and whitespace (represented by an underscore), and their frequencies are displayed in the table. b) At each iteration, the sub-word tokenizer looks for the most commonly occurring adjacent pair of characters (in this case se) and merges them. This creates a new token and decreases the counts for the original tokens s and e. c) At the second iteration, the algorithm merges e and the whitespace character_. Note that the last character of the first token to be merged cannot be whitespace, which prevents merging across words. d) After 22 iterations, the tokens consist of a mix of letters, word fragments, and commonly occurring words. e) If we continue this process indefinitely, the tokens just represent the words. f) Over time, the number of tokens increases as we add word fragments to the letters, and then decreases again as we merge these fragments. In a real situation, there would be a very large number of words and the algorithm would terminate when the vocabulary size (number of tokens) reached a predetermined value. Punctuation and capital letters would also be treated as separate input characters.

12.5 Transformers for natural language processing

A typical natural language processing (NLP) pipeline starts with a *tokenize*r that splits the text into words or word fragments. Then each of these tokens is mapped to a learned embedding. These embeddings are passed through a series of transformer layers. We now consider each of these stages in turn.

12.5.1 Tokenization

A text processing pipeline begins with a *tokenizer*. This splits the text into a *vocabulary* of smaller constituent units (tokens) that can be processed by the subsequent network. In the discussion above, we have implied that these tokens represent words, but there are several difficulties with this.

- Inevitably, some words (e.g., names) will not be in the vocabulary.
- It's not clear how to handle punctuation but this is important. If a sentence ends in a question mark, then we need to encode this information.
- The vocabulary would need different tokens for versions of the same word with different suffixes (e.g., walk, walks, walked, walking) and there is no way to clarify that these variations are related.

One approach would be just to use letters and punctuation marks as the vocabulary, but this would mean splitting text into many very small parts and requiring the subsequent network to re-learn the relations between them.

In practice, a compromise between using letters and full words is used, and the final vocabulary includes both common words and word fragments from which larger and less frequent words can be composed. The vocabulary is computed using a *sub-word tokenizer* such as *byte pair encoding* (figure 12.8) that greedily merges commonly occurring sub-strings based on their frequency.

Problem 12.7

12.5.2 Embeddings

Each token in the vocabulary $\mathcal{V}$ is mapped to a *word embedding*. Importantly, the same token always maps to the same embedding. To accomplish this, the N input tokens are encoded in the matrix $\mathbf{T} \in \mathbb{R}^{|\mathcal{V}| \times N}$, where n^{th} column corresponds to the n^{th} token and is a $|\mathcal{V}| \times 1$ *one-hot vector* (i.e., a vector where every entry is zero except for the entry corresponding to the token, which is set to one). The embeddings for the whole vocabulary are stored in a matrix $\boldsymbol{\Omega}_e \in \mathbb{R}^{D \times |\mathcal{V}|}$. The input embeddings are then computed as $\mathbf{X} = \boldsymbol{\Omega}_e \mathbf{T}$, and $\boldsymbol{\Omega}_e$ is treated like any other network parameter (figure 12.9). A typical embedding size D is 1024 and a typical total vocabulary size $|\mathcal{V}|$ is 30,000, so even before the main network, there are many parameters in $\boldsymbol{\Omega}_e$ to learn.

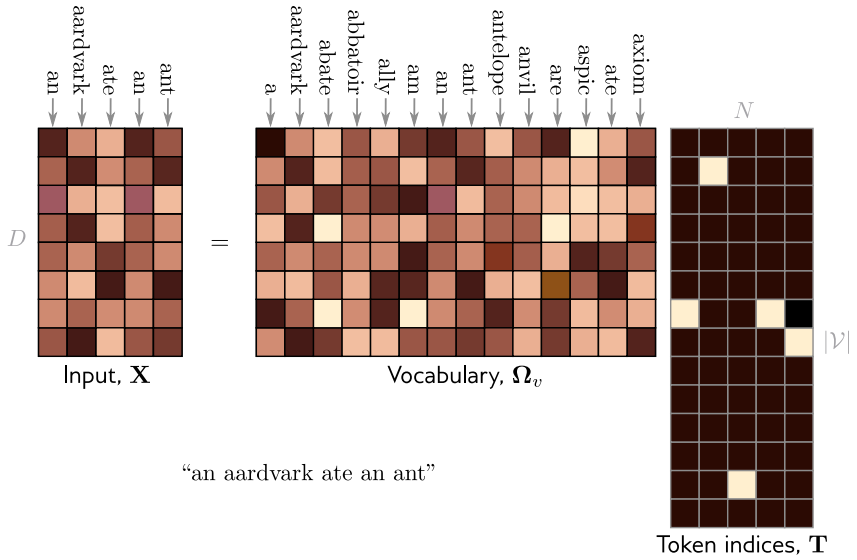

Figure 12.9 The input embedding matrix $\mathbf{X} \in \mathbb{R}^{D \times N}$ contains N embeddings of length D and is created by multiplying a matrix $\boldsymbol{\Omega}_e$ containing the embeddings for the entire vocabulary with a matrix containing one-hot vectors in its columns that correspond to the word or sub-word indices. The vocabulary matrix $\boldsymbol{\Omega}_e$ is considered a parameter of the model and is learned along with the other parameters. Note that the two embeddings for the word an in $\mathbf{X}$ are the same.

12.5.3 Transformer model

Finally, the input embedding matrix $\mathbf{X}$ is passed through a series of transformer layers, which we'll refer to as a *transformer model*. There are three types of transformer models. An *encoder* transforms the text into a representation that can support a variety of language tasks. A *decoder* generates a new token that continues the input text. *Encoder-decoder models* are used in *sequence-to-sequence tasks*, where one text string is converted into another (e.g., machine translation). These three variations are described in sections 12.6–12.8, respectively.

12.6 Encoder model example: BERT

BERT is an encoder model that uses a vocabulary of 30,000 tokens. The tokens are converted to 1024-dimensional word embeddings and passed through 24 transformer layers. Each contains a self-attention mechanism with 16 heads, and for each head, the queries, keys, and values are of dimension 64 (i.e., the matrices $\boldsymbol{\Omega}_{vh}, \boldsymbol{\Omega}_{qh}, \boldsymbol{\Omega}_{kh}$ are of

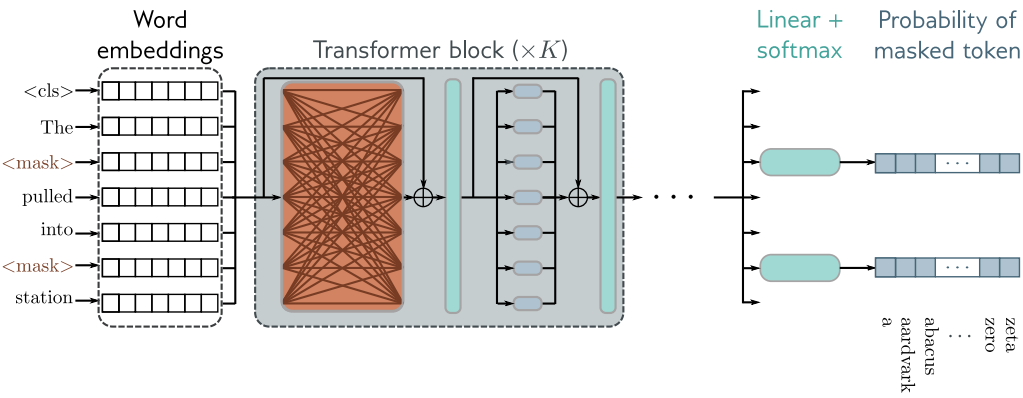

Figure 12.10 Pre-training for BERT-like encoder. The input tokens (and a special <cls> token denoting the start of the sequence) are converted to word embeddings. Here these are represented as rows rather than columns, and so the box labeled "word embeddings" is $\mathbf{X}^T$. These embeddings are passed through a series of transformer layers (orange connections indicate that every token attends to every other token in these layers) to create a set of output embeddings. A small fraction of the input tokens is replaced at random with a generic <mask> token. In pre-training, the goal is to predict the missing word from the associated output embedding. As such, the output embeddings are passed through a softmax function and the multiclass classification loss (section 5.24) is used. This task has the advantage that it uses both the left and right context to predict the missing word but has the disadvantage that it does not make efficient use of data; here, seven tokens need to be processed to add two terms to the objective function.

size 1024×64). The dimension of the hidden layer in the neural network layer of the transformer is 4096. The total number of parameters is $\sim$ 340 million. When BERT was introduced, this was considered large but is now orders of magnitude smaller than state-of-the-art models.

