Lecture Notes on Transformers

Chapter 12: Deep Learning

Based on Bishop & Bishop, "Deep Learning: Foundations and Concepts"

Abstract

These lecture notes provide a comprehensive introduction to transformer architectures, one of the most important developments in deep learning. We cover the fundamental attention mechanism, self-attention, multi-head attention, positional encoding, and the complete transformer layer architecture. The material follows a coherent progression from basic concepts to the full implementation.

Contents

1 Introduction to 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.

1.1 Key Characteristics

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.

1.2 Historical Context and Applications

Transformers were originally introduced in the context of natural language processing (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:

- **Vision transformers**: Often outperform CNNs in image processing tasks
- **Multimodal transformers**: Combine multiple types of data (text, images, audio, video)
- Among the most powerful deep learning models currently available

1.3 Advantages of Transformers

1. **Transfer learning**: 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**.
2. **Self-supervised learning**: 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.
3. **Scaling hypothesis**: 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.

4. **Parallel processing**: The transformer is especially well suited to massively parallel processing hardware such as graphical processing units (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).

2 Attention Mechanism

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, although it has much broader applicability.

2.1 Motivating Example: Word Disambiguation

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 1.

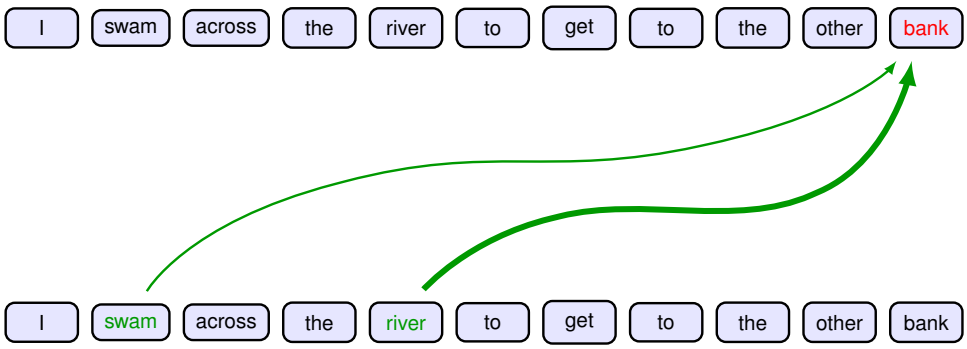

Figure 1: Schematic illustration of attention in which the interpretation of the word 'bank' is influenced by the words 'river' and 'swam', with the thickness of each line being indicative of the strength of its influence.

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.

2.2 Attention in Standard Neural Networks

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 2 shows the attention weights from a section of a transformer network trained on natural language.

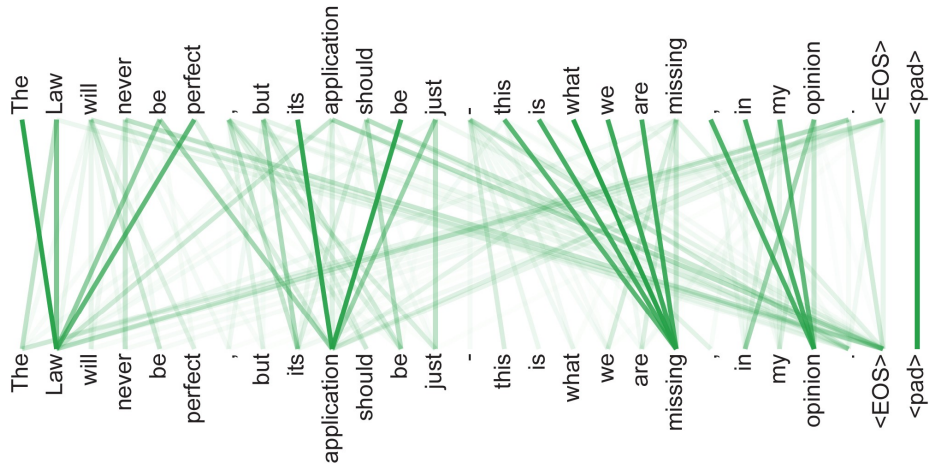

Figure 2: An example of learned attention weights showing the connections between words in a sentence.

2.3 Word Embeddings and Vector Representation

When we discuss natural language processing, we will see how word embeddings 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.

Transformers as Rich Embeddings

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".

