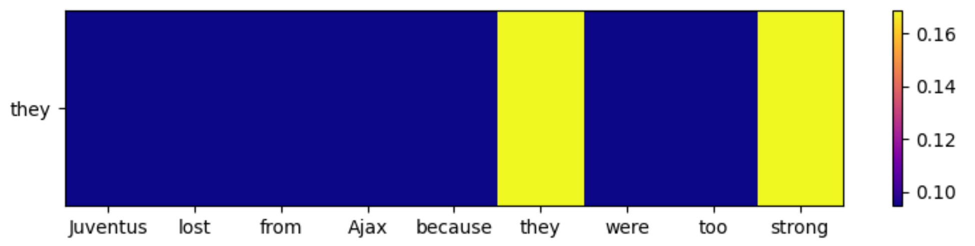
Transformer Language Models

The attention mechanism

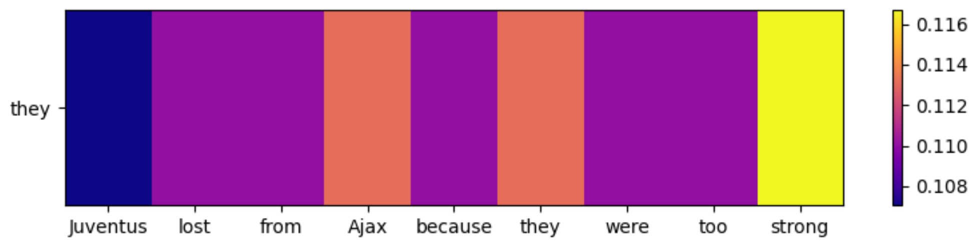
Question 1

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

np.random.seed(1)
plt.rcParams["figure.figsize"] = 8, 2

xaxis = np.array(range(0, 9))
first, last, step = xaxis[0], xaxis[-1], xaxis[1] - xaxis[0]
extent = [first - step / 2., last + step / 2., 0, 1]

s1 = "Juventus lost from Ajax because they were too strong".split(" ")
s2 = "Juventus lost from Ajax because they were too weak".split(" ")
reps1 = np.array([[1, 0, 0, -1, 0, 0, 0, 0, -1],
                  [0, 0, 0,  0, 0, 1, 0, 0,  1]])  # 2x9
reps2 = np.array([[1, 0, 0, -1, 0, 0, 0, 0,  1],
                  [0, 0, 0,  0, 0, 1, 0, 0,  1]])  # 2x9

Wk = np.array([[1, 0],
               [0, 1],
               [0, 0]])  # 3x2
Wq = np.array([[1, 0],
               [0, 1],
               [0, 0]])  # 3x2
Wv = np.array([[1, 0],
               [0, 1]])  # 2x2


def softmax(x, axis=0):
    """Compute softmax values for each sets of scores in x."""
    return np.exp(x) / np.sum(np.exp(x), axis=axis)


def attend(reps):
    k = Wk @ reps  # 3x2 x 2x9 = 3x9
    q = Wq @ reps  # 3x2 x 2x9 = 3x9
    v = Wv @ reps  # 2x2 x 2x9 = 2x9
    scaler = np.sqrt(Wq.shape[0])
    a = softmax((k.transpose() @ q) / scaler, axis=0)  # 9x3 x 3x9 = 9x9
    o = v @ a  # 2x9 x 9x9 = 2x9
    return a, o


def colormap(y):
    plt.imshow(y[np.newaxis, :], cmap="plasma", aspect="auto", extent=extent)
    plt.xticks(xaxis, s)
    plt.yticks([.5], ['they'])
    plt.colorbar()
    plt.tight_layout()
    plt.show()


