NLP Assignment 4 Deliverables
Sharvita Paithankar

Part 1:

```
mport random
from torch import manual_seed, randn, unsqueeze
from numpy import round as npround

random.seed(42)
manual_seed(42)


input_tensor = randn(5, 8)
# [batch=1, seq_len=5, d_model=8]
batched_input = unsqueeze(input_tensor, 0)

sh_attn = TransformerLayer(d_model=8, d_internal=32, d_ff=64, n_heads=1,
max_seq_len=5)
print('Single-head output')

output = sh_attn(batched_input).squeeze(0)
print(npround(output.detach().numpy(), 4), '\n')
print('Single-head attention map')
print(npround(sh_attn.last_map.detach().numpy(), 4), '\n')

print('='*60, '\n')



mh_attn = TransformerLayer(d_model=8, d_internal=16, d_ff=128, n_heads=2,
max_seq_len=5)

print('Multi-head output')
output = mh_attn(batched_input).squeeze(0)
print(npround(output.detach().numpy(), 4), '\n')
print('Multi-head attention maps')
print(npround(mh_attn.last_map.detach().numpy(), 4))
```

```
Single-head output
[[ 2.2838  1.5244  0.8004 -2.9931  0.589  -0.8144 -0.1878 -2.4291]
```

```
 [-1.1199  1.5985 -0.2764 -1.6976 -0.5537 -0.2992 -1.1663 -0.1034]
 [ 1.6057 -0.0639 -0.236   0.0499 -0.5856  1.5798  0.844   0.9629]
 [ 0.3906  0.0176  1.5224 -1.2355  1.6741  1.1942 -1.2787 -0.9399]
 [-1.5059  1.3385 -0.244   1.3421  1.5637  0.644  -1.9173 -1.3539]]

Single-head attention map
[[[0.1903 0.3172 0.1341 0.1319 0.2265]
  [0.2291 0.2353 0.2829 0.1432 0.1095]
  [0.2137 0.1621 0.1954 0.1957 0.2331]
  [0.1552 0.1517 0.1744 0.2308 0.288 ]
  [0.1125 0.1147 0.1225 0.1859 0.4644]]]

============================================================

Multi-head output
[[ 1.5158  1.0291  0.5479 -1.8851  0.5383 -1.0615 -0.0882 -1.7973]
 [-0.6814  1.7653 -0.6277 -1.1421 -0.2932 -0.0086 -0.9963  0.5146]
 [ 1.4643 -0.4265 -0.5724  0.9202 -0.3251  1.3543  1.1753  1.6216]
 [-0.0789  0.3658  1.0965 -0.6266  1.7939  1.0766 -1.3486 -0.6459]
 [-1.8123  0.8444 -0.5449  2.7515  2.1552 -0.337  -1.1889 -1.4461]]

Multi-head attention maps
[[[0.0688 0.2591 0.1634 0.1492 0.3595]
  [0.1031 0.2269 0.2934 0.1487 0.2279]
  [0.3352 0.1856 0.1292 0.2322 0.1178]
  [0.1696 0.2226 0.1257 0.1595 0.3227]
  [0.0853 0.2292 0.1713 0.1358 0.3784]]

 [[0.0787 0.0955 0.5467 0.0553 0.2238]
  [0.1075 0.1515 0.2709 0.1631 0.3069]
  [0.1018 0.2352 0.3694 0.1336 0.16  ]
  [0.1242 0.1698 0.3983 0.1144 0.1933]
  [0.4868 0.1637 0.0387 0.2139 0.097 ]]]
```

