RCDS AI Club
June 21st, 2025

This newsletter, prepared by the RCDS AI Club, compiles educational notes on artificial intelligence concepts, primarily drawn from *Tutorial 6: Transformers and multi-head attention - UVA DL notebooks v1.2 documentation* (n.d.), published by the University of Amsterdam. Significant portions of the content, including direct quotations and closely paraphrased text, are sourced from this tutorial and are attributed with in-text citations (e.g., ("Tutorial 6," n.d.)) throughout the document. Direct quotes are enclosed in quotation marks, and paraphrased content is credited to the original source. Additional sources include a YouTube video by Alelab (2020) on permutation equivariance in graph neural networks and the *torch.nn.init - PyTorch 2.7 documentation* (PyTorch, 2024) for initialization methods. Some explanations and rephrasings, noted as "ChatGPT rephrased" or similar, were generated by OpenAI's ChatGPT (personal communication, June 21, 2025) to clarify concepts. Citation formatting and additional clarifications were provided by xAI's Grok 3 (personal communication, August 1, 2025). All sources are credited to ensure transparency and academic integrity. These notes are intended for educational purposes and are shared under the copyright permissions of the University of Amsterdam's *UVA DL Notebooks* (n.d.). For questions or further details, contact the RCDS AI Club. (generated by Grok 3)

Notetaker: Felipe Quintero Ochoa, Rye Country Day School AI Club Co-President

Notes taken from (an excellent article on how to build a neural network from scratch): https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html

*Disclaimer: A lot of the notes in this document may have been directly quoted from the article provided above. A lot of the text might be the same or very similar. Any quotes are directly taken from the article provided: https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html

Alelab Alelab. (2020, October 5). *Lecture 5.2 - Permutation Equivariance of Graph Neural Networks*. YouTube. https://www.youtube.com/watch?v=7JuIRqwWNo4

APA Citation:

University of Amsterdam. (n.d.). Tutorial 6: Transformers and multi-head attention - UVA DL notebooks v1.2 documentation. Retrieved August 1, 2025, from https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAAttention.html

PyTorch. (2024). *torch.nn.init - PyTorch 2.7 documentation*. Retrieved August 1, 2025, from https://pytorch.org/docs/stable/nn.init.html

Alelab. (2020, October 5). Lecture 5.2 - Permutation equivariance of graph neural networks [Video]. YouTube. Retrieved August 1, 2025, from https://www.youtube.com/watch?v=7JuiR6qwWNo

AI Usage:

xAI's Grok 3, personal communication, August 1, 2025

OpenAI's ChatGPT, personal communication, various dates.

The Basics:

Okay, so this is where everything we learned so far basically feels like a piece of cake. The transformer architecture is probably one of the most, if not the most difficult concept to grasp in the entirety of Artificial Intelligence. Nevertheless, let's dive right in.

Attention:

Definition: "The attention mechanism describes a recent new group of layers in neural networks [... .] There are a lot of different possible definitions of 'attention' in the literature, but the one we will use here is the following: *the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and element's keys.* The goal is to take an average over the features of multiple elements. However, instead of weighting each element equally, we want to weigh them depending on their actual values. In other words, we want to dynamically decide on which inputs we want to 'attend' more than others. In particular, an attention mechanism has usually four parts we need to specify:

- **Query:** The query is a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.
- **Keys:** For each input element, we have a key which is again a feature vector. This feature vectors roughly describes what the element is 'offering', or when it might be important. The keys designed such that we can identify the elements we want to pay attention to based on the query.

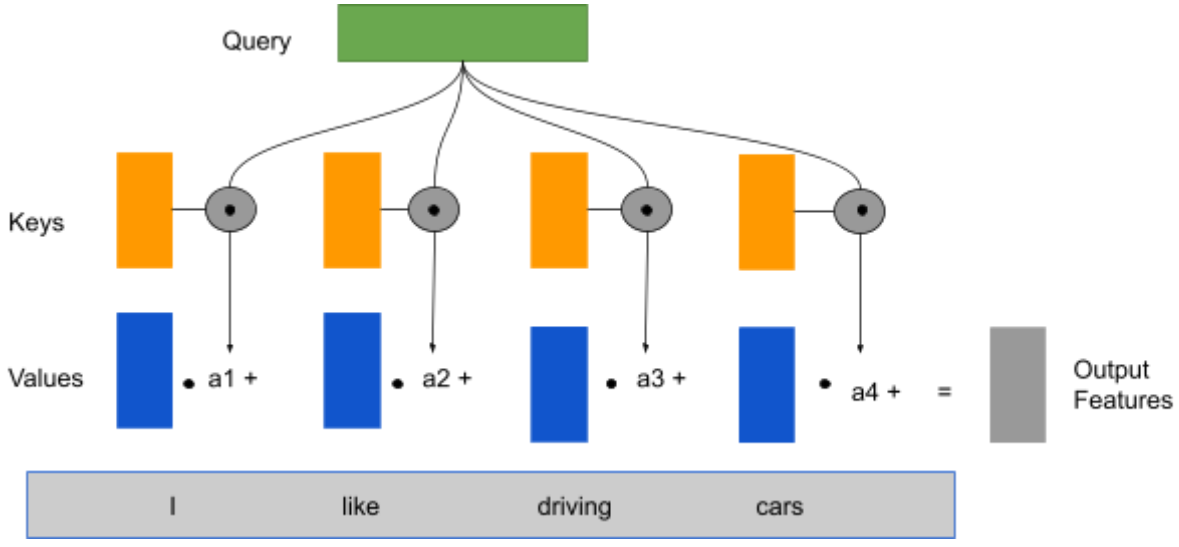
- **Values:** For each input element, we also have a value vector. This feature vector is the one we want to average over.
- **Score function:** To rate which elements we want to pay attention to, we need to specify a score function f_{attn}. The score function takes the query and a key as input, and output the score / attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product , or a small MLP.

The weights of the average are calculated by a softmax over all score function outputs. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query. If we try to describe it with pseudo math, we can write:

$$a_i = \frac{exp(f_{attn}(key_i, query))}{\sum_j exp(f_{attn}(key_i, query))}$$

$$out = \sum_i a_i * value_i$$

Visually, we can show the attention over a sequence of words as:



For every word, we have one key and one value vector. The query is compared to all keys with a score function. The query is compared to all keys with a score function (in this case the dot product) to determine the weights. The softmax is not visualized for simplicity. Finally, the value vectors of all words are averaged using the attention weights.

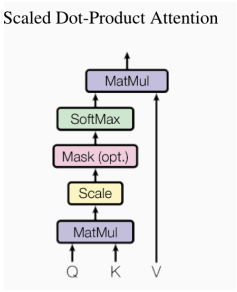
Most attention mechanisms differ in terms of what queries they use, how the key and value vectors are defined, and what score function is used. The attention applied inside the Transformer architecture is called self-attention. In self attention, each sequence element provides a key, value, and a query. For each element, we perform an attention layer where based on its query, we check the similarity of all the sequence elements' keys, and returned a different, averaged value vector for each element. We will now go into a bit more detail by first looking at the specific implementation of the attention mechanism which is in the Transformer case the scaled dot product attention." (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

"**Scaled Dot Product Attention**

The core concept behind self-attention is the scaled dot product attention. Our goal is to have an attention mechanism with which any element in a sequence can attend to any other while still being efficient to compute. The dot product attention takes as input a set of queries $Q \in \mathbb{R}^{T \times d_k}$, keys $K \in \mathbb{R}^{T \times d_k}$, and values $V \in \mathbb{R}^{T \times d_v}$ where T is the sequence length [in tokens (e.g. the sequence length, T, for "a cat sat on a mat," where each word is one token, is 6)], and d_k and d_v are the hidden dimensionality for queries / keys and values respectively [$\mathbb{R}^{T \times d_k}$ just means that there is a matrix with dimensions T - the number of tokens by d_k, which is the number of features for the query and key values, or d_v, which is the number of features for the value values]. For simplicity, we neglect the batch dimension for now. The attention value from i to j is based on the similarity of the query Q_i and key K_j, using the dot product as the similarity metric. In math we calculate the dot product attention as follows:

$$Attention(Q,\ K,\ V)\ =\ softmax(\frac{QK^T}{\sqrt{d_k}})V$$

The matrix multiplication QK^T performs the dot product of every possible pair of queries and keys, resulting in a matrix of the shape T x T [(because we multiply the Query matrix - with dimensions $T\ x\ d_k$ by the transpose of the Keys matrix - with transposed dimensions $d_k\ x\ T$)]. Each row represents the attention logits for a specific element i to all other elements in the sequence. On these, we apply a softmax and multiply with the value vector to obtain a weighted mean (the weights being determined by the attention). Another perspective on this attention mechanism the computation graph which is visualized below (figure credit - Vaswani et al., 2017)



”

(*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Wait what? Why matrix multiplication. I thought we were all about dot products. Well too bad. I was shocked too when I first read the article. So basically matrix multiplication and dot product are similar but not exactly the same. Here's the best way to explain this:

- For every machine learning task, we typically divide the dataset into batches to make training more efficient and stable. Each batch is a small subset of the full training dataset, and the number of data points in each batch is called the batch size. For example, if we have 50 training samples and a batch size of 2, that results in 25 batches per epoch — meaning the model performs 25 forward and backward passes to complete one full pass through the dataset. In general, the number of batches per epoch is equal to the total number of samples divided by the batch size; the batch size is the number of samples processed in one forward/backward pass; and an epoch is one complete pass through the entire dataset via all of its batches. (ChatGPT rephrased)
- The sequence length represents the length of the sequence in tokens (we explained this earlier)
- Heads - Number of heads - the number of heads in our multi head attention mechanism - this is a controllable hyperparameter (a parameter whose value is set before the machine learning process begins such as the learning rate).
- d_k - Embedding dimensions - must be divisible by the number of heads. This is simple.

