Intermediate Machine Learning: Assignment 5

Deadline

Assignment 5 is due Wednesday, December 4 by 11:59pm. Late work will not be accepted as per the course policies (see the Syllabus and Course policies on Canvas).

Directly sharing answers is not okay, but discussing problems with the course staff or with other students is encouraged.

You should start early so that you have time to get help if you're stuck. The drop-in office hours schedule can be found on Canvas. You can also post questions or start discussions on Ed Discussion. The assignment may look long at first glance, but the problems are broken up into steps that should help you to make steady progress.

Submission

Submit your assignment as a pdf file on Gradescope, and as a notebook (.ipynb) on Canvas. You can access Gradescope through Canvas on the left-side of the class home page. The problems in each homework assignment are numbered. Note: When submitting on Gradescope, please select the correct pages of your pdf that correspond to each problem. This will allow graders to more easily find your complete solution to each problem.

To produce the .pdf, please do the following in order to preserve the cell structure of the notebook:

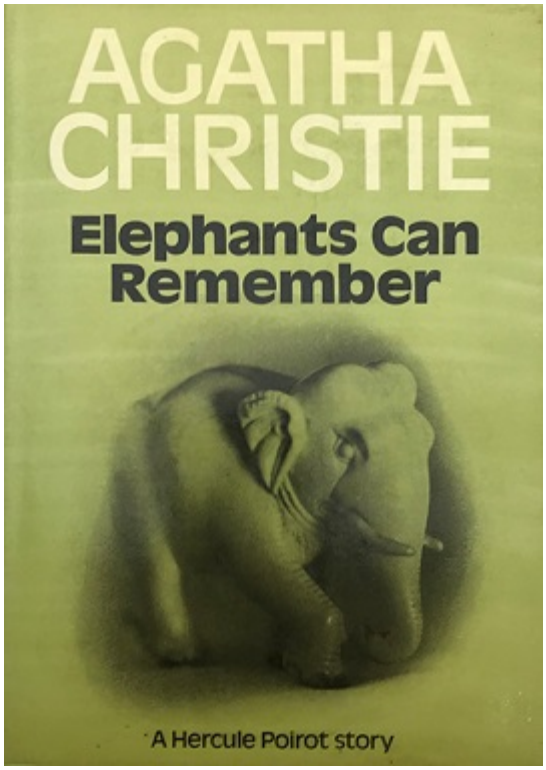
Go to "File" at the top-left of your Jupyter Notebook Under "Download as", select "HTML (.html)" After the .html has downloaded, open it and then select "File" and "Print" (note you will not actually be printing) From the print window, select the option to save as a .pdf

Topics

- RNNs and GRUs
- Transformers

This assignment will also help to solidify your Python skills.

Problem 1: Elephants Can Remember (25 points)



In this problem, we will work with "vanilla" Recurrent Neural Networks (RNNs) and Recurrent Neural Networks with Gated Recurrent Units (GRUs). The models in this part of the assignment will be character-based models, trained on an extract of the book Elephants Can Remember by Agatha Christie. To reduce the size of our vocabulary, the text is pre-processed by converting the letters to lower case and removing numbers. The code below shows some information about our training and test set. All the necessary files for this problem are available through Canvas, under the file name "problem1_data".

In [10]:
```
# note: used VSCode auto-complete suggestions for all code in this assignment


import numpy as np
from tqdm import tqdm
import tensorflow as tf
import random

from keras.models import Sequential, model_from_json
from keras.layers import Dense, Activation
from keras.layers import GRU, SimpleRNN
```

In [3]:
```
with open('problem1_data/Agatha_Christie_train.txt', 'r') as file:
    train_text = file.read()

with open('problem1_data/Agatha_Christie_test.txt', 'r') as file:
    test_text = file.read()

vocabulary = sorted(list(set(train_text + test_text)))
vocab_size = len(vocabulary)

# Dictionaries to go from a character to index and vice versa
```

```
char_to_indices = dict((c, i) for i, c in enumerate(vocabulary))
indices_to_char = dict((i, c) for i, c in enumerate(vocabulary))
```

In [4]:
```
# The first 500 characters of our training set
train_text[0:500]
```

Out[4]:
```
'mrs. oliver looked at herself in the glass. she gave a brief, sideways look towar
ds the clock on the mantelpiece, which she had some idea was twenty minutes slow.
then she resumed her study of her coiffure. the trouble with mrs. oliver was--and
she admitted it freely--that her styles of hairdressing were always being changed.
she had tried almost everything in turn. a severe pompadour at one time, then a wi
nd-swept style where you brushed back your locks to display an intellectual brow,
at least'
```

In [5]:
```
print("The vocabulary contains", vocab_size, "characters")
print("The training set contains", len(train_text) ,"characters")
print("The test set contains", len(test_text) ,"characters")
```

```
The vocabulary contains 44 characters
The training set contains 262174 characters
The test set contains 7209 characters
```

Problem 1.1: The Diversity of Language Models

Before jumping into coding, let's start with comparing the language models we will be using in this assigment.

