Lab 8: Transformers and Attention-based Explainability

In this laboratory, we will discuss the most impactful architecture over the last 5 years: the Transformer model. Since the paper Attention Is All You Need by Vaswani et al. had been published in 2017, the Transformer architecture has continued to beat benchmarks in many domains, prominently in Natural Language Processing but also in many related fields (e.g., Computer Vision).

Here are three examples of the amazing Transformer applications:

- The Guardian newspaper article written with GPT2
- DALL·E 2: text 2 image generator
- GATO: multi-modal multi-task learning model

As the hype of the Transformer architecture seems not to come to an end in the next years, it is important to understand how it works, and have implemented it yourself, which we will do in this notebook.

Setup

This notebook requires some packages besides pytorch-lightning. It may take a while to setup the environment. After that you will also need to **restart** the runtime.

Below, we import some standard libraries.

```
# Standard libraries
import math
import os
import urllib.request
from functools import partial
from urllib.error import HTTPError
from tqdm.notebook import tqdm
import random

# Plotting
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

# PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data


%matplotlib inline

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "data/"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "saved_models/"

# Set seed to ensure that all operations are deterministic for reproducibility
torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
torch.backends.cudnn.determinstic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", device)
```

```
Device: cuda:0
```

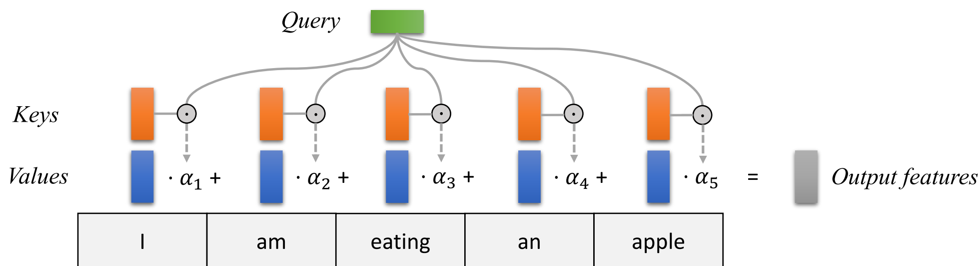
The Transformer architecture

In this notebook, we will implement the Transformer architecture by hand. As the architecture is so popular, there already exists a Pytorch module `nn.Transformer` (documentation) and a tutorial on how to use it for next token prediction. However, we will implement it here ourselves, to get through to the smallest details.

There are of course many more tutorials out there about attention and Transformers. Below, we list a few that are worth exploring if you are interested in the topic and might want yet another perspective on the topic after this one:

- Transformer: A Novel Neural Network Architecture for Language Understanding (Jakob Uszkoreit, 2017) - The original Google blog post about the Transformer paper, focusing on the application in machine translation.
- The Illustrated Transformer (Jay Alammar, 2018) - A very popular and great blog post intuitively explaining the Transformer architecture with many nice visualizations. The focus is on NLP.
- Attention? Attention! (Lilian Weng, 2018) - A nice blog post summarizing attention mechanisms in many domains including vision.
- Illustrated: Self-Attention (Raimi Karim, 2019) - A nice visualization of the steps of self-attention. Recommended going through if the explanation below is too abstract for you.
- The Transformer family (Lilian Weng, 2020) - A very detailed blog post reviewing more variants of Transformers besides the original one.

What is Attention?



The attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on input queries and elements' keys. This average has to represent:

- The numerical representation of the elements of the input sequence (**values**).
- The correlations among the elements (for self-attention) or with the output elements (for language models)(**queries**).
- The relevant features of each input elements with respect to the task at hand (**keys**).

Also, we need to specify a score function f_{attn} that takes the query and a key as input, and output the score/attention weight of the query-key pair. The weights of the average are calculated by a softmax over all score function outputs. Hence,

> We assign higher attention scores to those values whose corresponding key is most similar to the query.

If we try to describe it with pseudo-math, we can write:

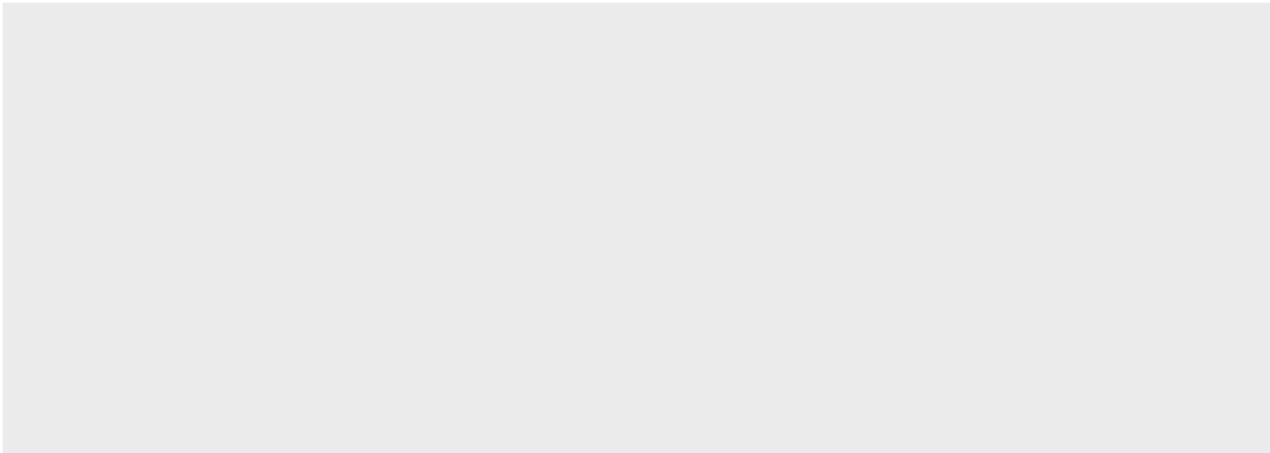
$$\alpha_i = \frac{\exp\left(f_{attn}\left(\text{key}_i, \text{query}\right)\right)}{\sum_j \exp\left(f_{attn}\left(\text{key}_j, \text{query}\right)\right)}, \quad \text{out} = \sum_i \alpha_i \cdot \text{value}_i$$