2.4 Example: Protein Structure Prediction

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).

2.5 Transformer Processing

Data Representation

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.

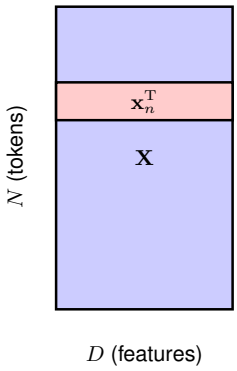

Figure 3: The structure of the data matrix $\mathbf{X}$, of dimension $N \times D$, in which row n represents the transposed data vector $\mathbf{x}_n^{\mathrm{T}}$.

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

2.5.1 Fundamental Transformer Operation

The fundamental building block of a transformer is a function that takes a data matrix as input and creates a transformed matrix $\tilde{\mathbf{X}}$ of the same dimensionality as the output. We can write this function in the form

$$\tilde{\mathbf{X}} = \mathrm{TransformerLayer}[\mathbf{X}]. \tag{1}$$

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.

2.6 Two-Stage Transformer Architecture

A single transformer layer itself comprises two stages:

1. **First stage (Attention)**: Implements the attention mechanism, mixes together the corresponding features from different token vectors across the columns of the data matrix.

2. **Second stage (Feed-forward)**: Acts on each row independently and transforms the features within each token vector.

We start by looking at the attention mechanism.

2.7 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}:

$$\mathbf{y}_n = \sum_{m=1}^{N} a_{nm}\mathbf{x}_m \tag{2}$$

where a_{nm} are called **attention weights**.

2.7.1 Constraints on 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 the 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:

$$a_{nm} \geqslant 0 \tag{3}$$

$$\sum_{m=1}^{N} a_{nm} = 1. \tag{4}$$

Note that we have a different set of coefficients for each output vector $\mathbf{y}_n$, and the constraints (3) and (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.

2.8 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.

2.8.1 Information Retrieval Analogy

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.

2.8.2 Applying the Analogy to Transformers

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 $\mathbf{x}_n$ as the query vector for output token $\mathbf{y}_m$, 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.

2.8.3 Measuring Similarity with Dot Product

One simple measure of similarity is to take their dot product $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m$. To impose the constraints (3) and (4), we can define the weighting coefficients a_{nm} by using the *softmax* function to transform the dot products:

$$a_{nm} = \frac{\exp(\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m)}{\sum_{m"=1}^{N}\exp(\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_{m'})}. \tag{5}$$

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.

2.8.4 Summary of Self-Attention

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 (2) in which the weight

a_{nm} applied to input vector $\mathbf{x}_m$ is given by the softmax function (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 $m = 1, \ldots, N$.

2.8.5 Matrix Notation

We can write (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

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{X}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X} \tag{6}$$

where 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*.

2.9 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

$$\tilde{\mathbf{x}} = \mathbf{X}\mathbf{U} \tag{7}$$

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

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{X}\mathbf{U}\mathbf{U}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X}\mathbf{U}. \tag{8}$$

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

$$\mathbf{X}\mathbf{U}\mathbf{U}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}} \tag{9}$$

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 (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.

2.9.1 Separate Query, Key, and Value Matrices

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

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(q)} \tag{10}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}^{(k)} \tag{11}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}^{(v)} \tag{12}$$

where the weight matrices $\mathbf{W}^{(q)}$, $\mathbf{W}^{(k)}$, and $\mathbf{W}^{(v)}$ represent parameters that will be learned during the training of the final transformer architecture.

Here the matrix $\mathbf{W}^{(k)}$ has dimensionality $D \times D_k$ where D_k is the length of the key vector. The matrix $\mathbf{W}^{(q)}$ must have the same dimensionality $D \times D_k$ as $\mathbf{W}^{(k)}$ so that we can form dot products between the query and the key vectors. A typical choice is $D_k = D$. Similarly, $\mathbf{W}^{(v)}$ is a matrix of size $D \times D_v$, where D_v governs the dimensionality of the output vectors. If we set $D_v = D$, so that the output representation has the same dimensionality as the input, this will facilitate the inclusion of residual connections. Also, multiple transformer layers can be stacked on top of each other if each layer has the same dimensionality. We can then generalize (6) to give

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{Q}\mathbf{K}^{\mathrm{T}}\right]\mathbf{V} \tag{13}$$

where $\mathbf{Q}\mathbf{K}^{\mathrm{T}}$ has dimension $N \times N$, and the matrix $\mathbf{Y}$ has dimension $N \times D_v$. The calculation of the matrix $\mathbf{Q}\mathbf{K}^{\mathrm{T}}$ is illustrated in Figure 4, whereas the evaluation of the matrix $\mathbf{Y}$ is illustrated in Figure 5.