for s, reps in zip((s1, s2), (reps1, reps2)):
    a1, v1 = attend(reps)
    a2, v2 = attend(v1)
    print("After applying self-attention once")
    colormap(a1[:, 5])  # 5 is the index of the word "they". We want to plot the attention when "they" is the query
    print("After applying self-attention twice")
    colormap(a2[:, 5])
```

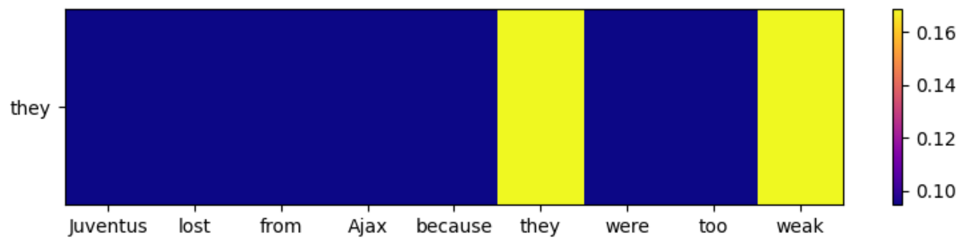
```
After applying self-attention once
```



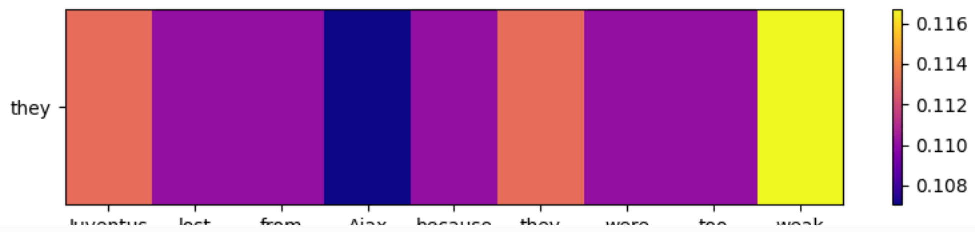
```
After applying self-attention twice
```



```
After applying self-attention once
```



```
After applying self-attention twice
```



Below is the solution in text form, as well as a handwritten writing-out of the formulas:

Let **s1** and **s2** be two sequences of word embeddings for sentences "Juventus lost from Ajax because they were too strong" and "Juventus lost from Ajax because they were too weak" respectively.
Let the representation matrices for **s1** and **s2** be **reps1** and **reps2** respectively, given by:

$$\mathbf{reps1} = \begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}, \mathbf{reps2} = \begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Let the weight matrices **Wk**, **Wq**, and **Wv** be given by:

$$\mathbf{Wk} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \mathbf{Wq} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \mathbf{Wv} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The softmax function applied to a 2D matrix can be defined as:

$$\text{softmax}_{ij}(\mathbf{X}, \text{axis}) = \begin{cases} \frac{\exp(\mathbf{X}_{ij})}{\sum_r \exp(\mathbf{X}_{rj})} & \text{if axis} = 0 \text{ (sum over rows)} \\ \frac{\exp(\mathbf{X}_{ij})}{\sum_c \exp(\mathbf{X}_{ic})} & \text{if axis} = 1 \text{ (sum over columns)} \end{cases}$$

Where $\mathbf{X}$ is a 2D matrix with elements $\mathbf{X}_{ij}$, and the denominator sums over elements from different rows but the same column if axis = 0, and different columns but the same row otherwise.
The attention mechanism can be described by the following steps:

1. Compute the key, query, and value matrices as:

$$\mathbf{K} = \mathbf{Wk} \cdot \mathbf{reps}, \quad \mathbf{Q} = \mathbf{Wq} \cdot \mathbf{reps}, \quad \mathbf{V} = \mathbf{Wv} \cdot \mathbf{reps}$$

2. Compute the scaled dot-product attention:

   Given a matrix $\mathbf{Q}$ representing queries and a matrix $\mathbf{K}$ representing keys, the attention weights matrix $\mathbf{A}$ can be computed as:

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{K}^\top \mathbf{Q}}{\sqrt{d_k}}, \text{axis} = 0\right)$$

   Here, d_k is the dimensionality of the keys, and the softmax function is applied over rows (axis = 0) of the scaled dot-product $\mathbf{K}^\top \mathbf{Q}$.

3. Obtain the output representation:

$$\mathbf{O} = \mathbf{V} \cdot \mathbf{A}$$

To apply self-attention to the representations, we perform the following operations:

$$\mathbf{A1}, \mathbf{O1} = \text{attend}(\mathbf{reps1}), \mathbf{A2}, \mathbf{O2} = \text{attend}(\mathbf{O1})$$

where attend function performs the operations as described in the attention mechanism above.

$$\overset{S = 9\ (\#\text{words})}{\longleftrightarrow}$$

$$\text{reps}_1 = \begin{bmatrix} 1&0&0&1&0&0&0&0&-1 \\ 0&0&0&0&0&0&0&1&0&1 \end{bmatrix} \updownarrow E=2 \text{ (embedding dimension)}$$

$$\text{reps}_2 = \begin{bmatrix} 1&0&0&1&0&0&0&0&1 \\ 0&0&0&0&0&0&0&1&0&-1 \end{bmatrix} \updownarrow E=2$$

$$W_K = \overset{\overset{E}{\longleftrightarrow}}{\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}} \updownarrow d_k \quad W_Q = \overset{\overset{E}{\longleftrightarrow}}{\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}} \updownarrow d_q \quad W_V = \overset{\overset{E}{\longleftrightarrow}}{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}} \updownarrow d_v$$

