Lecture 6: Self-Attention in Transformers

Outline

In Lecture 5, we learned about **recurrent neural networks (RNNs)** and their applications in **natural language processing (NLP)** through a sentiment analysis project. However, a new architecture has recently emerged that has been shown to outperform the RNN-based **sequence-to-sequence** (**seq2seq**) models in several NLP tasks. This is the so-called **transformer** architecture.

Transformers have revolutionized natural language processing and have been at the forefront of many impressive applications ranging from automated language translation and modeling fundamental properties of protein sequences to creating an AI that helps people write code.

In this lecture, we learn about the basic mechanisms of attention and self-attention and see how they are used in the original transformer architecture.

```
[1]: from IPython.display import Image
```

0.1 Adding an attention mechanism to RNNs

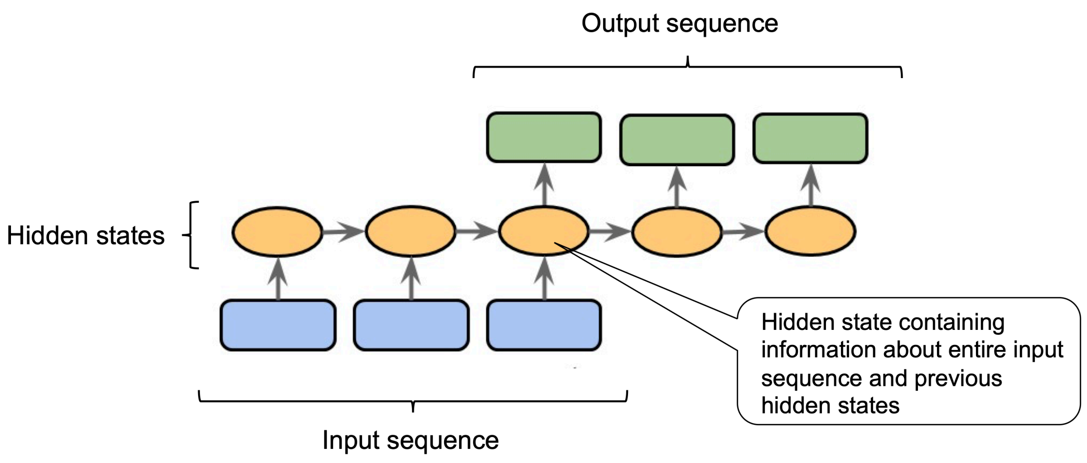
What is the motivation behind adding *attention* to RNNs? It helps predictive models to focus on certain parts of the input sequence more than others.

0.1.1 Attention helps RNNs with accessing information

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

```
[2]: Image(filename='figures/16_01.png', width=500)
```

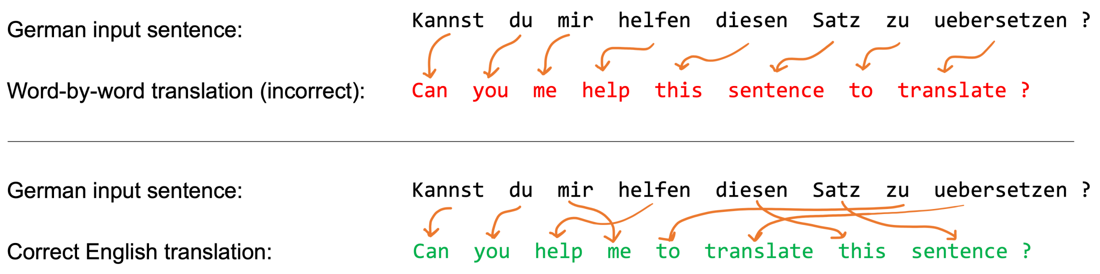
[2]:


Output sequence
Hidden states
Hidden state containing information about entire input sequence and previous hidden states
Input sequence

Why is the RNN parsing the whole input sentence before producing the first output? This is motivated by the fact that translating a sentence word by word would likely result in grammatical errors.

```
[3]: Image(filename='figures/16_02.png', width=700)
```

[3]:


German input sentence:
Kannst du mir helfen diesen Satz zu uebersetzen ?
Word-by-word translation (incorrect):
Can you me help this sentence to translate ?
German input sentence:
Kannst du mir helfen diesen Satz zu uebersetzen ?
Correct English translation:
Can you help me to translate this sentence ?

However, one limitation of this **seq2seq** approach is that the RNN is trying to remember the entire input sequence via one single hidden unit before translating it.

Compressing all the information into one hidden unit may cause loss of information, especially for long sequences.

Thus, similar to how humans translate sentences, it may be beneficial to have access to the whole input sequence at each time step.

In contrast to a regular RNN, an attention mechanism lets the RNN access all input elements at each given time step.

However, having access to all input sequence elements at each time step can be overwhelming.

So, to help the RNN focus on the most relevant elements of the input sequence, the attention mechanism assigns different attention weights to each input element.

These attention weights designate how important or relevant a given input sequence element is at a given time step.

