DL2 - Sheet 02: Attention (50 points)

```
# Install the transformers library if necessary
pip install transformers
```

Intro: Transformers for Sequence Classification

In this weeks notebook, we will implement our own Transformer model from scratch. Typical Transformers can be broken down into the following components: Remark: Here, we will focus on the encoding of sentences for the purpose of sentiment classification, the decoder used in sequence2sequences Transformers has a very similar structure.

1. Embedding: An embedding layer that transforms word tokens into vector representations.

2. Encoder: The encoder block consists of several multi-headed attention blocks

   2.1. Attention block 1

   2.2. Attention block 2

   ...

   2.L. Attention block L

3. Pooling: The final output of the encoder computes one vector representation for each token. To further summarize/pool this information for classification, the [CLS] token at sequence position 0 is typically selected.

4. Classification: A standard small MLP is used for classification and outputs probabilites for the most likely predicted class.

Preparation: From sentences to tokens

First, we will use hugginface's tokenizer to go from words to indices in the vocabulary. In the following, we will focus on the distilBERT model:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
sentence = "This was one of the best movies I have ever seen."
inputs = tokenizer(sentence, return_tensors = 'pt')
print(inputs)
```

```
{'input_ids': tensor([[ 101, 2023, 2001, 2028, 1997, 1996, 2190, 5691,
1045, 2031, 2412, 2464,
         1012,  102]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1]])}
```

```
# Now we download pretrained weights for the model.
! wget https://tubcloud.tu-
berlin.de/s/GfEoq4r8Sgb2727/download/distilbert.pt

# This config contains the model parameters, it will be important to
understand what representations
# have which dimensionality

from torch import nn
import torch
import math
import torch

class Config(object):
    def __init__(self):
        self.n_heads = 12
        self.n_layers = 6
        self.pad_token_id = 0
        self.dim = 768
        self.hidden_dim = 3072
        self.max_position_embeddings = 512
        self.vocab_size = 30522
        self.eps = 1e-12

        self.attention_head_size = int(self.dim / self.n_heads)
        self.all_head_size = self.n_heads * self.attention_head_size

        self.n_classes = 2
        self.device = 'cpu'

config = Config()
```

1. Embedding Layer (5p)

Next, we will have to implement the Embedding layer.

1. forward: compute the output embeddings from the input_embeds and position_embeds. (Think about how they are merged.)

```
torch.manual_seed(0) # set seed for reproducible random initialization
of weights

class Embeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.word_embeddings = nn.Embedding(config.vocab_size,
config.dim, padding_idx=config.pad_token_id)
        self.position_embeddings =
nn.Embedding(config.max_position_embeddings, config.dim)
        self.LayerNorm = nn.LayerNorm(config.dim, eps=config.eps)
```

```python
    def forward(self, input_ids: torch.Tensor) -> torch.Tensor:
        """
        Parameters:
            input_ids (torch.Tensor):
                torch.tensor(bs, max_seq_length) The token ids to embed.
        Returns: torch.tensor(bs, max_seq_length, dim) The embedded tokens (plus position embeddings)
        """

        # Embedding the input ids
        input_embeds = self.word_embeddings(input_ids)  # (bs, max_seq_length, dim)
        seq_length = input_embeds.size(1)

        # Creating and embedding the position ids
        position_ids = torch.arange(seq_length, dtype=torch.long, device=input_ids.device)  # (max_seq_length)
        position_ids = position_ids.unsqueeze(0).expand_as(input_ids)  # (bs, max_seq_length)
        position_embeddings = self.position_embeddings(position_ids)  # (bs, max_seq_length, dim)

        # Compute the output embeddings

        # 1. START YOUR CODE HERE #

        # 1. END YOUR CODE HERE #

        embeddings = self.LayerNorm(embeddings)  # (bs, max_seq_length, dim)
        return embeddings

embedding_layer = Embeddings(config)
# Test if your embedding layer computes an output
embeddings = embedding_layer(inputs['input_ids'])
```

2. Attention Block (20 points)

The next step is writing the attention block. It mainly consits of the self-attention function (that you have analyzed in the first part of the exercise sheet), a layer normalization followed by an additional projection layer.

Please add the missing code:

1. __init__: Add the Linear projections for the query, key and value functions. Make sure your set the correct dimensions.

2. forward: Write the main self-attention function. Follow the three main steps as indicated in the comments below.