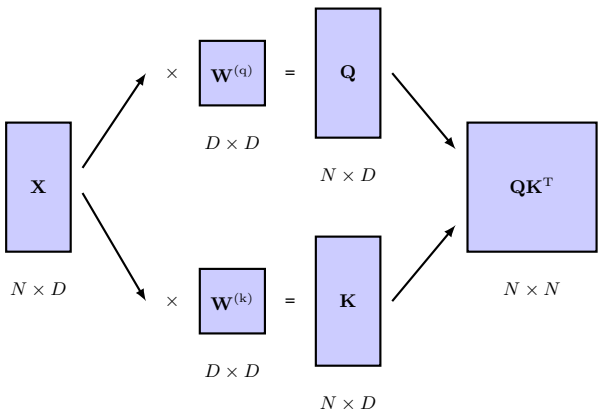

Figure 4: Illustration of the evaluation of the matrix $\mathbf{Q}\mathbf{K}^{\mathrm{T}}$, which determines the attention coefficients in a transformer. The input $\mathbf{X}$ is separately transformed using (10) and (11) to give the query matrix $\mathbf{Q}$ and key matrix $\mathbf{K}$, respectively, which are then multiplied together.

2.9.2 Including Bias Parameters

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 using homogenous matrices. From now on we will treat the bias parameters as implicit to avoid cluttering the notation.

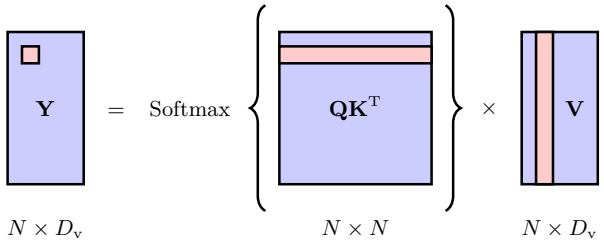

Figure 5: Illustration of the evaluation of the output from an attention layer given the query, key, and value matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, respectively. The entry in the output matrix $\mathbf{Y}$ is obtained from the dot product of the highlighted row and column of the Softmax[$\mathbf{QK}^{\mathrm{T}}$] and $\mathbf{V}$ matrices, respectively.

2.9.3 Comparison with Standard Neural Networks

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 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.

2.10 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 rescale 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

$$\mathbf{Y} = \mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \equiv \mathrm{Softmax}\left[\frac{\mathbf{QK}^{\mathrm{T}}}{\sqrt{D_k}}\right]\mathbf{V}. \tag{14}$$

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 6 and in Algorithm 1.

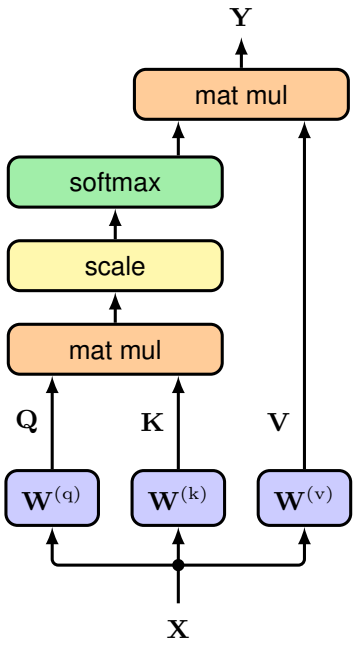

Figure 6: Information flow in a scaled dot-product self-attention neural network layer. Here 'mat mul' denotes matrix multiplication, and 'scale' refers to the normalization of the argument to the softmax using $\sqrt{D_k}$. This structure constitutes a single attention 'head'.