  So our embedding dimensions add context about the word (represented as 1 dimensional vectors).

Okay, so now that we have that out of our way, the Q, K, and V tensors have the dimensions [Batch Size, Heads, Sequence Length, d_k]. So here is the brilliance of transformer architectures.

When we are using models, we have a batch size of 1. When we are training models, the number of batches = training size/batch size (obviously). Let's provide an example:

```
import torch

# Batch of 2 instances, 1 head, 3 tokens, embedding dim 2
Q = torch.tensor([[[[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]]],   # Instance 1 queries
                  [[[0.0, 1.0], [1.0, 0.0], [1.0, -1.0]]]]) # Instance 2 queries
```

```
K = torch.tensor([[[[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]]],   # Instance 1 keys
                  [[[0.0, 1.0], [1.0, 0.0], [1.0, -1.0]]]]) # Instance 2 keys

V = torch.tensor([[[[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]]],   # Instance 1 values
                  [[[6.0, 5.0], [4.0, 3.0], [2.0, 1.0]]]])  # Instance 2 values
```

Okay, so it's very simple. The 2D matrices inside the 4D tensor are the embedding dimensions of the incoming tokens (all of which must have the same size). More advanced models have larger embedding vectors (more context assigned to a specific token). Also, inside each 4D tensor is a 2D tensor (with dimensions embedding dimensions x tokens), 3 1D tensors (with dimensions 1 x embedding dimensions), and a 3D tensor (with dimensions tokens x number of heads). The outermost layer of the 4D tensor is the batch size, the second outermost layer of the 4D tensor is the number of heads, the third most outermost layer of the 4D tensor is the number of tokens, and the innermost layer of the 4D matrix is the embedding dimensions.

So…Why is this important? Okay, so I don't know if we've noticed but ChatGPT is really fast - both for training (you probably haven't noticed) and for general use. Why? Because it uses vectorized matrix multiplication. This means that GPTs take in the vectorized representation of tokens of the same length and apply multihead attention to them all at once. So instead of using the dot product for each token in training or in general use (takes forever), we just do matrix multiplication and multiply everything at once.

So you may be asking. How on earth do we take the transpose of a 4 dimensional matrix? Good question. It's simple, if our matrix has dimensions x by y by z by a, we adjust the dimensions so that we get the new shape, x by y by a by z. In our case, we would take the Keys tensor with dimensions Batch Size by Number of Heads by Sequence Length by Embedding Dimensions into a tensor with dimensions Batch Size by Number of Heads by Embedding dimensions by Sequence Length.

Okay, I'm done. Back to the article.

"One aspect we haven't discussed yet is the scaling factor of $1 / \sqrt{d_k}$. This scaling factor is crucial to maintain an appropriate variance of attention values after initialization. Remember that we [initialize] our layers with the intention of having equal variance throughout the model, and hence, Q and K might also have a variance close to 1. However, performing a dot product over two vectors with a variance of σ^2 results in a scalar having d_k times higher variance [a scalar is a constant that the matrix is multiplied by, not necessarily subject to the rules of matrix multiplication - here the variance (standard deviation) for the two variables - Q and K^T is equal to σ^2. When we multiply σ^2 by itself (variance of Q multiplied by the variance of K^T) we get σ^4]:

$$q_i \sim N(0,\ \sigma^2),\ k_i \sim N(0,\sigma^2)\ \rightarrow Var(\sum_{i=1}^{d_k} q_i * k_i)\ =\ \sigma^4 * d_k$$

If we do not scale down the variance back to $\sim \sigma^2$, the softmax over the logits will already saturate to 1 for one random element and 0 for all others. The gradients through the softmax will be close to zero so that we can't learn the parameters appropriately. Note that the extra factor of σ^2, i.e., having σ^4 instead of σ^2, is usually not any issue, since we keep the original variance of σ^2 close to 1 anyways.

The block [titled] Mask (opt.) in the diagram above represents the optional masking of specific entries in the attention matrix. This is for instance used if we stack multiple sequences with different lengths into a batch [this goes back to the idea that for each batch you must be working with the same sequence length / number of tokens]. To still benefit from parallelization in PyTorch, we pad the sentences to the same length and mask out the padding tokens during the calculation of the attention values. This is usually done by setting the respective attention logits to a very low value.

After we have discussed the details of the scaled dot product attention block, we can write a function below which computes the output features given the triple of queries, keys, and values:

```
def scaled_dot_product(q, k, v, mask=None):
    d_k = q.size()[-1]
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits = attn_logits / math.sqrt(d_k)
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -9e15)
    attention = F.softmax(attn_logits, dim=-1)
    values = torch.matmul(attention, v)
    return values, attention
```

Note that our code above supports any additional dimensionality in front of the sequence length so that we can also use it for batches. However, for a better understanding, let's generate a few random queries, keys, and value vectors, and calculate the attention outputs:

```
seq_len, d_k = 3, 2
pl.seed_everything(42)
q = torch.randn(seq_len, d_k)
k = torch.randn(seq_len, d_k)
v = torch.randn(seq_len, d_k)
values, attention = scaled_dot_product(q, k, v)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Values\n", values)
```

```
print("Attention\n", attention)
```

Before continuing, make sure you can follow the calculation of the specific values here, and also check it by hand. It is important to fully understand how the scaled dot product attention is calculated." (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

So let's do just that. We will go through each line of code:

"

```
def scaled_dot_product(q, k, v, mask=None):
    d_k = q.size()[-1]
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits = attn_logits / math.sqrt(d_k)
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -9e15)
    attention = F.softmax(attn_logits, dim=-1)
    values = torch.matmul(attention, v)
    return values, attention
```

" (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Okay. Great. So basically, we are defining the function, scaled_dot_product, with the arguments q (the query values), k (the key values), and v (the value values).

First off, we define d_k (the embedding dimensions) as q.size()[-1] - this means that we take the size / dimensions of q. q has dimensions (batch_size, seq_len, d_k) and q.size() returns the shape of the query tensor. q.size()[-1] accesses the last dimension of q, which has dimensions d_k.

Then, we calculate the attention logits by using the torch matrix multiplication function. We multiply q with dimensions (batch_size, num_heads, sequence_length, embedding_dimensions) by the transpose of k (both q and k have the same original dimensions) which has dimensions (batch_size, num_heads, embedding_dimensions, sequence_length). This is a vectorized form instead of dot producing each individual head - we just take the dot product of all the heads all at once. Then, we calculate the scaled dot product by dividing by the square root of the embedding dimensions - giving us our final attention logits.

Now that we have that, we take the softmax of the attention logits. We apply the softmax across dim = -1 (which corresponds to the sequence length). Applying the softmax along these dimensions means for each query, the scores over all keys are converted into probabilities.

Finally, we multiply the softmax modified attention logits by the values tensor. Ok, and for the next piece of the code:

“

```
seq_len, d_k = 3, 2
pl.seed_everything(42)
q = torch.randn(seq_len, d_k)
k = torch.randn(seq_len, d_k)
v = torch.randn(seq_len, d_k)
values, attention = scaled_dot_product(q, k, v)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Values\n", values)
print("Attention\n", attention)
```

”

(*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

So, here we define our sequence length and embedding dimensions. We have sequence length 3 and embedding dimensions 2. The pl.seed_everything() function ensures that operations involving randomness within PyTorch, such as weight initialization or data sampling, produce the same results across different runs when the same seed is used (this is a little complicated - I still don’t fully understand why this is necessary). Then we assign the values for the q, k, and v tensors randomly. We then calculate our values and attention using the scaled_dot_product function.

Multi Head Attention:

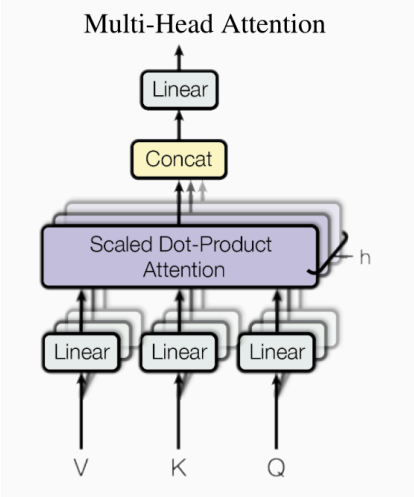
Ok, so that wasn’t actually all that bad. Let’s go to the next section of the article.