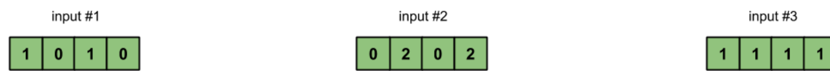
Visually, we can show the attention over a sequence of words as follows:

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 within the Encoder and the Decoder of Transformers is called **self-attention**. In self-attention, each sequence element provides a key, value, and query.
- In Language models (therefore also in Transformers) between the Encoder and the decoder we have the standard attention or encoder-decoder attention. We use as queries the output of the model, i.e. the decoded or generated output sequence.

Self-attention





We will now go into a bit more detail by first looking at the specific implementation of the attention mechanism which in the Transformer is the scaled dot product attention.

Scaled Dot Product Attention

The core concept behind self-attention is the scaled dot product attention. Our goal is to have an attention mechanism comparing each element in a sequence with any other (in an efficient way).

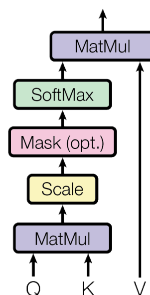
The dot product attention takes as input a set of queries $Q \in \mathbb{R}^{T \times d_k}$, keys $K \in \mathbb{R}^{T \times d_k}$ and values $V \in \mathbb{R}^{T \times d_v}$ where T is the sequence length, and d_k and d_v are the hidden dimensionality for queries/keys and values respectively. For simplicity, we neglect the batch dimension for now. The attention value from element i to j is based on its 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:

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

- The matrix multiplication QK^T performs the dot product for every possible pair of queries and keys, resulting in a matrix $T \times T$.
- 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 an average weighted by the attention.

The computation graph is visualized below:

Scaled Dot-Product Attention



The scaling factor of $1/\sqrt{d_k}$ is crucial to maintain an appropriate variance of attention values after initialization. Remember that we intialize our layers (therefore also Q and K) to have a variance close to 1. However, performing a dot product over two vectors with a variance σ results in a scalar having d_k-times higher variance:

> If we do not scale down the variance to σ, 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 ≈ 0 and we wont learn the parameters appropriately (*vanishing gradient*).

The masking block `Mask(opt.)` is used to stack **multiple sequences with different lengths into a batch**. 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 (negative) values, e.g. 10^{-14}.

Let's now write a function computing 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, -10e14)
    attention = F.softmax(attn_logits, dim=-1)

    # In order to save the gradient of the attention (we will need it later)
    attention.requires_grad_(True)
    attention.retain_grad()

    output_values = torch.matmul(attention, v)

    return output_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:

```
b_s, seq_len, d_k, d_v = 1, 3, 2, 1
q = torch.randn(b_s, seq_len, d_k)
k = torch.randn(b_s, seq_len, d_k)
v = torch.randn(b_s, seq_len, d_v)
# q = torch.tensor([[-0.8920, -1.5091],
#         [ 0.3704,  1.4565],
#         [ 0.9398,  0.7748]])
# k = torch.tensor([[ 0.1919,  1.2638],
#         [-1.2904, -0.7911],
#         [-0.0209, -0.7185]])
# v = torch.tensor([[ 0.5186, -1.3125],
#         [ 0.1920,  0.5428],
#         [-2.2188,  0.2590]])
print(f"Q: {q.shape}\n{q}\nK: {k.shape}\n{k}\nV: {v.shape}\n{v}")

output_values, attention = scaled_dot_product(q, k, v)

print(f"Attention: {attention.shape}\n{attention}")
print(f"Output: {output_values.shape}\n{output_values}")


# assert (output_values - torch.tensor([[-0.4846,  0.4063],
#                                       [ 0.2174, -1.0264],
#                                       [-0.0766, -0.8279]]) < 1e-4).all(), \
#     f"Error in computing the attention"
# assert (attention - torch.tensor([[0.0300, 0.6852, 0.2847],
#                                   [0.8302, 0.0678, 0.1019],
#                                   [0.7071, 0.0857, 0.2072]]) < 1e-4).all(), \
#     f"Error in computing the attention"
```

```
Q: torch.Size([1, 3, 2])
tensor([[[ 1.5410, -0.2934],
         [-2.1788,  0.5684],
         [-1.0845, -1.3986]]])
K: torch.Size([1, 3, 2])
tensor([[[ 0.4033,  0.8380],
         [-0.7193, -0.4033],
         [-0.5966,  0.1820]]])
V: torch.Size([1, 3, 1])
tensor([[[-0.8567],
         [ 1.1006],
         [-1.0712]]])
Attention: torch.Size([1, 3, 3])
tensor([[[0.5662, 0.2156, 0.2182],
         [0.1249, 0.4274, 0.4477],
         [0.0758, 0.6120, 0.3122]]], requires_grad=True)
Output: torch.Size([1, 3, 1])
tensor([[[-0.4815],
         [-0.1161],
         [ 0.2741]]], grad_fn=<UnsafeViewBackward0>)
```