```
torch.manual_seed(0)

class AttentionBlock(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config

        # self-attention components

        # 1. START YOUR CODE HERE #



        # 1. END YOUR CODE HERE #

        self.out_lin = nn.Linear(in_features=config.dim,
out_features=config.dim, bias=True)
        self.sa_layer_norm = nn.LayerNorm(normalized_shape=config.dim,
eps=config.eps)

        # feed-forward network
        self.lin1 = nn.Linear(in_features=config.dim,
out_features=3072, bias=True)
        self.lin2 = nn.Linear(in_features=3072,
out_features=config.dim, bias=True)
        self.output_layer_norm =
nn.LayerNorm(normalized_shape=config.dim, eps=config.eps)


    def forward(self, hidden_states):

        def shape(x):
            """ separate heads """
            return x.view(1, -1, 12, 64).transpose(1, 2)

        def unshape(x):
            """ group heads """
            return x.transpose(1, 2).contiguous().view(1, -1, 12 * 64)

        bs=hidden_states.shape[0]
        n_nodes= hidden_states.shape[1]

        query=key=value=hidden_states
        q = self.q_lin(query)
        k = self.k_lin(key)
        v = self.v_lin(value)
```

```
        # Separating the heads
        q = shape(q)  # (bs, n_heads, q_length, dim_per_head)
        k = shape(k)  # (bs, n_heads, k_length, dim_per_head)
        v = shape(v)  # (bs, n_heads, k_length, dim_per_head)

        # Normalizing the query-tensor
        q = q / math.sqrt(q.shape[-1])


        # 2. START YOUR CODE HERE #

        # Compute attention scores


        # Transform the scores into probability distribution via
softmax


        # Compute the weighted representation of the value-tensor (aka
context)


        # 2. END YOUR CODE HERE #

        # Merging the heads again
        context = unshape(context)  # (bs, q_length, dim)

        # Additional projection of the context to get the output of
the self-attention block
        sa_output = self.out_lin(context)
        sa_output = self.sa_layer_norm(sa_output + hidden_states)

        # Feed-forward network to compute the attention block output
        x = self.lin1(sa_output)
        x = nn.functional.gelu(x)
        ffn_output = self.lin2(x)
        ffn_output = self.output_layer_norm(ffn_output + sa_output)

        return  ffn_output, weights

block = AttentionBlock(config)
# Test if your attention block computes an output
block_output = block(embeddings)

block_output
```

```
(tensor([[[ 0.7114, -0.0722, -0.5230,  ..., -0.1903, -0.7819, -
0.5930],
          [ 0.2216, -0.9576,  0.9970,  ..., -0.0926, -1.1536,
0.8901],
```

```
          [ 0.4497, -0.6879,  1.1513,  ...,  0.0209, -0.9052,
1.1951],
          ...,
          [-0.1305, -0.3480,  0.5797,  ...,  0.3765, -0.8779,
0.5787],
          [ 0.6187,  0.2542,  0.1652,  ...,  1.0826, -1.0824,
0.9224],
          [-0.1873,  0.0490,  0.7358,  ..., -1.2114, -0.6134, -
0.7558]]],
        grad_fn=<NativeLayerNormBackward0>),
 tensor([[[[0.0434, 0.0909, 0.0583,  ..., 0.1567, 0.0598, 0.0668],
           [0.0969, 0.0588, 0.0794,  ..., 0.0579, 0.0654, 0.0741],
           [0.0529, 0.0764, 0.0608,  ..., 0.0522, 0.0904, 0.0868],
           ...,
           [0.0657, 0.0971, 0.1063,  ..., 0.0665, 0.0505, 0.0848],
           [0.0883, 0.0417, 0.1146,  ..., 0.1033, 0.0529, 0.0934],
           [0.1194, 0.0599, 0.0607,  ..., 0.0505, 0.0614, 0.0591]],