1. Describe the differences between a Vanilla RNN and a GRU network. In your explanation, make sure you mention the issues with vanilla RNNs and how GRUs try to solve them.

A recurrent neural network maintains a state vector which is updated based on the previous state and the current input, and from which the output is generated. While in theory the RNN state can encode past information, in practice the repeated multiplication by the state transition matrix numerically dilutes all far past information so that it can't significantly inform output generation. The GRU solves this problem by introducing a mechanism (the gate) to preserve and carry forward important past information by bypassing multiplication by the state transition matrix for automatically selected elements.

2. Describe at least two advantages of a character based language model over a word based language model.

1. A character based language model can generate any arbitrary text (IE "BAHAHA" for an evil laugh) rather than be confined to text comprised of full english dictionary words (IE "ha" for an evil laugh).
2. The number of parameters has an O(v) term, where v is the size of the vocabulary, due the encoding and decoding matrices that embedd and un-embedded the token input. A vocabulary of characters is much smaller than a vocabulary of words (there are 10s of

characters vs. tens of thousands of words) so reduces model size, making training easier

Problem 1.2: Generating Text with the Vanilla RNN

The code below loads in a pretrained vanilla RNN model with two layers. The model is set up exactly like in the lecture slides (with tanh activation layers in the recurrent layers) with the addition of biases (intercepts) in every layer (i.e. the recurrent layer and the dense layer). The training process consisted of 30 epochs.

In [7]:
```
# load json and create model
json_file = open('problem1_data/RNN_model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()

RNN_model = model_from_json(loaded_model_json)
RNN_model.load_weights("problem1_data/RNN_model.h5")
```

In [8]:
```
# load in the weights and show summary
weights_RNN = RNN_model.get_weights()
RNN_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Vanilla_RNN_1 (SimpleRNN)   (None, 100, 128)          22144

 Vanilla_RNN_2 (SimpleRNN)   (None, 64)                12352

 Dense_layer (Dense)         (None, 44)                2860

 Softmax_layer (Activation)  (None, 44)                0

=================================================================
Total params: 37,356
Trainable params: 37,356
Non-trainable params: 0
_________________________________________________________________
```

Finish the following function that uses a vanilla RNN architecture to generate text, given the weights of the RNN model, a text prompt, and the number of characters to return. The function should be completed by **only using numpy functions**. Use your knowledge of how every weight plays its role in the RNN architecture. Do not worry about the weight extraction part, this is already provided for you. The weight matrix W_{xh1}, for example, denotes the weight matrix to go from the input x to the first hidden state layer h1. The hidden states h_1 and h_2 are initialized to a vector of zeros.

The embedding of each character has to be done by a one-hot encoding, where you will need the dictionaries defined in the introduction to go from a character to an index position.

In [20]:

```
def sample_text_RNN(weights, prompt, N):
    '''
    Uses a pretrained RNN to generate text, starting from a prompt,
    only using the weights and numpy commands
            Parameters:
                    weights (list): Weights of the pretrained RNN model
                    prompt (string): Start of generated sentence
                    N (int): Length of output sentence (including prompt)
            Returns:
                    output_sentence (string): Text generated by RNN
    '''
    # Extracting weights and biases
    # Dimensions of matrices are same format as lecture slides

    # First Recurrent Layer
    W_xh1 = weights[0].T
    W_h1h1 = weights[1].T
    b_h1 = np.expand_dims(weights[2], axis=1)

    # Second Recurrent Layer
    W_h1h2 = weights[3].T
    W_h2h2 = weights[4].T
    b_h2 = np.expand_dims(weights[5], axis=1)

    # Linear (dense) layer
    W_h2y = weights[6].T
    b_y = np.expand_dims(weights[7], axis=1)

    # Initiate the hidden states
    h1 = np.zeros((W_h1h1.shape[0], 1))
    h2 = np.zeros((W_h2h2.shape[0], 1))

    # -----------------------------------------------

    # Your code starts here
    output_sentence = ""
    output_sentence += prompt

    # Encode prompt into hidden states by iterating over each prompt character
    for char in prompt:
        # One-hot encode the character
        x = np.zeros((vocab_size, 1))
        x[char_to_indices[char]] = 1

        h1 = np.tanh(np.dot(W_xh1, x) + np.dot(W_h1h1, h1) + b_h1)
        h2 = np.tanh(np.dot(W_h1h2, h1) + np.dot(W_h2h2, h2) + b_h2)

    # Generate the next N characters
    for i in range(N):
        # Calculate the output
        y = np.dot(W_h2y, h2) + b_y
        p = np.exp(y) / np.sum(np.exp(y))

        # Sample the next character
        x_index = np.random.choice(range(vocab_size), p=p.flatten())
        char = indices_to_char[x_index]
```

```
        output_sentence += char

        # Make the sampled character the new input
        x = np.zeros((vocab_size, 1))
        x[x_index] = 1

        # Update the hidden states
        h1 = np.tanh(np.dot(W_xh1, x) + np.dot(W_h1h1, h1) + b_h1)
        h2 = np.tanh(np.dot(W_h1h2, h1) + np.dot(W_h2h2, h2) + b_h2)

    return output_sentence
```

Test out your function by running the following code cell. Use it as a sanity check that your code is working. The generated text should not be perfect English, but at least you should be able to recognize some words.