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.

Multi-Head Attention

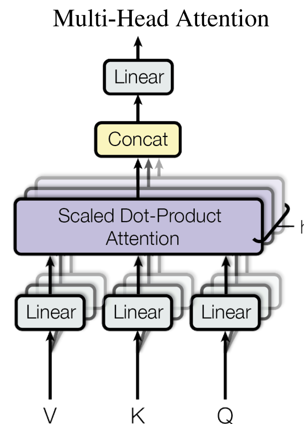
The scaled dot product attention allows a network to attend over a sequence. However, often there are multiple different aspects a sequence element wants to attend to, and a single weighted average is not a good option for it. This is why we extend the attention mechanisms to **multiple heads**, i.e. 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,
- We pass through the scaled dot product attention independently.
- We concatenate the heads and combine them with a final weight matrix.

Mathematically,

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
$$\text{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

We refer to this as Multi-Head Attention layer with the learnable parameters $W_{1\ldots h}^Q \in \mathbb{R}^{D\times d_k}$, $W_{1\ldots h}^K \in \mathbb{R}^{D\times d_k}$, $W_{1\ldots h}^V \in \mathbb{R}^{D\times d_v}$, and $W^O \in \mathbb{R}^{h\cdot d_k \times d_{out}}$ (D being the input dimensionality). Expressed in a computational graph:



What are the Query, the Key and the Value in a NN where we only have the output of the previous layer?

Looking at the computation graph above, a possible implementation is to obtain them from the current feature map $X \in \mathbb{R}^{B\times T\times d_{\text{model}}}$ (B being the batch size, T the sequence length, d_{model} the hidden dimensionality of X). The consecutive weight matrices W^Q, W^K, and W^V can transform X to the corresponding Queries, Keys, and Values of the input.

Using this approach, let's implement the Multi-Head Attention module below.

```
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 # dimension of concatenated heads
        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, embed_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, input_dim = x.size()
        qkv = self.qkv_proj(x)

        # Extract Q, K, V from linear projection of the input
        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)
        # print(f"Query, key, value shape: {q.shape} {k.shape}")

        # Apply Dot Product Attention
        values, attention = scaled_dot_product(q, k, v, mask=mask)
```

```
        # Concatenate heads
        values = values.permute(0, 2, 1, 3)  # [Batch, SeqLen, Head, Dims]
        values = values.reshape(batch_size, seq_length, self.embed_dim)

        # Output projection
        o = self.o_proj(values)

        if return_attention:
            return o, attention
        else:
            return o

input_d = 3
seq_l = 5
embed_d = 4
n_heads = 2
b_size = 1

mh_att = MultiheadAttention(input_d, embed_d, n_heads)

# x = torch.rand(b_size, seq_l, input_d)
x = torch.tensor([[[0.3360, 0.6676, 0.6393],
         [0.2083, 0.5484, 0.1204],
         [0.3533, 0.3038, 0.9383],
         [0.0499, 0.2048, 0.0107],
         [0.5019, 0.5082, 0.3027]]])
print(f"Input x: {x}")
print(f"Input x shape: {x.shape}")

att_output, attention = mh_att(x, return_attention=True)
print(f"MhA Output {att_output}")
assert att_output.shape == torch.Size([1, 5, 4]), "Error in computing multi-head attention"
print(attention.shape)
```

```
Input x: tensor([[[0.3360, 0.6676, 0.6393],
         [0.2083, 0.5484, 0.1204],
         [0.3533, 0.3038, 0.9383],
         [0.0499, 0.2048, 0.0107],
         [0.5019, 0.5082, 0.3027]]])
Input x shape: torch.Size([1, 5, 3])
MhA Output tensor([[[ 0.0895,  0.0139, -0.2404, -0.0016],
         [ 0.0892,  0.0107, -0.2436,  0.0013],
         [ 0.0905,  0.0200, -0.2399, -0.0054],
         [ 0.0897,  0.0133, -0.2458,  0.0004],
         [ 0.0893,  0.0123, -0.2416, -0.0006]]], grad_fn=<ViewBackward0>)
torch.Size([1, 2, 5, 5])
```