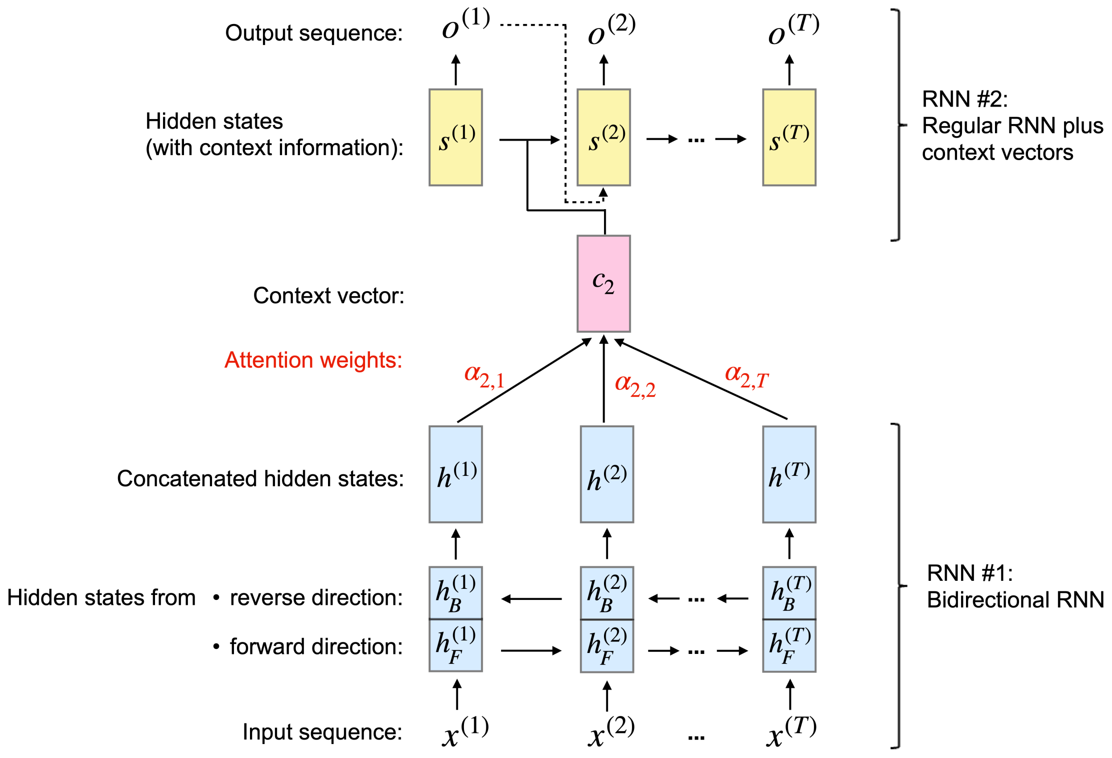
For example, in the above example, the words “mir, helfen, zu” may be more relevant for producing the output word “help” than the words “kannst, du, Satz.”

0.1.2 The original attention mechanism for RNNs

Let us look at the mechanics of the attention mechanism that was originally developed for language translation.

```
[4]: Image(filename='figures/16_03.png', width=600)
```

[4]:



Given an input sequence $x = (x^{(1)}, x^{(2)}, \ldots, x^{(T)})$, the attention mechanism assigns a weight to each element $x^{(i)}$ (or, to be more specific, its hidden representation) and helps the model identify which part of the input it should focus on.

For example, suppose our input is a sentence, and a word with a larger weight contributes more to our understanding of the whole sentence.

The RNN with the attention mechanism as shown above (modeled after the original paper on attention mechanism) illustrates the overall concept of generating the second output word.

As is clear above, there are two RNN models in the overall architecture.

0.1.3 Processing the inputs using a bidirectional RNN

The first RNN (RNN #1) of the attention-based RNN above is a bidirectional RNN that generates context vectors, c_i. We can think of a context vector as an augmented version of the input vector, $x^{(i)}$.

In other words, the c_i input vector also incorporates information from all other input elements via an attention mechanism.

As we can see in the figure above, RNN #2 then uses this context vector, prepared by RNN #1, to generate the outputs. We first discuss how RNN #1 works, and then we discuss RNN #2.

The bidirectional RNN #1 processes the input sequence x in the regular forward direction $(1 \dots T)$ as well as backward $(T \dots 1)$.

Parsing a sequence in the backward direction has the same effect as reversing the original input sequence—think of reading a sentence in reverse order.

The rationale behind this is to capture additional information since current inputs may have a dependence on sequence elements that came either before or after it in a sentence, or both.

Consequently, from reading the input sequence twice (that is, forward and backward), we have two hidden states for each input sequence element.

For instance, for the second input sequence element $x^{(2)}$, we obtain the hidden state $h_F^{(2)}$ from the forward pass and the hidden state $h_B^{(2)}$ from the backward pass.

These two hidden states are then concatenated to form the hidden state $h^{(2)}$.

For example, if both $h_F^{(2)}$ and $h_B^{(2)}$ are 128-dimensional vectors, the concatenated hidden state $h^{(2)}$ will consist of 256 elements.

We can consider this concatenated hidden state as the "annotation" of the source word since it contains the information of the jth word in both directions.

Given the concatenated hidden states, we use the second RNN to generate outputs.

0.1.4 Generating outputs from context vectors

We can consider RNN #2 as the main RNN that is generating the outputs.

In addition to the hidden states, it receives so-called context vectors as input.

A context vector c_i is a weighted version of the concatenated hidden states, $h^{(1)} \dots h^{(T)}$, which we obtained from RNN #1.

We can compute the context vector of the ith input as a weighted sum

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

Here α_{ij} represents the attention weights over the input sequence $j = 1 \dots T$ in the context of the ith input sequence element.

Note that each ith input sequence element has a unique set of attention weights.

We discuss the computation of the attention weights α_{ij} next.

Let us discuss how the context vectors are used via RNN #2.

Just like a vanilla (regular) RNN, RNN #2 also uses hidden states.

Considering the hidden layer between the aforementioned "annotation" and final output, let us denote the hidden state at time i as $s^{(i)}$.

Now, RNN #2 receives the aforementioned context vector c_i at each time step i as input.

In the model architecture, we saw that the hidden state $s^{(i)}$ depends on the previous hidden state $s^{(i-1)}$, the previous target word $y^{(i-1)}$ and the context vector $c^{(i)}$, which are used to generate the predicted output $o^{(i)}$ for target word $y^{(i)}$ at time i.

Note that the sequence vector $\mathbf{y}$ refers to the sequence vector representing the correct translation of input sequence $\mathbf{x}$ that is available during training.

During training, the true label (word) $y^{(i)}$ is fed into the next state $s^{(i+1)}$ since this true label information is not available for prediction (inference), we feed the predicted output $o^{(i)}$ instead.

To summarize, the attention-based RNN consists of two RNNs.

RNN #1 prepares context vectors from the input sequence elements, and RNN #2 receives the context vectors as input.

The context vectors are computed via a weighted sum over the inputs, where the weights are the attention weights α_{ij}. Next we discuss how to compute these attention weights.

0.1.5 Computing the attention weights

Because the weights pairwise connect the inputs (annotations) and the outputs (contexts), each attention weight α_{ij} has two subscripts: j refers to the index position of the input and i corresponds to the output index position.

The attention weight α_{ij} is a normalized version of the alignment score e_{ij} where the alignment score evaluates how well the input around position j matches with the output at position i.

To be more specific, the attention weight is computed by normalizing the alignment scores as follows

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T} exp(e_{ik})} \tag{2}$$

where T is the length of the sequence. Note that the above normalisation is similar to the `softmax` function.

Consequently, the attention weights $\alpha_{i1} ... \alpha_{iT}$ sum up to 1.