          [[0.0197, 0.0776, 0.0725,  ..., 0.0774, 0.0452, 0.1007],
           [0.0580, 0.0531, 0.1805,  ..., 0.0595, 0.0481, 0.0488],
           [0.0740, 0.0582, 0.0836,  ..., 0.0834, 0.0513, 0.0613],
           ...,
           [0.0609, 0.0808, 0.0940,  ..., 0.0750, 0.0579, 0.0563],
           [0.0940, 0.0579, 0.0625,  ..., 0.0605, 0.0566, 0.0800],
           [0.0347, 0.1090, 0.0651,  ..., 0.0415, 0.0822, 0.0602]],

          [[0.0713, 0.0696, 0.0610,  ..., 0.0518, 0.0681, 0.0424],
           [0.0995, 0.0449, 0.0531,  ..., 0.0194, 0.0455, 0.0827],
           [0.0792, 0.0897, 0.0594,  ..., 0.0341, 0.0447, 0.1238],
           ...,
           [0.0578, 0.1115, 0.0490,  ..., 0.0912, 0.0793, 0.0916],
           [0.1088, 0.0735, 0.0675,  ..., 0.0452, 0.0388, 0.0921],
           [0.0817, 0.0562, 0.0701,  ..., 0.0554, 0.0796, 0.0604]],

          ...,

          [[0.0477, 0.0677, 0.0459,  ..., 0.0571, 0.0909, 0.0790],
           [0.0645, 0.0570, 0.0538,  ..., 0.1012, 0.0618, 0.0555],
           [0.0779, 0.0998, 0.0554,  ..., 0.0543, 0.0790, 0.0684],
           ...,
           [0.0735, 0.0531, 0.0597,  ..., 0.0728, 0.0636, 0.0572],
           [0.0682, 0.0776, 0.0640,  ..., 0.0703, 0.0635, 0.0671],
           [0.0599, 0.0946, 0.0948,  ..., 0.1572, 0.0878, 0.0454]],

          [[0.0749, 0.0834, 0.0690,  ..., 0.0588, 0.0469, 0.0702],
           [0.0808, 0.0887, 0.0401,  ..., 0.1106, 0.0904, 0.0474],
           [0.0630, 0.0525, 0.0656,  ..., 0.0826, 0.0535, 0.0649],
           ...,
           [0.0954, 0.0685, 0.0830,  ..., 0.0772, 0.0390, 0.0527],
           [0.0751, 0.1311, 0.0592,  ..., 0.0547, 0.0957, 0.0692],
```

```
          [0.0841, 0.0517, 0.0462,  ..., 0.0726, 0.0990, 0.0743]],

         [[0.0448, 0.0504, 0.0980,  ..., 0.0628, 0.0405, 0.0326],
          [0.0600, 0.0585, 0.0775,  ..., 0.0950, 0.0999, 0.0700],
          [0.0575, 0.1256, 0.0897,  ..., 0.0512, 0.0349, 0.1047],
          ...,
          [0.0649, 0.0609, 0.0928,  ..., 0.0484, 0.0931, 0.0536],
          [0.0668, 0.0944, 0.0297,  ..., 0.0597, 0.0402, 0.0861],
          [0.0919, 0.0231, 0.0281,  ..., 0.0641, 0.0864, 0.0764]]]],
       grad_fn=<SoftmaxBackward0>))
```

3. Building the model (15 points)

Now, we can finally put it all together. For this, please, write the following missing code:

1. __init__: Add the attention layers (the encoder) to the model.
2. forward: Add the missing code for sequentially looping through the attention layers.
3. forward: Add the classifier, which consists of [1. pre_classifier, 2. ReLU activation, 3. classifier] and eventually returns the logit scores.