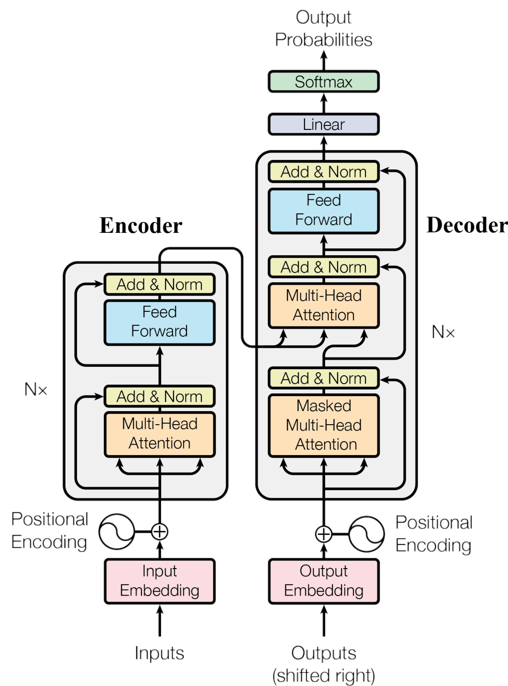
Attention is permutation equivariant

One crucial characteristic of the multi-head attention is that it is permutation-equivariant with respect to its inputs. 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 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 later (topic *Positional encodings* below).

Transformer Encoder

Next, we will look at how to apply the multi-head attention 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. The decoder, instead, 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, it is not always necessary and. Many advances in NLP have been made using pure encoder-based Transformer models (e.g. BERT-family, the Vision Transformer, and more). Therefore, we will focus here only 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:



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 $\text{LayerNorm}(x + \text{Multihead}(x, x, x))$ (x being Q, K and V input to the attention layer).

The **residual connection** is crucial in the Transformer architecture to:

1. Avoid vanishing gradients, as in ResNet, but valid for all deep architectures (Some models contain more than 24 blocks in the encoder)
2. Retain the information about the original sequence (remember that Self-Attention do not necessairly consider the input as a sequence)

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.

Additionally to the Multi-Head Attention, a small fully connected **Feed-Forward Network (FFN)** is added to the model, which is applied to each position separately and identically. Specifically, the model uses a Linear$\rightarrow$ReLU$\rightarrow$Linear MLP. The full transformation including the residual connection can be expressed as:

$$\begin{split}\text{FFN}(x) & = \max(0, xW_1 + b_1)W_2 + b_2\\ \text{output} & = \text{LayerNorm}(x + \text{FFN}(x))\end{split}$$

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 it for the next attention block. Usually, the inner dimensionality of the MLP is 2-8$\times$ 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.
- Also, we will assume now on a constant `input_dim = embed_dim` throughout the Transformer and therefore we will instantiate the attention as `MultiheadAttention(input_dim, input_dim, num_heads)`. The dimensionality of the first input will be addressed later.

```
class EncoderBlock(nn.Module):
    def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
        """
        Args:
            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.MLP = 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 (Layer Norm and Dropout)
        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None, return_attention=False):
        # Attention part
        attn_out, attention = self.self_attn(x, mask, return_attention=True)

        x = x + self.dropout(attn_out)
        x = self.norm1(x)

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

        if return_attention:
            return x, attention
        return 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. They helps us in understanding, and partially, explaining the model. Attention scores may not necessarily reflect the true interpretation of the model (it is disputed in literature, check 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 layer in self.layers:
            x = layer(x, mask=mask)
        return x

    def get_attention_maps(self, x, mask=None, return_outputs=False):
        attn_maps = []
        for layer in self.layers:
            x, attn_map = layer(x, mask, return_attention=True)
            attn_maps.append(attn_map)
        if return_outputs:
            return x, attn_maps
        return attn_maps
```

Positional encoding

We have discussed before that the Multi-Head Attention block is permutation-equivariant. In tasks like language understanding, however, the position is important for interpreting the input words. The position information is therefore added in the input features by means of feature patterns that the network can identify and potentially generalize to larger sequences. The specific pattern chosen by Vaswani et al. are sine and cosine functions of different frequencies, as follows:

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

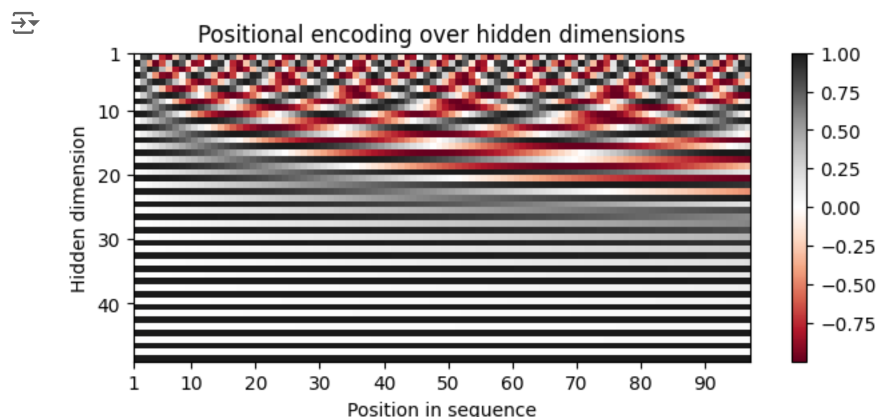
$PE_{(pos,i)}$ represents the position encoding at position pos in the sequence, and hidden dimensionality i. These values, concatenated for all hidden dimensions, are added to the original input features (in the Transformer visualization above, see "Positional encoding"), and constitute the position information. We distinguish between even ($i \bmod 2 = 0$) and uneven ($i \bmod 2 = 1$) 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 $10000 \cdot 2\pi$.