Encoder models like BERT exploit transfer learning (section 9.3.6). During *pre-training*, the parameters of the transformer architecture are learned using *self-supervision* from a large corpus of text. The goal here is for the model to learn general information about the statistics of language. In the *fine-tuning stage*, the resulting network is adapted to solve a particular task, using a smaller body of supervised training data.

12.6.1 Pre-training

In the pre-training stage, the network is trained using self-supervision. This allows the use of enormous amounts of data without the need for manual labels. For BERT, the self-supervision task consisted of predicting missing words from sentences from a large

Problem 12.8

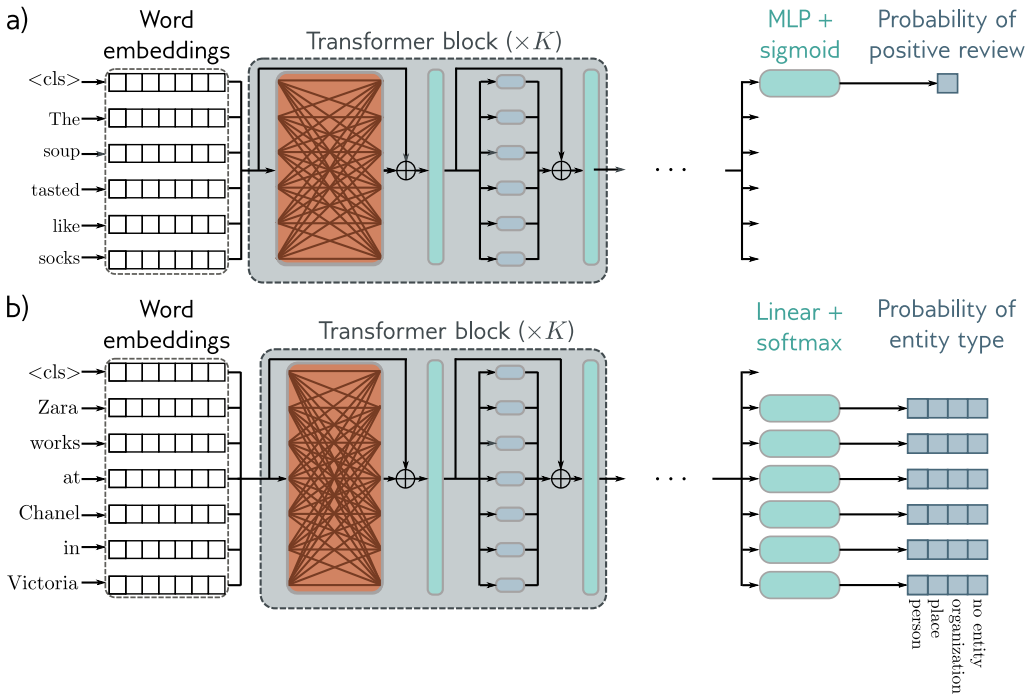

Figure 12.11 After pre-training, the encoder is fine-tuned using manually labeled data to solve a particular task. Usually, a linear transformation or a multi-layer perceptron (MLP) is appended to the encoder to produce whatever output is required for the task. a) Example text classification task. In this sentiment classification task, the <cls> token embedding is used to predict the probability that the review is positive. b) Example word classification task. In this named entity recognition problem, the embedding for each word is used to predict whether the word corresponds to a person, place, or organization, or is not an entity.

internet corpus (figure 12.10).[1] During training, the maximum input length was 512 tokens, and the batch size was 256. The system was trained for a million steps, which corresponded to roughly 50 epochs of the 3.3-billion word corpus.

Predicting missing words forces the transformer network to understand some syntax. For example, it might learn that the adjective red is often found before nouns like house or car but never before a verb like shout. It also allows the model to learn some superficial *common sense* about the world. For example, after training, the model will assign a higher probability to the missing word train in the sentence The <mask> pulled into the station, than it would to the word peanut. However, the degree of "understanding" that this type of model can ever have is limited.

[1]BERT also used a secondary task that involved predicting whether two sentences were originally adjacent in the text or not but this only marginally improved performance.

12.6.2 Fine-tuning

In the fine-tuning stage, the model parameters are adjusted to specialize the network to a particular task. An extra layer is appended onto the transformer network to convert the output vectors to the desired output format. Examples include:

Text classification: In BERT, there is a special token known as the classification or <cls> token that is placed at the start of each string during pre-training. For text classification tasks like *sentiment analysis* (in which the passage is labeled as having a positive or negative emotional tone), the vector associated with the <cls> token is mapped to a single number and passed through a logistic sigmoid (figure 12.11a). This contributes to a standard binary cross-entropy loss (section 5.4).

Word classification: The goal of *named entity recognition* is to classify each word as an entity type (e.g., person, place, organization, or no-entity). To this end, each input embedding $\mathbf{x}_n$ is mapped to a $K \times 1$ vector where K is the entity type. This is passed through a softmax function to create probabilities for each class, which contribute to a multiclass cross-entropy loss (figure 12.11b).

Text span prediction: In the SQuAD 1.1 question answering task, the question and a passage from Wikipedia containing the answer are concatenated and tokenized. BERT is then used to predict the text span in the passage that contains the answer. Each token maps to two numbers that indicate how likely it is that the text span begins and ends at this location. The resulting two sets of numbers are put through two softmax functions and the probability of any text span being the answer can then be derived by combining the probability of starting and ending at the appropriate places.

12.7 Decoder model example: GPT3

In this section, we present a high-level description of GPT3, which is an example of a decoder model. The basic architecture is extremely similar to the encoder model in that it consists of a series of transformer layers that operate on learned word embeddings. However, the goal is different. The encoder aimed to build a representation of the text that could be fine-tuned to solve a variety of more specific NLP tasks. Conversely, the decoder has one purpose, which is to generate the next token in a sequence. By feeding the extended sequence back into the model, it can generate a coherent text passage.

12.7.1 Language modeling

More formally, GPT3 constructs an autoregressive language model. For any sentence, it aims to model the joint probability $Pr(t_1, t_2, \ldots, t_N)$ of the N observed tokens and it does this by factorizing this joint probability into an autoregressive sequence:

$$Pr(t_1, t_2, \ldots, t_N) = Pr(t_1) \prod_{n=2}^{N} Pr(t_n | t_1, \ldots, t_{n-1}). \tag{12.14}$$

This is easiest to understand with a concrete example. Consider the sentence It takes great personal courage to let yourself appear weak. For simplicity, let's assume that the tokens are the full words. The probability of the full sentence is:

$$\begin{aligned} &Pr(\text{It takes great personal courage to let yourself appear weak}) \quad = \\ &\quad Pr(\text{It}) \times Pr(\text{takes}|\text{It}) \times Pr(\text{great}|\text{It takes}) \times Pr(\text{courage}|\text{It takes great}) \times \\ &\quad Pr(\text{to}|\text{It takes great courage}) \times Pr(\text{let}|\text{It takes great courage to}) \times \\ &\quad Pr(\text{yourself}|\text{It takes great courage to let}) \times \\ &\quad Pr(\text{appear}|\text{It takes great courage to let yourself}) \times \\ &\quad Pr(\text{weak}|\text{It takes great courage to let yourself appear}). \end{aligned} \tag{12.15}$$

This demonstrates the connection between the probabilistic formulation of the loss function and the next token prediction task.

12.7.2 Masked self-attention

To train a decoder, we maximize the log probability of the input text under the autoregressive model. Ideally, we would pass in the whole sentence and compute all of the log probabilities and gradients simultaneously. However, this poses a problem; if we pass in the full sentence, then the term computing $\log\left[Pr(\text{great}|\text{It takes})\right]$ has access to both the answer great and the right context courage to let yourself appear weak.

Fortunately, in a transformer network, the tokens only interact in the self-attention layers. Hence, the problem can be resolved by ensuring that the attention to the answer and the right context is zero. This can be achieved by setting the corresponding dot products in the self-attention computation (equation 12.5) to negative infinity before they are passed through the $\textbf{softmax}[\bullet]$ function. This is known as *masked self-attention*.