```
for reps in [reps1,reps2]:
    for layer in [first layer, second layer]:
        For self-attention, key/value tokens and query tokens are the same (there are S of them). We will
        write it however as S_k/v and S_q
        k = Wk . reps  Dimensions: dk×E . E×S_k/v -> dk×S_k/v
        q = Wq . reps  Dimensions: dq×E . E×S_q -> dq×S_q
        v = Wv . reps  Dimensions: dv×E . E×S_k/v -> dv×S_k/v
        scores = k^T . q  Dimensions: S_k/v×dk . dq×S_q -> S_k/v×S_q. Let's write this as S_k×S_q.
        n = normalization-ready-scores = e^(scores/sqrt(d_k))  Dimensions: S_k/v×S_q -> S_k/v×S_q(element-wise)
        probabilities: = (see matrix below)
        reps = v . probabilities  Dimensions: dv×S_k/v . S_k/v×S_q-> dv×S_q
        In our simple case, dv = E
```

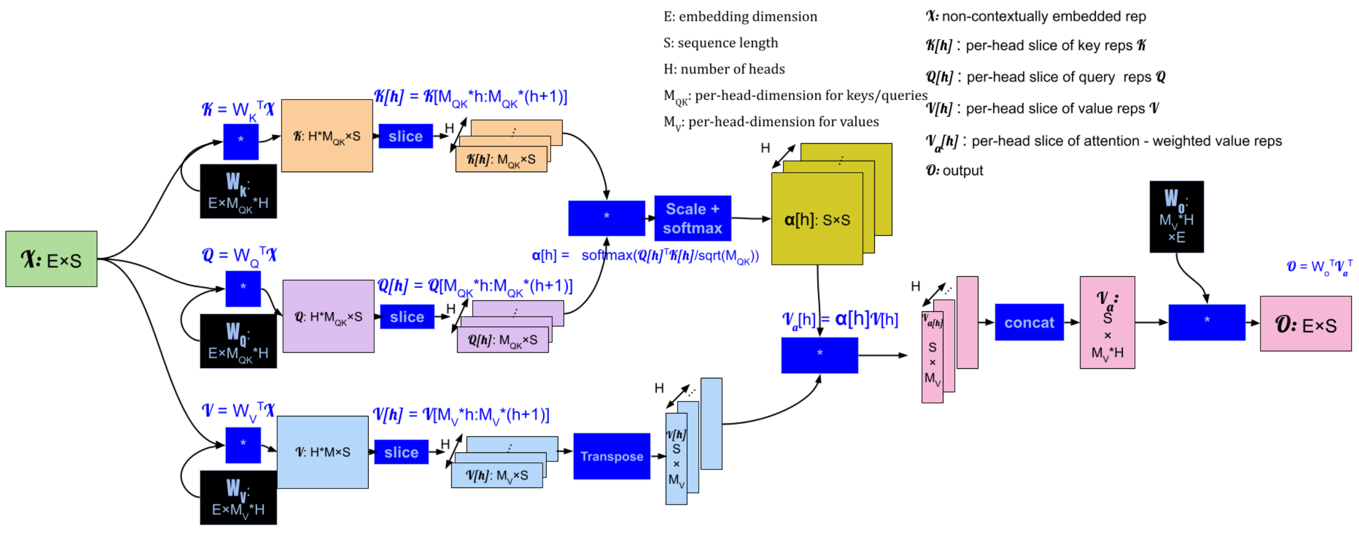
$$n = \text{normalization-ready-scores} = e^{\frac{scores}{\sqrt{d_k}}}$$

$$\text{probabilities:} = \begin{array}{c|ccccc} & 1 & \cdots & j & \cdots & S_q \\ \hline 1 & & & \vdots & & \\ \vdots & & & \vdots & & \\ i & \cdots & & \dfrac{n_{ij}}{\sum_i n_{ij}} & & \\ \vdots & & & & & \\ S_{k/v} & & & & & \\ \hline & \Sigma=1 & \Sigma=1 & \Sigma=1 & \Sigma=1 & \Sigma=1 \end{array}$$

Sum over key/value-elements is 1, because we want a weighted average of the key elements

Multi-head self-attention

Question 2


E: embedding dimension
S: sequence length
H: number of heads
M_QK: per-head-dimension for keys/queries
M_V: per-head-dimension for values
X: non-contextually embedded rep
K[h] : per-head slice of key reps K
Q[h] : per-head slice of query reps Q
V[h] : per-head slice of value reps V
V_a[h] : per-head slice of attention - weighted value reps
O: output
X: E×S
K = W_K^T X
Q = W_Q^T X
V = W_V^T X
W_K: E×M_QK*H
W_Q: E×M_QK*H
W_V: E×M_V*H