The positional encoding is implemented below.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        """
        Args
            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
```

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.

```
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()
```



Transformer Encoder Classifier

Finally, we can implement a template for a classifier based on the Transformer encoder.

Additionally to the Transformer architecture, we add:

- a small input network (maps input dimensions to model dimensions)
- the positional encoding
- an output network (transforming output encodings to predictions).

Notice that the output network will take in input a 3D tensor `<batch_samples, seq_len, model_dim>` and produces in output another 2D tensor `<batch_samples, seq_len>` where each output value represents the prediction of the corresponding reversed number.

If we would need a classifier over the whole sequence, instead, the common approach is to add an additional `[CLS]` token to the sequence, representing the classifier token and then get the prediction only from that output token.

```
class TransformerPredictor(nn.Module):
    def __init__(
        self,
        input_dim,
        model_dim,
        num_classes,
        num_heads,
        num_layers,
        dropout=0.0,
        input_dropout=0.0,
    ):
        """
        Args:
            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.input_dim = input_dim
        self.model_dim = model_dim
        self.num_classes = num_classes
        self.num_heads = num_heads
        self.num_layers = num_layers
        self.dropout = dropout
        self.input_dropout = input_dropout

        # Learn an Generic Input Encoder Input dim -> Model dim
        self.input_net = nn.Sequential(
            nn.Dropout(self.input_dropout),
            nn.Linear(self.input_dim, self.model_dim)
        )

        # Positional encoding for sequences
        self.positional_encoding = PositionalEncoding(d_model=self.model_dim)

        # Transformer
        self.transformer = TransformerEncoder(
            num_layers=self.num_layers,
            input_dim=self.model_dim,
            dim_feedforward=2 * self.model_dim,
            num_heads=self.num_heads,
            dropout=self.dropout,
        )
        # Output classifier per sequence element
        self.output_net = nn.Sequential(
            nn.Linear(self.model_dim, self.model_dim),
            nn.LayerNorm(self.model_dim),
            nn.ReLU(inplace=True),
            nn.Dropout(self.dropout),
            nn.Linear(self.model_dim, self.num_classes),
        )

    def forward(self, x, mask=None, add_positional_encoding=True):
        """
        Args:
            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

    def get_attention_maps(self, x, mask=None, add_positional_encoding=True,
                           return_outputs=False):
        """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)
        transf_outputs, attn_maps = self.transformer.get_attention_maps(x, mask, return_outputs=True)

        if return_outputs:
            transf_outputs = self.output_net(transf_outputs)
            return transf_outputs, attn_maps
        return attn_maps
```

Experiment: Sequence to Sequence

After having finished the implementation of the Transformer architecture, we can start experimenting.

A Seq-2-Seq task represents a task where the input *and* the output is a sequence, not necessarily of the same length. Popular tasks in this domain include machine translation and summarization. For this, we usually have a Transformer encoder for interpreting the input sequence, and a decoder for generating the output in an autoregressive manner. Here, however, we will go back to a much simpler example task and use only the encoder, since the output length is fixed. Given a sequence of N numbers between 0 and M, the task is to reverse the input sequence. In Numpy notation, if our input is x, the output should be x[::-1]. Although this task sounds very simple, RNNs can have issues with such because the task requires long-term dependencies. Transformers are built to support such, and hence, we expect it to perform very well.

First, let's create a dataset class below.

```
class ReverseDataset(data.Dataset):
    def __init__(self, num_categories, seq_len, size):
        super().__init__()
        self.num_categories = num_categories
        self.seq_len = seq_len
        self.size = size

        self.data = torch.randint(self.num_categories, size=(self.size, self.seq_len))

    def __len__(self):
        return self.size

    def __getitem__(self, idx):
        inp_data = self.data[idx]
        labels = torch.flip(inp_data, dims=(0,))
        return inp_data, labels
```

We create an arbitrary number of random sequences of numbers between 0 and `num_categories-1`. The label is simply the tensor flipped over the sequence dimension. We can create the corresponding data loaders below.

