ARCHITECTING INTELLIGENCE

ASSIGNMENT 2

Ans 1. A perceptron is a simple learning model that works as a linear classifier. This means it separates different classes of data using a single straight line. Because of this limitation, a perceptron can only solve problems that are linearly separable.

The XOR problem is not linearly separable. In XOR, the output is true when the two inputs are different and false when they are the same. When these input–output pairs are considered together, there is no single straight line that can correctly separate all the true outputs from the false ones. Hence, a single-layer perceptron cannot solve the XOR problem.

This limitation was overcome with the introduction of multi-layer perceptrons. Multi-layer perceptrons contain one or more hidden layers between the input and output layers. These hidden layers, along with non-linear activation functions, allow the network to learn more complex patterns and form non-linear decision boundaries. As a result, multi-layer perceptrons are able to solve the XOR problem, which is impossible for a single-layer perceptron.

Ans 2. A linear layer performs a linear transformation of the input. When multiple linear layers are stacked together without any non-linear activation in between, their combined effect is still just a single linear transformation. This is because the composition of linear operations results in another linear operation. Therefore, stacking linear layers does not increase the model's ability to represent complex patterns, and the network remains equivalent to a single linear layer.

In deep networks, gradients are computed by repeatedly applying the chain rule while moving backward through layers. During this process, gradients from each layer are multiplied together. If these values are small, which often happens due to certain activation functions or weight initializations, repeated multiplication causes the gradients to become extremely small as they move toward earlier layers. This effect is known as the vanishing gradient problem and makes it difficult for deep networks to learn effectively.

The sigmoid activation function compresses its input into a very small range and tends to saturate for large positive or negative values. In these saturated regions, the gradient becomes very small, which worsens the vanishing gradient problem. ReLU, on the other hand, does not saturate for positive inputs and has a constant gradient in that region. Because of this, gradients can flow more easily through the network when ReLU is used, making it more effective for training deep networks.

Ans 3. Transformers process all tokens in parallel and do not have an inherent sense of word order. Without positional encoding, the model would treat a sentence as a set of words with no sequence information. Positional encoding is therefore required to provide information about the position of each token so that the model can understand order and structure.

Absolute positional encoding assigns a unique position value or embedding to each token position, but it is tied to fixed positions and does not generalize well to sequences longer than those seen during training. Sinusoidal positional encoding uses sine and cosine functions to represent positions, allowing the model to generalize better and capture relative position information.

Rotary positional embeddings (RoPE) encode position by rotating token embeddings based on their position. This directly models relative positions between tokens, which is more useful for attention. As a result, RoPE helps transformers handle long contexts more effectively.

Ans 4. In the attention mechanism, Query, Key, and Value are different learned representations of the input tokens. The Query represents what a token is looking for, the Key represents what each token contains, and the Value represents the actual information that will be passed forward. Attention works by comparing Queries with Keys to decide how much importance to give to each Value.

The attention scores are scaled by sqrt{d_k} to prevent them from becoming too large when the dimensionality of the keys is high. Without scaling, large dot-product values can push the softmax function into saturation, making learning unstable. Scaling keeps the attention scores in a reasonable range and helps with smoother training.

Diagonal values in the attention matrix are usually the highest because a token is most similar to itself. Since the Query and Key of the same token are closely related, their similarity score tends to be higher compared to other tokens, resulting in stronger self-attention along the diagonal.

Ans 5. Attention is split into multiple heads so that the model can focus on different types of relationships in the input at the same time. Each attention head learns to attend to different aspects of the sequence, such as short-range dependencies, long-range dependencies, or different semantic patterns. This makes the attention mechanism more expressive and improves the model’s overall understanding.