Algorithm 1 Scaled Dot-Product Self-Attention

Require: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D}$
Require: Query weight matrix $\mathbf{W}^{(q)} \in \mathbb{R}^{D \times D_k}$
Require: Key weight matrix $\mathbf{W}^{(k)} \in \mathbb{R}^{D \times D_k}$
Require: Value weight matrix $\mathbf{W}^{(v)} \in \mathbb{R}^{D \times D_v}$
Ensure: Output matrix $\mathbf{Y} \in \mathbb{R}^{N \times D_v}$

// Compute query, key, and value matrices
$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(q)}$
$\mathbf{K} = \mathbf{X}\mathbf{W}^{(k)}$
$\mathbf{V} = \mathbf{X}\mathbf{W}^{(v)}$

// Compute scaled attention
$\mathbf{Y} = \text{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{D_k}}\right]\mathbf{V}$
return $\mathbf{Y}$

2.11 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 are typically many different kinds of relationships or patterns in the data that need to be captured.

2.11.1 Motivation for Multiple Heads

For example, in natural language processing, we might want one head to focus on syntactic relationships (such as subject-verb agreement) while another head captures semantic relationships (such as synonyms or antonyms). By having multiple attention heads operating in parallel, the model can simultaneously attend to different aspects of the input.

2.11.2 Multi-Head Architecture

In multi-head attention, we run H separate attention mechanisms in parallel, each with its own set of query, key, and value weight matrices. Each head h computes:

$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(q)} \tag{15}$$

$$\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(k)} \tag{16}$$

$$\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(v)} \tag{17}$$

$$\mathbf{H}_h = \text{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h) \tag{18}$$

The outputs from all heads are then concatenated and linearly projected back to the original dimensionality:

$$\mathbf{Y}(\mathbf{X}) = \text{Concat}[\mathbf{H}_1, \mathbf{H}_2, \ldots, \mathbf{H}_H]\mathbf{W}^{(o)} \tag{19}$$

where $\mathbf{W}^{(o)} \in \mathbb{R}^{HD_v \times D}$ is an output weight matrix.

Typically, the dimensionality D_v of each head's output is 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 2 and the information flow in a multi-head attention layer is illustrated in Figure 8.

Algorithm 2 Multi-Head Attention

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

```
// compute self-attention for each head (Algorithm 1)
for h = 1, ..., H do
    Q_h = X W_h^(q),   K_h = X W_h^(k),   V_h = X W_h^(v)
    H_h = Attention(Q_h, K_h, V_h) // H_h ∈ R^{N × D_v}
end for
H = Concat[H_1, ..., H_N] // concatenate heads
return Y(X) = H W^(o)
```

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 $\mathbf{W}^{(v)}$ matrix for each head and the output matrix $\mathbf{W}^{(o)}$. Removing this redundancy allows a multi-head self-attention layer to be written as a sum over contributions from each of the heads separately.

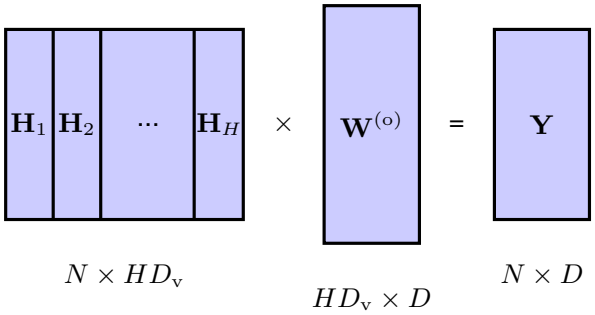

Figure 7: Network architecture for multi-head attention. Each head comprises the structure shown in Figure 6, and has its own key, query, and value parameters. The outputs of the heads are concatenated and then linearly projected back to the input data dimensionality.

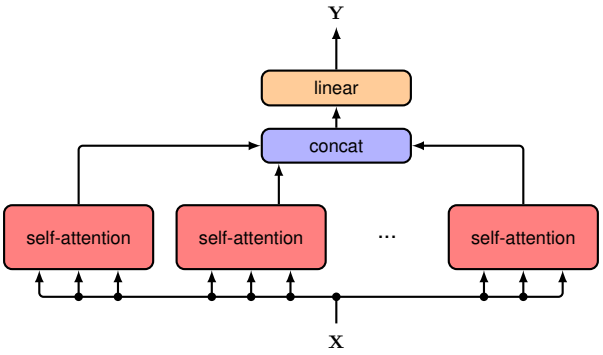

Figure 8: Information flow in a multi-head attention layer. The associated computation, given by Algorithm 2, is illustrated in Figure 7.

2.12 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 efficiency, we can introduce residual connections that bypass the multi-head structure.