```
dataset = partial(ReverseDataset, 10, 16)
train_dl = data.DataLoader(dataset(50000), batch_size=128, shuffle=True, drop_last=True, pin_memory=True)
val_dl = data.DataLoader(dataset(1000), batch_size=128)
test_dl = data.DataLoader(dataset(10000), batch_size=128)
```

Let's look at an arbitrary sample of the dataset:

```
inp_data, labels = train_dl.dataset[0]
print("Input data:", inp_data)
print("Labels:    ", labels)
```

```
Input data: tensor([2, 1, 4, 9, 1, 1, 0, 3, 0, 6, 7, 6, 9, 1, 6, 4])
Labels:     tensor([4, 6, 1, 9, 6, 7, 6, 0, 3, 0, 1, 1, 9, 4, 1, 2])
```

During training, we pass the input sequence through the Transformer encoder and predict the output for each input token. We use the standard Cross-Entropy loss to perform this. Every number is represented as a one-hot vector. Remember that representing the categories as single scalars decreases the expressiveness of the model extremely as 0 and 1 are not closer related than 0 and 9 in our example. An alternative to a one-hot vector is using a learned embedding vector as it is provided by the PyTorch module `nn.Embedding`. However, using a one-hot vector

with an additional linear layer as in our case has the same effect as an embedding layer (`self.input_net` maps one-hot vector to a dense vector, where each row of the weight matrix represents the embedding for a specific category).

In the following we will implement the training and evaluation step required for fitting the model

```
def train_step(model, x, y, optim):
    model.train()

    # Fetch data and transform categories to one-hot vectors
    inp_data = F.one_hot(x, num_classes=model.num_classes).float()
    inp_data, y = inp_data.to(device), y.to(device)


    # Perform prediction and calculate loss and accuracy
    preds = model(inp_data, add_positional_encoding=True)
    loss = F.cross_entropy(preds.view(-1, preds.size(-1)), y.view(-1))
    acc = (preds.argmax(dim=-1) == y).float().mean()

    # Backpropagate and update weights
    loss.backward()
    optim.step()
    model.zero_grad()

    return loss, acc

def eval_step(model, x, y):
    with torch.no_grad():
        model.eval()

        # Fetch data and transform categories to one-hot vectors
        inp_data = F.one_hot(x, num_classes=model.num_classes).float()
        inp_data, y = inp_data.to(device), y.to(device)

        # Perform prediction and calculate loss and accuracy
        preds = model(inp_data, add_positional_encoding=True)
        loss = F.cross_entropy(preds.view(-1, preds.size(-1)), y.view(-1))
        acc = (preds.argmax(dim=-1) == y).float().mean()

    return loss, acc
```

Finally, we can create a training function similar to the one we have seen in previous laboratories. We running for N epochs printing the training and validation loss and saving our best model based on the validation. Afterward, we test our models on the test set.

```
def train_model(model, train_loader, val_loader, test_loader,
                optim, epochs=5):
    best_acc = 0.
    pbar = tqdm(range(epochs))
    for e in range(epochs):
        # Train model
        train_loss, train_acc = 0., 0.
        for x, y in train_loader:
            loss, acc = train_step(model, x, y, optim)
            train_loss += loss
            train_acc += acc

        # Validate model
        val_loss, val_acc = 0., 0.
        for x, y in val_loader:
            loss, acc = eval_step(model, x, y)
            val_loss += loss
            val_acc += acc

        # Saving best model for early stopping
        if val_acc/len(val_loader) > best_acc:
            torch.save(model.state_dict(), "best_model.pt")
            best_acc = val_acc/len(val_loader)

        pbar.update()
        pbar.set_description(f"Train Acc: {train_acc/len(train_loader)* 100:.2f} "
                             f"Train Loss: {train_loss/len(train_loader):.2f} "
                             f"Val Acc: {val_acc/len(val_loader)* 100 :.2f}  "
                             f"Val loss: {val_loss/len(val_loader):.2f} ")

    pbar.close()
    # Load best model for early stopping
    model.load_state_dict(torch.load("best_model.pt"))

    # Test model
    test_loss, test_acc = 0., 0.
    for x, y in test_loader:
        loss, acc = eval_step(model, x, y)
        test_loss += loss
        test_acc += acc

    print(f"Test accuracy: {test_acc/len(test_loader)*100 :.2f}")


    return model
```

Finally, we can train the model. In this setup, we will use a single encoder block and a single head in the Multi-Head Attention. This is chosen because of the simplicity of the task, and in this case, the attention can actually be interpreted as an "explanation" of the predictions (compared to the other papers above dealing with deep Transformers).

```
reverse_model = TransformerPredictor(
    input_dim=train_dl.dataset.num_categories,
    model_dim=32,
    num_heads=1,
    num_classes=train_dl.dataset.num_categories,
    num_layers=1,
    dropout=0.0,
)
reverse_model = reverse_model.to(device)
optimizer = optim.AdamW(reverse_model.parameters(), lr=0.001)

reverse_model = train_model(reverse_model, train_dl, val_dl, test_dl, optimizer)
```

Show hidden output

As we would have expected, the Transformer can correctly solve the task. However, how does the attention in the Multi-Head Attention block looks like for an arbitrary input? Let's try to visualize it below.

```
data_input, labels = next(iter(val_dl))
inp_data = F.one_hot(data_input, num_classes=reverse_model.num_classes).float()
inp_data = inp_data.to(device)
attention_maps = reverse_model.get_attention_maps(inp_data)
```

The object `attention_maps` is a list of length N where N is the number of layers. Each element is a tensor of shape [Batch, Heads, SeqLen, SeqLen], which we can verify below.