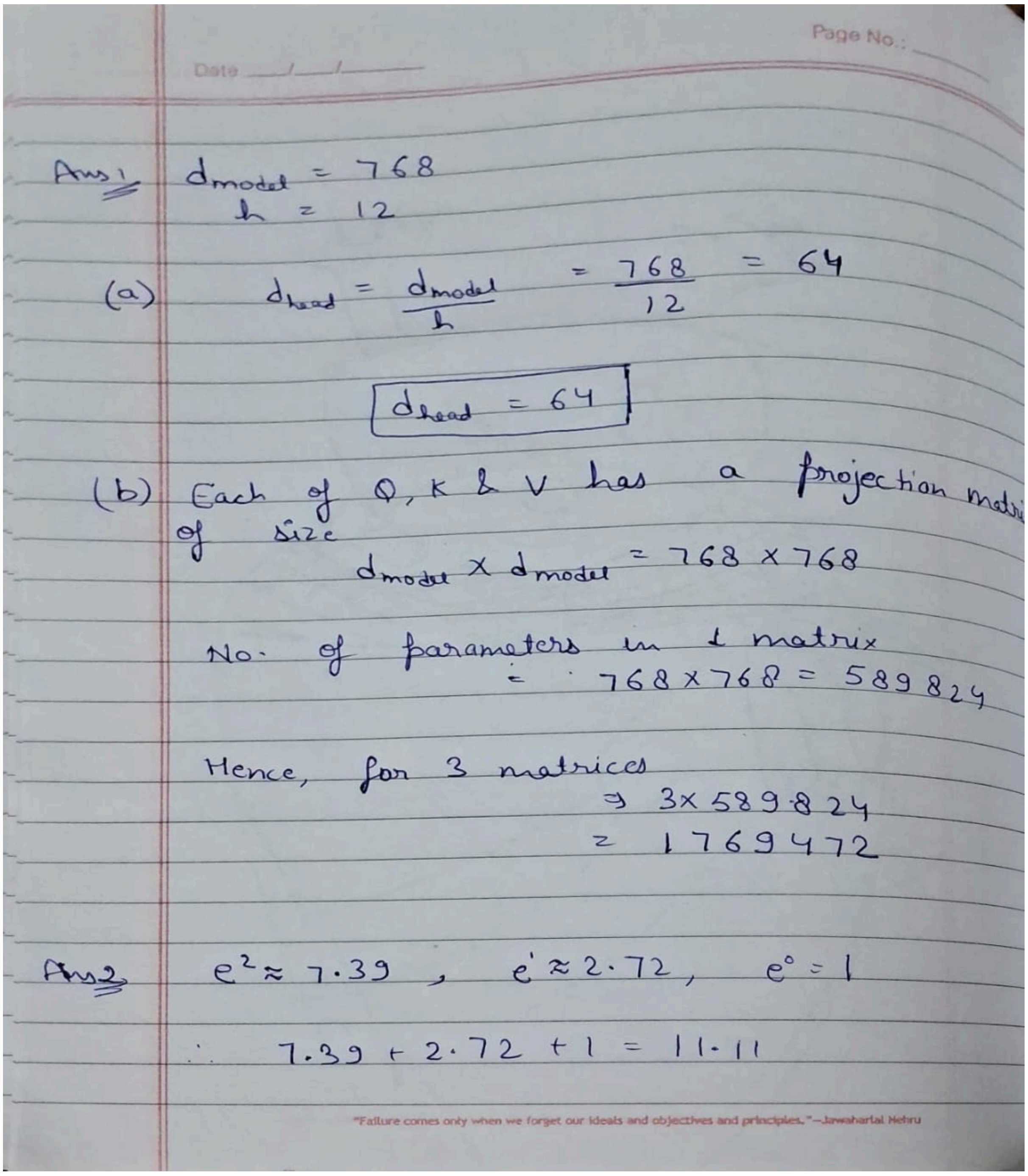
In multi-head attention, d_model represents the total embedding dimension of the model. The number of heads is denoted by h, which specifies how many parallel attention mechanisms are used. Each head works on a smaller subspace of the embedding, whose dimension is called d_head . Typically, the model dimension is split evenly across heads, so d_model = h X d_head .

Ans 6. Greedy decoding is suboptimal because it selects the most probable word at each time step without considering how that choice will affect the rest of the sequence. Once a word is chosen, the decision cannot be revised, so early mistakes can lead to poor overall sentences even if better sequences exist.

Beam search improves this by keeping multiple candidate sequences at each step instead of only one. By exploring several high-probability paths simultaneously, beam search is able to consider future outcomes and choose a sequence with higher overall probability. This makes beam search more effective at generating fluent and meaningful outputs.

For example, consider a translation task. Suppose the model generates the word “I” with highest probability first. At the next step, greedy decoding might choose “am” because it has the highest immediate probability, leading to the sequence “I am going…”. However, a slightly less probable choice like “will” at the second step could eventually produce a more correct sentence such as “I will go tomorrow.” Greedy decoding cannot recover from its early choice, while beam search can keep both possibilities and select the better full sentence.