2.12.1 Residual Connections and Layer Normalization

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

$$\mathbf{Z} = \mathrm{LayerNorm}\left[\mathbf{Y}(\mathbf{X}) + \mathbf{X}\right] \tag{20}$$

where $\mathbf{Y}$ is defined by (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

$$\mathbf{Z} = \mathbf{Y}(\mathbf{X}') + \mathbf{X}, \quad \text{where} \quad \mathbf{X}' = \mathrm{LayerNorm}[\mathbf{X}]. \tag{21}$$

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. Non-linearity 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 capability of the attention layer.

2.12.2 Feed-Forward Network (MLP)

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 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 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

$$\tilde{\mathbf{X}} = \text{LayerNorm}\left[\text{MLP}[\mathbf{Z}] + \mathbf{Z}\right]. \tag{22}$$

This leads to an overall architecture for a transformer layer shown in Figure 9 and summarized in Algorithm 3. Again, we can use a pre-norm instead, in which case the final output is given by

$$\tilde{\mathbf{X}} = \text{MLP}(\mathbf{Z}') + \mathbf{Z}, \quad \text{where} \quad \mathbf{Z}' = \text{LayerNorm}[\mathbf{Z}]. \tag{23}$$

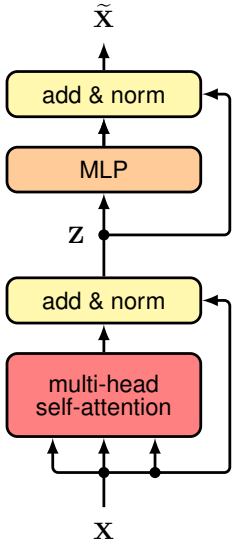

Figure 9: One layer of the transformer architecture that implements the transformation (1). Here 'MLP' stands for multilayer perceptron, while 'add & norm' denotes a residual connection followed by layer normalization.

Algorithm 3 Transformer Layer

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

1:
2: $\mathbf{Z} = \text{LayerNorm}\,[\mathbf{Y}(\mathbf{X}) + \mathbf{X}]$ // $\mathbf{Y}(\mathbf{X})$ from Algorithm 2
3: $\tilde{\mathbf{X}} = \text{LayerNorm}\,[\text{MLP}[\mathbf{Z}] + \mathbf{Z}]$ // shared neural network
4: **return** $\tilde{\mathbf{X}}$

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.

2.13 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 $\mathbf{W}^{(q)}$, $\mathbf{W}^{(k)}$, and $\mathbf{W}^{(v)}$ are shared across input tokens, and therefore the number of independent parameters is $\mathcal{O}(D^2)$, assuming $D_k \simeq D_v \simeq D$. Since there are N input tokens, the number of computational steps in evaluating the dot products in a self-attention layer is $\mathcal{O}(N^2D)$.

2.14 Positional Encoding

In the transformer architecture, the matrices $\mathbf{W}^{(q)}$, $\mathbf{W}^{(k)}$, and $\mathbf{W}^{(v)}$ 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 $\tilde{\mathbf{X}}$. 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.

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 instead 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 $\mathbf{x}_n$.

2.14.1 Requirements for Positional Encoding

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

$$\tilde{\mathbf{x}}_n = \mathbf{x}_n + \mathbf{r}_n. \tag{24}$$

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 token vector 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.

2.14.2 Constructing Position Embeddings

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.

2.14.3 Sinusoidal Positional Encoding

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

$$r_{ni} = \begin{cases} \sin\left(\frac{n}{L^{i/D}}\right), & \text{if } i \text{ is even,} \\ \cos\left(\frac{n}{L^{(i-1)/D}}\right), & \text{if } i \text{ is odd.} \end{cases} \tag{25}$$

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 10.