To summarise, we can structure the attention-based RNN model into three parts.

- The first part computes bidirectional annotations of the input.
- The second part consists of the recurrent block, which is very much like the original RNN, except that it uses context vectors instead of the original input.

- The last part concerns the computation of the attention weights and context vectors, which describe the relationship between each pair of input and output elements

The transformer architecture also utilizes an attention mechanism, but unlike the attention-based RNN, it solely relies on the **self-attention** mechanism and does not include the recurrent process found in the RNN.

In other words, a transformer model processes the whole input sequence all at once instead of reading and processing the sequence one element at a time.

Next we introduce a basic form of the self-attention mechanism before we discuss the transformer architecture in more detail.

0.2 Introducing the self-attention mechanism

We can have an architecture entirely based on attention, without the recurrent parts of an RNN. This attention-based architecture is known as **transformer**.

In fact, transformers can appear a bit complicated at first glance.

So, before we discuss transformers, let us dive into the self-attention mechanism used in transformers.

In fact, as we will see, this self-attention mechanism is just a different flavour of the attention mechanism that we earlier.

We can think of the previously discussed attention mechanism as an operation that connects two different modules, that is, the encoder and decoder of the RNN.

As we will see, self-attention focuses only on the input and captures only dependencies between the input elements, without connecting two modules.

First, we introduce a basic form of self-attention without any learning parameters, which is very much like a pre-processing step to the input.

Next, we introduce the common version of self-attention that is used in the transformer architecture and involves learnable parameters.

0.2.1 Starting with a basic form of self-attention

To introduce self-attention, let us assume we have an input sequence of length T, $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)}$ as well as an output sequence $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \dots, \mathbf{z}^{(T)}$.

We use $\mathbf{o}$ as the final output of the transformer model and $\mathbf{z}$ as the output of the self-attention layer because it is an intermediate step in the model.

Each ith element in these sequences, $\mathbf{x}^{(i)}$ and $\mathbf{z}^{(i)}$, are vectors of size d (that is, $\mathbf{x}^{(i)} \in \mathbb{R}^d$) representing the feature information for the input at position i, which is similar to RNNs.

Then, for a `seq2seq` task, the goal of self-attention is to model the dependencies of the current input element to all other input elements.

To achieve this, self-attention mechanisms are composed of three stages.

- First, we derive importance weights based on the similarity between the current element and all other elements in the sequence.

- Second, we normalize the weights, which usually involves the use of the already familiar `softmax` function.
- Third, we use these weights in combination with the corresponding sequence elements to compute the attention value.

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

$$\mathbf{z}^{(i)} = \sum_{j=1}^{T} \alpha_{ij} \mathbf{x}^{(j)}. \tag{3}$$

Hence, we can think of $\mathbf{z}^{(i)}$ as a context-aware embedding vector in input vector $\mathbf{x}^{(i)}$ that involves all other input sequence elements weighted by their respective attention weights.

Here, the attention weights, α_{ij} are computed based on the similarity between the current input element, $\mathbf{x}^{(i)}$, and all other elements in the input sequence, $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$.

More concretely, this similarity is computed in two steps.

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

$$\omega_{ij} = \mathbf{x}^{(i)^\top} \mathbf{x}^{(j)}. \tag{4}$$

Before we normalize the ω_{ij} values to obtain the attention weights, a_{ij}, let us illustrate how we compute the ω_{ij} values with a code example.

Here, let us assume we have an input sentence "can you help me to translate this sentence" that has already been mapped to an integer representation via a dictionary as explained in Lecure 5.

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

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

Let us also assume that we already encoded this sentence into a real-number vector representation via an embedding layer.

Here, our embedding size is 16, and we assume that the dictionary size is 10.

The following code will produce the word embeddings of our eight words.

[6]:
```
torch.manual_seed(123)
embed = torch.nn.Embedding(10, 16)
embedded_sentence = embed(sentence).detach()
embedded_sentence.shape
```

[6]:
```
torch.Size([8, 16])
```

- The goal is to compute the context vectors $z^{(i)} = \sum_{j=1}^{T} \alpha_{ij} x^{(j)}$, which involve attention weights α_{ij}.
- In turn, the attention weights α_{ij} involve the ω_{ij} values
- Let's start with the ω_{ij}'s first, which are computed as dot-products:

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

[7]:
```
omega = torch.empty(8, 8)

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

- Actually, let's compute this more efficiently by replacing the nested for-loops with a matrix multiplication:

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

[8]:
```
omega_mat = embedded_sentence.matmul(embedded_sentence.T)
```

We can use the `torch.allclose` function to check that this matrix multiplication produces the expected results.

If two tensors contain the same values, `torch.allclose` returns `True`, as we can see here:

[9]:
```
torch.allclose(omega_mat, omega)
```

[9]:
```
True
```

We have learned how to compute the similarity-based weights for the ith input and all inputs in the sequence $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$, the "raw" weights ω_{i1} to ω_{iT}.

We can obtain the attention weights, α_{ij}, by normalizing the ω_{ij} values via the familiar `softmax` function, as follows

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

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

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

[10]:
```
import torch.nn.functional as F

attention_weights = F.softmax(omega, dim=1)
attention_weights.shape
```

[10]:
```
torch.Size([8, 8])
```

Note that `attention_weights` is an 8×8 matrix, where each element represents an attention weight, α_{ij}.

For instance, if we are processing the ith input word, the ith row of this matrix contains the corresponding attention weights for all words in the sentence.

These attention weights indicate how relevant each word is to the ith word. Hence, the columns in this attention matrix should sum to 1, which we can confirm via the following code:

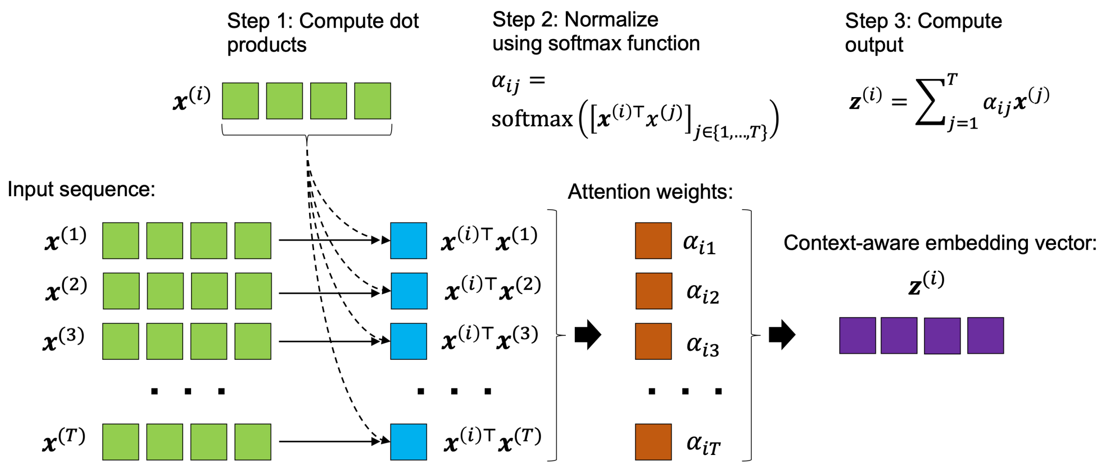
[11]:
```
attention_weights.sum(dim=1)
```

[11]:
```
tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