In [29]:
```
print(sample_text_RNN(weights_RNN,
                      'mrs. oliver looked at herself in the glass. she gave a brief
                      1000))
```

```
mrs. oliver looked at herself in the glass. she gave a brief, sideways looke things,
" said mrs. oliver, were time of mrs. cirland her. it went happened, but she wasticj
to must sayes. then thesime. and all the dout sigrt. no swork on look afor the ges o
nd of sady father them. yes. i subtechod anywhere elved not to sennice.. his sopulio
uld her lask did, it's ale i was what be suse ives." "yes, yes. there was anlig? the
y boad scerdiaf celtarbed in and they restage. had and poirot. and elephants?" "yes,
i saying at you dot work or a metterven. "well, plopined when qhimny! inverseind apy
eces or that." "yes, yy, i'th a lot leakedland any-like a motairy the looked at erse
it lines happenes got they widing," said poirot. "what don't was mrs. oambouthere an
d sayt of alwon very rivens if the lifely refor gow firsnce you cientasen somebofien
, that is all about it." saige--haven'w rese now. she's he didn't," erising to see y
out she was fis some heo had frithes deary and does it?" "are alowe be somantersend.
"i said been i can't know, chirict to tust cainiraly as your nea
```

Problem 1.3: Generating Text with the GRU

The code below loads in a pretrained GRU model. The model is set up exactly like in the lecture slides (with sigmoid activation layers for the gates and tanh activation layers in the recurrent layer). The model is trained for only 10 epochs.

In [22]:
```
# load json and create model
json_file = open('problem1_data/GRU_model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()

GRU_model = model_from_json(loaded_model_json)
GRU_model.load_weights("problem1_data/GRU_model.h5")
```

In [23]:
```
# load in the weights and show summary
weights_GRU = GRU_model.get_weights()
GRU_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 512)               857088

 dense (Dense)               (None, 44)                22572

 activation (Activation)     (None, 44)                0

=================================================================
Total params: 879,660
Trainable params: 879,660
Non-trainable params: 0
_________________________________________________________________
```

Finish the following function that uses a GRU architecture to generate text, given the weights of the GRU model, a text prompt, and the number of characters to return. The function should be completed by **only using numpy functions**. Use your knowledge of how every weight plays its role in the GRU architecture. Do not worry about the weight extraction part, this is already provided for you. The hidden state h is initialized to a vector of zeros.

The embedding of each character has to be done by a one-hot encoding, where you will need the dictionaries defined in the introduction to go from a character to an index position.

Note: a slightly different version of the GRU is used, where the candidate state c_t is calculated as:

$$c_t = \tanh\left(W_{hx}x_t \ + \ \Gamma_t^r \odot (W_{hh}h_{t-1}) \ + \ b_h\right)$$

In [26]:
```python
# Helper function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sample_text_GRU(weights, prompt, N):
    '''
    Uses a pretrained GRU to generate text, starting from a prompt,
    only using the weights and numpy commands
            Parameters:
                    weights (list): Weights of the pretrained GRU model
                    prompt (string): Start of generated sentence
                    N (int): Total length of output sentence
            Returns:
                    output_sentence (string): Text generated by GRU
    '''
    # Extracting weights and biases
    # Dimensions of matrices are same format as lecture slides

    # GRU Layer
    W_ux, W_rx, W_hx = np.split(weights[0].T, 3, axis = 0)
    W_uh, W_rh, W_hh = np.split(weights[1].T, 3, axis = 0)

    bias = np.sum(weights[2], axis=0)
```

```
    b_u, b_r, b_h = np.split(np.expand_dims(bias, axis=1), 3)

    # Linear (dense) layer
    W_y = weights[3].T
    b_y = np.expand_dims(weights[4], axis=1)

    # Initiate hidden state
    h = np.zeros((W_hh.shape[0], 1))

    # ------------------------------------------------

    # Your code starts here
    output_sentence = ""
    output_sentence += prompt

    # Encode prompt into hidden states by iterating over each prompt character
    for char in prompt:
        # One-hot encode the character
        x = np.zeros((vocab_size, 1))
        x[char_to_indices[char]] = 1

        u = sigmoid(np.dot(W_ux, x) + np.dot(W_uh, h) + b_u)
        r = sigmoid(np.dot(W_rx, x) + np.dot(W_rh, h) + b_r)
        c = np.tanh(np.dot(W_hx, x) + r * np.dot(W_hh, h) + b_h)
        h = (1 - u) * c + u * h

    # Generate the next N characters
    for i in range(N):
        # Calculate the output
        y = np.dot(W_y, h) + b_y
        p = np.exp(y) / np.sum(np.exp(y))

        # Sample the next character
        x_index = np.random.choice(range(vocab_size), p=p.flatten())
        char = indices_to_char[x_index]
        output_sentence += char

        # Make the sampled character the new input
        x = np.zeros((vocab_size, 1))
        x[x_index] = 1

        # Update the hidden states
        u = sigmoid(np.dot(W_ux, x) + np.dot(W_uh, h) + b_u)
        r = sigmoid(np.dot(W_rx, x) + np.dot(W_rh, h) + b_r)
        c = np.tanh(np.dot(W_hx, x) + r * np.dot(W_hh, h) + b_h)
        h = (1 - u) * c + u * h

    return output_sentence
```