K: H*M_QK×S
Q: H*M_QK×S
V: H*M×S
slice
K[h] = K[M_QK*h:M_QK*(h+1)]
Q[h] = Q[M_QK*h:M_QK*(h+1)]
V[h] = V[M_V*h:M_V*(h+1)]
K[h]: M_QK×S
Q[h]: M_QK×S
V[h]: M_V×S
H
Transpose
V[h] S × M_V
Scale + softmax
α[h] = softmax(Q[h]^T K[h]/sqrt(M_QK))
α[h]: S×S
V_a[h] = α[h]V[h]
V_a[h] S × M_V
concat
V_a: S × M_V*H
W_O: M_V*H ×E
O = W_o^T V_a^T
O: E×S

Pretraining, freezing and finetuning

Question 3

We call these objectives 'self-supervised' because the ground-truth labels that need to be guessed by the pretraining objective are provided without human labelers. For example, we can let the model mask random words in text, and have it then try to guess what is behind the mask. The benefit is that it is much cheaper to generate input-target_label combos, and hence we will be able to generate many more than if we were to need manual labeling.

Question 4

You can leverage the unlabeled tweets by training a model that tries to solve a self-supervised task (such as Language Modelling), and subsequently reusing part of those trained parameters when you start trainnig on the target task (sentiment classification).

Question 5

For inference, both the full-fine-tuning approach and the LoRa approach are best. Because the LoRa matrices are merged into the original matrix during inference, they do not lead to additional inference time.

For storage, we will want to avoid fine-tuning. Both the adapter-method and the LoRa approach can be beneficial, depending on W_out and R.

Full-fine-tuning requires updating 200*50 = 10 000 parameters, while LoRa requires 200*5 + 5*50 = 1250 parameters

Byte-pair encoding

Question 6

First, we split the text into words and their frequency, and add a delimiter to each word (we use '_' as the delimiter, '\w' is also often used):