```
torch.manual_seed(0)

class DistillBertAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.n_layers=config.n_layers

        # embedding
        self.embeddings = Embeddings(config)

        # encoder
        # 1. START YOUR CODE HERE #

        # 1. END YOUR CODE HERE #

        # classification
        self.pre_classifier =  nn.Linear(in_features=config.dim, 
out_features=config.dim, bias=True)
        self.classifier =  nn.Linear(in_features=config.dim, 
out_features=config.n_classes, bias=True)

        self.attention_probs = {i: [] for i in range(config.n_layers)}

    def forward(self, input_ids):
        """
        Parameters:
            input_ids (torch.Tensor): torch.tensor(bs, max_seq_length) 
The token ids to embed.
```

```
        Returns: torch.tensor(bs, n_classes) The computed logit scores
for each class.
        """

        # Computing the embeddings
        hidden_states =
self.embeddings(input_ids=input_ids).to(self.config.device)

        # Iteratively going through the attention layers
        encoder_input = hidden_states
        # 2. START YOUR CODE HERE #




        # 2. END YOUR CODE HERE #

        # Pooling by selection the [CLS] token
        pooled_output = output[:, 0]  # (bs, dim)

        # Classification
        # 3. START YOUR CODE HERE #



        # 3. END YOUR CODE HERE #

        return logits

model = DistillBertAttention(config)
state_dict = torch.load('distilbert.pt')
_ = model.load_state_dict(state_dict)
_ = model.eval()