Test out your function by running the following code cell. Use it as a sanity check that your code is working. The generated text should not be perfect English, but at least you should be able to recognize some words.

In [28]:
```
print(sample_text_GRU(weights_GRU,
                      'mrs. oliver looked at herself in the glass. she gave a brief
```

```
                    1000))
```

```
mrs. oliver looked at herself in the glass. she gave a brief, sideways looked at the
ravenscrofts. or some of treach time becture some there or a wmere strently eniean o
r stian insion to her howe wo know, the it. that it come the it a riall sor and came
to anyormation taybrestanded and stwents of thes treat onelle some there is the dous
t it, i really about i  eee tice they cent a sme and a child her and way there all a
will have oole a grave if so mind a more all a gatire. monat she had the hooding, wh
en she had a realy able to about her, betinite marrare teanctupe as the ator. s me t
aike to wear a certen more than starien a waing the adopted to hersilt the two. and
i kind her heact. it was still ofter that she was seewal tade. i want till them beat
ewely in a lot to leave it were conkind a doed her lives in. a nowes ouch of the str
ave lock where." "ableatire simeling a now about." "yes, we that woman rather it all
the reasing oftan thought it it case a told you at that she was a nice, and you reme
mber that a tried sometimes it one of the recert an resol! a d
```

Problem 1.4: Can Elephants Remember Better?

Perplexity is a measure to quantify how "good" a language model M is, based on a test (or validation) set. The perplexity on a sequence s of characters a_i of size N is defined as:

$$\text{Perplexity}(M) = M(s)^{(-1/N)} = \{p(a_1, \ldots, a_N)\}^{(-1/N)} = \{p(a_1)\, p(a_2|a_1)\, \ldots\, p(a_N|a_1, \ldots$$

> The intuition behind this metric is that, if a model assigns a high probability to a test set, it is not surprised to see it (not perplexed by it), which means the model M has a good understanding of how the language works. Hence, a good model has, in theory, a lower perplexity. The exponent $(-1/N)$ in the formula is just a normalizing strategy (geometric average), because adding more characters to a test set would otherwise introduce more uncertainty (i.e. larger test sets would have lower probability). So by introducing the geometric average, we have a metric that is independent of the size of the test set.

When calculating the perplexity, it is important to know that taking the product of a bunch of probabilities will most likely lead to a zero value by the computer. To prevent this, make use of a log-transformation:

$$\text{Log-Perplexity}(M) = -\frac{1}{N} log\, \{p(a_1, \ldots, a_N)\} = -\frac{1}{N} \{log\, p(a_1) + \ log\, p(a_2|a_1) + \ \ldots$$

Don't forget to go back to the normal perplexity after this transformation.

1. Before calculating the perplexity of a test sequence, start with comparing the outputs of 2.2 and 2.3. Do you see any differences in the generated text of the Vanilla RNN model and the GRU model? Rerun your functions a couple of times (because of stochasticity) and use different prompts. Briefly discuss why you would expect (or not expect) certain differences.

Generated text from each (The ... is my own addition, so I can highlight both the start and end of the generated text):

1. Sample from RNN: mrs. oliver looked at herself in the glass. she gave a brief, sideways looke things," said mrs. oliver, were time of mrs. cirland her. it went happened, but she wasticj to must sayes. then thesime. and all the dout sigrt. ... rese now. she's he didn't," erising to see yout she was fis some heo had frithes deary and does it?" "are alowe be somantersend. "i said been i can't know, chirict to tust cainiraly as your nea

2. Sample from GRU: mrs. oliver looked at herself in the glass. she gave a brief, sideways looked at the ravenscrofts. or some of treach time becture some there or a wmere strently eniean or stian insion to her howe wo know, the it. ... "yes, we that woman rather it all the reasing oftan thought it it case a told you at that she was a nice, and you remember that a tried sometimes it one of the recert an resol! a d

Both generated samples vaguely suggest real english sentences, but are incoherant. Many words are not real, but even when real words do follow each other, the phrases don't make sense together. The RNN generated text seems to struggle even correctly spelling words, especially near the end of the generated text. The GRU seems to do better, which isn't suprising to me because making a complete valid word requires "remembering" how the word started and the GRU is better at memory. Even still, it doesn't form complete thoughts which might be in part a result of it only being trained for 10 epochs.