Part 2:

```
vocab_size = 10
d_model = 16
max_seq_len = 5
num_classes = 3
d_internal = 8
d_ff = 32
```

```
num_layers = 1
num_heads = 1


model = Transformer(
   vocab_size=vocab_size,
   max_seq_len=max_seq_len,
   d_model=d_model,
   d_internal=d_internal,
   num_classes=num_classes,
   d_ff=d_ff,
   num_layers=num_layers,
   num_heads=num_heads
)

input_indices = torch.tensor([[1, 1, 2, 2, 3]])  # Shape: [1, 5]


unbatched_indices = torch.tensor([1, 1, 2, 2, 3])
embedding_output = model.word_embedding(unbatched_indices) +
model.positional_encoding[:5]
print("Embedding (word + positional) for [1, 1, 2, 2, 3]:")
print(embedding_output)
```

```
Embedding (word + positional) for [1, 1, 2, 2, 3]:
tensor([[-0.2324,  0.8818, -0.7065,  1.8570, -0.6066,  1.5692,  0.0560,
1.9572,
         -0.0072, -0.8654, -1.2593,  1.4450,  1.5879, -0.4607,  1.9040,
0.1749],
        [ 0.6091,  0.4221, -0.3955,  1.8075, -0.5067,  1.5642,  0.0877,
1.9567,
          0.0028, -0.8655, -1.2561,  1.4450,  1.5889, -0.4607,  1.9043,
0.1749],
        [ 0.2923,  0.4208,  0.3784, -0.2312, -0.3332,  2.1780, -1.6367,
0.5988,
         -0.7891, -1.6494,  0.6515,  2.6632, -0.6078,  2.5516, -0.3205,
-0.5025],
        [-0.4758, -0.1530,  0.5999, -0.4550, -0.2364,  2.1533, -1.6052,
0.5963,
         -0.7791, -1.6496,  0.6547,  2.6631, -0.6068,  2.5516, -0.3202,
-0.5025],
        [-1.0145,  0.0799,  0.3063,  0.7578, -0.1166,  1.9184, -0.0338,
1.4021,
         -0.1250,  0.1371,  1.0538,  0.8103,  2.0546,  1.6958,  0.1865,
1.7890]],
       grad_fn=<AddBackward0>)
```

Part 3:

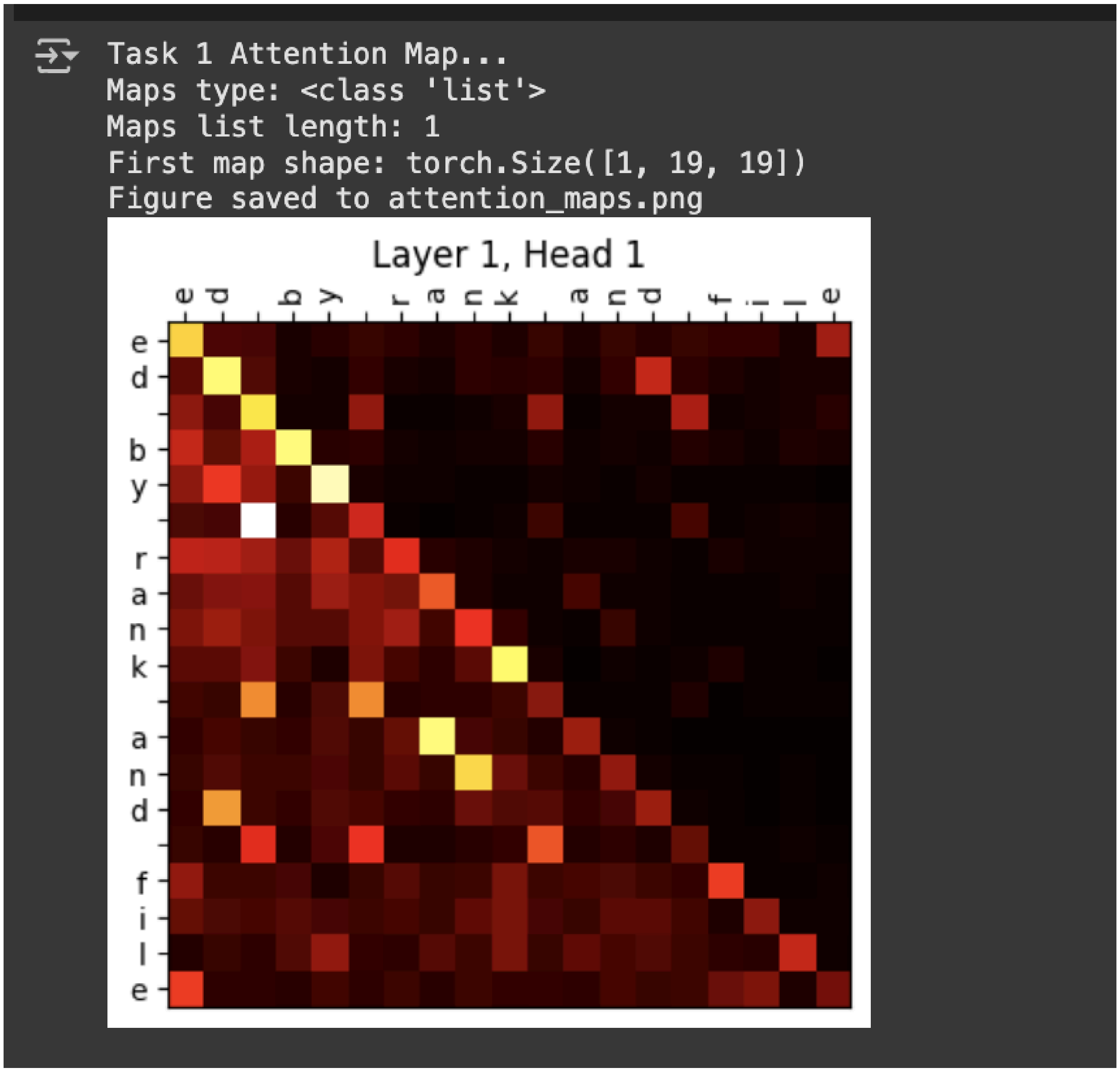
```
test_string = "ed by rank and file"
test_id_tensor = indexer(test_string)
test_task1_tensor = task1(test_string)
test_task2_tensor = task2(test_string)

print("String:", test_string)
print("ID tensor:", test_id_tensor)
print("Task1 tensor:", test_task1_tensor)
print("Task2 tensor:", test_task2_tensor)
```