The full decoder network operates as follows. The input text is tokenized, and the tokens are converted to embeddings. The embeddings are passed into the transformer network, but now the transformer layers use masked self-attention so that they can only attend to the current and previous tokens. Each of the output embeddings can be thought of as representing a partial sentence, and for each, the goal is to predict the next token in the sequence. Consequently, after the transformer layers, a linear layer maps each word embedding to the size of the vocabulary, followed by a $\textbf{softmax}[\bullet]$ function that converts these values to probabilities. We aim to maximize the sum of the log probabilities of the next token in the ground truth sequence at every position using a standard multiclass cross-entropy loss (figure 12.12).

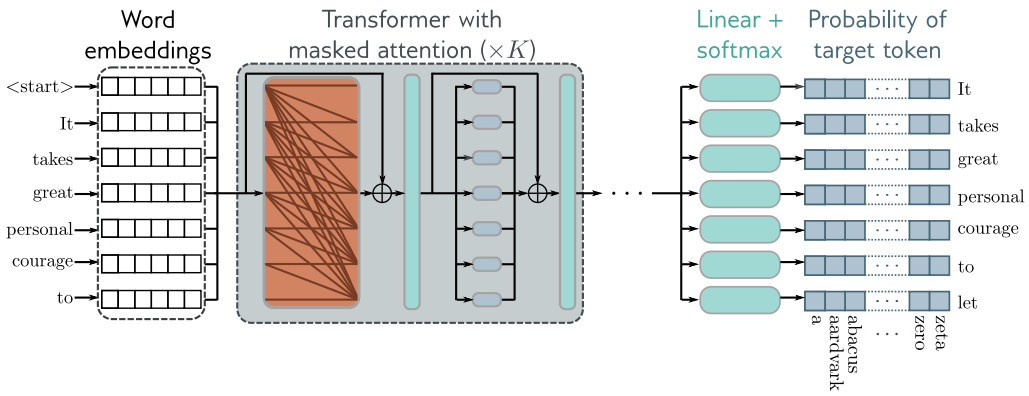

Figure 12.12 GPT3-type decoder network. The tokens are mapped to word embeddings with a special <start> token at the beginning of the sequence. The embeddings are passed through a series of transformers that use masked self-attention. Here, each position in the sentence can only attend to its own embedding and the embeddings of tokens earlier in the sequence (orange connections). The goal at each position is to maximize the probability of the next ground truth token in the sequence. In other words, at position one we want to maximize the probability of the token It, at position two we want to maximize the probability of the token takes, and so on. Masked self-attention is designed to ensure that the system cannot cheat by looking at subsequent inputs. This system has the advantage that it makes efficient use of the data since every word contributes a term to the loss function. However, it has the disadvantage that it only exploits the left context of each word.

12.7.3 Generating text from a decoder

The autoregressive language model is the first example of a *generative model* discussed in this book. Since it defines a probability model over text sequences, it can be used to sample new examples of plausible text. To generate from the model, we start with an input sequence of text (which might be just a special <start> token) and feed this into the network which then outputs the probabilities over possible next tokens. We can then either pick the most likely token or sample from this probability distribution. The new extended sequence can be fed back into the decoder network that outputs the probability distribution over the next token and in this way, we can generate large bodies of text. The computation can be made quite efficient as prior embeddings do not depend on subsequent ones due to the masked self-attention and hence much of the earlier computation can be recycled as we generate subsequent tokens.

In practice, many strategies can be employed to help make the output text more coherent. For example, *beam search* keeps track of multiple possible sentence completions with the aim of finding the overall most likely (which is not necessarily found by greedily choosing the most likely next word at each step). *Top-k sampling* randomly draws the

next word from only the top-K most likely possibilities to prevent the system from accidentally choosing from the long tail of low probability tokens and leading to an unnecessary linguistic dead end.

12.7.4 GPT3 and few-shot learning

GPT3 applies these ideas on a massive scale. The sequence lengths are 2048 tokens long and since multiple spans of 2048 tokens are processed at once, the total batch size is 3.2 million tokens. There are 96 transformer layers (some of which implement a sparse version of attention), each of which processes a word embedding of size 12288. There are 96 heads in the self-attention layers and the value, query, and key dimension is 128. It is trained with 300 billion tokens and learns a total of 175 billion parameters.

Here's an example of completing text from the GPT3 model where the text provided to the model is in orange and the generated text is in cyan:

Understanding Deep Learning is a new textbook from MIT Press by Simon Prince that's designed to offer an accessible, broad introduction to the field. Deep learning is a branch of machine learning that is concerned with algorithms that learn from data that is unstructured or unlabeled. The book is divided into four sections:

1. Introduction to deep learning
2. Deep learning architecture
3. Deep learning algorithms
4. Applications of deep learning

The first section offers an introduction to deep learning, including its history and origins. The second section covers deep learning architecture, discussing various types of neural networks and their applications. The third section dives into deep learning algorithms, including supervised and unsupervised learning, reinforcement learning, and more. The fourth section applies deep learning to various domains, such as computer vision, natural language processing, and robotics.

This continuation is plausible, if not entirely accurate.

One surprising property of learning models on this scale is that they can perform many tasks without the need for fine-tuning. If we provide several examples of correct question/answer pairs, and then another question, they often answer the final question correctly by completing the sequence. One example is correcting English grammar:

Poor English input: I eated the purple berries.
Good English output: I ate the purple berries.
Poor English input: Thank you for picking me as your designer. I'd appreciate it.
Good English output: Thank you for choosing me as your designer. I appreciate it.
Poor English input: The mentioned changes have done. or I did the alteration that you requested. or I changed things you wanted and did the modifications.
Good English output: The requested changes have been made. or I made the alteration that

you requested. or I changed things you wanted and made the modifications.
Poor English input: I'd be more than happy to work with you in another project.
Good English output: I'd be more than happy to work with you on another project.

Here, the text containing the paired examples in cyan was provided as context for GPT3 and the system then generated the correct answer in cyan. This phenomenon extends to many situations including generating code snippets based on natural language descriptions, arithmetic, translating between languages, and answering questions about text passages. Consequently, it is argued that enormous language models are *few-shot learners*; they can learn to do novel tasks based on just a few examples. However, in practice performance is erratic, and it is not clear the extent to which it is extrapolating from learned examples rather than merely interpolating, or copying verbatim.

12.8 Encoder-decoder model example: machine translation

Translation between languages is an example of a *sequence-to-sequence* task. This requires an encoder (to compute a good representation of the source sentence) and a decoder (to generate the sentence in the target language). This task can be tackled using an *encoder-decoder* model.

Consider the example of translating from English to French. The encoder receives the sentence in English and processes it through a series of transformer layers to create an output representation for each token. During training, the decoder receives the sentence in French and passes it through a series of transformer layers that use masked self-attention and produce the subsequent word at each position. However, the decoder layers also attend to the output of the encoder. Consequently, each French output word is conditioned not only on the previous output words but also on the entire English sentence that it is translating (figure 12.13).

This is achieved by modifying the transformer layers in the decoder. The original transformer layer in the decoder (figure 12.12) consisted of a masked self-attention layer followed by a neural network applied individually to each embedding. A new self-attention layer is added between these two components, in which the decoder embeddings attend to the encoder embeddings. This uses a version of self-attention known as *encoder-decoder attention* or *cross-attention* where the queries are computed from the decoder embeddings and the keys and values from the encoder embeddings (figure 12.14).

12.9 Transformers for long sequences

Since each token in a transformer encoder model interacts with every other token, the computational complexity scales quadratically with the length of the sequence. For a decoder model, each token only interacts with previous tokens, so there are roughly