2. Calculate the perplexity of each language model by using test_text, an unseen extract of the book. Choose the prompt as the first m letters of the test set, where m is a parameter that you can choose yourself. You should be able to reuse the majority of your previous code in this calculation. Discuss your results at the end.

In [34]:
```
# Calculated perplexity of a pretrained RNN with given weights on first m character
def RNN_perplexity(weights, test_text, m):
    # Extracting weights and biases
    # Dimensions of matrices are same format as lecture slides

    # First Recurrent Layer
    W_xh1 = weights[0].T
    W_h1h1 = weights[1].T
    b_h1 = np.expand_dims(weights[2], axis=1)

    # Second Recurrent Layer
    W_h1h2 = weights[3].T
    W_h2h2 = weights[4].T
    b_h2 = np.expand_dims(weights[5], axis=1)

    # Linear (dense) layer
    W_h2y = weights[6].T
    b_y = np.expand_dims(weights[7], axis=1)

    # Initiate the hidden states
    h1 = np.zeros((W_h1h1.shape[0], 1))
    h2 = np.zeros((W_h2h2.shape[0], 1))
```

```
    # ------------------------------------------------

    log_probs = 0

    for i in range(m):
        # One-hot encode the character
        x = np.zeros((vocab_size, 1))
        x[char_to_indices[test_text[i]]] = 1

        # update hidden states
        h1 = np.tanh(np.dot(W_xh1, x) + np.dot(W_h1h1, h1) + b_h1)
        h2 = np.tanh(np.dot(W_h1h2, h1) + np.dot(W_h2h2, h2) + b_h2)

        # Calculate the output
        y = np.dot(W_h2y, h2) + b_y
        p = np.exp(y) / np.sum(np.exp(y))

        # Calculate the log probability of the true character
        log_prob = np.log(p[char_to_indices[test_text[i+1]]])
        log_probs += log_prob

    # Calculate perplexity
    log_perplexity = -log_probs / m
    perplexity = np.exp(log_perplexity)

    return perplexity

# Calculated perplexity of a pretrained GRU with given weights on first m characters
def GRU_perplexity(weights, test_text, m):
    # Extracting weights and biases
    # Dimensions of matrices are same format as lecture slides

    # GRU Layer
    W_ux, W_rx, W_hx = np.split(weights[0].T, 3, axis = 0)
    W_uh, W_rh, W_hh = np.split(weights[1].T, 3, axis = 0)

    bias = np.sum(weights[2], axis=0)
    b_u, b_r, b_h = np.split(np.expand_dims(bias, axis=1), 3)

    # Linear (dense) layer
    W_y = weights[3].T
    b_y = np.expand_dims(weights[4], axis=1)

    # Initiate hidden state
    h = np.zeros((W_hh.shape[0], 1))

    # ------------------------------------------------

    log_probs = 0

    for i in range(m):
        # One-hot encode the character
        x = np.zeros((vocab_size, 1))
        x[char_to_indices[test_text[i]]] = 1

        # update hidden states
```

```
        u = sigmoid(np.dot(W_ux, x) + np.dot(W_uh, h) + b_u)
        r = sigmoid(np.dot(W_rx, x) + np.dot(W_rh, h) + b_r)
        c = np.tanh(np.dot(W_hx, x) + r * np.dot(W_hh, h) + b_h)
        h = (1 - u) * c + u * h

        # Calculate the output
        y = np.dot(W_y, h) + b_y
        p = np.exp(y) / np.sum(np.exp(y))

        # Calculate the log probability of the true character
        log_prob = np.log(p[char_to_indices[test_text[i+1]]])
        log_probs += log_prob

    # Calculate perplexity
    log_perplexity = -log_probs / m
    perplexity = np.exp(log_perplexity)

    return perplexity
```

In [37]:
```
print("RNN perplexity on test set:", RNN_perplexity(weights_RNN, test_text, 5000))
print("GRU perplexity on test set:", GRU_perplexity(weights_GRU, test_text, 5000))
```

```
RNN perplexity on test set: [5.65447466]
GRU perplexity on test set: [3.9099905]
```

The perplexity scores indicate that the RNN has, on average, a probability of 1/5.65 = 17.7% and the GRU has a probability of 1/3.91 = 25.6% of correctly generating the next letter. This is quite impressive to me considering there are 44 possible characters so if the model was picking uniformily randomly it'd only get the next letter correct 2.27% of the time. It also makes sense to me that the GRU gets the next letter correct more often because letters have a strong dependence on previous letters and the GRU has a better memory mechanism.