Now that we have seen how to compute the attention weights, let us summarize the three main steps behind the self-attention operation: 1. For a given input element, $\mathbf{x}^{(i)}$, and each jth element in the set $\{1, \ldots, T\}$, compute the dot product, $\mathbf{x}^{(i)\top}\mathbf{x}^{(j)}$. 2. Obtain the attention weight, α_{ij} by normalizing the dot products using the `softmax` function. 3. Compute the output, $\mathbf{z}^{(i)}$, as the weighted sum over the entire input sequence: $\mathbf{z}^{(i)} = \sum_{j=1}^{T} \alpha_{ij}\mathbf{x}^{(j)}$.

[12]:
```
Image(filename='figures/16_04.png', width=700)
```

[12]:



Step 1: Compute dot products
Step 2: Normalize using softmax function
$\alpha_{ij} = \text{softmax}\left(\left[\boldsymbol{x}^{(i)\top}\boldsymbol{x}^{(j)}\right]_{j\in\{1,\ldots,T\}}\right)$
Step 3: Compute output
$\boldsymbol{z}^{(i)} = \sum_{j=1}^{T} \alpha_{ij}\boldsymbol{x}^{(j)}$
$\boldsymbol{x}^{(i)}$
Input sequence:
$\boldsymbol{x}^{(1)}$
$\boldsymbol{x}^{(2)}$
$\boldsymbol{x}^{(3)}$
$\boldsymbol{x}^{(T)}$
$\boldsymbol{x}^{(i)\top}\boldsymbol{x}^{(1)}$
$\boldsymbol{x}^{(i)\top}\boldsymbol{x}^{(2)}$
$\boldsymbol{x}^{(i)\top}\boldsymbol{x}^{(3)}$
$\boldsymbol{x}^{(i)\top}\boldsymbol{x}^{(T)}$
Attention weights:
α_{i1}
α_{i2}
α_{i3}
α_{iT}
Context-aware embedding vector:
$\boldsymbol{z}^{(i)}$

For instance, to compute the context-vector of the 2nd input element (the element at index 1), that is $\mathbf{z}^{(2)}$, we can perform the following computation:

[13]:
```
x_2 = embedded_sentence[1, :]
context_vec_2 = torch.zeros(x_2.shape)
for j in range(8):
    x_j = embedded_sentence[j, :]
    context_vec_2 += attention_weights[1, j] * x_j

context_vec_2
```

[13]:
```
tensor([-9.3975e-01, -4.6856e-01,  1.0311e+00, -2.8192e-01,  4.9373e-01,
        -1.2896e-02, -2.7327e-01, -7.6358e-01,  1.3958e+00, -9.9543e-01,
        -7.1287e-04,  1.2449e+00, -7.8077e-02,  1.2765e+00, -1.4589e+00,
        -2.1601e+00])
```

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

[14]:
```
context_vectors = torch.matmul(attention_weights, embedded_sentence)


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

[14]:
```
True
```

Here we implemented a basic form of self-attention. Next, we modify this implementation using learnable parameter matrices that can be optimized during neural network training.

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

Let us now look at the the more advanced self-attention mechanism called **scaled dot-product attention** that is used in the transformer architecture.

Note that we did not involve any learnable parameters when computing the outputs.

In other words, using the previously introduced basic self-attention mechanism, the transformer model is rather limited regarding how it can update or change the attention values during model optimization for a given sequence.

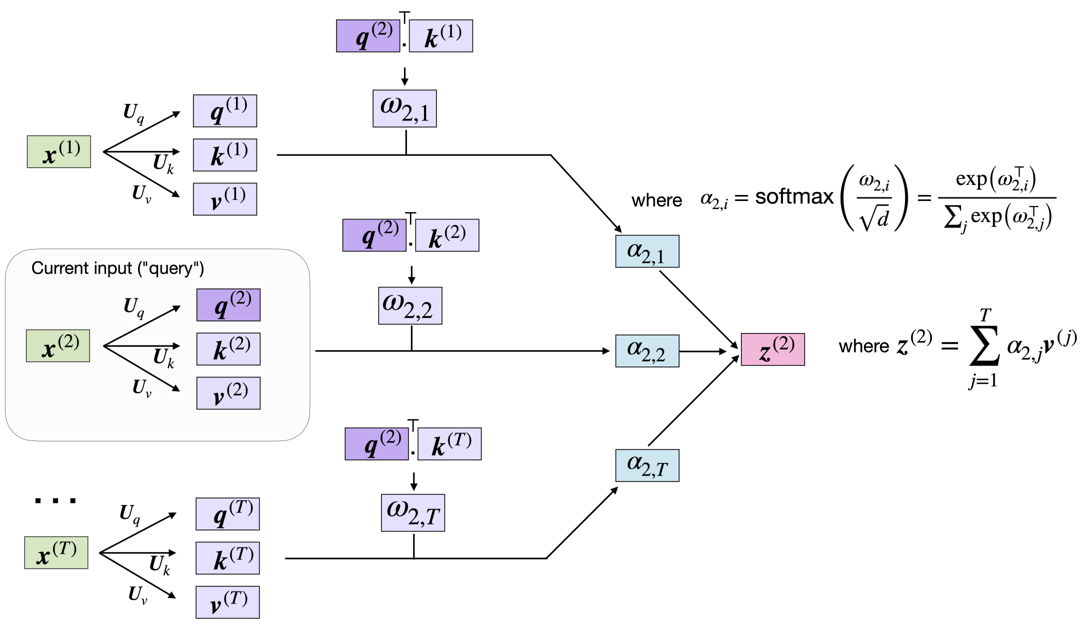
To make the self-attention mechanism more flexible and amenable to model optimization, we introduce three additional weight matrices that can be fit as model parameters during model training.

We denote these three weight matrices as $\mathbf{U}_q$, $\mathbf{U}_k$ and $\mathbf{U}_v$. They are used to project the inputs into query, key, and value sequence elements, as follows: - Query sequence: $q^{(i)} = \mathbf{U}_q\mathbf{x}^{(i)}$ for $i \in [1, \ldots, T]$ - Key sequence: $k^{(i)} = \mathbf{U}_k\mathbf{x}^{(i)}$ for $i \in [1, \ldots, T]$ - Value sequence: $v^{(i)} = \mathbf{U}_v\mathbf{x}^{(i)}$ for $i \in [1, \ldots, T]$

Below we show how individual components are used to compute the context-aware embedding vector corresponding to the second input element:

[15]:
```
Image(filename='figures/16_05.png', width=700)
```

[15]:


Current input ("query")
where $\alpha_{2,i} = \text{softmax}\left(\frac{\omega_{2,i}}{\sqrt{d}}\right) = \frac{\exp(\omega_{2,i}^\top)}{\sum_j \exp(\omega_{2,j}^\top)}$
where $z^{(2)} = \sum_{j=1}^{T} \alpha_{2,j} v^{(j)}$

Here, both $\mathbf{q}^{(i)}$ and $\mathbf{k}^{(i)}$ are vectors of size d_k.

Therefore, the projection matrices $\mathbf{U}_q$ and $\mathbf{U}_k$ have the shape $d_k \times d$, while $\mathbf{U}_v$ has the shape $d_v \times d$. (Note that d is the dimensionality of each word vector, $\mathbf{x}^{(i)}$.)

For simplicity, we can design these vectors to have the same shape, for example, using $d_k = d_v = d$.

To provide additional intuition via code, we can initialize these projection matrices as follows:

[16]:
```
torch.manual_seed(123)