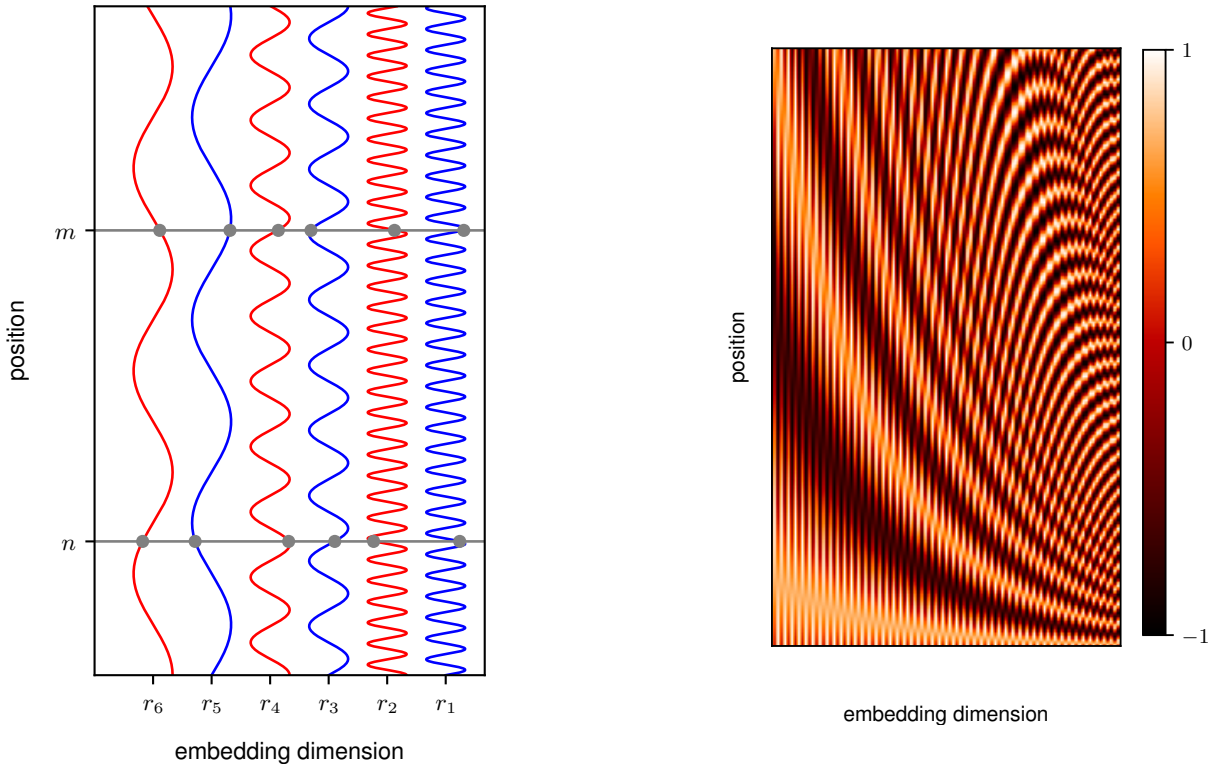

Figure 10: Illustrations of the functions defined by (25) and used to construct position-encoding vectors. (a) A plot in which the horizontal axis shows the different components of the embedding vector $\mathbf{r}$ whereas the vertical axis shows the position in the sequence. The values of the vector 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 (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:

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

For the encoding given by (25), however, the vector elements are continuous variables rather than binary. A plot of the position-encoding vectors is shown in Figure 10(b).

One nice property of the sinusoidal representation given by (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.

2.14.4 Learned Positional Embeddings

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.

3 Summary

In this chapter, we have developed a comprehensive understanding of transformer architectures:

- **Attention Mechanism**: The fundamental concept allowing networks to weight inputs based on their relevance to specific outputs
- **Self-Attention**: Using the same sequence for queries, keys, and values
- **Scaled Dot-Product Attention**: Normalizing by $\sqrt{D_k}$ to prevent gradient vanishing
- **Multi-Head Attention**: Parallel attention mechanisms to capture different types of relationships
- **Transformer Layers**: Combining multi-head attention with feed-forward networks and residual connections
- **Positional Encoding**: Injecting sequence order information while maintaining permutation equivariance

These components work together to create powerful models capable of processing sequential data, with applications ranging from natural language processing to computer vision and protein structure prediction.

4 Exercises

Exercise 1

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

$$a_{nm} \geqslant 0 \tag{26}$$

$$\sum_m a_{nm} = 1. \tag{27}$$

By using a Lagrange multiplier show that the coefficients must also satisfy

$$a_{nm} \leqslant 1 \quad \text{for} \quad n = 1, \ldots, N. \tag{28}$$

Exercise 2

Verify that the softmax function (5) satisfies the constraints (3) and (4) for any values of the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$.