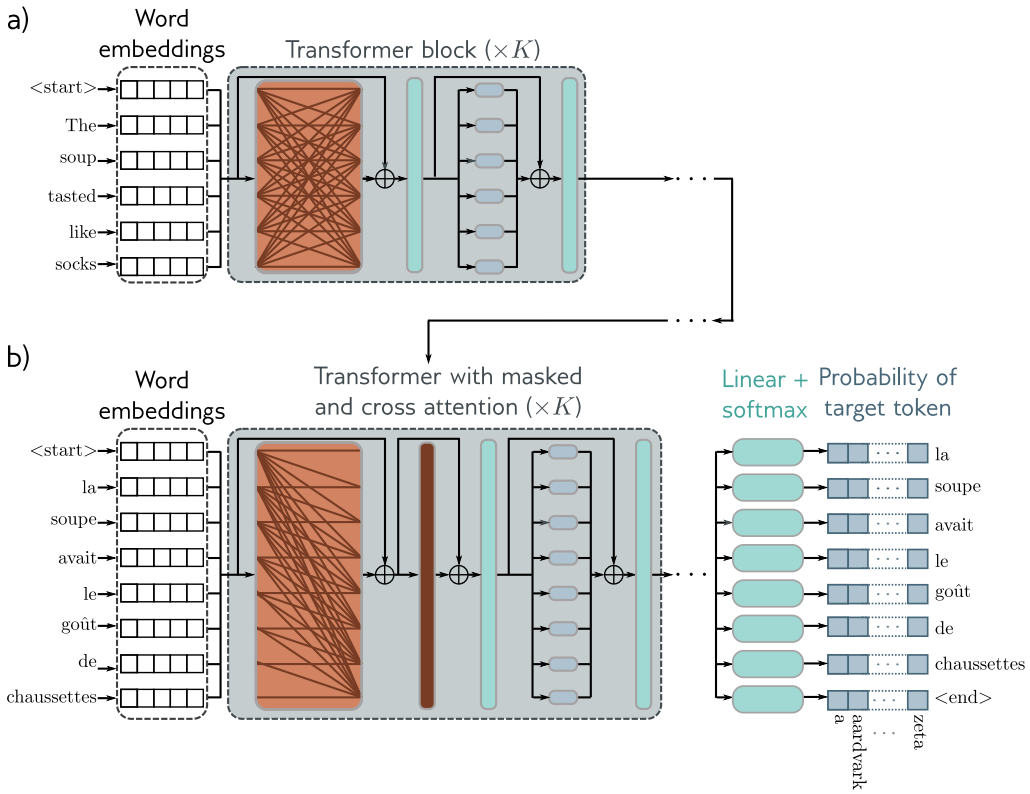

Figure 12.13 Encoder-decoder architecture. Two sentences are passed to the system with the goal of learning to translate the first into the second. The first sentence is passed through a standard encoder. The second sentence is passed through a decoder that uses masked self-attention but also attends to the output embeddings of the encoder using cross-attention (brown rectangle). The loss function is the same as for the decoder; we want to maximize the probability of the next word in the output sequence.

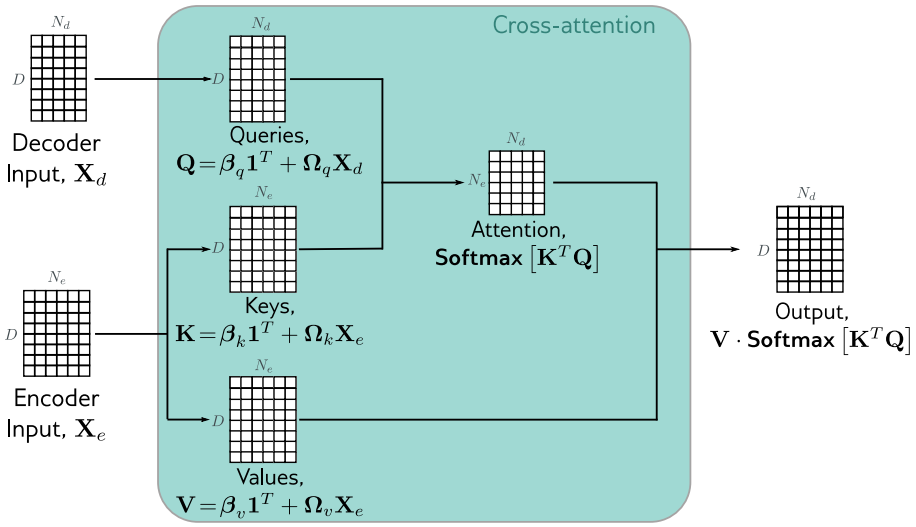

Figure 12.14 cross-attention. The flow of computation is the same as in standard self-attention. However, the queries are now calculated from the decoder embeddings $\mathbf{X}_d$, and the keys and values from the encoder embeddings $\mathbf{X}_e$.

half the number of interactions, but the complexity still scales quadratically. These relationships can be visualized as interaction matrices (figure 12.15a–b).

This quadratic increase in the amount of computation ultimately limits the length of sequences that can be used. Many methods have been developed to extend the transformer to cope with longer sequences. An important subset of these prunes the self-attention interactions, or equivalently sparsify the interaction matrix. One possibility is to use a convolutional structure so that each token only interacts with a few neighboring tokens. Across multiple layers, tokens still interact at larger distances as the receptive field expands. As for convolution in images, the kernel can vary in size and dilation rate.

A pure convolutional approach requires many layers to integrate information over large distances. One way to speed up this process is to allow select tokens (perhaps at the start of every sentence) to attend to all other tokens (encoder model) or all previous tokens (decoder model). A similar idea is to have a small number of global tokens that connect to all the other tokens and themselves. Like the <cls> token, these do not represent any word, but serve to provide long-distance connections.

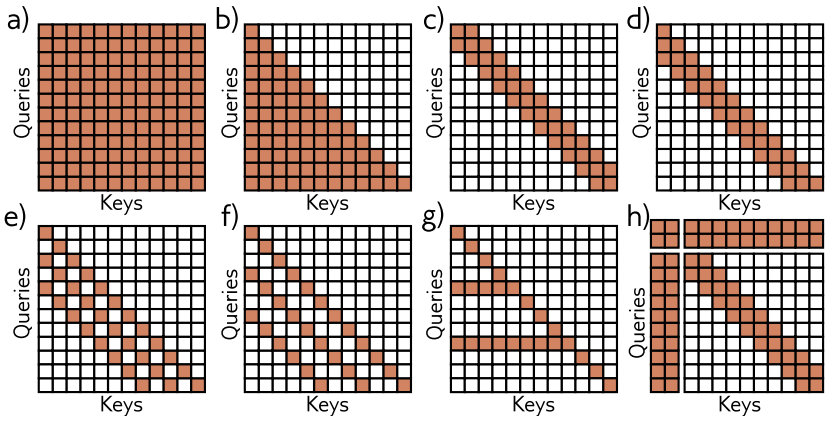

Figure 12.15 Interaction matrices for self-attention. a) In an encoder, every token interacts with every other token and computation expands quadratically with the number of tokens. b) In a decoder, each token only interacts with the previous tokens, but complexity is still quadratic. c) Complexity can be reduced by using a convolutional structure (encoder case) d) Convolutional structure for decoder case. e–f) Convolutional structure with dilation rate of two and three (decoder case). g) Another strategy is to allow selected tokens to interact with all the other tokens (encoder case) or all the previous tokens (decoder case pictured). h) Alternatively, global tokens can be introduced (left two columns and top two rows). These interact with all of the tokens as well as with each other.

12.10 Transformers for images

Transformers were initially developed for text data. Their enormous success in this area led to experimentation on images. This was not an obviously promising idea for two reasons. First, there are many more pixels in an image than words in a sentence, and so the quadratic complexity of self-attention poses a practical bottleneck. Second, convolutional nets are designed to have a good inductive bias because each layer is equivariant to spatial translation. However, this must be learned in a transformer network.

Regardless of these apparent disadvantages, transformer networks for images have now eclipsed the performance of convolutional networks for image classification and other tasks. This is partly because of the enormous scale at which they can be built, and the large amounts of data that can be used to pre-train the networks. This section describes transformer models for images.

12.10.1 ImageGPT

ImageGPT is a transformer decoder; it builds an autoregressive model of image pixels that ingests a partial image and predicts the subsequent pixel value. The quadratic

Figure 12.16 ImageGPT. a) Images generated from the autoregressive ImageGPT model. The top-left pixel is drawn from the estimated empirical distribution at this position. Subsequent pixels are generated in turn conditioned on the previous ones, working along the rows until the bottom-right of the image is reached. For each pixel, the transformer decoder generates a conditional distribution as in equation 12.14, and a sample is drawn. The extended sequence is then fed back into the network to generate the next pixel, and so on. b) Image completion. In each case, the lower half of the image is removed (top row) and ImageGPT completes the remaining part pixel by pixel (three different completions shown). Adapted from https://openai.com/blog/image-gpt/.

complexity of the transformer network means that the largest model (which contained 6.8 billion parameters) could still only operate on 64×64 images. Moreover, to make this tractable, the original 24-bit RGB color space had to be quantized into a nine-bit color space, so the system ingests (and predicts) one of 512 possible tokens at each position.

Images are naturally 2D objects, but ImageGPT simply learns a different position embedding at each pixel. Hence it must learn that each pixel not only has a close relationship with its preceding neighbors but also with nearby pixels in the row above. Figure 12.16 shows example generation results.