# Predict your output
logits = model(inputs['input_ids'])
logits

tensor([[-4.1982,  4.5568]], grad_fn=<AddmmBackward0>)
```

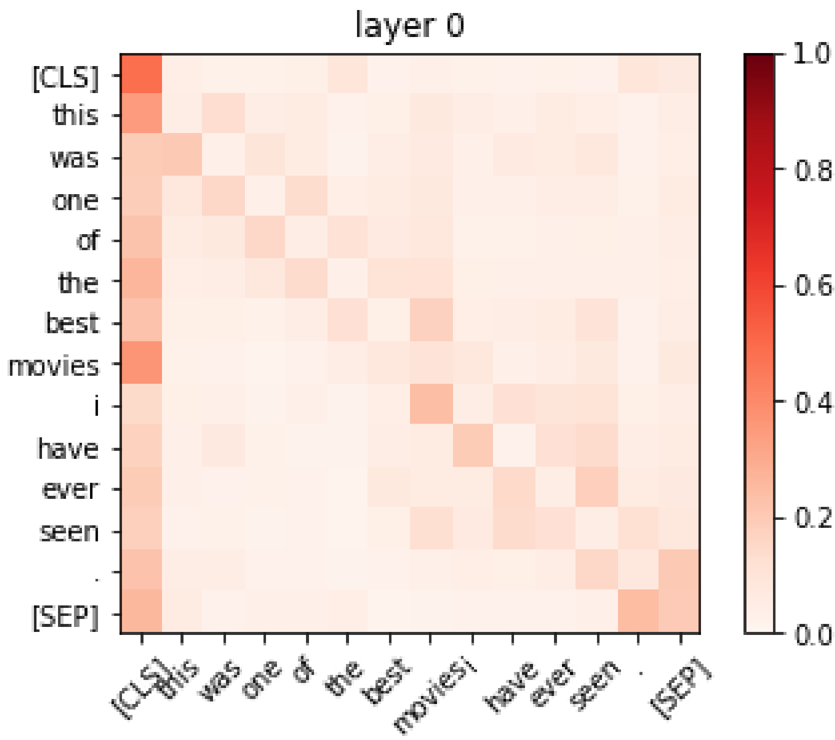
4. Visualize the attention weights (10 points)

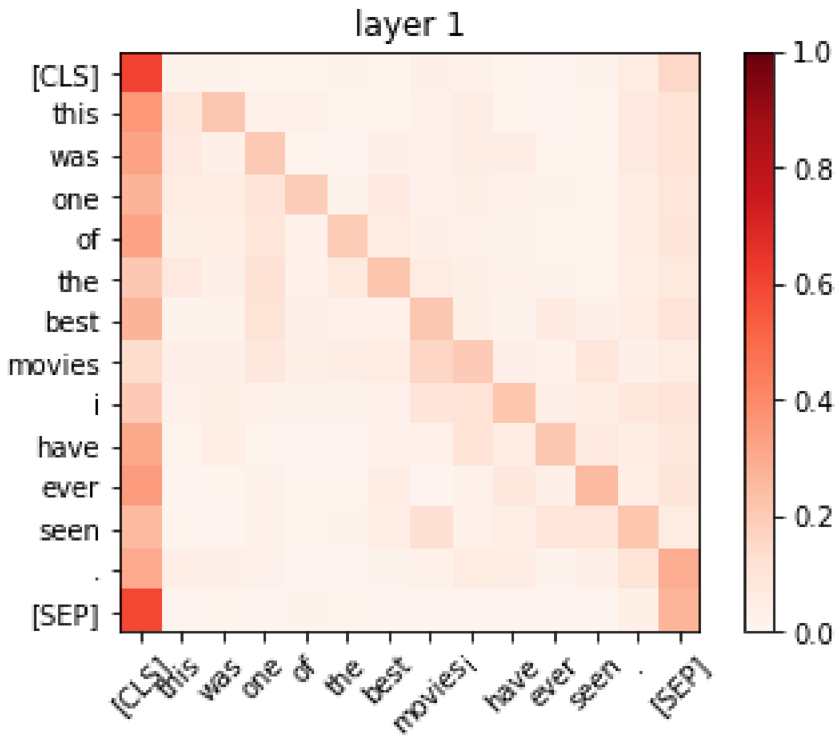
Let's now look at what tokens the model selects in its self-attention blocks.

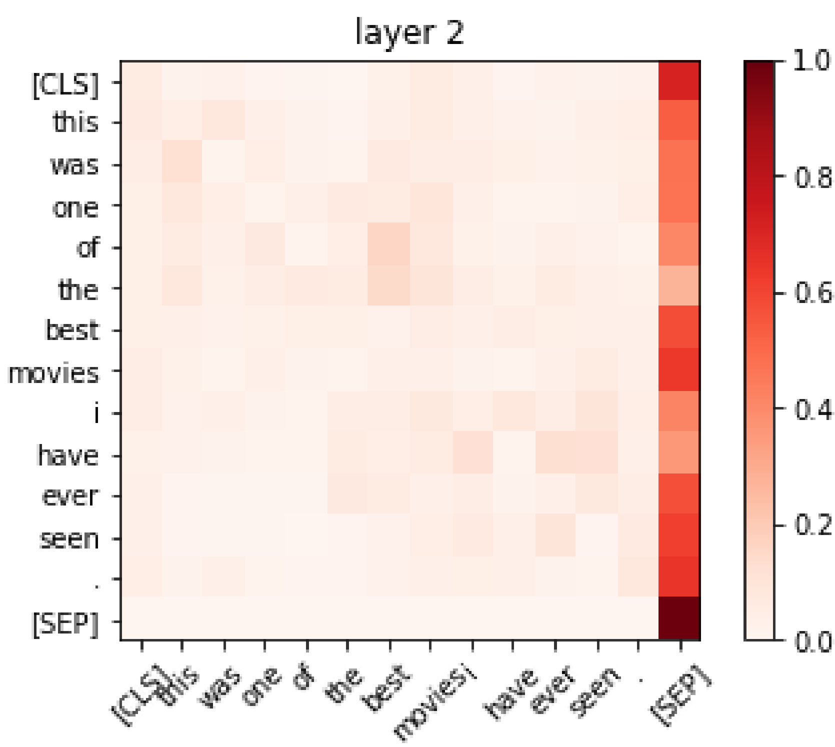
1. Use the tokenizer to map the 'input_ids' back to 'tokens'.