SOLVING and CODING



Page No.:

Date __/__/____

Ans 1 $d_{model} = 768$
$h = 12$

(a) $d_{head} = \frac{d_{model}}{h} = \frac{768}{12} = 64$

$$\boxed{d_{head} = 64}$$

(b) Each of Q, K & V has a projection matri
of size
$d_{model} \times d_{model} = 768 \times 768$

No. of parameters in 1 matrix
$= 768 \times 768 = 589824$

Hence, for 3 matrices
$\Rightarrow 3 \times 589824$
$= 1769472$

Ans 2 $e^2 \approx 7.39$, $e^1 \approx 2.72$, $e^0 = 1$

$\therefore 7.39 + 2.72 + 1 = 11.11$

Date __/__/___ Page No.: ____

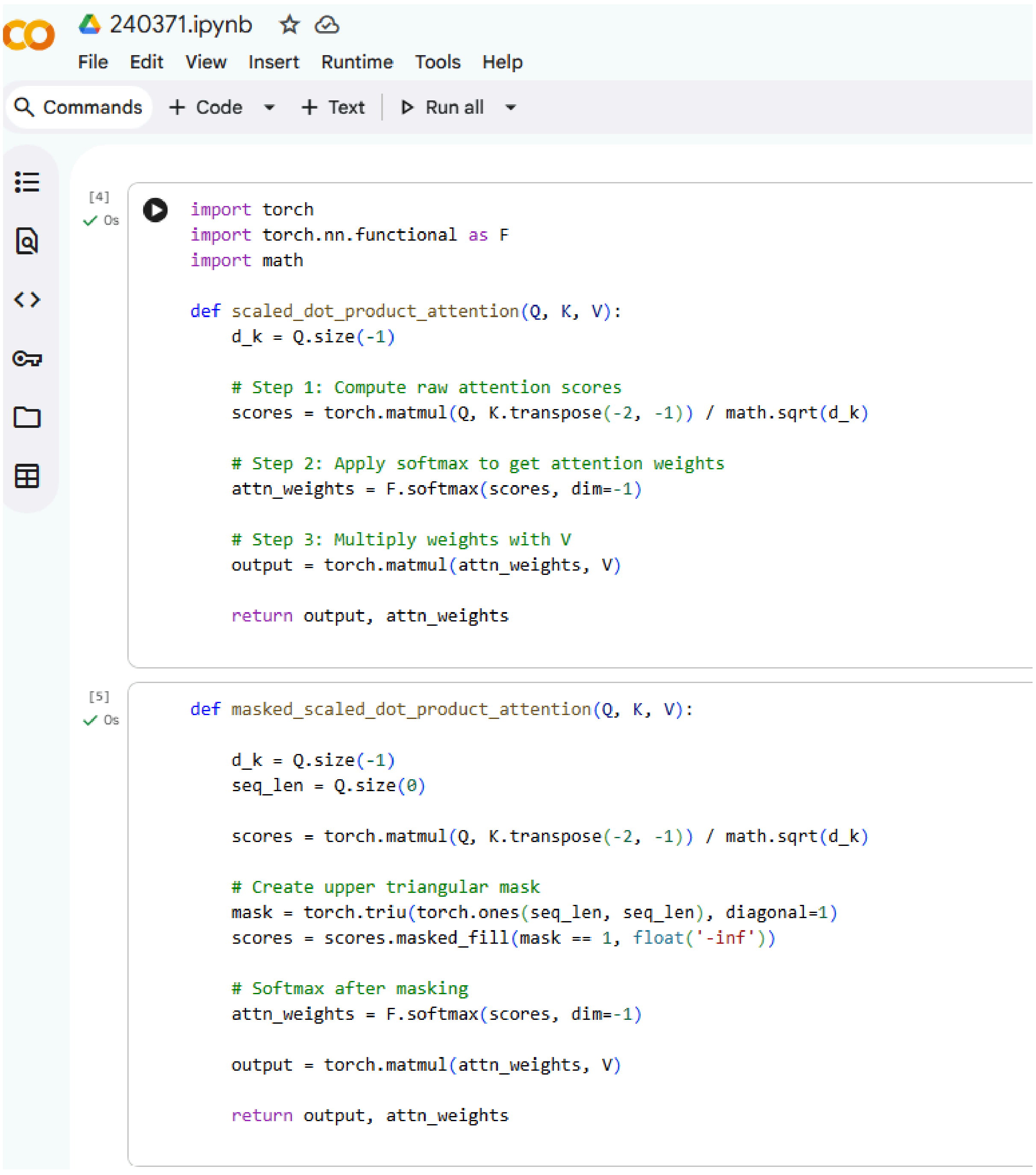
$$\text{Softmax}(2.0) = \frac{7.39}{11.11} \approx 0.665$$