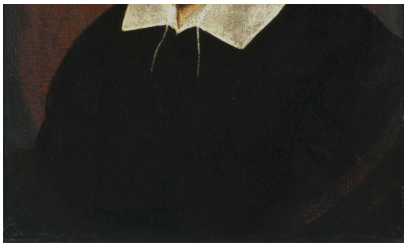
3. As seen in part 2 and 3 of this problem, the text generation is not perfect. Describe some possible model improvements that could make the quality of the generated text better.

Possible model improvements including training on more text (the training set contained less than 300,000 characters, which is not a lot), training for more epochs (the GRU was only trained for 10), and adding more hidden layers (the GRU for example only had 1)

Problem 2: Be the Bard (30 points)

Transformer models are the current state of the art in many sequence modeling tasks, and the Transformer architecture underlies most Large Language Models (LLMs), including ChatGPT, Llama, Mistral, etc.

In this problem, we will implement a Transformer language model from scratch in `numpy`. You will be



provided with the weights of a small, slightly simplified Transformer language model that we trained on the works of Shakespeare. We will walk through implementing each component of the Transformer architecture and ultimately assemble this into a language model that can generate some text in the style of the Bard.

In [4]:
```
import numpy as np
import pickle
import tiktoken
import matplotlib.pyplot as plt
import seaborn as sns
import math
```

In [5]:
```
d_model = 512
n_layers = 4
n_heads = 8
d_ff = 1024
vocab_size = 1024
block_size = 128

transformer_model_weights = np.load(f'problem2_model_parameters/model_weights_D{d_m
```

In [6]:
```
print('Parameters:')
for key in transformer_model_weights.keys():
    print(f'    {key}')
```

```
Parameters:
    word_embedding.weight
    position_embedding.inv_freq
    layers.0.attention.wq.weight
    layers.0.attention.wk.weight
    layers.0.attention.wv.weight
    layers.0.attention.wo.weight
    layers.0.feed_forward.0.weight
    layers.0.feed_forward.2.weight
    layers.1.attention.wq.weight
    layers.1.attention.wk.weight
    layers.1.attention.wv.weight
    layers.1.attention.wo.weight
    layers.1.feed_forward.0.weight
    layers.1.feed_forward.2.weight
    layers.2.attention.wq.weight
    layers.2.attention.wk.weight
    layers.2.attention.wv.weight
    layers.2.attention.wo.weight
    layers.2.feed_forward.0.weight
    layers.2.feed_forward.2.weight
    layers.3.attention.wq.weight
    layers.3.attention.wk.weight
    layers.3.attention.wv.weight
    layers.3.attention.wo.weight
    layers.3.feed_forward.0.weight
    layers.3.feed_forward.2.weight
    fc_out.weight
    fc_out.bias
```

In [22]:
```
transformer_model_weights['layers.0.feed_forward.0.weight'].shape
```

Out[22]:
```
(2048, 512)
```

Problem 2.1: Describe the model parameters

Describe the role of the parameters of the model. What are the weights `wq.weight`, `wk.weight`, `wv.weight`, and `wo.weight`? What are the dimensions of these matrices? What about the role and dimensions of the feedforward weights `feed_forward.0.weight` and `feed_forward.2.weight`?

You can refer to the descriptions below as well as lecture notes. Note, in particular, in the comments preceding Problem 2.4 that the attention matrices across heads are "packed together" when you read in the parameters in each layer.

- word_embedding.weight: converts one-hot encoded token vectors to vector embeddings (vocab_size, d_model)
- wq.weight: converts embedded input vectors to queries (d_model, d_model)
- wk.weight: converts embeddded input vectors to keys (d_model, d_model)
- wv.weight: converts embedded input vectors to values (d_model, d_model)

- attention.wo.weight: converts head output to transformer layer output (d_model, d_model)
- feed_forward.0.weight: MLP layer 1 weights, contributing nonlinearity (d_model, d_ff)
- feed_forward.2.weight: MLP layer 2 weights, reshaping tokens to be length d_model (d_ff, d_model)
- fc_out.weight: converts output vectors from transformer layers into logits over the vocabulary (d_model, vocab_size)
- fc_out.bias: bias for fc_out.weight conversion (vocab_size,)

where vocab_size is 1024, d_model is 512, d_ff is 2048

Tokenizer

To process text with a neural network model, we need to first tokenize it in order to convert it to a numerical format that the model can understand and process. The tokenizer converts strings into sequences of integer tokens in a fixed vocabulary. There are different ways to do this. For this problem, we trained a custom Byte-Pair Encoding (BPE) tokenizer on Shakespeare text, setting the vocabulary size to 1024. The code below demonstrates how it works.