```
attention_maps[0].shape
```

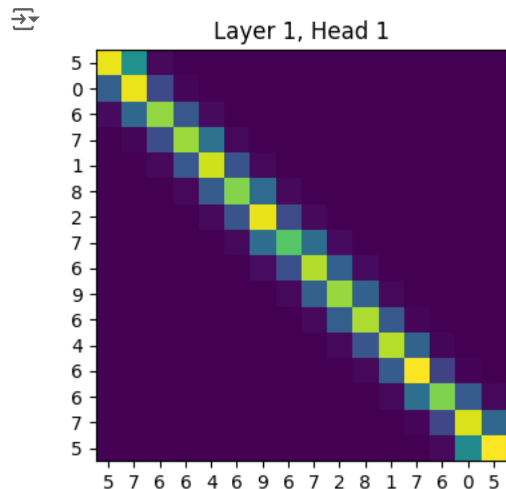
```
attention_maps[0].shape
```

Next, we will write a plotting function that takes as input the sequences, attention maps, and an index indicating for which batch element we want to visualize the attention map. We will create a plot where over the rows we have different layers, while over columns we show the different heads. Remember that the softmax has been applied for each row separately.

```
def plot_attention_maps(input_data, attn_maps, idx=0):
    if input_data is not None:
        input_data = input_data[idx].detach().cpu().numpy()
    else:
        input_data = np.arange(attn_maps[0][idx].shape[-1])
    attn_maps = [m[idx].detach().cpu().numpy() for m in attn_maps]

    num_heads = attn_maps[0].shape[0]
    num_layers = len(attn_maps)
    seq_len = input_data.shape[0]
    fig_size = 4 if num_heads == 1 else 3
    fig, ax = plt.subplots(num_layers, num_heads, figsize=(num_heads * fig_size, num_layers * fig_size))
    if num_layers == 1:
        ax = [ax]
    if num_heads == 1:
        ax = [[a] for a in ax]
    for row in range(num_layers):
        for column in range(num_heads):
            ax[row][column].imshow(attn_maps[row][column], origin="lower", vmin=0)
            ax[row][column].set_xticks(list(range(seq_len)))
            ax[row][column].set_xticklabels(input_data.tolist())
            ax[row][column].set_yticks(list(range(seq_len)))
            ax[row][column].set_yticklabels(input_data.tolist())
            ax[row][column].set_title("Layer %i, Head %i" % (row + 1, column + 1))
    fig.subplots_adjust(hspace=0.5)
    plt.show()
```

Finally, we can plot the attention maps of our trained Transformer on the reverse task:

```
plot_attention_maps(data_input, attention_maps, idx=5)
```



The model has learned to attend to the token that is on the flipped index of itself. Hence, it actually does what we intended it to do. We see that it however also pays some attention to values close to the flipped index. This is because the model doesn't need the perfect, hard attention to solve this problem, but is fine with this approximate, noisy attention map. The close-by indices are caused by the similarity of the positional encoding, which we also intended with the positional encoding.

Now let's perform some more advanced attention-based explanation!

Retrain your model with now 4 heads and 3 layers. What happens to the visualization?

Compute and visualize with the previous function:

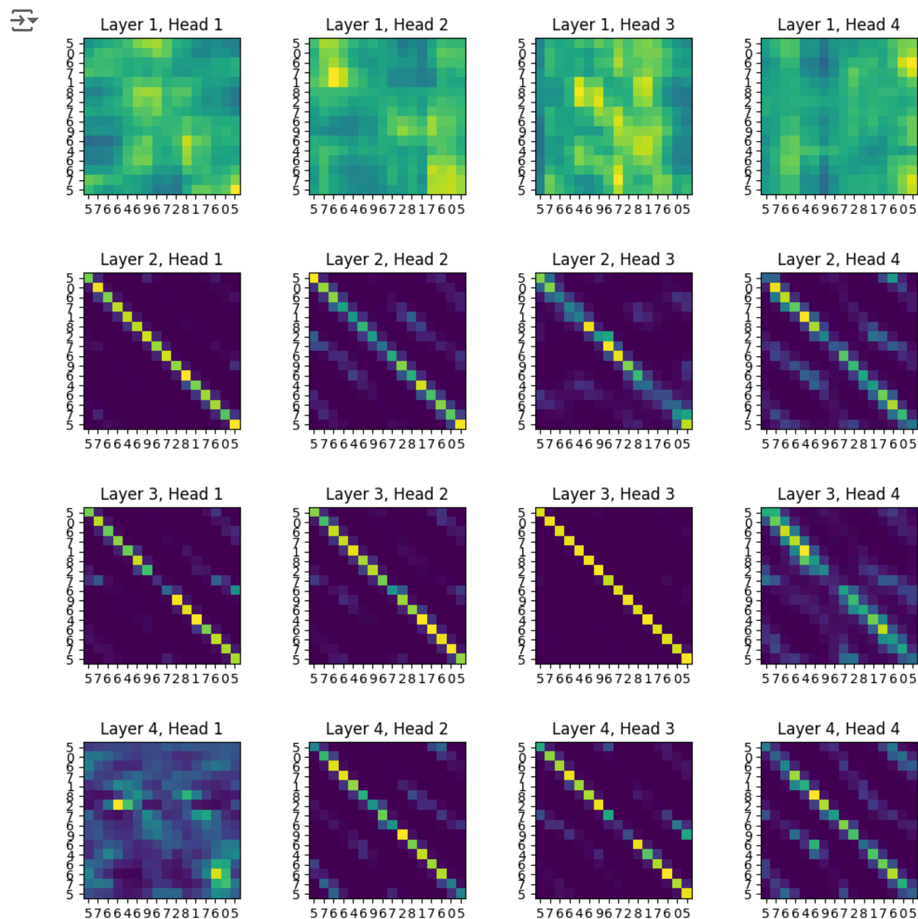
- Standard attention explanation for 1st and last layer
- Rollout
- Transformer Attention (Attention x Gradient)

Note: don't forget to aggregate over the heads!

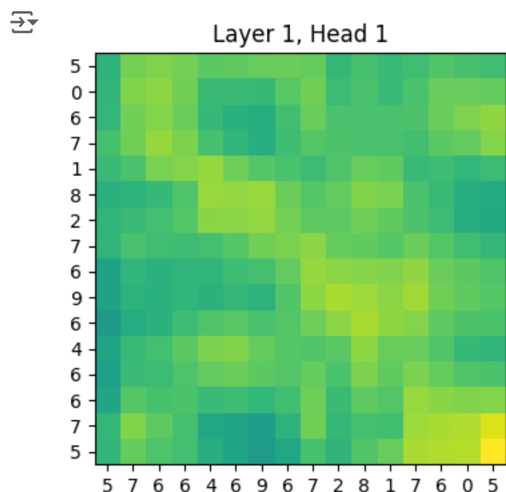
```
### Retraining with 2 attention heads and 3 layers
reverse_model = TransformerPredictor(
    input_dim=train_dl.dataset.num_categories,
    model_dim=32,
    num_heads=4,
    num_classes=train_dl.dataset.num_categories,
    num_layers=4,
    dropout=0.0,
)
reverse_model = reverse_model.to(device)
optimizer = optim.AdamW(reverse_model.parameters(), lr=0.001)