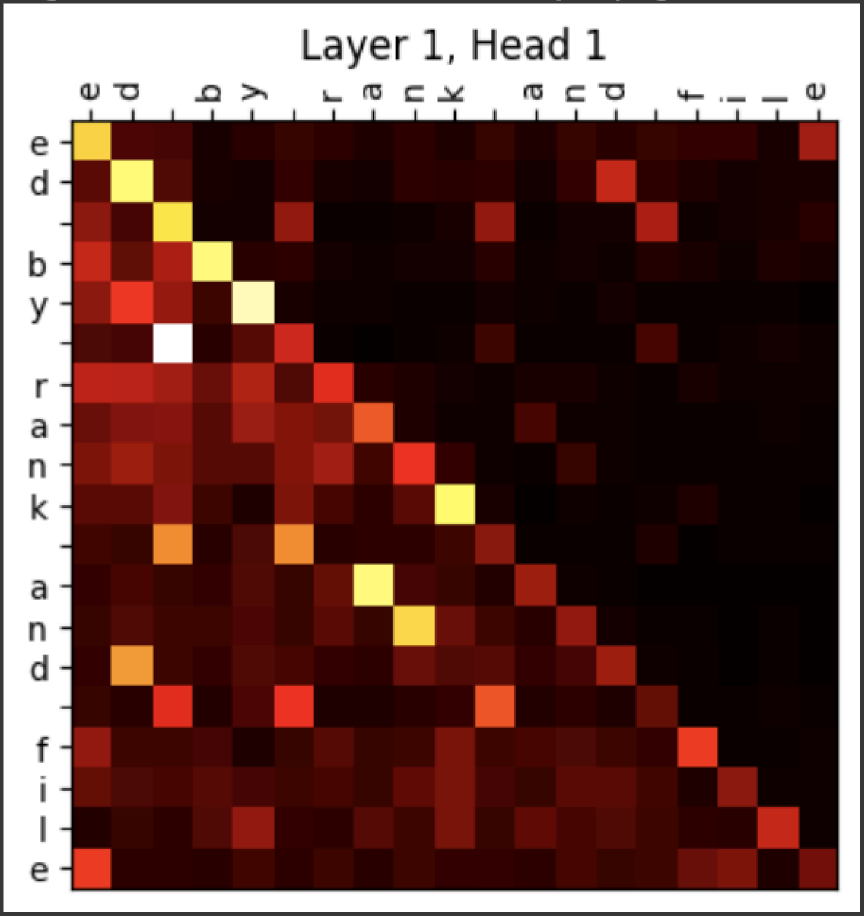
```
String: ed by rank and file
ID tensor: tensor([ 5,  4,  0,  2, 25,  0, 18,  1, 14, 11,  0,  1, 14,  4,
0,  6,  9, 12,
         5])
Task1 tensor: tensor([0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 1, 1, 1, 2, 0, 0,
0, 1])
Task2 tensor: tensor([1, 1, 2, 0, 0, 2, 0, 1, 1, 0, 2, 1, 1, 1, 2, 0, 0,
0, 1])
```