In [7]:
```
# load the BPE tokenizer
with open('problem2_model_parameters/bpe1024_enc_full.pkl', 'rb') as pickle_file:
    enc = pickle.load(pickle_file)
```

In [8]:
```
# text to tokenize
text = """What's in a name? that which we call a rose
By any other name would smell as sweet;"""

print('Original text:')
print(text)
encoded = enc.encode(text)
print()

print('Encoded:')
print(encoded)
print()

print('Tokens: ')
print([enc.decode([idx]) for idx in encoded])
decoded = enc.decode(encoded)
print()

print('Decoded text:')
print(decoded)
```

```
Original text:
What's in a name? that which we call a rose
By any other name would smell as sweet;

Encoded:
[462, 320, 307, 258, 813, 63, 323, 621, 331, 800, 258, 697, 305, 10, 889, 801, 845,
813, 504, 260, 109, 408, 366, 818, 59]

Tokens:
['What', "'s", ' in', ' a', ' name', '?', ' that', ' which', ' we', ' call', ' a', '
ro', 'se', '\n', 'By', ' any', ' other', ' name', ' would', ' s', 'm', 'ell', ' as',
' sweet', ';']

Decoded text:
What's in a name? that which we call a rose
By any other name would smell as sweet;
```

Problem 2.2a: Token Embeddings

After tokenization, we get a sequence of integers that represent the text to processed, with each integer index corresponding to a particular token in the vocabulary (e.g., a word or word-part). The first step in processing this text is to turn it into a vector representation. This is done via a learned embedding look-up table. For each token in the vocabulary $t \in \mathcal{V}$, we learn an embedding $E_t \in \mathbb{R}^d$. A sequence of tokens $(t_1, \ldots, t_n)$ is transformed to a vector representation by mapping each token to its embedding $(E_{t_1}, \ldots, E_{t_n}) \in \mathbb{R}^{n \times d}$. This sequence of vectors is what the neural network model ultimately operates over.

In [9]:
```python
def embed_tokens(tokens, params):
    """
    Embed tokens using the input embeddings.

    Args:
        tokens (np.array): array of token indices, shape (n_tokens,)
        params (dict): dictionary containing the model parameters
    """

    # get needed parameters
    embeddings = params['word_embedding.weight'] # shape (vocab_size, d_model)
    # embeddings is a look up table for embeddings, with rows corresponding to toke
    # i.e., embeddings[token_index] returns the embedding for the token with index

    # look up embeddings
    embedded_tokens = embeddings[tokens] # shape (n_tokens, d_model)
    return embedded_tokens
```

In [10]:
```python
embed_tokens(np.array([1, 2, 3]), params=transformer_model_weights)[:3, :5]
```

Out[10]:
```
array([[-0.4494016 ,  0.8637606 ,  0.64816946, -1.0005027 ,  0.65828127],
       [ 1.1595719 , -0.09202019,  1.4256238 ,  1.7668523 , -1.366581  ],
       [ 0.7071919 ,  0.45128715, -1.1132193 , -0.18074755, -0.36634383]],
      dtype=float32)
```

Expected answer:

```
array([[-0.4494016 ,  0.8637606 ,  0.64816946, -1.0005027 ,
0.65828127],
       [ 1.1595719 , -0.09202019,  1.4256238 ,  1.7668523 ,
-1.366581  ],
       [ 0.7071919 ,  0.45128715, -1.1132193 , -0.18074755,
-0.36634383]],
      dtype=float32)
```

Positional Encoding

Transformer models are by-default permutation-equivariant. That is, they don't understand order or position. To make them understand positional information, we need to encode it directly in the token embeddings. One way to do this is to represent each possible position i with its own position embedding $PE_i \in \mathbb{R}^d$. One way to encode the position of each token is to simply add a positional embedding representing the position of the token.

In the original Transformer paper, the authors propose a particular choice for $PE_i \in \mathbb{R}^d$ based on sines and cosines with frequencies depending on the position i. Since the original proposal, many follow-up works proposed different positional encoding methods aiming to improve performance and length-generalization. In this problem, we'll use the sinusoidal positional encodings of the original Transformer paper.

We provide the code for computing these sinusoidal positional embeddings below. To give some intuition about the structure of the sinusoidal positional embeddings, we also plot a heatmap of the pairwise inner products $\langle PE_i, PE_j \rangle$. We see that that positions that are closer together have more similar positional embeddings, with additional oscillatory behavior on top of that.

Problem 2.2b: Describe the positional embeddings

Below is an implementation of the positional embeddings.