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

Using the query projection matrix, we can then compute the query sequence. For this example, consider the second input element, $\mathbf{x}^{(2)}$ as our query.

[17]:
```
x_2 = embedded_sentence[1]
query_2 = U_query.matmul(x_2)
```

Similarly, we can compute the key and value sequences, $\mathbf{k}^{(2)}$ and $\mathbf{v}^{(2)}$:

[18]:
```
key_2 = U_key.matmul(x_2)
value_2 = U_value.matmul(x_2)
```

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

[19]:
```
keys = U_key.matmul(embedded_sentence.T).T
torch.allclose(key_2, keys[1])
```

[19]:
```
True
```

In the key matrix, the ith row corresponds to the key sequence of the ith input element, and the same applies to the value matrix.

[20]:
```
values = U_value.matmul(embedded_sentence.T).T
torch.allclose(value_2, values[1])
```

[20]:
```
True
```

Previously, we computed the unnormalized weights, ω_{ij} as the pairwise dot product between the given input sequence element,$\mathbf{x}^{(i)}$ and the jth sequence element,$\mathbf{x}^{(J)}$.

Now, in this parameterized version of self-attention, we compute ω_{ij} as the dot product between the query and key:

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

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

[21]:
```
omega_23 = query_2.dot(keys[2])
omega_23
```

[21]:
```
tensor(14.3667)
```

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

[22]:
```
omega_2 = query_2.matmul(keys.T)
omega_2
```

[22]:
```
tensor([-25.1623,   9.3602,  14.3667,  32.1482,  53.8976,  46.6626,  -1.2131,
        -32.9392])
```

The next step in self-attention is to go from the unnormalized attention weights, ω_{ij}, to the normalized attention weights, α_{ij}, using the `softmax` function.

We can then further use $\frac{1}{\sqrt{m}}$ to scale ω_{ij} before normalizing it via the softmax function.

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

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

[23]:
```
attention_weights_2 = F.softmax(omega_2 / d**0.5, dim=0)
attention_weights_2
```

[23]:
```
tensor([2.2317e-09, 1.2499e-05, 4.3696e-05, 3.7242e-03, 8.5596e-01, 1.4026e-01,
        8.8897e-07, 3.1935e-10])
```

[24]:
```
#context_vector_2nd = torch.zeros(values[1, :].shape)
#for j in range(8):
#     context_vector_2nd += attention_weights_2[j] * values[j, :]

#context_vector_2nd
```

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

[25]:
```
context_vector_2 = attention_weights_2.matmul(values)
context_vector_2
```

[25]:
```
tensor([-1.2226, -3.4387, -4.3928, -5.2125, -1.1249, -3.3041, -1.4316, -3.2765,
        -2.5114, -2.6105, -1.5793, -2.8433, -2.4142, -0.3998, -1.9917, -3.3499])
```

0.3 Attention is all we need: introducing the original transformer architecture

After using RNNs with attention, researchers found that an attention-based language model was even more powerful when the recurrent layers were deleted. This led to the development of the **transformer** architecture.

Thanks to the self-attention mechanism, a transformer model can capture long-range dependencies among the elements in an input sequence—in an NLP context; for example, this helps the model better “understand” the meaning of an input sentence.

The transformer architecture was first proposed in the NeurIPS 2017 paper Attention Is All You Need by *A. Vaswani* and colleagues.