“The scaled dot product attention allows a network to attend over a sequence. However, often there are multiple different aspects a sequence element wants to attend to, and a single weighted average is not a good option for it. This is why we extend the attention mechanism to multiple heads, i.e. multiple different query-key-value-triplets on the same features. Specifically, given a query, key, and value matrix, we transform those into *h* sub-queries, sub-keys, and sub-values, which we pass through the scaled dot product attention independently. Afterward, we concatenate [(e.g. concatenating [1 2 3] and [4 5 6] gives you [1 2 3 4 5 6]) the heads and combine them with a final weight matrix. Mathematically, we can express this operation as

$$Multihead(Q,K,V) = Concat(head_1, \ldots, head_h)W^O$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

We refer to this as Multi-Head Attention layer with learnable parameters $W^{Q}_{i...h} \in \mathbb{R}^{D*d_k}$, $W^{K}_{i...h} \in \mathbb{R}^{D*d_k}$, $W^{V}_{i...h} \in \mathbb{R}^{D*d_v}$, and $W^{0}_{i...h} \in \mathbb{R}^{h*d_v*d_{out}}$ (D being the input dimensionality [and dout being the dimensions of the output]). Expressed in a computational graph, we can visualize it as below (figure credit - Vaswani et al.)



How are we applying a Multi-Head attention layer in a neural network where we don't have an arbitrary query, key, and value vector as input? Looking at the computation graph above, a simple but effective implementation is to set the current feature map in a NN, $X \in \mathbb{R}^{BxTxd_{model}}$, as Q, K and V (B being the batch size, T the sequence length, d_{model} the hidden dimensionality of X). The consecutive weight matrices, W^Q, W^K, W^V can transform X to the corresponding feature vectors that represent the queries, keys, and values of the input. Using this approach we can implement the Multi-Head Attention Module below.

```
# Helper function to support different mask shapes.
# Output shape supports (batch_size, number of heads, seq length, seq length)
# If 2D: broadcasted over batch size and number of heads
# If 3D: broadcasted over number of heads
# If 4D: leave as is
def expand_mask(mask):
    assert mask.ndim >= 2, "Mask must be at least 2-dimensional with seq_length x
seq_length"
    if mask.ndim == 3:
        mask = mask.unsqueeze(1)
    while mask.ndim < 4:
        mask = mask.unsqueeze(0)
    return mask
```

” (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

So here, it’s quite simple. If our tensor dimensions are two dimensional, we broadcast the tensor dimensions over the number of heads and batch size. If our tensor dimensions are three dimensional, we broadcast the tensor dimensions across the number of heads, and if our tensor dimensions are four dimensional we leave the tensor dimensions as is. Okay so that’s very basic. Like it’s not complicated - we just need to ensure that we have a 4 dimensional tensor with dimensions: batch size x number of heads x sequence length (in tokens) x embedding dimensions. Here is where everything gets complicated:

“

```
class MultiheadAttention(nn.Module):

    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()
        assert embed_dim % num_heads == 0, "Embedding dimension must be 0 modulo number of heads."

        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        # Stack all weight matrices 1...h together for efficiency
        # Note that in many implementations you see "bias=False" which is optional
        self.qkv_proj = nn.Linear(input_dim, 3*embed_dim)
        self.o_proj = nn.Linear(embed_dim, input_dim)

        self._reset_parameters()

    def _reset_parameters(self):
        # Original Transformer initialization, see PyTorch documentation
        nn.init.xavier_uniform_(self.qkv_proj.weight)
        self.qkv_proj.bias.data.fill_(0)
        nn.init.xavier_uniform_(self.o_proj.weight)
        self.o_proj.bias.data.fill_(0)

    def forward(self, x, mask=None, return_attention=False):
        batch_size, seq_length, _ = x.size()
        if mask is not None:
            mask = expand_mask(mask)
        qkv = self.qkv_proj(x)

        # Separate Q, K, V from linear output
        qkv = qkv.reshape(batch_size, seq_length, self.num_heads, 3*self.head_dim)
        qkv = qkv.permute(0, 2, 1, 3) # [Batch, Head, SeqLen, Dims]
        q, k, v = qkv.chunk(3, dim=-1)

        # Determine value outputs
        values, attention = scaled_dot_product(q, k, v, mask=mask)
        values = values.permute(0, 2, 1, 3) # [Batch, SeqLen, Head, Dims]
```

```
        values = values.reshape(batch_size, seq_length, self.embed_dim)
        o = self.o_proj(values)

        if return_attention:
            return o, attention
        else:
            return o
```

" (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Okay so we create a class called "MultiheadAttention" which is a subclass of nn.Module. We initialize the class for the variables "input_dim," "embed_dim," and "num_heads," We also need to confirm that the number of embedding dimensions is divisible by the number of heads. The reason why this is important is because if we have 8 heads for example, and we have 64 embedding dimensions, we want to split the embedding dimensions into 8 heads with 8 embedding dimensions each. After that we define "self" for the embed_dim, num_heads, head_dim. Then, and here is where it gets complicated, we "stack all weight matrices 1…h together for efficiency," (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.) so self.qkv_project basically does nn.Linear(input_dim, 3 * embed_dim) where input_dim is the number of inputs and 3*embed_dim is the number of neurons in one hidden layer. Then we find o_proj (the output) which is nn.Linear(embed_dim, input_dim) where embed_dim is the number of input dimensions (in the previous hidden layer for the output layer), and input_dim is the number of neurons for the output (meaning that the number of neurons in both the input and output layers are exactly the same).

Now that we have initialized the MultiheadAttention class, we need to define the functions within the class. So for the reset_paramers, we use the function nn.init.xavier_uniform_() - which takes the argument tensor, gain, and generator. So our argument is the weight matrices of the queries, keys, and values. So on doc.pytroch.org, "the method is described in *Understanding the difficulty of training deep feedforward neural networks* - Glorot, X. & Bengio, Y. (2010). The resulting tensor will have values sampled from $\upsilon(-\ a,\ a)$ where

$a\ =\ gain\ x\sqrt{\frac{6}{[fan.in + fan.out]}}$ " (*Torch.nn.init — PyTorch 2.7 Documentation*, 2024)

Okay, so the fan in and fan out is the only thing that is really confusing for me. It wasn't clearly described in the torch documentation but I assume that fan in means condensing all the values of the previous layer and fan out means expanding all the values of the previous layer. Right now we won't worry about that. If I learn more about the fanning in and out, I will describe it further. Then, we do self.qkv_proj.bias.data.fill_(0) - meaning we set all bias values equal to 0 (refer to Notes Document III if you don't know what biases are). We then do the same for the output layer

- both with the xavier_uniform_() function and the bias.data.fill_(). So there we go, we've reset the parameters.

Now we are about to go through the forward pass of the multi head attention model. Okay, so we find the size of batch_size, seq_length, but we exclude input_dim because those remain constant. Then we have a mask function. If the mask function does not exist, we do not do anything related to the mask function. If the mask function is there, we will set all values for a sequence whose length is not equal to the length expected by the mask with missing or additional values equal to 0 for those missing or additional values. Then, we define qkv, where we adjust for the input x.

After we do this, we "Separate Q, K, V from [the] linear output" (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.). Then we reshape qkv for the batch size and sequence length (adjusted by the input - x) while keeping self.num_heads and 3*self.head_dim. Okay, then we adjust the dimensions of the qkv tensor from [0, 1, 2, 3] to [0, 2, 1, 3] - so from [Batch, Sequence Length, Number of Heads, 3*Head Dimensions] to [Batch, Number of Heads, Sequence Length, 3*Head Dimensions]. Next, we split the concatenated tensor (remember we concatenated the queries, keys, and values tensors), into three tensors. In other words we go from 1 tensor with [Batch, Sequence Length, Number of Heads, 3*Head Dimensions] to 3 tensors with [Batch, Sequence Length, Number of Heads, Head Dimensions].

We are almost done with this section of the code. Okay, so we defined our scaled_dot_product function earlier so that's already done. From there we just determine the values and attention values. We then adjust the dimensions of the values to be Batch x Sequence Length x Head x Dimensions. We then reshape the values (getting rid of the number of heads) to have the dimensions batch size, sequence length, and embedding dimensions. We then return the outputs for the values. If we want to return the attention as well, we will return the output and the attention values; otherwise, we will just return the outputs. Okay so that's enough from me for now - back to the article.

"One crucial characteristic of multi-head attention is that it is permutation-equivariant with respect to its inputs [permutation equivariance "means a model's output changes in the same way as the input when the input's element order is permuted." (Alelab, 2020, 7:45)] This means that if we switch two input elements in the sequence, e.g. $X_1 \leftrightarrow X_2$ (neglecting the batch dimension for now), the output is exactly the same besides the elements 1 and 2 switched. Hence, the multi-head attention is actually looking at the input not as a sequence, but as a set of elements. This property makes the multi-head attention block and the Transformer architecture so powerful and widely applicable! But what if the order of the input is actually important for solving the task, like language modeling? The answer is to encode the position in the input features, which we will take a closer look at ladder (topic *Positional Encodings* below).