The internal representation of this decoder was used as a basis for image classification. The final pixel embeddings are averaged and a linear layer maps these to activations which are passed through a softmax layer to predict class probabilities. The system is pre-trained on a large corpus of web images and then fine-tuned on the ImageNet database resized to 48×48 pixels using a loss function that contains both a cross-entropy term for image classification and a generative loss term for predicting the pixels. Despite

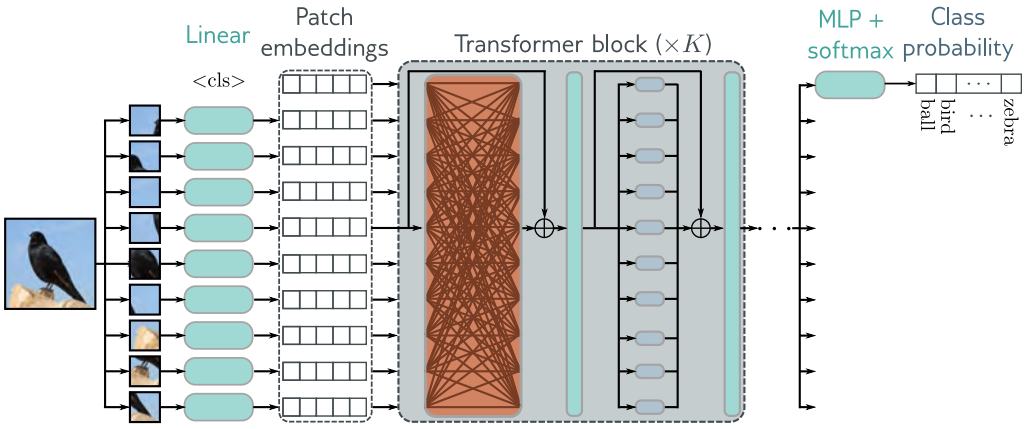

Figure 12.17 Vision transformer. The Vision Transformer (ViT) breaks the image into a grid of patches (16×16 in the original implementation) and each of these is projected via a learned linear transformation to become a patch embedding. These patch embeddings are fed into a transformer encoder network and the <cls> token is used to predict the class probabilities.

using a large amount of external training data, the system achieved only a 27.4% top-1 error rate on ImageNet (figure 10.15). This was less than convolutional architectures of the time (see figure 10.22) but is still impressive given the small input image size; it is unsurprising that it fails to classify images where the target object is small or thin.

12.10.2 Vision Transformer (ViT)

The *Vision Transformer* tackled the problem of image resolution by dividing the image into 16×16 patches (figure 12.17). Each of these is mapped to a lower dimension via a learned linear transformation and it is these representations that are fed into the transformer network. Once again, standard 1D position embeddings were learned.

Problem 12.9

This is an encoder model with a <cls> token (see figures 12.10–12.11). However, unlike BERT it used *supervised* pre-training on a large database of 303 million labeled images from 18,000 classes. The <cls> token was mapped via a final network layer to create activations that are fed into a softmax function to generate class probabilities. After pre-training, the system is applied to the final classification task by replacing this final layer with one that maps to the desired number of classes and is fine-tuned.

For the ImageNet benchmark, this system achieved an 11.45% top-1 error rate. However, it did not perform as well as the best contemporary convolutional networks without supervised pre-training. It appears that the strong inductive bias of convolutional networks can only be superseded by employing extremely large amounts of training data.

12.10.3 Multi-scale vision transformers

The Vision Transformer differs from convolutional architectures in that it operates on a single scale. Many transformer models that process the image at multiple scales have subsequently been proposed. Similarly to convolutional networks, these generally start with high-resolution patches and few channels and gradually decrease the resolution, while simultaneously increasing the number of channels.

A representative example of a multi-scale transformer is the *shifted-window* or *SWIN* transformer. This is an encoder transformer that divides the image into patches and groups these patches into a grid of windows within which self-attention is applied independently (figure 12.18). These windows are shifted in adjacent transformer blocks, so the effective receptive field at a given patch can expand beyond the window border.

The scale is reduced periodically by concatenating features from non-overlapping 2×2 patches and applying a linear transformation that maps these concatenated features to twice the original number of channels. This architecture does not have a <cls> token but instead averages the output features at the last layer. These are then mapped via a linear layer to the desired number of classes and passed through a softmax function to output class probabilities. At the time of writing, the most sophisticated version of this architecture achieves a 9.89% top-1 error rate on the ImageNet database.

A related idea is to periodically integrate information from across the whole image. *Dual attention vision transformers* (DaVIT) achieve this by alternating two types of transformer blocks. In the first, different patches of the image attend to one another, and the self-attention computation uses all the channels. In the second, the different channels attend to one another, and the self-attention computation uses all of the spatial positions. This architecture reaches a 9.60% top-1 error rate on ImageNet and is close to the state-of-the-art at the time of writing.

Problem 12.10

12.11 Summary

This chapter introduced self-attention and then described how this contributes to the transformer architecture. The encoder, decoder, and encoder-decoder architectures were then described. The transformer operates on sets of high-dimensional embeddings. It has a low computational complexity per layer and much of the computation can be performed in parallel, using the matrix form. Since every input embedding interacts with every other, it can describe long-range dependencies in text. Ultimately though, the computation scales quadratically with the sequence length; one approach to reducing the complexity is to sparsify the interaction matrix.

One of the advantages of transformers is that they can be trained with extremely large unlabeled datasets. This is the first example of *unsupervised learning* (learning without labels) in this book. The encoder model trains a text representation that can be used for other tasks by predicting missing tokens. The decoder model builds an autoregressive probability model over the input tokens. This is the first example of a *generative model* in this book; sampling from generative models creates new data examples and examples

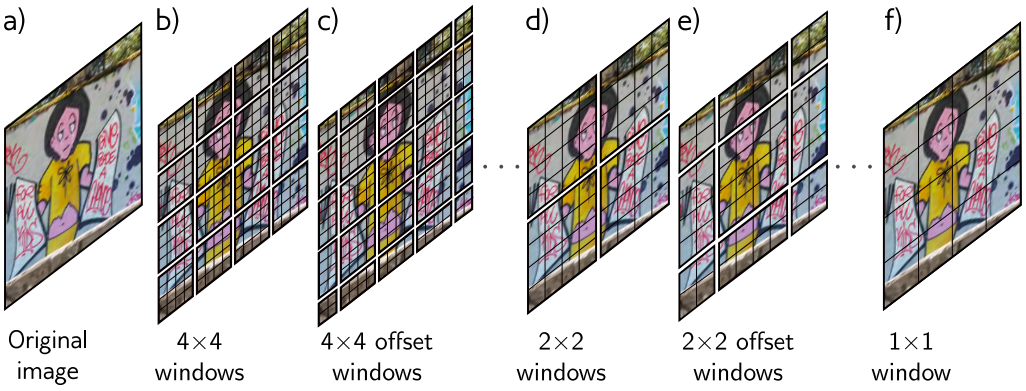

Figure 12.18 Shifted window (SWIN) transformer (Liu et al., 2021c). a) Original image. b) The SWIN transformer breaks the image into a grid of windows and each of these windows into a sub-grid of patches. The transformer network applies self-attention to the patches within each window independently. c) Each alternate layer shifts the windows so that the subsets of patches that interact with one another change and information can propagate across the whole image. d) After several layers, the 2×2 blocks of patch representations are concatenated so the effective patch (and window) size increase. e) Alternate layers use shifted windows at this new lower resolution. f) Eventually, the resolution is such that there is just a single window and the patches span the entire image.

of text and image generation were provided.

The next chapter considers networks for processing graph data. These have close connections with transformers in that they involve a series of network layers in which the nodes of the graph interact with each other. Chapters 14–18, return to unsupervised learning and generative models.

Notes

Natural language processing: Transformers were developed for natural language processing (NLP) tasks. This is an enormous area that deals with text analysis, categorization, generation, and manipulation. Example tasks include part of speech tagging, translation, text classification, entity recognition (people, places, companies, etc.), text summarization, question answering, word sense disambiguation, and document clustering. NLP was originally tackled by rule-based methods that exploited the structure and statistics of grammar. See Manning & Schutze (1999) and Jurafsky & Martin (2000) for early approaches.