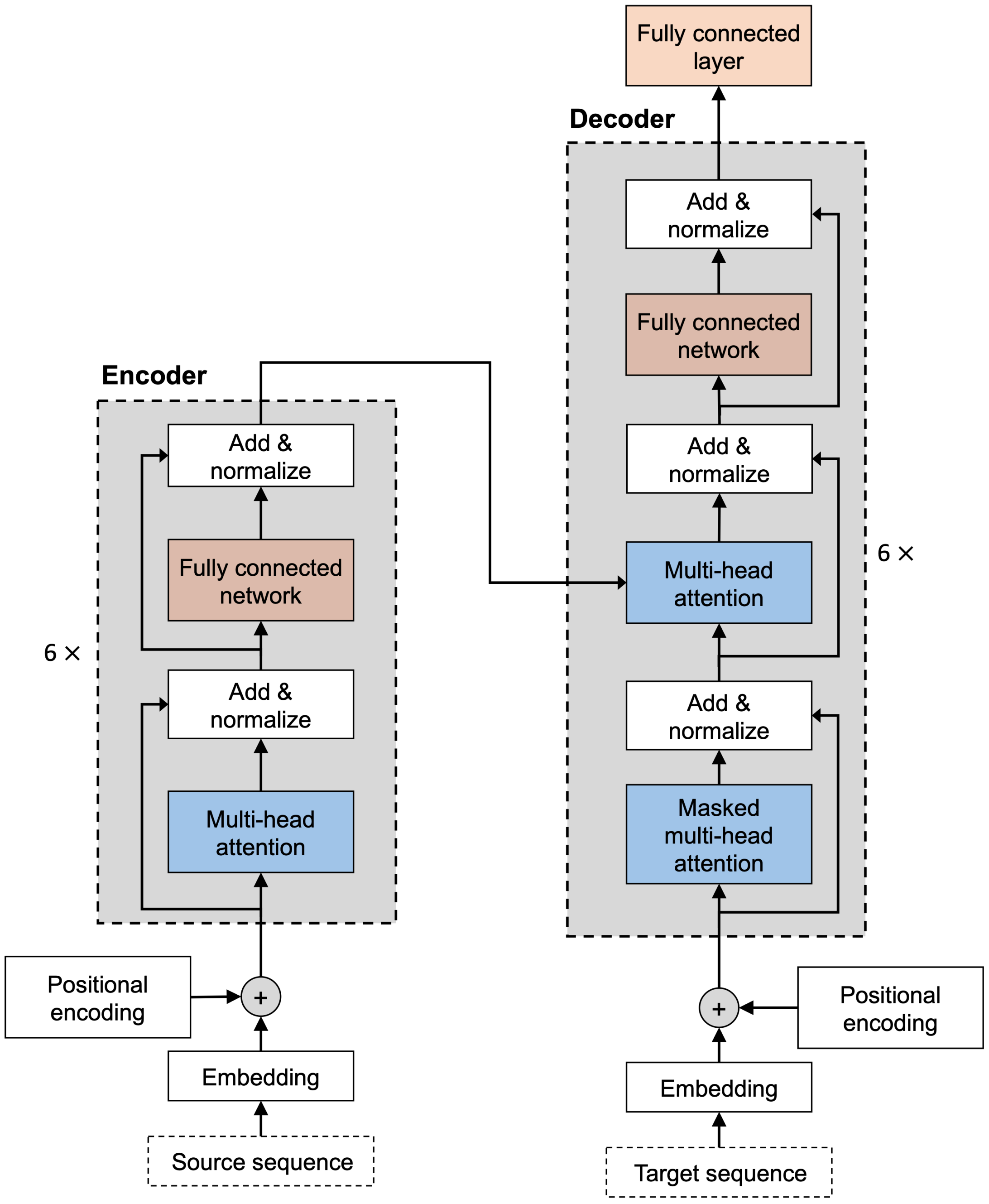
Although this transformer architecture was originally designed for language translation, it can be generalized to other tasks such as text generation, and text classification.

Later on, we discuss popular language models, such as BERT and GPT, which were derived from this original transformer architecture.

The following figure adapted from the original transformer paper illustrates the main architecture and components.

[26]:
```
Image(filename='figures/16_06.png', width=600)
```

[26]:



Let us go over the this original transformer model step by step, by decomposing it into two main blocks: an encoder and a decoder.

The encoder receives the original sequential input and encodes the embeddings using a multi-head self-attention module.

The decoder takes in the processed input and outputs the resulting sequence (for instance, the translated sentence) using a masked form of self-attention.

0.3.1 Encoding context embeddings via multi-head attention

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

The encoder is a stack of six identical layers. Six is not a magic number here but merely a hyperparameter choice made in the original transformer paper. You can adjust the number of layers according to the model performance.

Inside each of these identical layers, there are two sublayers: - one computes the multi-head self-attention, which we discuss below, and, - the other one is a fully connected layer, which we have already encountered previously in the course

Multi-head self-attention, which is a simple modification of the **scaled dot-product attention** covered earlier.

In the scaled dot-product attention, we used three matrices (corresponding to query, value, and key) to transform the input sequence. In the context of multi-head attention, we can think of this set of three matrices as *one attention head*.

As indicated by its name, in multi-head attention, we now have a multiple of such heads (sets of query, value, and key matrices).

To explain the concept of multi-head self-attention with h heads in more detail, let us break it down into the following steps. First, we read in the sequential input $\mathbf{X} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)})$.

Suppose each element is embedded by a vector of length d. Here, the input can be embedded into a $T \times d$ matrix. Then, we create h sets of the query, key, and value learning parameter matrices.

- $\mathbf{U}_{q1}, \mathbf{U}_{k1}, \mathbf{U}_{v1}$
- $\mathbf{U}_{q2}, \mathbf{U}_{k2}, \mathbf{U}_{v2}$
- $\vdots$
- $\mathbf{U}_{qh}, \mathbf{U}_{kh}, \mathbf{U}_{vh}$

Because we are using these weight matrices to project each element $\mathbf{x}^{(i)}$ for the required dimension-matching in the matrix multiplications, both $\mathbf{U}_{qj}$ and $\mathbf{U}_{kj}$ have the shape $d_k \times d$, and $\mathbf{U}_{vj}$ has the shape $d_v \times d$.

As a result, both resulting sequences, query and key, have length d_k, and the resulting value sequence has length dv.

In practice, people often choose $d_k = d_v = m$ for simplicity.

To illustrate the multi-head self-attention stack in code, first consider how we created the single query projection matrix previously.

[27]:
```
torch.manual_seed(123)

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

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

```
[28]: h = 8
      multihead_U_query = torch.rand(h, d, d)
      multihead_U_key = torch.rand(h, d, d)
      multihead_U_value = torch.rand(h, d, d)
```

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

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

$$\mathbf{q}_j^{(i)} = \mathbf{U}_{q_j}\mathbf{x}^{(i)}.$$

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

For the second input word as the query, the code looks like as follows:

```
[29]: multihead_query_2 = multihead_U_query.matmul(x_2)
      multihead_query_2.shape
```

```
[29]: torch.Size([8, 16])
```

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

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

```
[30]: multihead_key_2 = multihead_U_key.matmul(x_2)
      multihead_value_2 = multihead_U_value.matmul(x_2)
```

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

```
[31]: multihead_key_2[2]
```

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

However, remember that we need to repeat the key and value computations for all input sequence elements, not just x_2 — we need this to compute self-attention later.

A simple and illustrative way to do this is by expanding the input sequence embeddings to size 8 as the first dimension, which is the number of attention heads. We use the `.repeat()` method for this.