$$\text{Softmax}(1.0) = \frac{2.72}{11.11} \approx 0.245$$

$$\text{Softmax}(0.0) = \frac{1}{11.11} \approx 0.090$$

Hence,
Softmax output $\approx [0.665, 0.245, 0.090]$

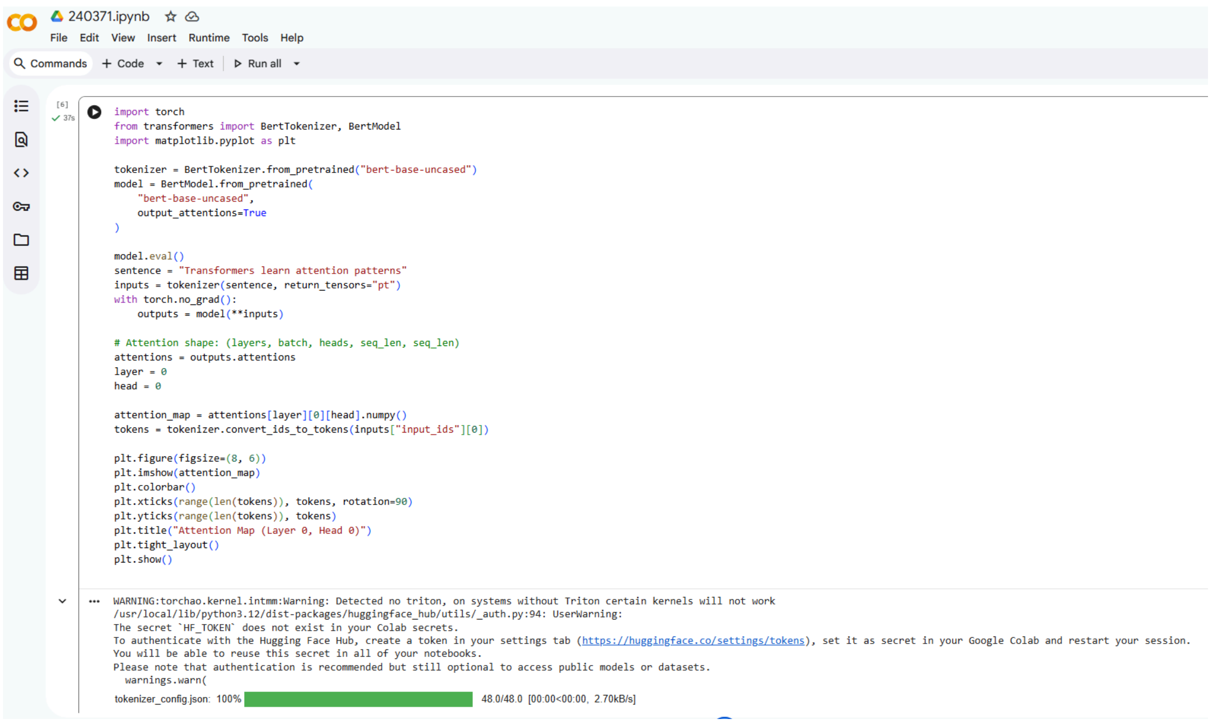
CODES OF ANSWERS 3 AND 4


240371.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
[4] 0s
[5] 0s

```
import torch
import torch.nn.functional as F
import math

def scaled_dot_product_attention(Q, K, V):
    d_k = Q.size(-1)

    # Step 1: Compute raw attention scores
    scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)

    # Step 2: Apply softmax to get attention weights
    attn_weights = F.softmax(scores, dim=-1)

    # Step 3: Multiply weights with V
    output = torch.matmul(attn_weights, V)

    return output, attn_weights
```

```
def masked_scaled_dot_product_attention(Q, K, V):

    d_k = Q.size(-1)
    seq_len = Q.size(0)

    scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)

    # Create upper triangular mask
    mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1)
    scores = scores.masked_fill(mask == 1, float('-inf'))

    # Softmax after masking
    attn_weights = F.softmax(scores, dim=-1)

    output = torch.matmul(attn_weights, V)

    return output, attn_weights
```