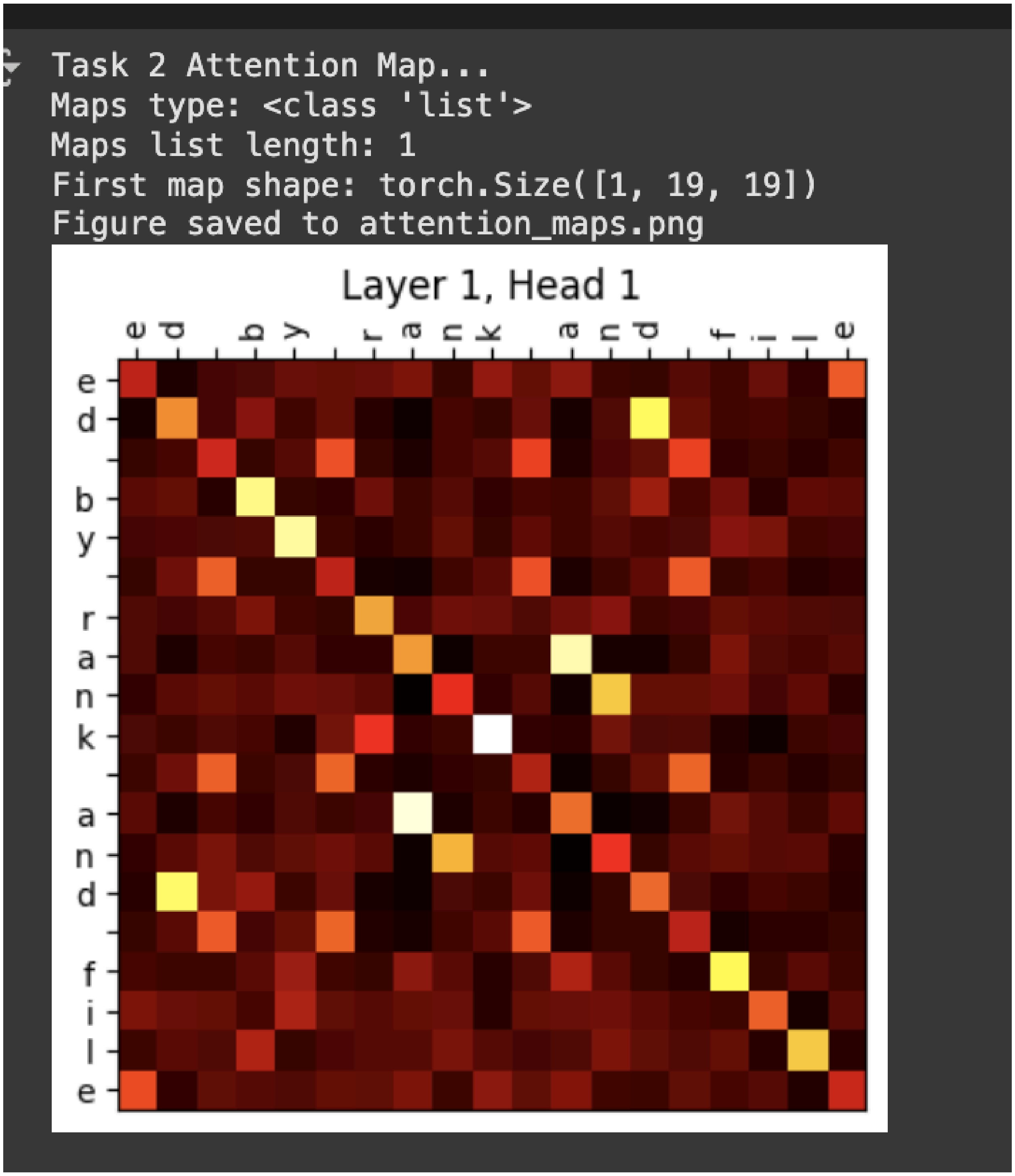
Part 4:

Finding the right learning rate for this Transformer model was tricky. If set too high, the model's training became unstable and diverged. If too low, training progressed too slowly and got stuck in poor solutions. I struggled with choosing an appropriate loss function too - the classes in the letter counting tasks aren't evenly distributed (most characters appear 0 times), which caused the model to predict the majority class too often. Other challenges included deciding on the batch size (larger batches were more stable but slower, smaller batches were faster but noisier), determining when to stop training to avoid overfitting, and figuring out the right model size. The small dataset size made these decisions harder since there wasn't enough validation data to reliably test different configurations.