Exercise 3

Consider the input vectors $\mathbf{x}_n$ in the simple transformation defined by (2), in which the weighting coefficients a_{nm} are defined by (5). Show that if all the input vectors are orthogonal, so that $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m = 0$ for $n \neq m$, then the output vectors will simply be equal to the input vectors so that $\mathbf{y}_n = \mathbf{x}_n$ for $n = 1, \ldots, N$.

Exercise 4

Consider two independent random vectors $\mathbf{a}$ and $\mathbf{b}$ each of dimension D and each being drawn from a Gaussian distribution with zero mean and unit variance $\mathcal{N}(\cdot|\mathbf{0}, \mathbf{I})$. Show that the expected value of $(\mathbf{a}^{\mathrm{T}}\mathbf{b})^2$ is given by D.

Exercise 5

Show that multi-head attention defined by (19) can be rewritten in the form

$$\mathbf{Y} = \sum_{h=1}^{H} \mathbf{H}_h \mathbf{X} \bar{\mathbf{W}}^{(h)} \tag{29}$$

where $\mathbf{H}_h$ is given by (12.15) and we have defined

$$\bar{\mathbf{W}}^{(h)} = \mathbf{W}_h^{(v)} \mathbf{W}_h^{(o)}. \tag{30}$$

Here we have partitioned the matrix $\mathbf{W}^{(o)}$ horizontally into sub-matrices denoted $\mathbf{W}_h^{(o)}$ each of dimension $D_v \times D$, corresponding to the vertical segments of the concatenated attention matrix. Since D_v is typically smaller than D, for example $D_v = D/H$ is a common choice, this combined matrix is rank deficient. Therefore, using a fully flexible matrix to replace $\mathbf{W}_h^{(v)}\mathbf{W}_h^{(o)}$ would not be equivalent to the original formulation given in the text.

See Figure 7 for reference.

Exercise 6

Express the self-attention function (14) as a fully connected network in the form of a matrix that maps the full input sequence of concatenated word vectors into an output vector of the same dimension. Note that such a matrix would have $\mathcal{O}(N^2D^2)$ parameters. Show that the self-attention network corresponds to a sparse version of this matrix with parameter sharing. Draw a sketch showing the structure of this matrix, indicating which blocks of parameters are shared and which blocks have all elements equal to zero.

Exercise 7

Show that if we omit the positional encoding of input vectors then the outputs of a multi-head attention layer defined by (19) are equivariant with respect to a reordering of the input sequence.

Exercise 8

Consider two D-dimensional unit vectors $\mathbf{a}$ and $\mathbf{b}$, satisfying $\|\mathbf{a}\| = 1$ and $\|\mathbf{b}\| = 1$, drawn from a random distribution. Assume that the distribution is symmetrical around the origin, i.e., it depends only on the distance from the origin and not the direction. Show that for large values of D the magnitude of the cosine of the angle between these vectors is close to zero and hence that these random vectors are nearly orthogonal in a high-dimensional space. To do this, consider an orthonormal basis set $\{\mathbf{u}_i\}$ where $\mathbf{u}_i^{\mathrm{T}}\mathbf{u}_j = \delta_{ij}$ and express $\mathbf{a}$ and $\mathbf{b}$ as expansions in this basis.

Exercise 9

Consider a position encoding in which the input token vector $\mathbf{x}$ is concatenated with a position-encoding vector $\mathbf{e}$. Show that when this concatenated vector undergoes a general linear transformation by multiplication using a matrix, the result can be expressed as the sum of a linearly transformed input and a linearly transformed position vector.

Exercise 10

Show that the positional encoding defined by (25) has the property that, for a fixed offset k, the encoding at position $n+k$ can be represented as a linear combination of the encoding at position n with coefficients that depend only on k and not on n. To do this make use of the following trigonometric identities:

$$\cos(A+B) = \cos A \cos B - \sin A \sin B \tag{12.44}$$

$$\sin(A+B) = \cos A \sin B + \sin A \cos B. \tag{12.45}$$

Show that if the encoding is based purely on sine functions, without cosine functions, then this property no longer holds.

5 References

- Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate.

- Vaswani, A., et al. (2017). Attention is all you need.
- Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization.
- Vig, J., et al. (2020). BERTology meets biology: Interpreting attention in protein language models.
- Bubeck, S., et al. (2023). Sparks of artificial general intelligence: Early experiments with GPT-4.
- Bishop, C. M., and Bishop, H. (2024). Deep Learning: Foundations and Concepts. Springer Nature Switzerland AG.