Before moving on to creating the Transformer architecture, we can compare the self-attention operation with our other common layer competitors for sequence data: convolutions and recurrent neural networks. Below you can find a table by Vaswani et al. (2017) on the complexity per layer, the number of sequential operations, and maximum path length. The complexity is measured by the upper bound of the operations to perform, while the maximum path length represents the maximum number of steps a forward or backward signal has to traverse to reach any other position. The lower this length, the better gradient signals can backpropagate for long-range dependencies. Let's take a look at the table below:

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

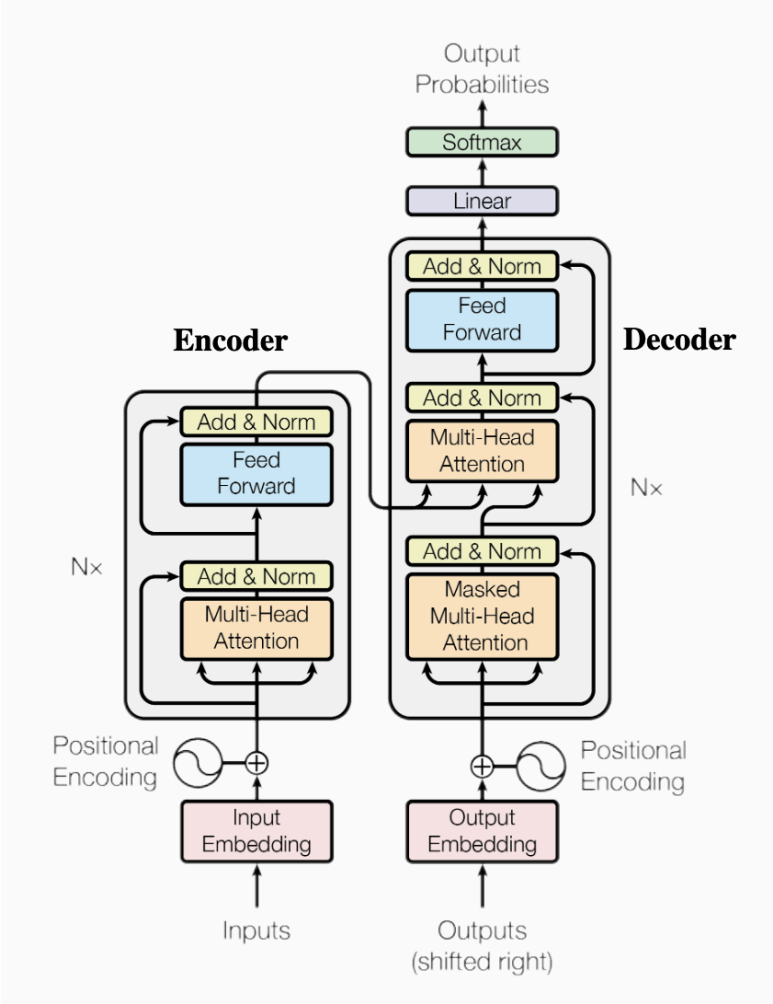
n is the sequence length, d is the representation dimension and k is the kernel size of convolutions. In contrast to recurrent networks, the self-attention layer can parallelize all its operations making it much faster to execute for smaller sequence lengths. However, when the sequence length exceeds the hidden dimensionality, self-attention becomes more expensive than RNNs. One way of reducing the computational cost of long sequences is by restricting the self-attention to a neighborhood of inputs to attend over, denoted by *r*. Nevertheless, there has been recently a lot of work on more efficient Transformer architectures that still allow long dependencies, of which you can find an overview in the paper by Tay et. al (2020) if interested." (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

So that's Multi-Head attention. Good job to everyone who has understood everything up to this point. Let's continue with the Transformer Encoder.

Transformer Encoder:

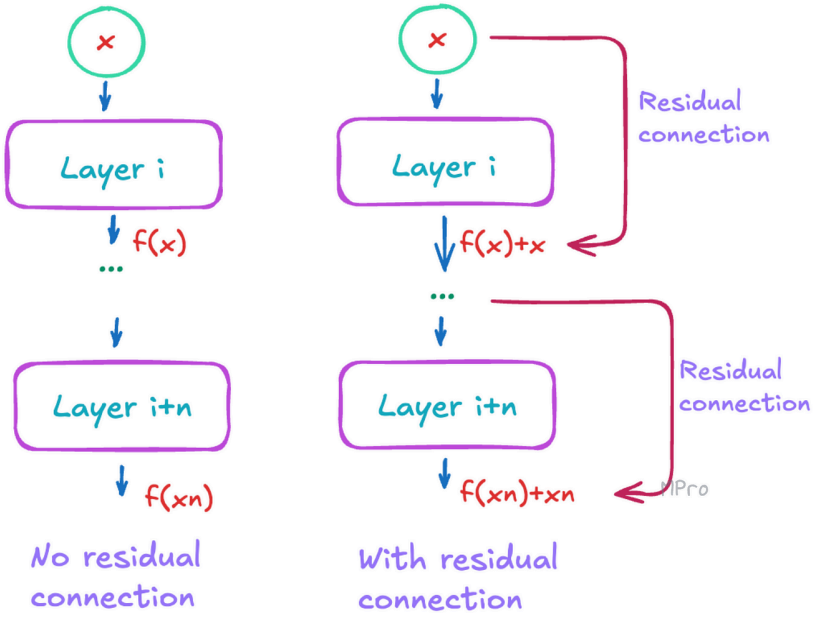
"Next, we will look at how to apply the mutli-head attention block inside the Transformer architecture. Originally, the Transformer model was designed for machine translation. Hence, it got an encoder-decoder structure where the encoder takes as input the sentence in the original language and generates an attention-based representation. On the other hand, the decoder attends over the encoded information and generates the translated sentence in an autoregressive manner, as in a standard RNN. While this structure is extremely useful for Sequence-to-Sequence tasks with the necessity of autoregressive decoding, we will focus here on the encoder part. Many advances in NLP [(natural language processing)] have been made using pure encoder-based

Transformer models (if interested, models include the BERT-family, the VIison transformer, and more), and in our tutorial, we will also mainly focus on the encoder part. If you have understood the encoder architecture, the decoder is a very small step to implement as well. THe full transformer architecture looks as follows (figure credit - Vaswani et al., 2017).:



The encoder consists of *N* identical blocks that are applied in sequence. Taking as input *x*, it is first passed through a Multi-Head Attention block as we have implemented above. The output is added to the original input using a residual connection, and we apply a consecutive Layer Normalization on the sum. Overall, it calculates LayerNorm(x + Multihead(x, x, x)) (x being Q, K and V input to the attention layer). The residual is crucial in the Transformer architecture for two reasons. [Here is an example of a residual connection (image credit - https://medium.com/@mariaprokofieva/attention-in-transformers-residual-connection-layer-a-sh

ortcut-that-makes-it-work-165b52566167) :



]

1. Similar to ResNets, Transformers are designed to be very deep. Some models contain more than 24 blocks in the encoder. Hence, the residual connections are crucial for enabling a smooth gradient flow in the model. [In other words, as the model keeps going through the transformer layers, it is likely that some original information will vanish - information that is both relevant and likely necessary. Thus, we use residual connections to ensure that this information is kept (to some extent). You will see more about this below]
2. Without the residual connection, the information about the original sequence is lost. Remember that the Mult-Head Attention lawyer ignores the position of elements in a sequence, and can only learn it based on the input features. Removing the residual connections would mean that this information is lost after the first attention layer (after initialization), and with a randomly initialized query and key vector, the output are likely to represent similar/same, and there is no chance for the model to distinguish which information came from which input element. An alternative option to residual connection would be to fix at least one head to focus on its original input, but this is very inefficient and does not have the benefits of improved gradient flow.

The Layer Normalization also plays an important role in the Transformer architecture as it enables faster training and provides small regularization. Additionally, it ensures that the features are in a similar magnitude among the elements in the sequence. We are not using Batch Normalization because it depends on the batch size which is often small with Transformers (they require a lot of GPU memory), and BatchNorm has shown to perform particularly bad in

language as the features of words tend to have a much higher variance (there are many, very words which need to be considered for a good distribution estimate).

Additionally to the Multi-Head Attention, a small fully connected feed-forward network is added to the model, which is applied to each position separately and identically. Specifically, the model uses a Linear —> ReLU —> Linear MLP. The full transformation including the residual connection can be expressed as:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$
$$x = Layernorm(x + FFN(x))$$

[*Note that MLP stands for mutli-layer perception - Linear MLP's are FeedForward Networks where a linear transformation is followed by a nonlinear activation function.] This MLP adds extra complexity to the model and allows transformations on each sequence element separately. You can imagine as this allows the model to "post-process" the new information added by the previous Multi-Head Attention, and prepare for the next attention block. Usually, the inner dimensionality of the MLP is 2-8 x larger than d_{model}, i.e. the dimensionality of the original input x. The general advantage of a wider layer instead of a narrow, multi-layer MLP is the faster, parallelizable execution.