```
corpus = "abbc abb abc abb"
counts = dict()
for i in corpus.split(" "):
  i += "_"
  counts[i] = counts.get(i, 0) + 1
print("counts: ", counts)
```

```
counts:  {'abbc_': 1, 'abb_': 2, 'abc_': 1}
```

Then, we list byte pairs and their frequencies

```
def get_bp_freqs(cnt):
  byte_pairs = dict()
  for word, freq in cnt.items():
    for c1,c2 in zip(word[:-1],word[1:]):
      byte_pairs[(c1,c2)] = byte_pairs.get((c1,c2), 0) + freq
  return byte_pairs
byte_pairs = get_bp_freqs(counts)
print("byte_pairs: ", byte_pairs)
```

```
byte_pairs:  {('a', 'b'): 4, ('b', 'b'): 3, ('b', 'c'): 2, ('c', '_'): 2, ('b', '_'): 2}
```

Then, we take the byte pair with the highest frequency, use a new symbol for it, and replace that in our counts.

```
new_symbols = ['A','B','C','D','E']
new_symbol = new_symbols[0]
symbol_table = []
def get_symbol_table_and_counts(byte_pairs, counts, new_symbol, symbol_table):
  import re
  max_bp = max(byte_pairs, key=byte_pairs.get)
  old_pair = "".join(max_bp)
  symbol_table.append((new_symbol, old_pair))
  for word, freq in dict(counts).items():
    new_word = re.sub(old_pair,new_symbol,word)
    del counts[word]
    counts[new_word] = freq
  return symbol_table, counts
symbol_table, counts = get_symbol_table_and_counts(byte_pairs, counts, new_symbol, symbol_table)
print("symbol_table: ", symbol_table)
print("counts: ", counts)
```

```
symbol_table:  [('A', 'ab')]
counts:  {'Abc_': 1, 'Ab_': 2, 'Ac_': 1}
```

And we recalculate byte pair frequencies

```
byte_pairs = get_bp_freqs(counts)
print("byte_pairs: ", byte_pairs)
```

```
byte_pairs:  {('A', 'b'): 3, ('b', 'c'): 1, ('c', '_'): 2, ('b', '_'): 2, ('A', 'c'): 1}
```

```
new_symbol = new_symbols[1]
symbol_table, counts = get_symbol_table_and_counts(byte_pairs, counts, new_symbol, symbol_table)
print("symbol_table: ", symbol_table)
print("counts: ", counts)
```

```
symbol_table:  [('A', 'ab'), ('B', 'Ab')]
counts:  {'Bc_': 1, 'B_': 2, 'Ac_': 1}
```

Once more