```
[32]: stacked_inputs = embedded_sentence.T.repeat(8, 1, 1)
      stacked_inputs.shape
```

```
[32]: torch.Size([8, 16, 8])
```

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

```
[33]: multihead_keys = torch.bmm(multihead_U_key, stacked_inputs)
      multihead_keys.shape
```

```
[33]: torch.Size([8, 16, 8])
```

In this code, we now have a tensor that refers to the eight attention heads in its first dimension. The second and third dimensions refer to the embedding size and the number of words, respectively.

Let us swap the second and third dimensions so that the keys have a more intuitive representation, that is, the same dimensionality as the original input sequence embedded_sentence:

```
[34]: multihead_keys = multihead_keys.permute(0, 2, 1)
      multihead_keys.shape
```

```
[34]: torch.Size([8, 8, 16])
```

```
[35]: multihead_keys[2, 1] # index: [2nd attention head, 2nd key]
```

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

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

```
[36]: multihead_values = torch.matmul(multihead_U_value, stacked_inputs)
      multihead_values = multihead_values.permute(0, 2, 1)
```

We follow the steps of the single head attention calculation to calculate the context vectors next.

Let us skip the intermediate steps and assume that we have computed the context vectors for the second input element as the query and the eight different attention heads, which we represent as `multihead_z_2` via some random data:

```
[37]: multihead_z_2 = torch.rand(8, 16)
```

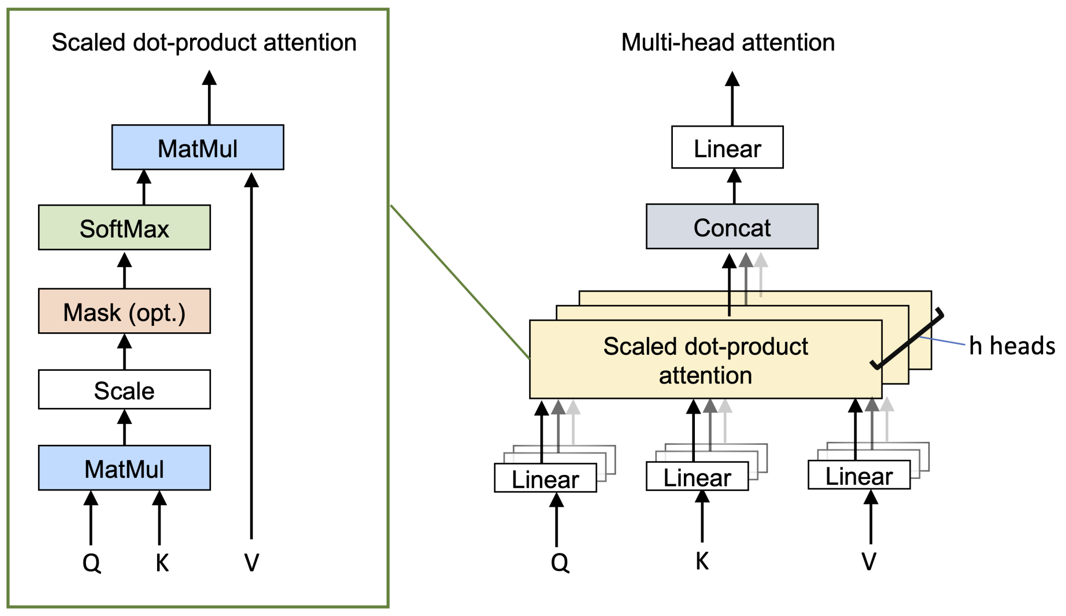
Note that the first dimension indexes over the eight attention heads, and the context vectors, similar to the input sentences, are 16-dimensional vectors.

We can think of `multihead_z_2` as eight copies of the $z^{(2)}$ we computed in the case of single-head attention; that is, we have one $z^{(2)}$ for each of the eight attention heads.

Then, we concatenate these vectors into one long vector of length $d_v \times h$ and use a linear projection (via a fully connected layer) to map it back to a vector of length d_v. This process is illustrated in the figure below. Left hand side of the figure shows us the detailed steps involved in the operation of computing scaled dot-product attention.

```
[38]: Image(filename='figures/16_07.png', width=700)
```

```
[38]:
```



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

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