2. Extract attention probabilities for every layer by averaging over the attention heads in each layer. You should get a matrix of size [n_layers x seq_length x seq_length]

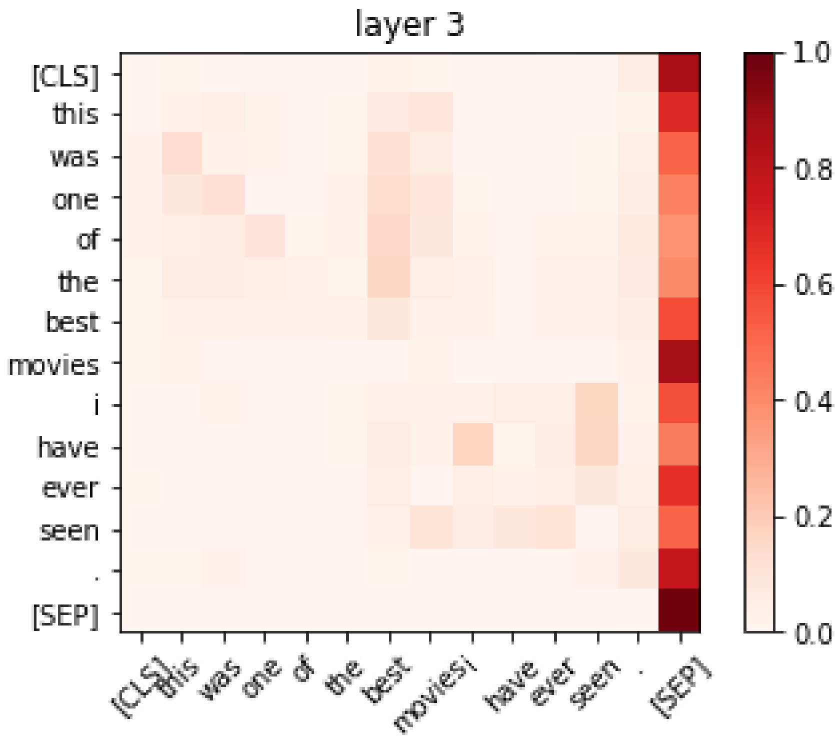
3. For each layer plot the resulting attention matrix. Hint: Use cmap='Reds' and vmin=0, vmax=1.

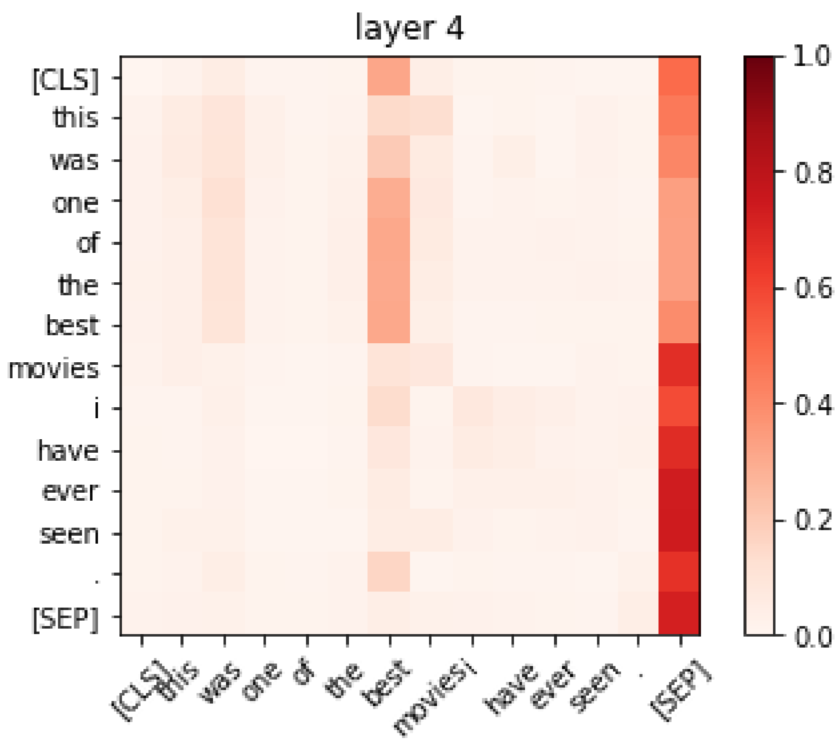
```
import numpy as np
import matplotlib.pyplot as plt