Finally, after looking at all parts of the encoder architecture, we can start implementing it below. We first start by implementing a single encoder block. Additionally to the layers described above, we will add dropout layers in the MLP and on the output of the MLP and Multi-Head Attention for regularization.

```
class EncoderBlock(nn.Module):

    def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
        """
        Inputs:
            input_dim - Dimensionality of the input
            num_heads - Number of heads to use in the attention block
            dim_feedforward - Dimensionality of the hidden layer in the MLP
            dropout - Dropout probability to use in the dropout layers
        """
        super().__init__()

        # Attention layer
        self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)

        # Two-layer MLP
        self.linear_net = nn.Sequential(
            nn.Linear(input_dim, dim_feedforward),
            nn.Dropout(dropout),
            nn.ReLU(inplace=True),
```

```
            nn.Linear(dim_feedforward, input_dim)
        )

        # Layers to apply in between the main layers
        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Attention part
        attn_out = self.self_attn(x, mask=mask)
        x = x + self.dropout(attn_out)
        x = self.norm1(x)

        # MLP part
        linear_out = self.linear_net(x)
        x = x + self.dropout(linear_out)
        x = self.norm2(x)

        return x
```

" (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Okay, time for one (of many more) coding explanation sessions. Okay, so we initialize the EncoderBlock() class which is a subclass of nn.Module. We define the initialization of the Encoder Block Class, which will take the inputs, input_dim, num_heads, dim_feedforward, dropout = 0.0). We then define self.self_attn which is equal to the output of the MultiheadAttention() class described earlier.

Then we define the self.linear_net where we use the sequential API key (from PyTorch this time) with two layers - the layer that connects the input and hidden layer with dimensions input_dim and the dim_feedforward and the layer that connects the hidden layer and the output layer with dimensions dim_feedforward, input_dim. We also apply dropout between the input and hidden layer, which, for those of you who don't know, randomly gives weights values of zero to prevent overfitting. We also apply the ReLU activation function for the hidden layer.

Now, we have to define "[l]ayers to apply in between the main layers." These layers are the two normalization layers and the dropout layer where we simply take the Layer Normalization of the two forward passes in the Two-Layer MLP and then apply the dropout layer to prevent overfitting.

After defining the ___init__() function, we define the forward() function, which takes input x, and has no mask. We define the "Attention part" with variables attn_out which is the self_attn of the input where the mask is applied (if applicable), we then modify x by adding the attention output values (with dropout) and then normalize x. We repeat this process, now for the "MLP part,": where we apply the "Two-layer MLP" to the modified x value. We then add the values of the linear output (with dropout) to x and redefine x. Then, we normalize x. Ultimately, our class returns x.

“Based on this block, we can implement a module for the full Transformer encoder. Additionally to a forward function that iterates through the sequence of encoder blocks, we also provide a function called get_attention_maps. The idea of this function is to return the attention probabilities for all Multi-Head Attention blocks in the encoder. This helps us in understanding and in a sense, explaining the model. However, the attention probabilities should be interpreted with a grain of salt as it does not necessarily reflect the true interpretation of the model (there is a series of papers about this, including Attention is not Explanation and Attention is not not Explanation).

```
class TransformerEncoder(nn.Module):

    def __init__(self, num_layers, **block_args):
        super().__init__()
        self.layers = nn.ModuleList([EncoderBlock(**block_args) for _ in
range(num_layers)])

    def forward(self, x, mask=None):
        for l in self.layers:
            x = l(x, mask=mask)
        return x

    def get_attention_maps(self, x, mask=None):
        attention_maps = []
        for l in self.layers:
            _, attn_map = l.self_attn(x, mask=mask, return_attention=True)
            attention_maps.append(attn_map)
            x = l(x)
        return attention_maps
```

” (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Okay, we are back to the code explanation only after a paragraph, sorry. We define the class “TransformerEncoder” which is a subclass of nn.Module. We initialize the class with __init__ function that takes arguments self, num_layers, and **block_args. We define “self.layers” with a function from the PyTorch library called nn.ModuleList, which ensures that all parameters within its submodules are visible to PyTorch’s optimization and model-saving mechanisms (e.g., model.parameters(), model.state_dict()). This is crucial for training and managing your neural network models (taken from the google AI overview). In other words, regular python lists won’t store this information properly. To fit it, we use the encoder block function with argument **block_args and iterate for the number of times equal to the number of layers.

Then, we define the forward function with a for loop for l in self.layers where we use l(x, mask = mask), where we call the forward function of PyTorch. This works because PyTorch’s nn.Module implements the __call__ method, which is equivalent to l.__call__(x) which internally calls l.forward(x).

Finally, for the get_attention_maps function, the function takes the arguments self, x, and mask = None. We define the attention_maps variable as a list. We then use a for loop - for l in self.layers, we return the attn_map but ignore the output tensor. Since we are using the MultiHeadAtention class from earlier, the class expects to return the output tensor - however, we only want the attention. Thus, we use _, to ignore the output tensor while acknowledging its presence (because if not the l.self_attn() would return the output tensor and the attention for the attn_map variable). We use the self.self_attn() function defined in the encoder block (where l is self) to find the attention for a specific instance l - we do not run the full encoder block jus he self_attn module. Since the attention map is that of all instances, we append each attention value for each instance, giving us the complete attention map. Then we just repeat that x = l(x) or l.forward(x) - the entire forward block. According to ChatGPT, we use l(x) instead of l.forward(x) because l(x) invokes some extra PyTorch internals (like hooks) and ensures consistent behavior, while calling forward() directly skips those. So the preferred usage is l(x). Then, as rephrased by ChaGPT, we return the list of attention maps collected from all layers.

Another benchmark hit. We have finished the Transformer Encoder. Congrats everyone! We are far from done though. Taking a quick look at the article, we aren't even half way through. Let's move on to Positional Encoding.

Positional Encoding:

"We have discussed before that the Multi-Head Attention block is permutation-equivariant, and cannot distinguish whether an input comes before another one in the sequence or not. In tasks like language understanding, however, the position is important for interpreting the input words. The position information can therefore be added via input features. We could learn [an] embedding for every possible position, but this would not generalize to a dynamical input sequence length. [So, in other words, the model needs to understand the position of each input - otherwise, position is not taken into account. Without positional encoding, according to ChatGPT, the model thinks "The cat sat down" and the "cat down sat the" are the same phrase.] Hence, the better option is to use feature patterns that the network can't identify from the features and potentially generalize to larger sequences. The specific pattern chosen by Vaswani et A. are sine and cosine functions of different frequencies, as follows:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{\text{model}}}}\right) & \text{if } i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{\text{model}}}}\right) & \text{otherwise} \end{cases}$$

$PE_{(pos,\,i)}$ represents the position encoding at position pos in the sequence, and hidden dimensionality i. These values, concatenated for all hidden dimensions, are added to the original

input features (in the Transformer visualization above, see "Positional encoding"), and constitute the position information. We distinguish between even (i mod 2 = 0) and uneven (i mod 2 = 1) hidden dimensionalities where we apply a sine/cosine respectively. The intuition behind this encoding is that you can represent $PE_{(pos+k,:)}$ as a linear function of $Pe_{(pos,:)}$, which might allow the model to easily attend to relative positions. The wavelengths in different dimensions range from 2π to 10000*2π." (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Okay, so before we continue to the code and the explanation of the code, we are quickly going to address what was previously stated. We are going to use ChatGPT, as usual, to help us understand this. *Pos* is a specific word in the sequence (i.e., the 3rd or 4th word) and *i* is a specific dimension of the model's hidden representation (the 10th element in a 512 dimensional vector - where the dimensions of the 512 dimensional vector determined are determined by the number of embedding dimensions). Then the thing about $PE_{(pos+k,:)}$ being a linear function of $Pe_{(pos,:)}$ can be expressed / explained mathematically - I will do so below. But, the key idea is very simple: the relationship between the inputs for the sin and cos functions for the first word in the sequence and the third word in the sequence (both are odd numbers in the sequence) is linear - meaning that the rate of increase is linear. However, the actual positional encoding themselves vary sinusoidally - meaning that they go up and down like a wave pattern. Okay, now let's get into why this is a linear relationship - this took me like 20 minutes to really understand and I will have to quote ChatGPT directly to explain the math.

First of all, let's recap the key formulas (with regular i):

"For any *i*, define

- If *i* is even:
  $$PE(pos,\ i) = sin(\frac{pos}{10000^{i/d_{model}}})$$
- If *i* is odd:
  $$PE(pos,\ i) = cos(\frac{pos}{10000^{(i-1)/d_{model}}})$$

For *pos* + k, and define

$$a = \frac{pos}{10000^{[i/2]/d_{model}}} \qquad b = \frac{k}{10000^{[i/2]/d_{model}}}$$

where [i/2] gives the "pair index' corresponding to *i*.

Then:

- For even i:

$$PE(pos + k, i) = sin(a + b) = sin(a)cos(b) + cos(a)sin(b) = PE(pos, i)cos(b) + PE(pos, i - 1)sin(b)$$

- For odd i:

$$PE(pos + k, i) = cos(a + b) = cos(a)cos(b) - sin(a)sin(b) = PE(pos, i)cos(b) - PE(pos, i + 1)sin(b)$$

Remember, sina = PE(*pos*, *i*) for the even numbers and cosa = PE(*pos*, *i*) for the odd numbers.