```
[39]: torch.Size([16])
```

To summarize, the multi-head self-attention is repeating the scaled dot-product attention computation multiple times in parallel and combining the results.

It works very well in practice because the multiple heads help the model to capture information from different parts of the input.

Lastly, while the multi-head attention sounds computationally expensive, in practice it is not since the computations across different heads can all be done in parallel because of no dependencies between the multiple heads.

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

Similar to the encoder, the **decoder** also contains several repeated layers.

Besides the two sublayers that we have already introduced in the encoder (the multi-head self-attention layer and fully connected layer), each repeated layer also contains a masked multi-head attention sublayer.

Masked attention is a variation of the original attention mechanism, where masked attention only passes a limited input sequence into the model by "masking" out a certain number of words.

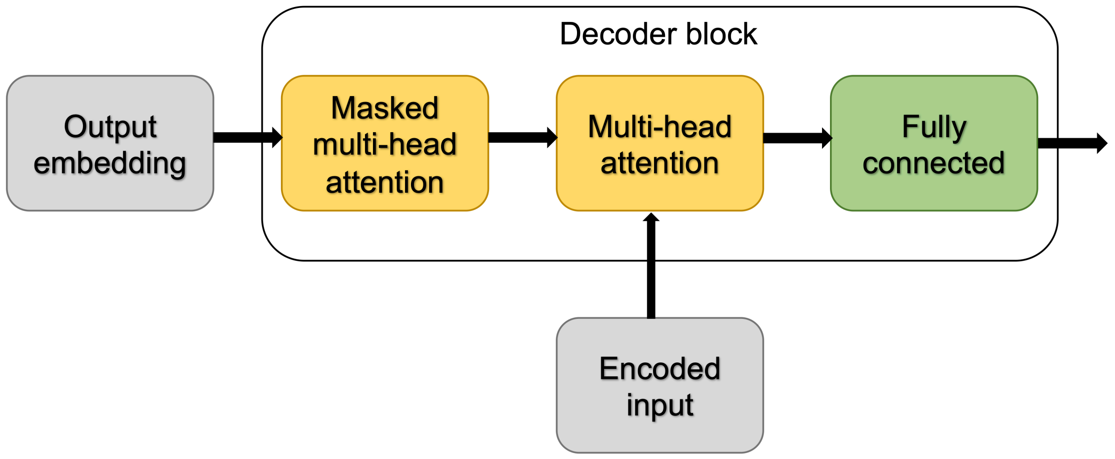
For example, if we are building a language translation model with a labelled dataset, at sequence position i during the training procedure, we only feed in the correct output words from positions $1, \ldots, i-1$. All other words (for instance, those that come after the current position) are hidden from the model to prevent the model from "cheating."

This is also consistent with the nature of text generation: although the true translated words are known during training, we know nothing about the ground truth in practice. Thus, we can only feed the model the solutions to what it has already generated, at position i.

The figure below shows how the layers are arranged in the decoder block.

[40]:
```
Image(filename='figures/16_08.png', width=600)
```

[40]:



First, the previous output words (output embeddings) are passed into the masked multi-head attention layer.

Then, the second layer receives both the encoded inputs from the encoder block and the output of the masked multi-head attention layer into a multi-head attention layer.

Finally, we pass the multi-head attention outputs into a fully connected layer that generates the overall model output: a probability vector corresponding to the output words.

Note that we can use the `argmax` function to obtain the predicted words from these word probabilities.

Comparing the decoder with the encoder block, the main difference is the range of sequence elements that the model can attend to. In the encoder, for each given word, the attention is calculated across all the words in a sentence, which can be considered as a form of bidirectional input parsing.

The decoder also receives the bidirectionally parsed inputs from the encoder. However, when it comes to the output sequence, the decoder only considers those elements that are preceding the current input position, which can be interpreted as a form of unidirectional input parsing.

0.3.3 Implementation details: positional encodings and layer normalization

There are some more implementation details that we need to be careful about before implementing the transformers.

First, let us consider the positional encodings that were part of the original transformer architecture.

Positional encodings help with capturing information about the input sequence ordering and are a crucial part of transformers because both scaled dot-product attention layers and fully connected layers are permutation-invariant.

This means, without positional encoding, the order of words is ignored and does not make any difference to the attention-based encodings.

However, we know that word order is essential for understanding a sentence. For example, consider the following two sentences:

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

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

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

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

Here i is the position of the word and k denotes the length of the encoding vector, where we choose k to have the same dimension as the input word embeddings so that the positional encoding and word embeddings can be added together.

Sinusoidal functions are used to prevent positional encodings from becoming too large.

For instance, if we used absolute position $1, 2, 3, \ldots, n$ to be positional encodings, they would dominate the word encoding and make the word embedding values negligible.

In general, there are two types of positional encodings, an absolute one (as shown in the previous formula) and a relative one.

The former will record absolute positions of words and is sensitive to word shifts in a sentence. That is to say, absolute positional encodings are fixed vectors for each given position.

On the other hand, relative encodings only maintain the relative position of words and are invariant to sentence shift.

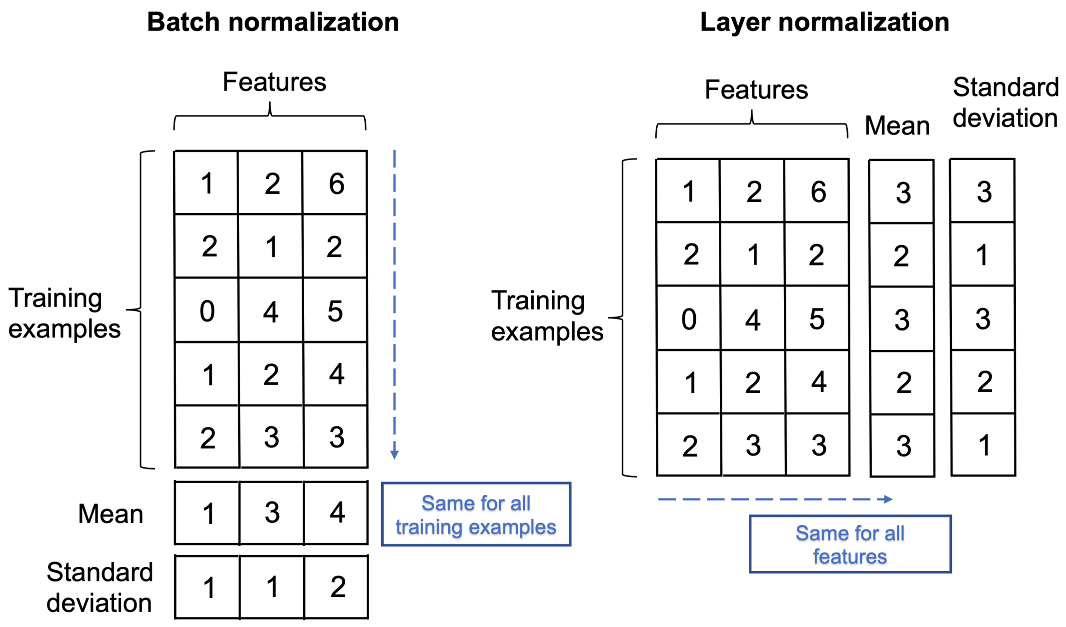
Next is the concept of layer normalization.

While batch normalization, which we will discuss in more detail when we study Generative Adversarial Networks (GANs), is a popular choice in computer vision contexts, layer normalization is the preferred choice in natural language processing (NLP) contexts where sentence length can vary.

The figure below illustrates the main difference between batch and layer normalization.

```
[41]: Image(filename='figures/16_09.png', width=600)
```

[41]:



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

Since layer normalization computes mean and standard deviation for each training example, it relaxes minibatch size constraints or dependencies.

In contrast to batch normalization, layer normalization is thus capable of learning from data with small minibatch sizes and varying lengths.

However, note that the original transformer architecture does not have varying-length inputs (sentences are padded when needed just like we did when implementing RNNs).

How do we justify the use of layer normalization over batch normalization?

Transformers are usually trained on very large text corpora, which requires parallel computation; this can be challenging to achieve with batch normalization, which has a dependency between training examples.

Layer normalization has no such dependency and is thus a more natural choice for transformers.