Recurrent neural networks: Prior to the introduction of transformers, many state-of-the-art NLP applications used *recurrent neural networks*, or *RNNs* for short (figure 12.19). The term "recurrent" was introduced by Rumelhart et al. (1985), but the main idea dates to at least

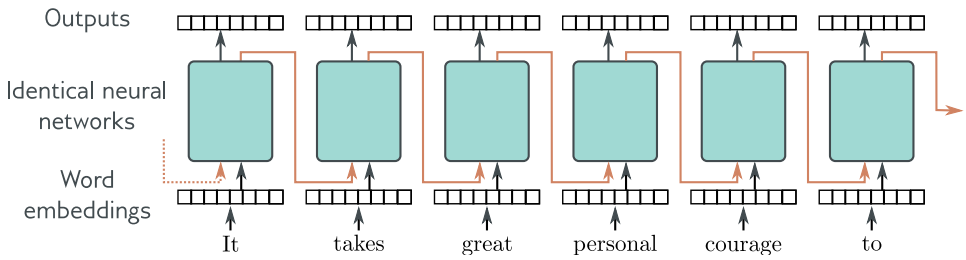

Figure 12.19 Recurrent neural networks. The word embeddings are passed sequentially through a series of identical neural networks. Each network has two outputs; one is the output embedding and the other (orange arrows) feeds back into the next neural network, along with the next word embedding. Each output embedding contains information about the word itself and its context in the preceding sentence fragment. In principle, the final output contains information about the entire sentence and could be used to support classification tasks in a similar way to the <cls> token in a transformer encoder model. However, the network sometimes gradually forgets about tokens that are further back in time.

Minsky & Papert (1969). RNNs ingest a sequence of inputs (words in NLP) one at a time. At each step, the network receives both the new input and a hidden representation computed from the previous time step (the recurrent connection). The final output contains information about the whole input. This representation can then be used to support NLP tasks like classification or translation. They have also been used in a decoding context in which generated tokens are fed back into the model to form the next input to the sequence. For example, the PixelRNN (Van Oord et al., 2016) used RNNs to build an autoregressive model of images.

From RNNs to transformers: One of the problems with RNNs is that they can forget information that is further back in the sequence. More sophisticated versions of this architecture such as *long short-term memory networks* or *LSTMs* (Hochreiter & Schmidhuber, 1997) and *gated recurrent units* or *GRUs* (Cho et al., 2014; Chung et al., 2014) partially addressed this problem. However, in machine translation, the idea emerged that all of the intermediate representations in the RNN could be exploited to produce the output sentence. Moreover, certain output words should *attend* more to certain input words, according to their relation (Bahdanau et al., 2015). This ultimately led to dispensing with the recurrent structure altogether and replacing it with the encoder-decoder transformer (Vaswani et al., 2017). Here input tokens attend to one another (self-attention), output tokens attend to those earlier in the sequence (masked self-attention), and output tokens also attend to the input tokens (cross-attention). A formal algorithmic description of the transformer can be found in Phuong & Hutter (2022), and a survey of work can be found in Lin et al. (2022). The literature should be approached with caution, as many enhancements to transformers do not make meaningful performance improvements when carefully assessed in controlled experiments (Narang et al., 2021).

Applications: Models based on self-attention and/or the transformer architecture have been applied to text sequences (Vaswani et al., 2017), image patches (Dosovitskiy et al., 2021), protein sequences (Rives et al., 2021), graphs (Veličković et al., 2019), database schema (Xu et al., 2021b), speech (Wang et al., 2020c), mathematical integration which can be formulated as a translation problem (Lample & Charton, 2020), and time series (Wu et al., 2020b). However,

their most celebrated successes have been in building language models, and more recently as a replacement for convolutional networks in computer vision.

Language models: The work of Vaswani et al. (2017) targeted translation tasks, but transformers are now more usually used to build either pure encoder or pure decoder models, the most famous of which are BERT (Devlin et al., 2019) and GPT2/GPT3 (Radford et al., 2019; Brown et al., 2020), respectively. These models are usually tested against benchmarks like GLUE (Wang et al., 2019b), which includes the SQuAD question-answering task (Rajpurkar et al., 2016) described in section 12.6.2, SuperGLUE (Wang et al., 2019a) and BIG-bench (Srivastava et al., 2022), which combine many NLP tasks to create an aggregate score for measuring language ability. Decoder models are generally not fine-tuned for these tasks but can perform well anyway when given a few examples of questions and answers and asked to complete the text from the next question. This is referred to as *few-shot learning* (Brown et al., 2020).

Since GPT3, many decoder language models have been released with steady improvement in few-shot results. These include GLaM (Du et al., 2022), Gopher (Rae et al., 2021), Chinchilla (Hoffmann et al., 2023), Megatron-Turing NLG (Smith et al., 2022), and LaMDa (Thoppilan et al., 2022). Most of the performance improvement is attributable to increased model size, using sparsely activated modules, and exploiting larger datasets. At the time of writing, the most recent model is PaLM (Chowdhery et al., 2022), which has 540 billion parameters and was trained on 780 billion tokens across 6144 processors. It is interesting to note that since text is highly compressible, this model has more than enough capacity to memorize the entire training dataset. This is true for many language models. Many bold statements have been made about how large language models exceed human performance. This is probably true for some tasks, but such statements should be treated with caution (see Ribeiro et al.,2021; McCoy et al., 2019; Bowman & Dahl, 2021; and Dehghani et al., 2021).

These models have considerable knowledge about the world. For example, in the example in section 12.7.4, the model knows key facts about deep learning, including that it is a type of machine learning, and that it has associated algorithms and applications. Indeed, one such model has been mistakenly identified as being sentient (Clark, 2022). However, there are persuasive arguments that the degree of "understanding" that this type of model can ever have is limited (Bender & Koller, 2020).

Tokenizers: Schuster & Nakajima (2012) and Sennrich et al. (2015) introduced *WordPiece* and *byte pair encoding* (*BPE*) respectively. Both of these methods greedily merge pairs of tokens based on their frequency of adjacency (figure 12.8), with the main difference being in how the initial tokens are chosen. For example, in BPE, the initial tokens are characters or punctuation with a special token to denote whitespace. The merges cannot occur over the whitespace. As the algorithm proceeds, new tokens are formed by combining characters recursively so that subword and word tokens emerge. The unigram language model (Kudo, 2018) generates several possible candidate merges and chooses the best one based on the likelihood in a language model. Provilkov et al. (2020) develop BPE dropout, which generates the candidates more efficiently by introducing randomness into the process of counting frequencies. Versions of both byte pair encoding and the unigram language model are included in the SentencePiece library (Kudo & Richardson, 2018), which works directly on Unicode characters, and so can work with any language. He et al. (2020) introduce a method that treats the sub-word segmentation as a latent variable that should be marginalized out for learning and inference.

Decoding algorithms: Transformer decoder models take a body of text and return a probability over the next token. This is then added to the preceding text and the model is run again. The process of choosing tokens from these probability distributions is known as *decoding*. Näive ways to do this would just be to either (i) greedily choose the most likely token or (ii) choose a token at random according to the distribution. However, neither of these methods works well in practice. In the former case, the results may be very generic, and the latter case may lead

to degraded quality outputs (Holtzman et al., 2020). This is partly because during training the model was only exposed to sequences of ground truth tokens (known as *teacher forcing*) but sees its own output when deployed.

It is not computationally feasible to try every combination of tokens in the output sequence, but it is possible to maintain a fixed number of parallel hypotheses and choose the most likely overall sequence. This is known as *beam search*. Beam search tends to produce many similar hypotheses and has been modified to investigate more diverse sequences (Vijayakumar et al., 2016; Kulikov et al., 2018). One possible problem with random sampling is that there is a very long tail of unlikely following words that collectively have a significant probability. This has led to the development of *top-K sampling*, in which tokens are sampled from only the K most likely hypotheses (Fan et al., 2018). This still sometimes allows unreasonable token choices when there are only a few high-probability choices. To resolve this problem Holtzman et al. (2020) proposed *nucleus sampling*, in which tokens are sampled from a fixed proportion of the total probability mass. These issues are discussed in more depth by El Asri & Prince (2020).

Types of attention: Scaled dot-product attention (Vaswani et al., 2017) is just one of a family of attention mechanisms that includes additive attention (Bahdanau et al., 2015), multiplicative attention (Luong et al., 2015), key-value attention (Daniluk et al., 2017), and memory-compressed attention (Liu et al., 2019b). Other work (Zhai et al., 2021) has constructed "attention-free" transformers, in which the tokens interact in a way that does not have quadratic complexity. Multi-head attention was also introduced by Vaswani et al. (2017). Interestingly, it appears that most of the heads can be pruned after training without critically affecting the performance (Voita et al., 2019); it has been suggested that their role is to guard against bad initializations. Hu et al. (2018b) propose squeeze-and-excitation networks which are an attention-like mechanism which re-weights the channels in a convoluational layer based on globally computed features.