# YOUR CODE HERE #
```

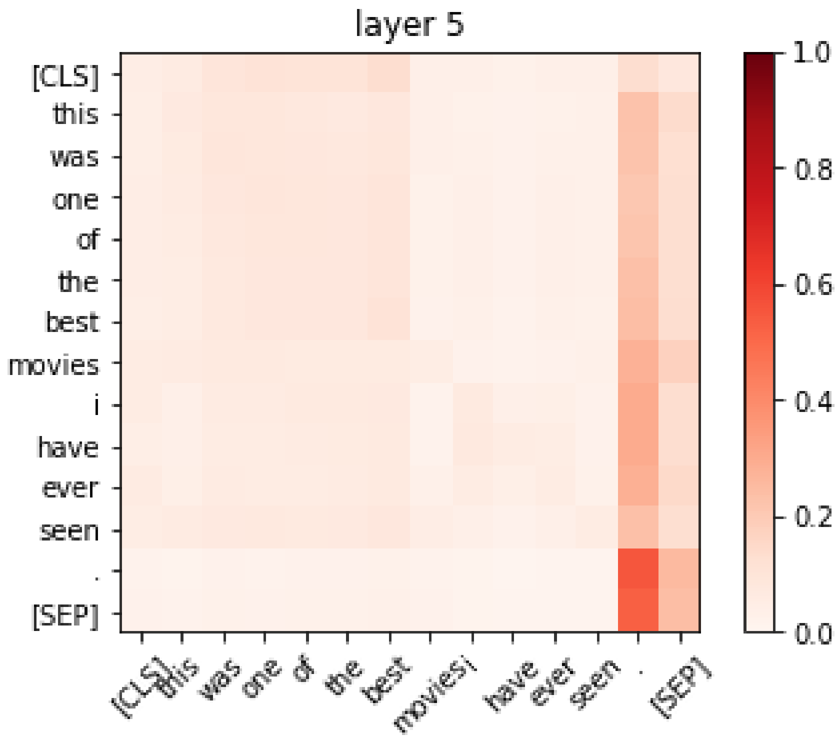



layer 1
[CLS]
this
was
one
of
the
best
movies
i
have
ever
seen
.
[SEP]
1.0
0.8
0.6
0.4
0.2
0.0


layer 2
[CLS]
this
was
one
of
the
best
movies
i
have
ever
seen
.
[SEP]
1.0
0.8
0.6
0.4
0.2
0.0


layer 3
[CLS]
this
was
one
of
the
best
movies
i
have
ever
seen
.
[SEP]
1.0
0.8
0.6
0.4
0.2
0.0


layer 4
[CLS]
this
was
one
of
the
best
movies
i
have
ever
seen
.
[SEP]
1.0
0.8
0.6
0.4
0.2
0.0


layer 5
[CLS]
this
was
one
of
the
best
movies
i
have
ever
seen
.
[SEP]
1.0
0.8
0.6
0.4
0.2
0.0