Step 1: Group Dimensions into Pairs
Consider the pair of dimensions i - 1, i (with i even). At position *pos*:

$$v(pos) = \begin{pmatrix} PE(pos, i-1) \\ PE(pos, i) \end{pmatrix} = \begin{pmatrix} cos(a) \\ sin(a) \end{pmatrix}$$

At position $pos + k$:

$$v(pos) = \begin{pmatrix} PE(pos+k, i-1) \\ PE(pos\_k, i) \end{pmatrix} = \begin{pmatrix} cos(a+b) \\ sin(a+b) \end{pmatrix}$$

Step 2: Express v($pos + k$) as a matrix multiplication of v(pos)

From the angle addition formulas:

$$cos(a + b) = cos(a)cos(b) - sin(a)sin(b)$$

$$sin(a + b) = sin(a)cos(b) + cos(a)sin(b)$$

Therefore,

$$v(pos+k) = \begin{pmatrix} \cos(a+b) \\ \sin(a+b) \end{pmatrix} = \begin{pmatrix} \cos b & -\sin b \\ \sin b & \cos b \end{pmatrix} \begin{pmatrix} \cos a \\ \sin a \end{pmatrix} = R(b)v(pos)$$

$$R(b) = \begin{pmatrix} \cos b & -\sin b \\ \sin b & \cos b \end{pmatrix}$$

where

$$R(b) = \begin{pmatrix} \cos b & -\sin b \\ \sin b & \cos b \end{pmatrix}$$

Is a rotation matrix.

Step 3: Interpretation

- The rotation matrix R(b) is a linear operator
- Applying R(b) to v(*pos*) yields v(*pos* + *k*)
- This means the vector encoding at position *pos* + *k* is a linear transformation (specifically, a rotation) of the vector encoding at position *pos*

Step 4: Why is this linearity important?

- Linearity here means no nonlinear terms like powers or products of the components of v(*pos*) are needed to get v(*pos*+*k*)
- Instead, each component at *pos*+*k* is a sum of the components at *pos* multiplied by constants (cos(b), sin(b)), which is the definition of a linear function

Step 5: What about the whole positional encoding vector?

- The full positional encoding vector is concatenated form many such pairs v_i for different frequencies (different *i*).
- Each pair undergoes a different rotation R(b_i) corresponding to different frequencies
- Hence, the entire positional encoding at *pos* + *k* is obtained by applying a block-diagonal linear operator (made of these rotation matrices) to the positional encoding at *pos.”* (ChatGPT - with some very, very small adjustments such as adding () for sina to be sin(a))

I would recommend that you look through what ChatGPT wrote because it is very useful. Also, as we can tell, this process seems to be ridiculously computationally intensive. To mitigate the

computational cost we can vectorize these matrices into tensors. In other words, if each matrix is 2 dimensional, we can vectorize all those matrices into a 3 dimensional tensor. Okay, now back to the article.

“The positional encoding is implemented below. The code is taken from the PyTorch tutorial about transformers on NLP and adjusted for our purposes.

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model, max_len=5000):
        """
        Inputs
            d_model - Hidden dimensionality of the input.
            max_len - Maximum length of a sequence to expect.
        """
        super().__init__()

        # Create matrix of [SeqLen, HiddenDim] representing the positional encoding
for max_len inputs
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0)
/ d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)

        # register_buffer => Tensor which is not a parameter, but should be part of
the modules state.
        # Used for tensors that need to be on the same device as the module.
        # persistent=False tells PyTorch to not add the buffer to the state dict (e.g.
when we save the model)
        self.register_buffer('pe', pe, persistent=False)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1)]
        return x
```

” (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Okay so great, we have our code for the positional encoding, let’s explain it. We define a class called "PositionalEncoding" that is a subclass of nn.Module. We then initialize the class with the __init__ function which takes the argument self, d_model, and the max_len (which is set to 5000). According to the article, “d_model [is the h]idden dimensionality of the input” and “max_len [is the m]aximum length of a sequence to expect.” We then, according to the article, “[c]reate matrix of [SeqLen, HiddenDim] representing the positional encoding for max_len inputs. We then define a variable, pe, which is equal to torch.zeros(max_len, d_model), which returns a tensor with dimensions max_len and d_model with scalar values 0 filling the tensor. We then define the position variable. The position variable uses the torch.arrange function to create a

1 dimensional tensor whose dimension is the max sequence length and values are the position of each token in the sequence. We use unsqueeze(1) to add a dimension after the max_len dimension and make the number type "floats" instead of integers. Then, we define the div_term, which is simply $\frac{pos}{10000^{i/d_{model}}}$ for even values and $\frac{pos}{10000^{(i-1)/d_{model}}}$ for odd values. To go into this further, the torch.arrange(0, d_model, 2) creates a tensor with even indices [0, 2, 4, …, d_model -2] if d_model is even and vice versa if the model is odd. Then this tensor is multiplied by -log(10000)/d_model, and exponentiated with torch.exp(...). The div_term is now equal to $10000^{-2i/d\_model}$ which is the same thing as $\frac{pos}{10000^{i/d_{model}}}$. Then, we for all values of pe[:, 0::2] - even values - we apply the positional encoding function for even values - the sin of position * $10000^{i/d\_model}$. And then for pe[:, 1::2] - the odd values - we apply the positional encoding function for odd values - the cos of position * $100000^{(i-1)/d\_model}$. We then use the unsqueeze to shift the dimensions of the positional encoding values, shifting the dimensions from [max_len, d_model] to [1, max_len, d_model].

Okay, now we have to look at what the self.register_buffer is going to do. According to the article "register buffer [is a t]ensor which is not a parameter, but should be part of the modules state[. It is u]sed for tensors that need to be on the same device as the module[.] persistent=False tells PyTorch to not add the buffer to the state dict (e.g., when we save the model)." (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.) So, as we can see here, we define self.register_buffer where, according to ChatGPT, pe is a tensor you want to associate with the module (a nonlearnable parameter like weights or biases). According to ChatGPT, by registering it as a buffer, PyTorch knows to move it automatically to the right device (CPU / GPU) when you call .to(device) or .cuda(). It also includes it in the module's state for tracking (like when you do .eval(), .train() or save/load state dict), unless you specify persistent=False. Buffers are useful for things like fixed tensors during forward passes (e.g., positional encodings in transformers - what we are using it for here), running statistics in batch normalization (running_mean, running_var), and anything you want to move to GPU with the model but don't want to optimize/update (also according to ChatGPT).

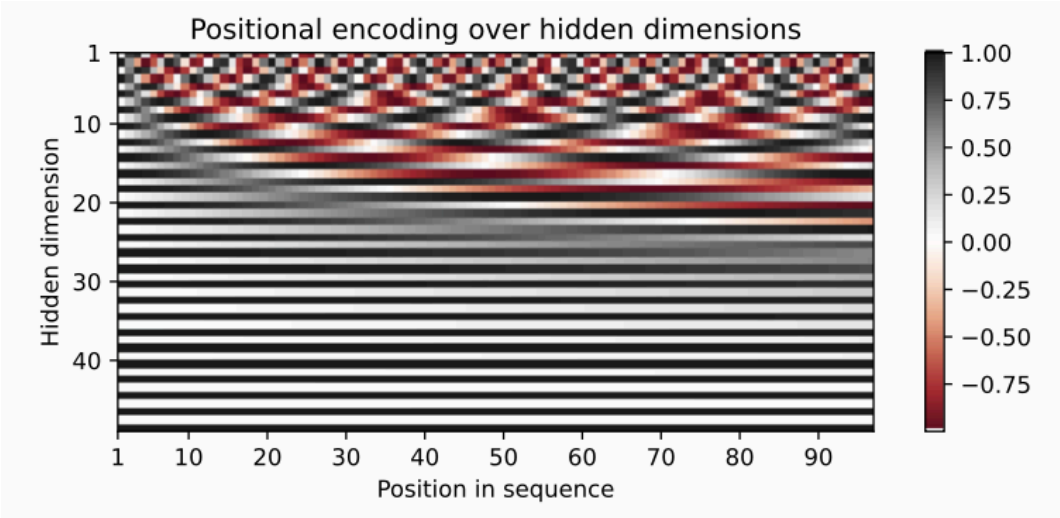
After we have initialized the Positional Encoding class, we define the forward function, which takes the arguments "self" and "x" (the input). We then rewrite x as x + self.pe[:, :x.size(1)]. This, according to ChatGPT, means that from self.pe, we take all elements along the first dimension (:), but slice the second dimension from the start up to x.size(1) - the sequence length of x. Then, we simply return x.

And yeah, that's pretty much it for positional encoding. Note, that when I say "according to ChatGPT" it means that I am basically saying that I copied directly from ChatGPT. Before I continue to another section, I make sure I fully understand the previous section though, which is important. Regardless, let's get back to the article.

"To understand the positional encoding, we can visualize it below. We will generate an image of the positional encoding over hidden dimensionality and position in a sequence. Each pixel,

therefore, represents the change of the input feature we perform to encode the specific position. Let's do it below [(since the code here visualizes the essential code, I won't explain the code)].

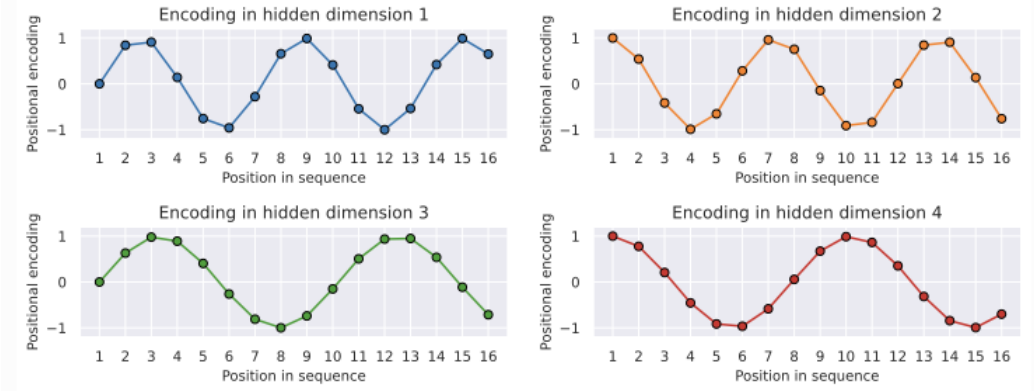
```
encod_block = PositionalEncoding(d_model=48, max_len=96)
pe = encod_block.pe.squeeze().T.cpu().numpy()