reverse_model = train_model(reverse_model, train_dl, val_dl, test_dl, optimizer)
```

Show hidden output

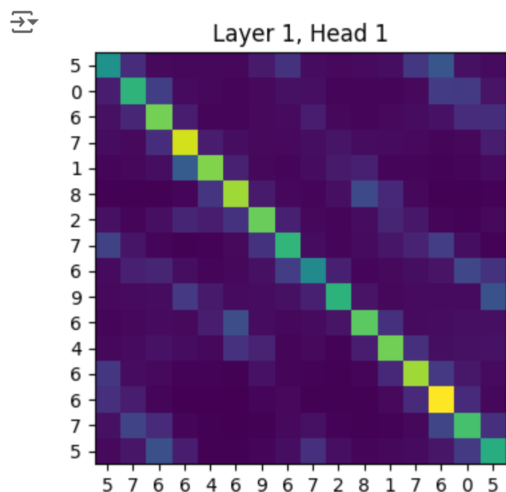
```
data_input, labels = next(iter(val_dl))
inp_data = F.one_hot(data_input, num_classes=reverse_model.num_classes).float()
inp_data, labels = inp_data.to(device), labels.to(device)
attention_maps = reverse_model.get_attention_maps(inp_data)

# Plot the matrix of attention maps
plot_attention_maps(data_input, attention_maps, idx=5)
```



```
# Plot STANDARD ATTENTION FIRST LAYER: aggregated attentions over the first layer
standard_attention_first = attention_maps[0].mean(dim=-3) # batch_size x heads x seq_len x seq_len

plot_attention_maps(data_input, [standard_attention_first.unsqueeze(dim=-3)], idx=5) # the function still requires list of attention
```




```
# Plot STANDARD ATTENTION LAST: aggregated attentions over the last layer
standard_attention_last = attention_maps[-1].mean(dim=-3) # batch_size x heads x seq_len x seq_len

plot_attention_maps(data_input, [standard_attention_last.unsqueeze(dim=-3)], idx=5) # the function still requires list of attention wit
```




```
def get_rollout_attention(attentions):
    """Factorize attentions over the layers after aggregating the heads"""

    cumulated_attention = None
    for i, attention in enumerate(attentions):
        # aggregated the attention heads of the current layer
        aggregated_attention = attention.mean(dim=-3)
        if i == 0:
            cumulated_attention = aggregated_attention # initialize cumulated_attention
        else:
            # factorize attention over the layers
            cumulated_attention = cumulated_attention * aggregated_attention

        # print(f"Iteration: {i}")
        # plot_attention_maps(data_input, [aggregated_attention.unsqueeze(dim=-3)], idx=5) # the function still requires list of attent
        # plot_attention_maps(data_input, [cumulated_attention.unsqueeze(dim=-3)], idx=5) # the function still requires list of attentic

    return cumulated_attention


# Plot ROLLOUT ATTENTION
rollout_attention = get_rollout_attention(attention_maps)

plot_attention_maps(data_input, [rollout_attention.unsqueeze(dim=-3)], idx=5) # the function still requires list of attention with head
```

Layer 1, Head 1

5
0
6
7
1
8
2
7
6
9
6
4
6