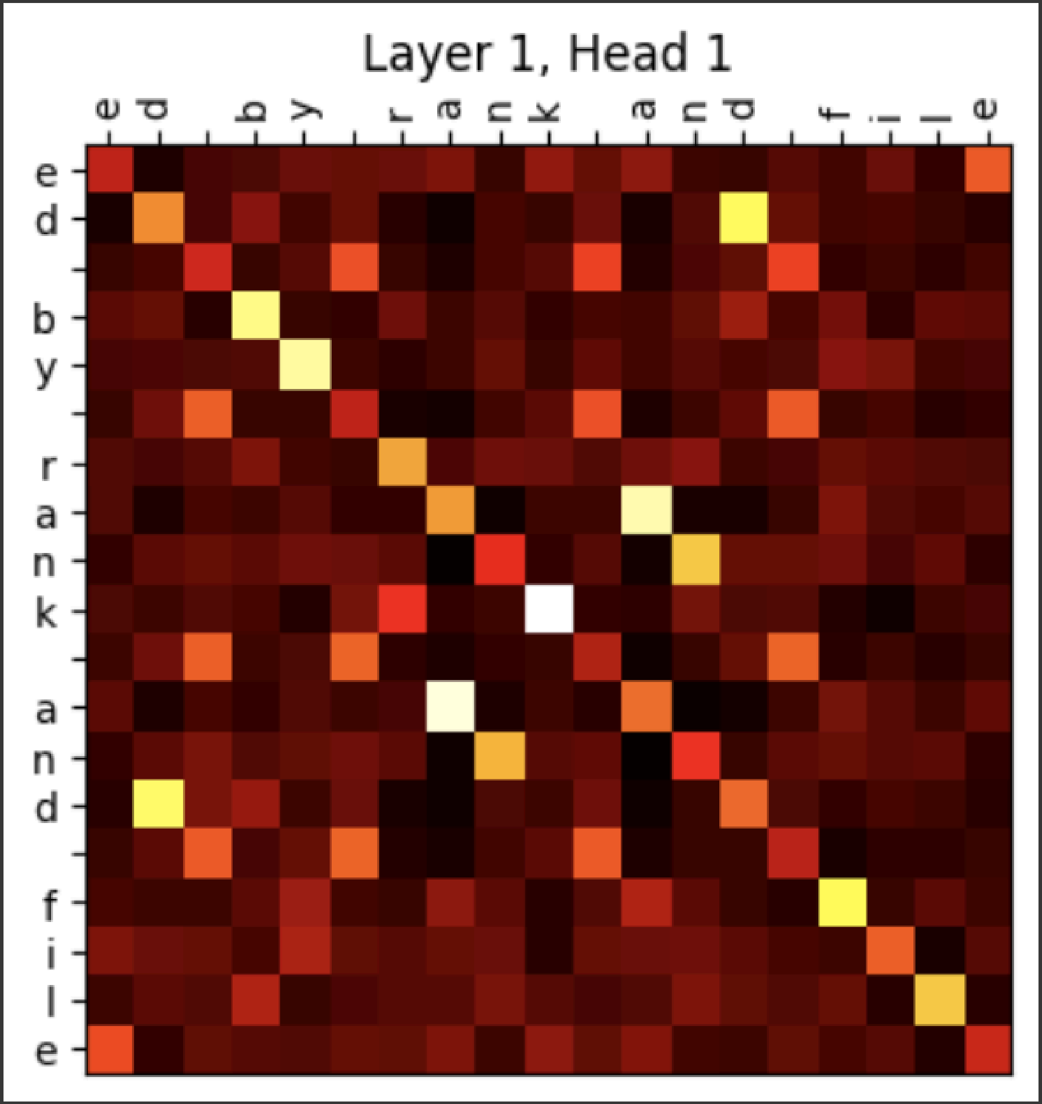


```
Task 1 Attention Map...
Maps type: <class 'list'>
Maps list length: 1
First map shape: torch.Size([1, 19, 19])
Figure saved to attention_maps.png
```