```
byte_pairs = get_bp_freqs(counts)
new_symbol = new_symbols[2]
symbol_table, counts = get_symbol_table_and_counts(byte_pairs, counts, new_symbol, symbol_table)
print("byte_pairs: ", byte_pairs)
print("symbol_table: ", symbol_table)
print("counts: ", counts)
```

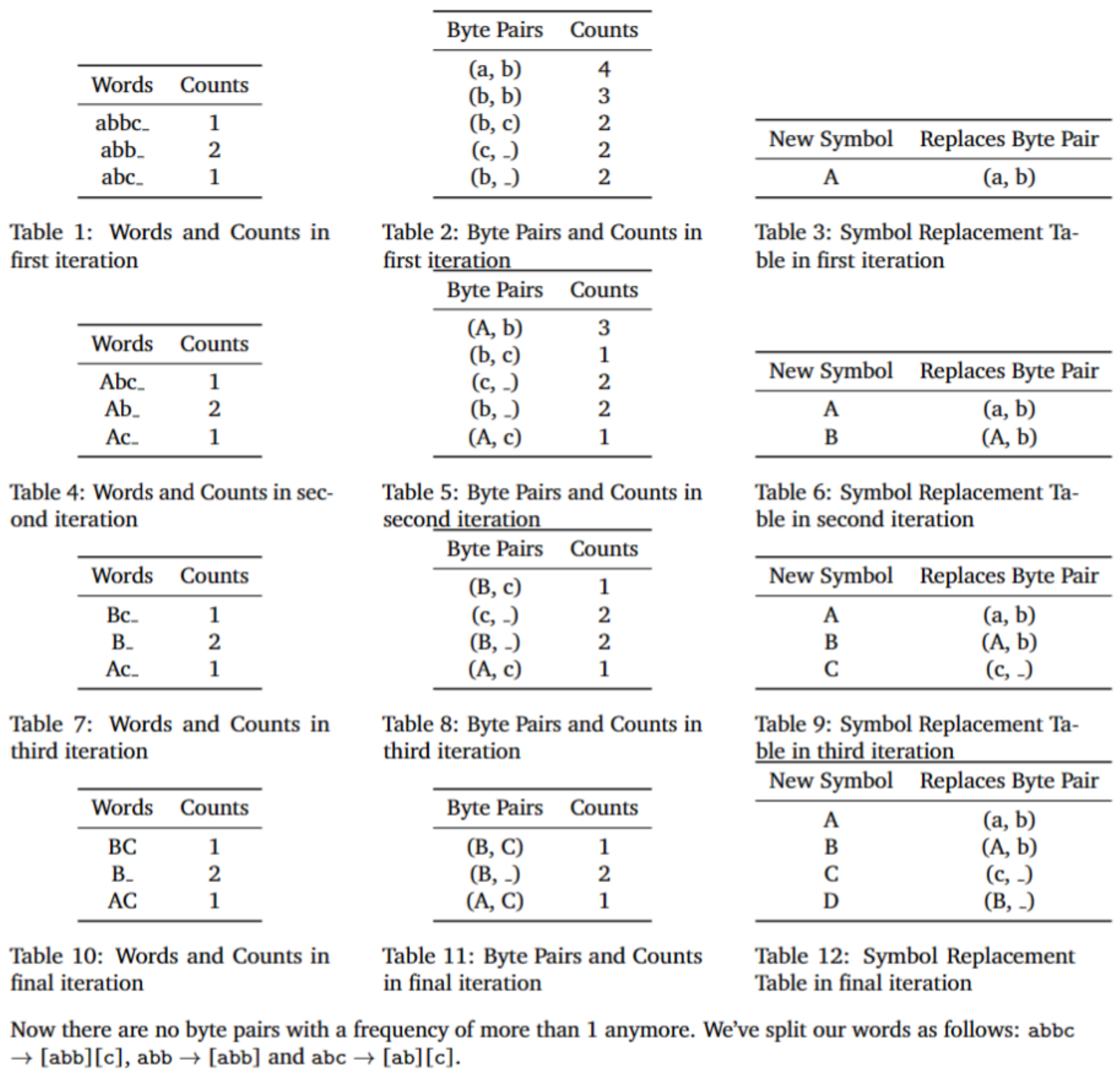
```
byte_pairs:  {('B', 'c'): 1, ('c', '_'): 2, ('B', '_'): 2, ('A', 'c'): 1}
symbol_table:  [('A', 'ab'), ('B', 'Ab'), ('C', 'c_')]
counts:  {'BC': 1, 'B_': 2, 'AC': 1}
```

And a final time

```
byte_pairs = get_bp_freqs(counts)
new_symbol = new_symbols[3]
symbol_table, counts = get_symbol_table_and_counts(byte_pairs, counts, new_symbol, symbol_table)
print("byte_pairs: ", byte_pairs)
print("symbol_table: ", symbol_table)
print("counts: ", counts)
```