Relationship of self-attention to other models: The self-attention computation has close connections to other models. First, it is a case of a hypernetwork (Ha et al., 2017) in that it uses one part of the network to choose the weights of another part: the attention matrix forms the weights of a sparse network layer that maps the values to the outputs (figure 12.3). The *synthesizer* (Tay et al., 2021) simplifies this idea by simply using a neural network to create each row of the attention matrix from the corresponding input. Even though the input tokens no longer interact with each other to create the attention weights, this works surprisingly well. Wu et al. (2019) present a similar system that produces an attention matrix with a convolutional structure, so the tokens attend to their neighbors. The gated multi-layer perceptron (Wu et al., 2019) computes a matrix that pointwise multiplies the values, and so modifies them without mixing them. Transformers are also closely related to *fast weight memory systems*, which were the intellectual forerunners of hypernetworks (Schlag et al., 2021).

Self-attention can also be thought of as a routing mechanism (figure 12.1) and from this viewpoint, there is a connection to capsule networks (Sabour et al., 2017). These capture hierarchical relations in images so lower network levels might detect facial parts (noses, mouths), which are then combined (routed) in higher level capsules that represent a face. However, capsule networks use *routing by agreement*. In self-attention, the inputs compete with each other for how much they contribute to a given output (via the softmax operation). In capsule networks, the outputs of the layer compete with each other for inputs from earlier layers. Once we consider self-attention as a routing network, we can question whether it is necessary to make this routing dynamic (i.e., dependent on the data). The random synthesizer (Tay et al., 2021) removed the dependence of the attention matrix on the inputs entirely and either used predetermined random values or learned values. This performed surprisingly well across a variety of tasks.

Multi-head self-attention also has close connections to graph neural networks (see chapter 13), convolution (Cordonnier et al., 2020), recurrent neural networks (Choromanski et al., 2020),

and memory retrieval in Hopfield networks (Ramsauer et al., 2021). For more information on the relationships between transformers and other models, consult Prince (2021a).

Position encoding: The original transformer paper (Vaswani et al., 2017) experimented with pre-defining the position embedding matrix $\boldsymbol{\Pi}$, and learning the position embedding $\boldsymbol{\Pi}$. It might seem odd to *add* the position embeddings to the $D \times N$ data matrix $\mathbf{X}$ rather than concatenate them. However, since the data dimension D is usually much greater than the number of tokens N, the position embedding lies in a subspace. The word embeddings in $\mathbf{X}$ are learned, and so it's possible in theory for the system to keep the two components in orthogonal subspaces and retrieve the position embeddings as required. The predefined embeddings chosen by Vaswani et al. (2017) were a family of sinusoidal components that had two attractive properties: (i) the relative position of two embeddings is easy to recover using a linear operation and (ii) their dot product generally decreased as the distance between positions increased (see Prince, 2021a, for more details). Many systems such as GPT3 and BERT used learned embedding matrices. Wang et al. (2020a) examined the cosine similarities of the learned position embeddings in these models and showed that they generally decline with relative distance, although they also have a periodic component.

Much subsequent work has modified just the attention matrix, so that in the scaled dot product self-attention equation:

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V} \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{D_q}}\right] \tag{12.16}$$

only the keys and values contain position information:

$$\begin{aligned}\mathbf{V} &= \boldsymbol{\beta}_v\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{v}\mathbf{X} \\ \mathbf{Q} &= \boldsymbol{\beta}_q\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{q}(\mathbf{X}+\boldsymbol{\Pi}) \\ \mathbf{K} &= \boldsymbol{\beta}_k\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{k}(\mathbf{X}+\boldsymbol{\Pi}).\end{aligned} \tag{12.17}$$

This has led to the idea of multiplying out the quadratic term in the numerator of equation 12.16 and retaining only some of the terms. For example, Ke et al. (2021) decouple or *untie* the content and position information by retaining only the content-content and position-position terms and using different projection matrices $\boldsymbol{\Omega}_\bullet$ for each.

Another modification is to directly inject information about the relative position. This is more important than absolute position since a batch of text can start at an arbitrary place in a document. Shaw et al. (2018), Raffel et al. (2020), and Huang et al. (2020b) all developed systems where a single term was learned for each relative position offset, and the attention matrix was modified in various ways using these *relative position embeddings*. Wei et al. (2019) investigated relative position embeddings based on predefined sinusoidal embeddings rather than learned values. DeBERTa (He et al., 2021) combines all of these ideas; they retain only a subset of terms from the quadratic expansion, apply different projection matrices to them, and use relative position embeddings. Other work has explored sinusoidal embeddings that encode absolute and relative position information in more complex ways (Su et al., 2021).

Wang et al. (2020a) compare the performance of transformers in BERT with different position embeddings. They found that relative position embeddings perform better than absolute position embeddings, but there was not much difference between using sinusoidal and learned embeddings. A survey of position embeddings can be found in Dufter et al. (2021).

Extending transformers to longer sequences: The complexity of the self-attention mechanism increases quadratically with the sequence length. Some tasks like summarization or

question answering may require long inputs, and so this quadratic dependence limits performance. Three lines of work have attempted to address this problem. The first decreases the size of the attention matrix, the second makes the attention sparse, and the third modifies the attention mechanism to make it more efficient.

To decrease the size of the attention matrix, Liu et al. (2018b) introduced *memory compressed attention*. This applies strided convolution to the keys and values, which reduces the number of positions in a very similar way to a downsampling operation in a convolutional network. One way to think about this is that attention is applied between weighted combinations of neighboring positions, where the weights are learned. Along similar lines, Wang et al. (2020b) observed that the quantities in the attention mechanism are often low rank in practice and developed the *LinFormer*, which projects the keys and values onto a smaller subspace before computing the attention matrix.

To make attention sparse, Liu et al. (2018b) proposed *local attention*, in which neighboring blocks of tokens only attend to one another. This creates a block diagonal interaction matrix (see figure 12.15). Obviously this means that information cannot pass from block to block, and so such layers are typically alternated with full attention. Along the same lines, GPT3 (Brown et al., 2020) uses a convolutional interaction matrix and alternates this with full attention. Child et al. (2019) and Beltagy et al. (2020) experimented with various interaction matrices, including using convolutional structures with different dilation rates but allowing some queries to interact with every other key. Ainslie et al. (2020) introduced the *extended transformer construction* (figure 12.15h), which uses a set of global embeddings that interact with every other token. This can only be done in the encoder version, or these implicitly allow the system to "look ahead". When combined with relative position encoding, this scheme requires special encodings for mapping to, from, and between these global embeddings. *BigBird* (Ainslie et al., 2020) combined global embeddings and a convolutional structure with a random sampling of possible connections. Other work has investigated learning the sparsity pattern of the attention matrix (Roy et al., 2021; Kitaev et al., 2020; Tay et al., 2020).

Finally, it has been noted that the terms in the numerator and denominator of the softmax operation that computes attention have the form $\exp[\mathbf{k}^T\mathbf{q}]$. This can be treated as a kernel function and as such can be expressed as the dot product $\mathbf{g}[\mathbf{k}]^T\mathbf{g}[\mathbf{q}]$ where $\mathbf{g}[\bullet]$ is a nonlinear transformation. This formulation decouples the queries and keys and makes the attention computation more efficient. Unfortunately, to replicate the form of the exponential terms, the transformation $\mathrm{g}[\bullet]$ must map the inputs to the infinite space. The linear transformer (Katharopoulos et al., 2020) recognizes this and replaces the exponential term with a different measure of similarity. The *Performer* (Choromanski et al., 2020) approximates this infinite mapping with a finite-dimensional one.

Problem 12.11

More details about extending transformers to longer sequences can be found in Tay et al. (2023) and Prince (2021a).

Training transformers: Training transformers is challenging and requires both learning rate warm-up (Goyal et al., 2018) and Adam (Kingma & Ba, 2015). Indeed Xiong et al. (2020a) and Huang et al. (2020a) show experimentally that gradients vanish, and the Adam updates decrease in magnitude without learning rate warm-up. Several interacting factors cause these problems. The residual connections cause the gradients to explode (figure 11.6) and normalization layers are required to prevent this. Vaswani et al. (2017) used LayerNorm rather than BatchNorm because NLP statistics are highly variable between batches, although subsequent work has modified BatchNorm for transformers (Shen et al., 2020a). The positioning of the LayerNorm outside of the residual block causes gradients to shrink as they pass back through the network (Xiong et al., 2020a). In addition, the relative weight of the residual connections and main self-attention mechanism varies as we move through the network upon initialization (see figure 11.6c), and there is the additional complication that the gradients for the query and key parameter are much smaller than for the value parameters (Liu et al., 2020), and this necessitates the use of Adam. These factors interact in a complex way and make training unstable,

necessitating the use of learning rate warm-up.