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,3))
pos = ax.imshow(pe, cmap="RdGy", extent=(1,pe.shape[1]+1,pe.shape[0]+1,1))
fig.colorbar(pos, ax=ax)
ax.set_xlabel("Position in sequence")
ax.set_ylabel("Hidden dimension")
ax.set_title("Positional encoding over hidden dimensions")
ax.set_xticks([1]+[i*10 for i in range(1,1+pe.shape[1]//10)])
ax.set_yticks([1]+[i*10 for i in range(1,1+pe.shape[0]//10)])
plt.show()
```



You can clearly see the sine and cosine waves with different wavelengths that encode the position in the hidden dimensions. Specifically, we can look at the sine/cosine wave for each hidden dimensionality separately, to get a better intuition of the pattern. Below we visualize hte positional encoding for the hidden dimensions 1,2,3, and 4 [((since the code here visualizes the essential code, I won't explain the code)].

```
sns.set_theme()
fig, ax = plt.subplots(2, 2, figsize=(12,4))
ax = [a for a_list in ax for a in a_list]
for i in range(len(ax)):
    ax[i].plot(np.arange(1,17), pe[i,:16], color=f'C{i}', marker="o", markersize=6,
markeredgecolor="black")
    ax[i].set_title(f"Encoding in hidden dimension {i+1}")
    ax[i].set_xlabel("Position in sequence", fontsize=10)
    ax[i].set_ylabel("Positional encoding", fontsize=10)
    ax[i].set_xticks(np.arange(1,17))
    ax[i].tick_params(axis='both', which='major', labelsize=10)
```

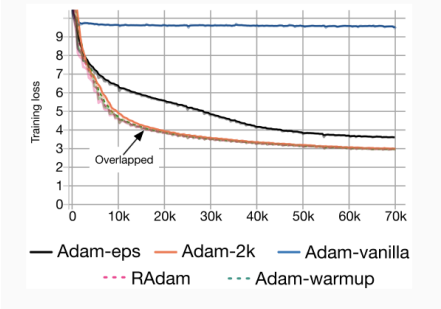
```
    ax[i].tick_params(axis='both', which='minor', labelsize=8)
    ax[i].set_ylim(-1.2, 1.2)
fig.subplots_adjust(hspace=0.8)
sns.reset_orig()
plt.show()
```



As we can see, the patterns between the hidden dimensions 1 and 2 only differ in the starting angle. The wavelength is 2π, hence the repetition after 6. The hidden dimensions 2 and 3 have about twice the wavelength." (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.) Note that by hidden dimension 1, 2,3,4 etc. we are referring to a specific element in the vector of the embedding / hidden dimensions for each token. And then the position in the sequence just refers to the specific token we are referring to.

Learning Rate Warm-up:

"One commonly used technique for training a Transformer is learning rate warm-up. This means that we gradually increase the learning rate from 0 to our originally specified learning rate in the first few iterations. Thus, we slowly start learning instead of taking very large steps from the beginning. In fact, training a deep Transformer without learning rate warm-up can make the model diverge and achieve a much worse performance on training and testing. Take for instance the following plot by Liu et al. (2019) comparing Adam-vanilla (i.e. Adam without warm-up) vs. Adam with a warm-up:



Clearly the warm-up is a crucial hyperparameter in the Transformer architecture. Why is it so important? There are currently two common explanations. Firstly, Adam uses the bias correction

factors which however can lead to a higher variance in the adaptive learning rate during the first iterations. Improve optimizers like RAdam have been shown to overcome this issue, not requiring warm-up training for training Transformers. Secondly, the iteratively applied Layer Normalization across all layers can lead to very high gradients during the first iterations, which can be solved by using Pre-Layer Normalization (similar to Pre-Activation ResNet), or replacing Layer Normalization by other techniques (Adaptive Normalization, Power Normalization).

Nevertheless, many applications and papers still use the original Transformer architecture with Adam, because warm-up is a simple, yet effective way of solving the gradient problem in the first iterations. There are many different schedulers [(schedulers control the learning rate)] we could use. For instance, the original Transformer paper used an exponential decay scheduler with a warm-up. However, the currently most popular scheduler is the cosine warm-up scheduler, which combines warm-up with a cosine-shaped learning rate decay. We can implement it below, and visualize the learning rate factor over epochs.

```
class CosineWarmupScheduler(optim.lr_scheduler._LRScheduler):

    def __init__(self, optimizer, warmup, max_iters):
        self.warmup = warmup
        self.max_num_iters = max_iters
        super().__init__(optimizer)

    def get_lr(self):
        lr_factor = self.get_lr_factor(epoch=self.last_epoch)
        return [base_lr * lr_factor for base_lr in self.base_lrs]

    def get_lr_factor(self, epoch):
        lr_factor = 0.5 * (1 + np.cos(np.pi * epoch / self.max_num_iters))
        if epoch <= self.warmup:
            lr_factor *= epoch * 1.0 / self.warmup
        return lr_factor
```

(*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Okay, we do have to explain this code. So we define a class titled "CosineWarmupScheduler" which is a subclass of (optim.lr_scheduler._LRSscheduler) - note that I might have been mixing up argument and class (where the class we define is a subclass of a larger class), so apologies for that. Then, we initialize the class with the function __init__ which takes the arguments self, optimizer, warmup, and max_iters. We define self.warmup and self.max_num_iters as warmup and max_iters respectively. Then from the parent class - optim.lr_scheduler._LRSscheduler - we call the constructor and it handles things like self.optimizer, self.base_lrs, and self.last_epoch using super().__init__(optimizer).

After initializing the CosineWarmupScheduler class, we define the get_lr (or get learning rate function). We define the lr_factor or learning rate factor as being equal to self.get_lr_factor (likely part of the parent class) and state that epoch is equal to self.last epoch. Then, we return

the base_lr or base learning rate times the lr_factor for base_lr in self.base_lrs where base_lrs is a list of the base learning rate for various parameter group (parameter groups could be groups of weights or biases - different from hyperparameters).

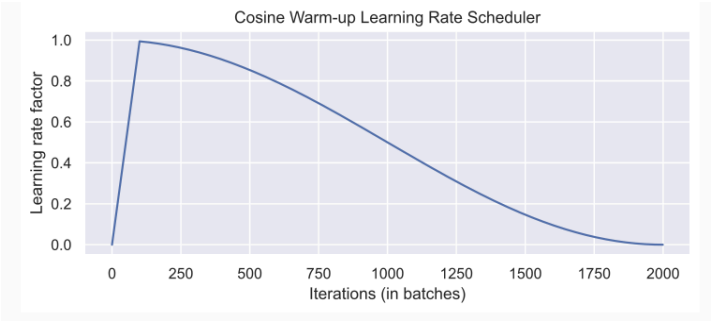
After defining the get_lr function, we define the get_lr_factor function which takes the argument self and epoch. We redefine the lr_factor as being equal to 0.5 * (1 + cos(epoch*π)/self.max_num_iters) - just using the numpy library to ensure that this is applied across tensors. Then, we state that if the epoch is less than or equal to the self.warmup, epoch + 1.0 / self.warmup is added to the lr_factor. Finally, we return the lr_factor.

Now, let's visualize the code. I will not be explaining this because its a visualization - also the visualization code is from the article.

"

```
# Needed for initializing the lr scheduler
p = nn.Parameter(torch.empty(4,4))
optimizer = optim.Adam([p], lr=1e-3)
lr_scheduler = CosineWarmupScheduler(optimizer=optimizer, warmup=100, max_iters=2000)

# Plotting
epochs = list(range(2000))
sns.set()
plt.figure(figsize=(8,3))
plt.plot(epochs, [lr_scheduler.get_lr_factor(e) for e in epochs])
plt.ylabel("Learning rate factor")
plt.xlabel("Iterations (in batches)")
plt.title("Cosine Warm-up Learning Rate Scheduler")
plt.show()
sns.reset_orig()
```

" (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)



In the first 100 iterations, we increase the learning rate factor from 0 to 1, whereas for all later iterations, we decay it using the cosine wave. Pre-impleentations of this scheduler can be found in the popular NLP Transformer library huggingface.

Pytorch Lightning Module

Finally, we can embed the transformer architecture into a PyTorch lightning module. From Tutorial 5, you know that PyTorch lightning simplifies our training and test code, as well as

structures the code nicely in separate functions. We implement a template for a classifier based on the Transformer encoder. Thereby, we have a prediction output per sequence element. If we would need a classifier over the whole sequence, the common approach is to add an additional [CLS] token to the sequence, representing the classifier token. However, here we focus on tasks where we have an output per element.

Additionally to the Transformer architecture, we add a small input network (maps input dimensions to mode dimensions), the positional encoding, and an output network (transforms output encodings to predictions). We also add the learning rate scheduler, which takes a step each iteration instead of once per epoch. This is ended for the warmup and the smooth cosine decay. The training, validation, and test step is left empty for now and will be filled for our task specific models.