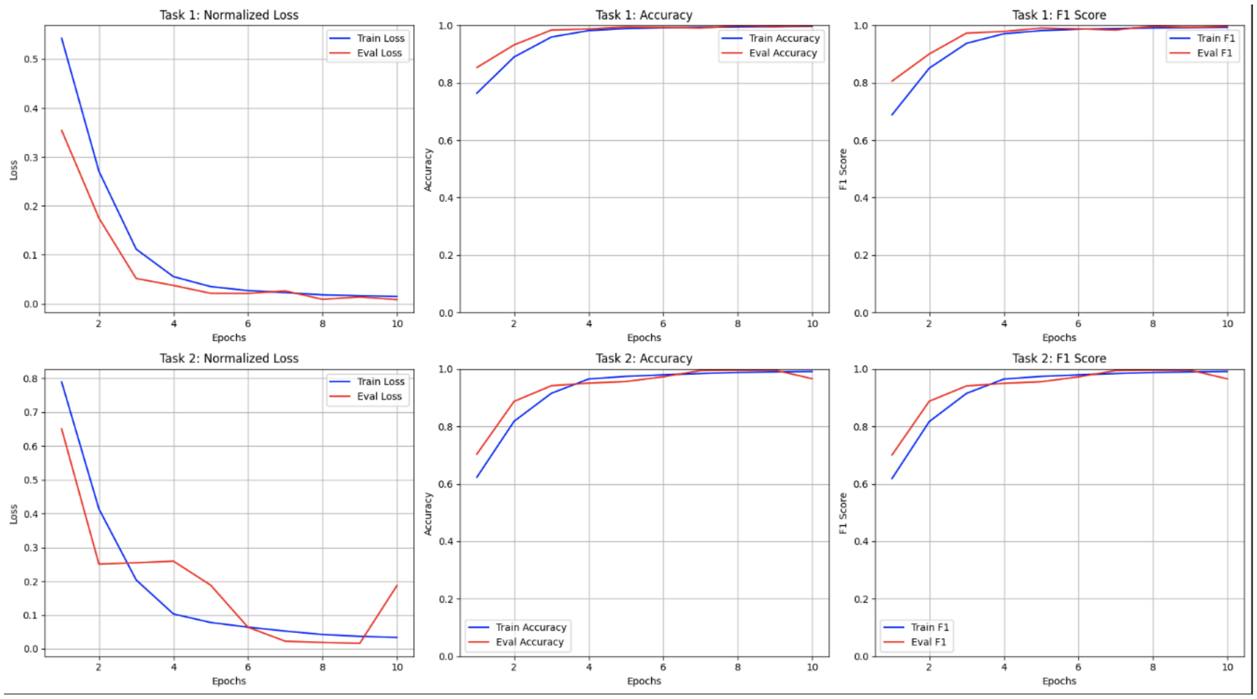




```
Task 2 Attention Map...
Maps type: <class 'list'>
Maps list length: 1
First map shape: torch.Size([1, 19, 19])
Figure saved to attention_maps.png
```



```
Training Task 1 model...
Epoch 1: Train Loss = 0.5426, Acc = 0.7633, F1 = 0.6882
Epoch 1: Eval Loss = 0.3545, Acc = 0.8527, F1 = 0.8053
Epoch 2: Train Loss = 0.2708, Acc = 0.8894, F1 = 0.8505
Epoch 2: Eval Loss = 0.1752, Acc = 0.9315, F1 = 0.8999
Epoch 3: Train Loss = 0.1117, Acc = 0.9588, F1 = 0.9369
Epoch 3: Eval Loss = 0.0517, Acc = 0.9831, F1 = 0.9725
Epoch 4: Train Loss = 0.0555, Acc = 0.9813, F1 = 0.9702
Epoch 4: Eval Loss = 0.0375, Acc = 0.9870, F1 = 0.9783
Epoch 5: Train Loss = 0.0351, Acc = 0.9882, F1 = 0.9813
Epoch 5: Eval Loss = 0.0214, Acc = 0.9938, F1 = 0.9900
Epoch 6: Train Loss = 0.0270, Acc = 0.9913, F1 = 0.9860
Epoch 6: Eval Loss = 0.0212, Acc = 0.9926, F1 = 0.9872
Epoch 7: Train Loss = 0.0228, Acc = 0.9929, F1 = 0.9884
Epoch 7: Eval Loss = 0.0263, Acc = 0.9902, F1 = 0.9835
Epoch 8: Train Loss = 0.0182, Acc = 0.9943, F1 = 0.9905
Epoch 8: Eval Loss = 0.0091, Acc = 0.9976, F1 = 0.9964
Epoch 9: Train Loss = 0.0165, Acc = 0.9950, F1 = 0.9918
Epoch 9: Eval Loss = 0.0140, Acc = 0.9953, F1 = 0.9926
Epoch 10: Train Loss = 0.0150, Acc = 0.9957, F1 = 0.9928
Epoch 10: Eval Loss = 0.0086, Acc = 0.9973, F1 = 0.9957
Training Task 2 model...
Epoch 1: Train Loss = 0.7895, Acc = 0.6230, F1 = 0.6186
Epoch 1: Eval Loss = 0.6505, Acc = 0.7037, F1 = 0.7006
Epoch 2: Train Loss = 0.4142, Acc = 0.8179, F1 = 0.8168
Epoch 2: Eval Loss = 0.2510, Acc = 0.8870, F1 = 0.8876
Epoch 3: Train Loss = 0.2039, Acc = 0.9152, F1 = 0.9149
Epoch 3: Eval Loss = 0.2548, Acc = 0.9416, F1 = 0.9408
Epoch 4: Train Loss = 0.1033, Acc = 0.9648, F1 = 0.9646
Epoch 4: Eval Loss = 0.2595, Acc = 0.9506, F1 = 0.9499
Epoch 5: Train Loss = 0.0781, Acc = 0.9742, F1 = 0.9740
Epoch 5: Eval Loss = 0.1884, Acc = 0.9561, F1 = 0.9554
Epoch 6: Train Loss = 0.0643, Acc = 0.9794, F1 = 0.9793
Epoch 6: Eval Loss = 0.0640, Acc = 0.9725, F1 = 0.9722
Epoch 7: Train Loss = 0.0526, Acc = 0.9840, F1 = 0.9839
Epoch 7: Eval Loss = 0.0227, Acc = 0.9943, F1 = 0.9943
Epoch 8: Train Loss = 0.0426, Acc = 0.9878, F1 = 0.9877
Epoch 8: Eval Loss = 0.0190, Acc = 0.9954, F1 = 0.9953
Epoch 9: Train Loss = 0.0371, Acc = 0.9897, F1 = 0.9897
Epoch 9: Eval Loss = 0.0165, Acc = 0.9958, F1 = 0.9957
Epoch 10: Train Loss = 0.0339, Acc = 0.9912, F1 = 0.9911
Epoch 10: Eval Loss = 0.1877, Acc = 0.9659, F1 = 0.9655
```