```
byte_pairs:  {('B', 'C'): 1, ('B', '_'): 2, ('A', 'C'): 1}
symbol_table:  [('A', 'ab'), ('B', 'Ab'), ('C', 'c_'), ('D', 'B_')]
counts:  {'BC': 1, 'D': 2, 'AC': 1}
```

Printed out, it looks like so:



Words	Counts
abbc_	1
abb_	2
abc_	1

Table 1: Words and Counts in first iteration

Byte Pairs	Counts
(a, b)	4
(b, b)	3
(b, c)	2
(c, _)	2
(b, _)	2

Table 2: Byte Pairs and Counts in first iteration

New Symbol	Replaces Byte Pair
A	(a, b)

Table 3: Symbol Replacement Table in first iteration

Words	Counts
Abc_	1
Ab_	2
Ac_	1

Table 4: Words and Counts in second iteration

Byte Pairs	Counts
(A, b)	3
(b, c)	1
(c, _)	2
(b, _)	2
(A, c)	1

Table 5: Byte Pairs and Counts in second iteration

New Symbol	Replaces Byte Pair
A	(a, b)
B	(A, b)

Table 6: Symbol Replacement Table in second iteration

Words	Counts
Bc_	1
B_	2
Ac_	1

Table 7: Words and Counts in third iteration

Byte Pairs	Counts
(B, c)	1
(c, _)	2
(B, _)	2
(A, c)	1

Table 8: Byte Pairs and Counts in third iteration

New Symbol	Replaces Byte Pair
A	(a, b)
B	(A, b)
C	(c, _)

Table 9: Symbol Replacement Table in third iteration

Words	Counts
BC	1
B_	2
AC	1

Table 10: Words and Counts in final iteration

Byte Pairs	Counts
(B, C)	1
(B, _)	2
(A, C)	1

Table 11: Byte Pairs and Counts in final iteration

New Symbol	Replaces Byte Pair
A	(a, b)
B	(A, b)
C	(c, _)
D	(B, _)

Table 12: Symbol Replacement Table in final iteration

Now there are no byte pairs with a frequency of more than 1 anymore. We've split our words as follows: abbc → [abb][c], abb → [abb] and abc → [ab][c].

Cross-modal search

Question 7

```
import numpy as np
I = np.array([[0.2,   -1.5, 0.6,  0.3],
              [-0.4,  -0.5, 1.2,  -0.1]])
T = np.array([[0.9,   0.3],
              [-0.8,  -0.1],
              [0.3,   -0.5]])
Wi = np.array([[0.6, 0.3, -0.2],
               [1.3, 0.3, -0.6],
               [0.8, 0.2, -0.3],
               [0.4, 0.4, 0.7]])
Wt = np.array([[-1.2, 0.5,  0.8],
               [0.4,  -0.1, -1.6]])
I_emb = I @ Wi # (Ni, Di) @ (Di, De) = (Ni, De)
T_emb = T @ Wt # (Nt, Dt) @ (Dt, De) = (Nt, De)
print("I_emb\r\n", I_emb)
print("T_emb\r\n", T_emb)
normalized_I_emb = I_emb / np.sqrt((I_emb**2).sum(1))[:,np.newaxis]
normalized_T_emb = T_emb / np.sqrt((T_emb**2).sum(1))[:,np.newaxis]
print("normalized I_emb\r\n", normalized_I_emb)
print("normalized T_emb\r\n", normalized_T_emb)
cos_match = normalized_I_emb @ normalized_T_emb.T # (Ni, De) @ (De, Nt) = (Ni, Nt)
print("cos_match matrix\r\n", cos_match)
# Max match per image
max_match = np.argmax(cos_match, axis=1)
print("max_match indexes\r\n",max_match)
```

```
I_emb
 [[-1.23 -0.15  0.89]
 [ 0.03 -0.07 -0.05]]
T_emb
 [[-0.96  0.42  0.24]
 [ 0.92 -0.39 -0.48]
 [-0.56  0.2   1.04]]
normalized I_emb
 [[-0.80623244 -0.09832103  0.58337144]
 [ 0.32929278  0.76834982  0.5488213 ]]
```

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.