```
class TransformerPredictor(pl.LightningModule):

    def __init__(self, input_dim, model_dim, num_classes, num_heads, num_layers, lr,
warmup, max_iters, dropout=0.0, input_dropout=0.0):
        """
        Inputs:
            input_dim - Hidden dimensionality of the input
            model_dim - Hidden dimensionality to use inside the Transformer
            num_classes - Number of classes to predict per sequence element
            num_heads - Number of heads to use in the Multi-Head Attention blocks
            num_layers - Number of encoder blocks to use.
            lr - Learning rate in the optimizer
            warmup - Number of warmup steps. Usually between 50 and 500
            max_iters - Number of maximum iterations the model is trained for. This is
needed for the CosineWarmup scheduler
            dropout - Dropout to apply inside the model
            input_dropout - Dropout to apply on the input features
        """
        super().__init__()
        self.save_hyperparameters()
        self._create_model()

    def _create_model(self):
        # Input dim -> Model dim
        self.input_net = nn.Sequential(
            nn.Dropout(self.hparams.input_dropout),
            nn.Linear(self.hparams.input_dim, self.hparams.model_dim)
        )
        # Positional encoding for sequences
        self.positional_encoding = PositionalEncoding(d_model=self.hparams.model_dim)
        # Transformer
        self.transformer = TransformerEncoder(num_layers=self.hparams.num_layers,
                                              input_dim=self.hparams.model_dim,

dim_feedforward=2*self.hparams.model_dim,
                                              num_heads=self.hparams.num_heads,
                                              dropout=self.hparams.dropout)
```

```
        # Output classifier per sequence lement
        self.output_net = nn.Sequential(
            nn.Linear(self.hparams.model_dim, self.hparams.model_dim),
            nn.LayerNorm(self.hparams.model_dim),
            nn.ReLU(inplace=True),
            nn.Dropout(self.hparams.dropout),
            nn.Linear(self.hparams.model_dim, self.hparams.num_classes)
        )

    def forward(self, x, mask=None, add_positional_encoding=True):
        """
        Inputs:
            x - Input features of shape [Batch, SeqLen, input_dim]
            mask - Mask to apply on the attention outputs (optional)
            add_positional_encoding - If True, we add the positional encoding to the
input.
                                      Might not be desired for some tasks.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        x = self.transformer(x, mask=mask)
        x = self.output_net(x)
        return x

    @torch.no_grad()
    def get_attention_maps(self, x, mask=None, add_positional_encoding=True):
        """
        Function for extracting the attention matrices of the whole Transformer for a
single batch.
        Input arguments same as the forward pass.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        attention_maps = self.transformer.get_attention_maps(x, mask=mask)
        return attention_maps

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=self.hparams.lr)

        # Apply lr scheduler per step
        lr_scheduler = CosineWarmupScheduler(optimizer,
                                             warmup=self.hparams.warmup,
                                             max_iters=self.hparams.max_iters)
        return [optimizer], [{'scheduler': lr_scheduler, 'interval': 'step'}]

    def training_step(self, batch, batch_idx):
        raise NotImplementedError

    def validation_step(self, batch, batch_idx):
        raise NotImplementedError

    def test_step(self, batch, batch_idx):
        raise NotImplementedError
```

” (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Okay, here we go, this is going to take a while to explain. This is the final boss; putting everything together. First, we define a subclass, TransformerPredictor from the parent class pl.Lightning Module. We initialize the class using the __init__ function, which takes the arguments self, input_dim, model_dim, num_classes, num_heads, num_layers, lr, warmup, max_iters, dropout (set to 0), and input_dropout (also set to 0). I will use the articles description of each of these arguments:

“

Inputs:

input_dim - Hidden dimensionality of the input
model_dim - Hidden dimensionality to use inside the transformer
num_classes - number of classes to predict per sequence element
num_heads - Number of heads to use in the Multi-Head Attention block
num_layers - Number of encoder blocks to use
lr - learning rate
warmup - Number of warmup steps. Usually between 50 and 500
max_iters - Number of maximum iterations the model is trained for. This is need for the CosineWarmup scheduler.
dropout - Dropout to apply inside the model
input_dropout - Dropout to apply on the input features

” (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Then, from the parent class, we use the functions, save_hyperparameters() - which we attribute to self and _create_model() - which we also attribute to self.

After initializing the TransformerPredictor class, we define the _create_model() function for arguments self. First, we transform the input_dim to the model_dim. This is done by defining self.input_net with the sequential API. In the sequential API, we define that the Dropout of the neural network is equal to the input_dropout defined in the hyperparameters (set to 0). Then, we apply a linear transformation to the input data, connecting the input layer and hidden layer (in the nn.Linear, we define the model dimensions as being self.hparams.input_dim and self.hparams.model_dim, which is the same thing as just saying nn.Linear(2, 3) - telling the model what dimensions to expect for the initial and transformed layer).

Then, after going from the input dimensions to the model dimensions, we define the “[p]ositional encoding for sequences.” (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.) We simply do this by defining self.positional_encoding and then using our Positional Encoding function from earlier, which requires the input of the d_model (remember the __init__ function for the PositionalEncoding

class was defined as follows: `def __init__(self, d_model, max_len=5000):` - which is called from the model_dim stored in hparams.

After defining the "[p]ositional encoding for sequences," (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.) we define the Transformer. We say that self.transformer = TransformerEncoder(), using the TransformerEncoder class we defined earlier. We define all the necessary inputs, including the num_layers, input_dim, dim_feedforward (which is 2 times the model_dimensions - meaning that dimensions of the feedforward layer are 2 times the size of the input layer), num_heads, and dropout. Everything else, however, is handled by the TransformerEncoder class.

Then, we define the "[o]utput classifier per sequence [e]lement," which is a simple neural network. We apply a Linear transformation to the output of the TransformerEncoder, producing the same dimensions. Then we apply Layer Normalization, then ReLU, and then dropout. Then, we apply another Linear transformation, transforming the input dimensions from the dimensions of the model to the dimensions of the number of classes.

Okay, now we define the forward function, which basically defines how the model processes input data to output data. The forward function takes argument self, x, mask (set to None), and add_positional encoding (set to True). According to the article, the inputs are:

"

x - Input features of shape [Batch, SeqLen, input_dim]
mask - Mask to apply on the attention outputs (optional)
Add_positional_encoding - If True, we add the positional encoding to the input.
Might not be desired for some tasks.

" (*Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks V1.2 Documentation*, n.d.)

Then, we define x as being self.input_net(x) - which basically means passing x through the input neural network we defined earlier. Then, we state that if add_positional encoding, we simply add the positional encoding to x. Then, we pass x through the transformer architecture using self.transformer(). Finally, we produce the outputs for x (likely probabilities for a number of classes) and then return x.

We also state - @torch.no_grad() which basically tells the architecture to not do backpropagation or compute gradients (saving computational cost for tasks where backprop is not necessary - like during post-training where all our parameters are set).

After defining the forward function, we define the function, get_attention_maps(), which takes the arguments self, x, mask, and add_positional encodings. According to the article, the get_attention_maps function is a "[f]unction for extracting the attention matrices of the whole Transformer for a single batch," and that the "[i]nput arguments [are the] same as the forward pass." Okay, so we do the same thing as the forward pass, except, instead of returning x, we are trying to return attention maps. Thus, we state that attention_maps = self.transformer.,get_attention_maps(), a function defined in the TransformerEncoder class.

get_attention_maps takes the arguments x and mask (which is set to None) and then returns the attention maps.

After returning the attention maps, we define the configure_optimizers function, which takes optimizers self. We define the variable, optimizer as being equal to optim.Adam(), which takes the arguments self,.parameters and the learning rate (which is defined as the model parameters). Then, we apply the learning rate scheduler, for which we use the CosineWarmupScheduler class from earlier, which receives the arguments optimizer, warmup (defined in the model hyperparameters), and the max_iters (also defined in the model hyperparameters). The "return [optimizer], [{'scheduler' : lr_scheduler, 'interval' : step}] tells the training framework both what optimizer(s) to use and how to use learning rate scheduler(s) according to ChatGPT.

Finally, we define the training_step, validation_step, and test_step functions, which we do not define (since we are not computing gradients and have not yet moved on to experiments - which will come soon).