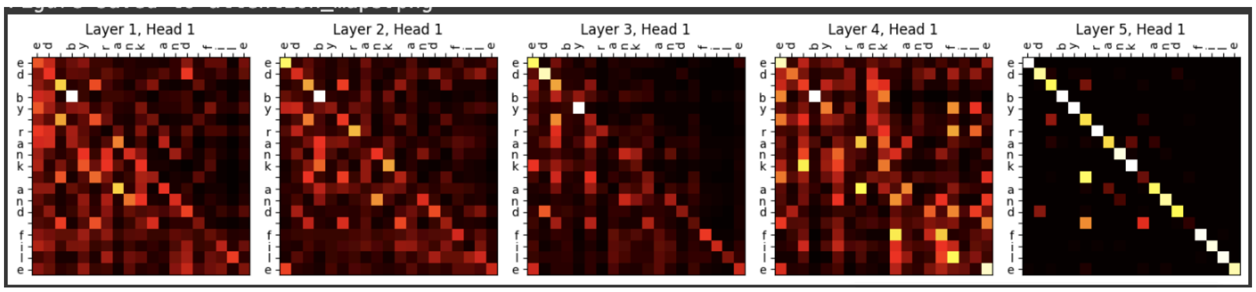
Part 5:

5.1)

The attention maps show how each token attends to other tokens in the sequence "ed by rank and file". The brighter spots (yellow/white) show stronger attention weights. Task 1 - Left Image: 'a': Shows clear bright spots connecting the two 'a's in "rank" and "and". 'n': The 'n' tokens strongly attend to previous occurrences of 'n'. ' ' (space): Has some attention patterns but less distinctive. 'e': Shows bright spots at the first occurrence (top-left) and when looking at other 'e's in the sequence (the final 'e'). This pattern helps the model count how many 'e's appeared before each position. 'd': Shows bright attention to the first 'd' when processing the second 'd'.
Task 2 (Before/After task) - Right Image:
Overall more diffuse attention patterns spread across the entire sequence.
'a': The two 'a's show mutual attention, indicating the model is tracking all occurrences. 'n': Both 'n's show attention to each other. 'd': Shows connections between the two 'd's but more diffuse than in Task 1. 'e': Attends to all 'e' occurrences in the sequence with moderate intensity.
Similarities:
Both models show high attention between identical characters and both create distinctive patterns for repeated characters.
Differences:
Task 1's attention is more focused and precise, with brighter spots for previous occurrences, which aligns with its goal of counting previous occurrences only. Task 2's attention is more distributed, as it needs to count all occurrences throughout the sequence, not just previous