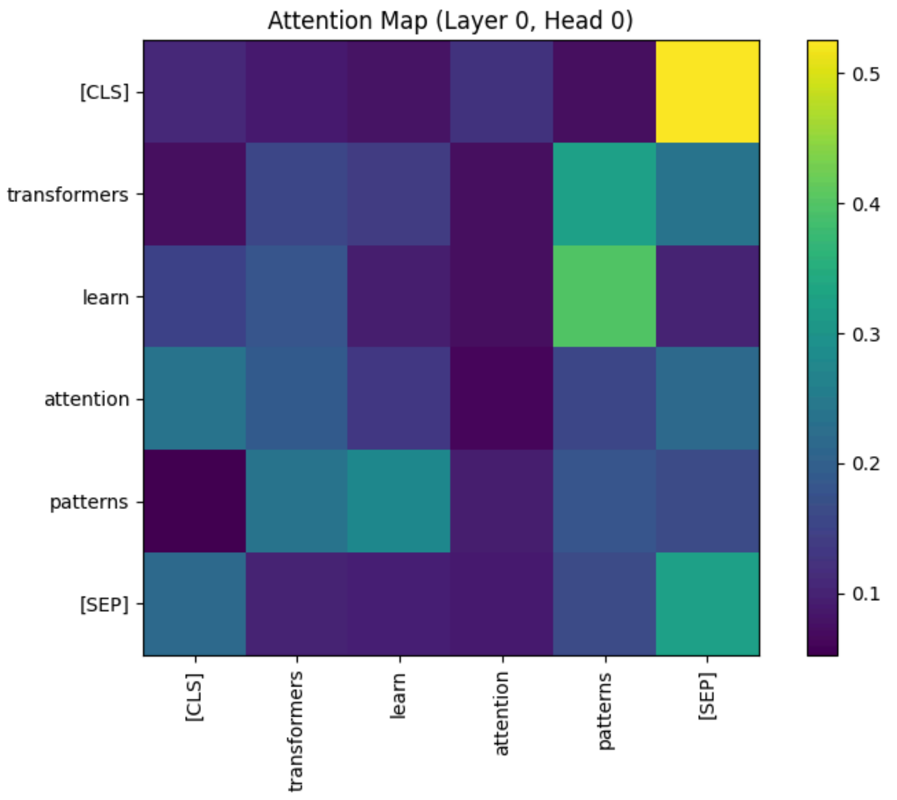
ANSWER 5


240371.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
[6]
37s

```python
import torch
from transformers import BertTokenizer, BertModel
import matplotlib.pyplot as plt

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertModel.from_pretrained(
    "bert-base-uncased",
    output_attentions=True
)

model.eval()
sentence = "Transformers learn attention patterns"
inputs = tokenizer(sentence, return_tensors="pt")
with torch.no_grad():
    outputs = model(**inputs)

# Attention shape: (layers, batch, heads, seq_len, seq_len)
attentions = outputs.attentions
layer = 0
head = 0

attention_map = attentions[layer][0][head].numpy()
tokens = tokenizer.convert_ids_to_tokens(inputs["input_ids"][0])

plt.figure(figsize=(8, 6))
plt.imshow(attention_map)
plt.colorbar()
plt.xticks(range(len(tokens)), tokens, rotation=90)
plt.yticks(range(len(tokens)), tokens)
plt.title("Attention Map (Layer 0, Head 0)")
plt.tight_layout()
plt.show()
```

```
WARNING:torchao.kernel.intmm:Warning: Detected no triton, on systems without Triton certain kernels will not work
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%  48.0/48.0 [00:00<00:00, 2.70kB/s]
```

```
  warnings.warn(
••• tokenizer_config.json: 100%  48.0/48.0 [00:00<00:00, 2.70kB/s]
    vocab.txt: 100%  232k/232k [00:00<00:00, 6.86MB/s]
    tokenizer.json: 100%  466k/466k [00:00<00:00, 3.62MB/s]
    config.json: 100%  570/570 [00:00<00:00, 24.7kB/s]
    model.safetensors: 100%  440M/440M [00:05<00:00, 91.4MB/s]
```


Attention Map (Layer 0, Head 0)
[CLS]
transformers
learn
attention
patterns
[SEP]