There have been various attempts to make stabilize training including (i) a variation of FixUp called *TFixup* (Huang et al., 2020a) that allows the LayerNorm components to be removed, (ii) changing the position of the LayerNorm components in the network (Liu et al., 2020), and (iii) reweighting the two paths in the residual branches (Liu et al., 2020; Bachlechner et al., 2021). Xu et al. (2021b) introduce an initialization scheme called *DTFixup* that allows transformers to be trained with smaller datasets. A detailed discussion can be found in Prince (2021b).

Applications in vision: ImageGPT (Chen et al., 2020a) and the Vision Transformer (Dosovitskiy et al., 2021) were both early transformer architectures applied to images. Transformers have been used for image classification (Dosovitskiy et al., 2021; Touvron et al., 2021), object detection (Carion et al., 2020; Zhu et al., 2020b; Fang et al., 2021), semantic segmentation (Ye et al., 2019; Xie et al., 2021; Gu et al., 2022), super-resolution (Yang et al., 2020a), action recognition (Sun et al., 2019; Girdhar et al., 2019), image generation (Chen et al., 2021a; Nash et al., 2021), visual question answering (Su et al., 2019b; Tan & Bansal, 2019), inpainting (Wan et al., 2021; Zheng et al., 2021; Zhao et al., 2020b; Li et al., 2022), colorization (Kumar et al., 2021), and many other vision tasks. Surveys of transformers for vision can be found in Khan et al. (2022) and Liu et al. (2023b).

Transformers and convolutional networks: Transformers have been combined with convolutional neural networks to solve diverse computer vision tasks including image classification (Wu et al., 2020a), object detection (Hu et al., 2018a; Carion et al., 2020), video processing (Wang et al., 2018c; Sun et al., 2019), unsupervised object discovery (Locatello et al., 2020) and various text/vision tasks (Chen et al., 2020d; Lu et al., 2019; Li et al., 2019). Transformers can outperform convolutional networks for vision tasks but usually require large quantities of data to achieve superior performance. Often, they are pre-trained on the enormous JFT dataset (Sun et al., 2017). The transformer does not have the inductive bias of convolutional networks but it appears that by using gargantuan amounts of data they can surmount this disadvantage.

From pixels to video: Non-local networks (Wang et al., 2018c) were an early application of self-attention to image data. Transformers themselves were initially applied to pixels in local neighborhoods (Parmar et al., 2018; Hu et al., 2019; Parmar et al., 2019; Zhao et al., 2020a). ImageGPT (Chen et al., 2020a) scaled this to model all of the pixels in a (small) image. The Vision Transformer (ViT) (Dosovitskiy et al., 2021) used non-overlapping patches to analyze bigger images.

Since then, many multi-scale systems have been developed including the SWIN transformer (Liu et al., 2021c), SWINV2 (Liu et al., 2022), multi-scale transformers (MVIT) (Fan et al., 2021), and pyramid vision transformers (Wang et al., 2021). The Crossformer (Wang et al., 2022a) models interactions between spatial scales. Ali et al. (2021) introduced cross-covariance image transformers, in which the channels rather than spatial positions attend to one another, hence making the size of the attention matrix indifferent to the image size. The dual attention vision transformer was developed by Ding et al. (2022) and alternates between local spatial attention within sub-windows and spatially global attention between channels. Chu et al. (2021) similarly alternate between local attention within sub-windows and global attention by subsampling the spatial domain. Dong et al. (2022) adapt the ideas of figure 12.15, in which the interactions between elements are sparsified to the 2D image domain.

Transformers were subsequently adapted to video processing (Arnab et al., 2021; Bertasius et al., 2021; Liu et al., 2021c; Neimark et al., 2021; Patrick et al., 2021). A survey of transformers applied to video can be found in Selva et al. (2022).

Combining images and text: CLIP (Radford et al., 2021) learns a joint encoder for images and their captions using a contrastive pre-training task. The system ingests N images and

their captions and produces a matrix of compatibility between images and captions. The loss function encourages the correct pairs to have a high score and the incorrect pairs to have a low score. Ramesh et al. (2021) and Ramesh et al. (2022) train a diffusion decoder to invert the CLIP image encoder for text-conditional image generation (see chapter 18).

Problems

Problem 12.1 Consider a self-attention mechanism that processes N inputs of length D to produce N outputs of the same size. How many weights and biases are used to compute the values? How many attention weights $\mathrm{a}[\bullet, \bullet]$ will there be? How many weights and biases would there be if we build a fully connected network relating all DN inputs to all DN outputs?

Problem 12.2 Why might we want to ensure that the input to the self-attention mechanism is the same size as the output?

Problem 12.3 The self-attention mechanism is defined as:

$$\mathbf{X} = (\boldsymbol{\beta}_v \mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{v}\mathbf{X})^\mathbf{T}\mathbf{Softmax}[(\boldsymbol{\beta}_\mathbf{q}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{q}\mathbf{X})^\mathbf{T}(\boldsymbol{\beta}_\mathbf{v}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{k}\mathbf{X})]. \tag{12.18}$$

Find the derivatives of the output $\mathbf{X}$ with respect to the parameters $\boldsymbol{\beta}_v, \boldsymbol{\Omega}_v, \boldsymbol{\beta}_q, \boldsymbol{\Omega}_q, \boldsymbol{\beta}_k$ and $\boldsymbol{\Omega}_k$.

Problem 12.4 Show that the self-attention mechanism (equation 12.8) is invariant to a permutation $\mathbf{PX}$ of the data matrix $\mathbf{X}$.

Problem 12.5 Consider the softmax operation:

$$y_k = \mathrm{softmax}_k[\mathbf{z}] = \frac{\exp[z_k]}{\sum_{k'=1}^{K}\exp[z_{k'}]}, \tag{12.19}$$

in the case where there are 5 inputs with values: $z_1 = -3$, $z_2 = 1$, $z_3 = 1000$, $z_4 = 5$, $z_5 = -1$. Compute the 25 derivatives, $\partial y_k[\mathbf{z}]/\partial z_j$ for all $j, k \in \{1, 2, 3, 4, 5\}$. What do you conclude?

Problem 12.6 Why is implementation more efficient if the values, queries, and keys in each of the H heads each have dimension D/H where D is the original dimension of the data?

Problem 12.7 Write Python code to create sub-word tokens for the nursery rhyme in figure 12.8 using byte-pair encoding.

Problem 12.8 BERT was pre-trained using two tasks. The first task requires the system to predict missing (masked) words. The second task requires the system to classify pairs of sentences as being adjacent or not in the original text. Identify whether each of these tasks is generative or contrastive (see section 9.3.6). Why do you think they used two tasks? Propose two novel contrastive tasks that could be used to pre-train a language model.

Problem 12.9 One problem with vision transformers is that the computation expands quadratically with the number of patches. Devise two methods to reduce the amount of computation using the principles from figure 12.15.

Problem 12.10 Consider representing an image with a grid of 16×16 patches, each of which is represented by a patch embedding of length 512. Compare the amount of computation required in the DaViT transformer to perform attention (i) between the patches, using all of the channels, and (ii) between the channels, using all of the patches.

Problem 12.11 Attention values are normally computed as:

$$\begin{aligned} a[\mathbf{x}_n, \mathbf{x}_m] &= \text{softmax}_m\left[\mathbf{q}_m^T\mathbf{k}_n\right] \\ &= \frac{\exp\left[\mathbf{q}_m^T\mathbf{k}_n\right]}{\sum_{m'=1}^{N}\exp\left[\mathbf{q}_{m'}^T\mathbf{k}_n)\right]}. \end{aligned} \tag{12.20}$$

Consider replacing $\exp\left[\mathbf{q}_m^T\mathbf{k}_n\right]$ with the dot product $\mathbf{g}[\mathbf{q}_m]^T\mathbf{g}[\mathbf{k}_n]$ where $\mathbf{g}[\bullet]$ is a nonlinear transformation. Show how this makes the computation of the attention values more efficient.