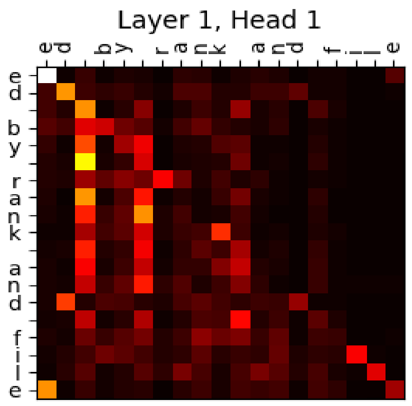
ones. Task 1 shows clear sequential dependency (looking backward), while Task 2 shows more global awareness (looking both ways).
These patterns show their different objectives: Task 1 needs to look backward to count previous occurrences, while Task 2 needs a global view to count total occurrences minus one.

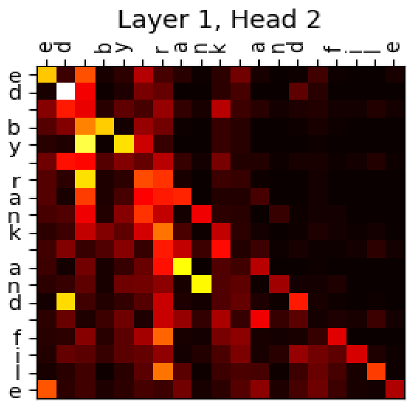
5.2)

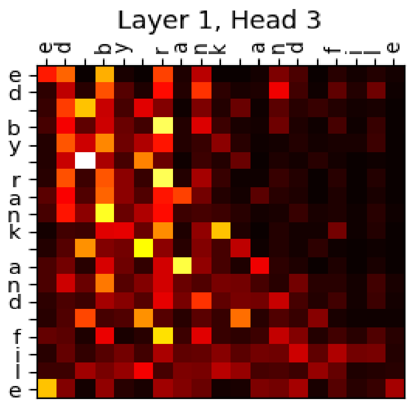


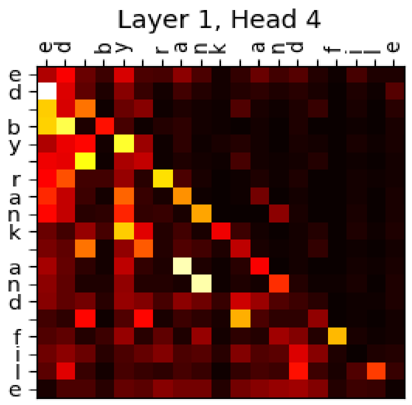
Layer 1: Notices when the same character appears close together. Layer 2: Gathers information about repeated characters and improves on what Layer 1 found. Layer 3: Combines information from earlier layers to create more advanced patterns. Layer 4: Starts changing the attention patterns to help make the final decision. Layer 5: Fine-tunes the focus on repeated characters that matter most for the task, leading to the final answer.

5.3)


Layer 1, Head 1
e d b y r a n k a n d f i l e
e d b y r a n k a n d f i l e


Layer 1, Head 2
e d b y r a n k a n d f i l e
e d b y r a n k a n d f i l e


Layer 1, Head 3
e d b y r a n k a n d f i l e
e d b y r a n k a n d f i l e


Layer 1, Head 4
e d b y r a n k a n d f i l e
e d b y r a n k a n d f i l e

The first head notices when a character appears for the very first time.
The second head pays attention to when characters show up again, especially the second time they appear.
The third head looks at the big picture or specific patterns of characters.
The fourth head combines information about where characters are positioned and how often they appear.
By splitting the work this way, the system can handle the complex task of counting characters more effectively. Each head specializes in one aspect of the job - like spotting new characters or tracking repeats. Then all this information gets combined to make the final decision.