In [11]:
```python
def get_sinusoidal_positional_embeddings(sequence_length, dim, base=10000):
    inv_freq = 1.0 / (base ** (np.arange(0, dim, 2) / dim))
    t = np.arange(sequence_length)
    sinusoid_inp = np.einsum("i,j->ij", t, inv_freq)

    sin, cos = np.sin(sinusoid_inp), np.cos(sinusoid_inp)

    emb = np.concatenate((sin, cos), axis=-1)
    # return emb[None, :, :]
    return emb[:, :]
```

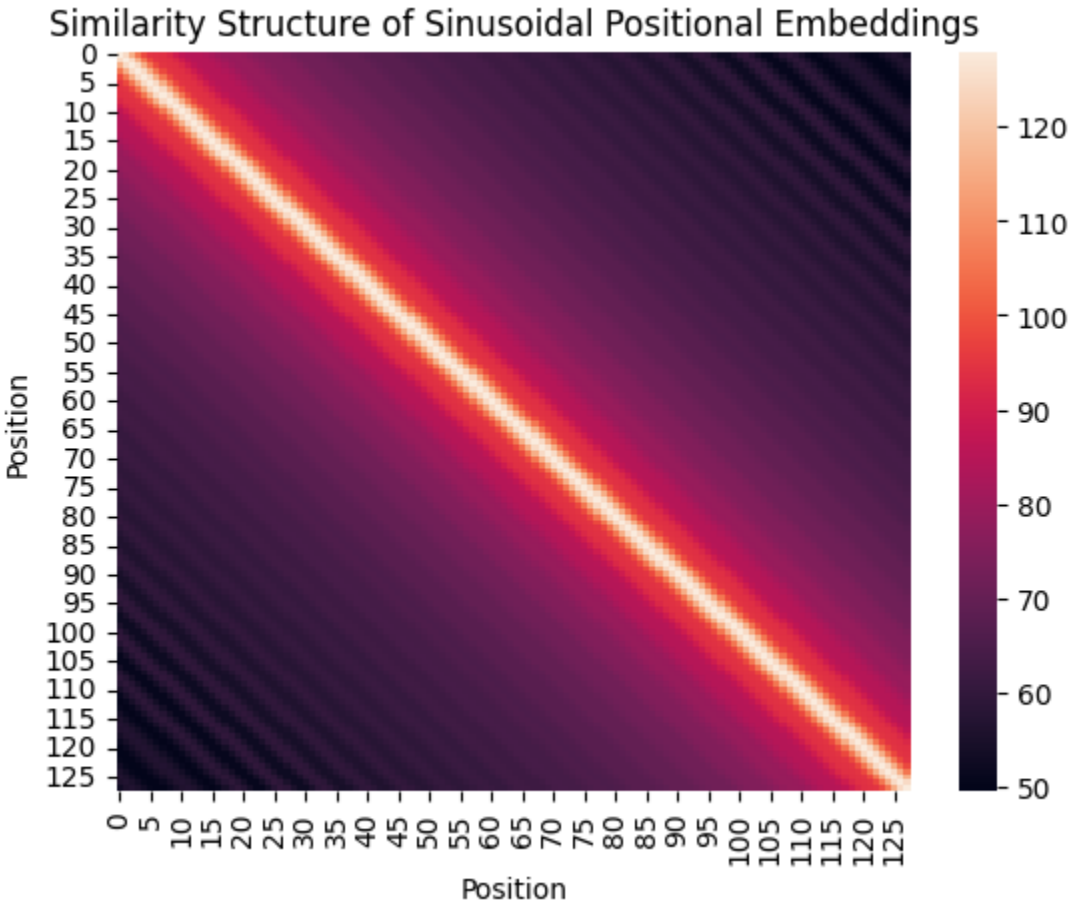
Explore the positional embeddings by computing the inner-product between all pairs of positional embeddings, and then plotting the similarity matrix as a heat map. Do the results make sense? What are the positional embeddings designed to model? Do they do this

effectively? Comment below.

In [12]:
```
pe = get_sinusoidal_positional_embeddings(128, 256)

#Your code here
pe_similarity = np.dot(pe, pe.T)

sns.heatmap(pe_similarity)
plt.title('Similarity Structure of Sinusoidal Positional Embeddings')
plt.xlabel('Position')
plt.ylabel('Position')
plt.show()
```



The heat map shows a spike in value along the identity, with a gradient of decreasing value extending out from it. This indicates that similarity between positional embeddings corresponds with closeness in sequence order of the associated tokens. This behaviour makes sense, because sinusoids are continous, so similar input values (positions) to a sinusoid will produce similar output values (embedding values), which make similar vectors. I would expect a token to be most predicted by the tokens closest to it, so it makes sense to use a positional embedding algorithm that clearly represents closeness in the sequence.

Multi-Head Attention

The core operation in a Transformer is multi-head attention, sometimes called self-attention.

In this problem, we will implement multi-head attention in numpy from scratch, given the trained parameters of the model.

The input is a sequence of vectors $X = (x_1, \ldots, x_n) \in \mathbb{R}^{n \times d_{model}}$. For each attention head $h \in [n_h]$, the parameters consist of a query projection matrix $W_q^h \in \mathbb{R}^{d_{model} \times d_h}$, a key projection matrix $W_k^h \in \mathbb{R}^{d_{model} \times d_h}$, and a value projection matrix $W_k^h \in \mathbb{R}^{d_{model} \times d_h}$. Here, d_h is the "head dimension", taken to be d_{model}/n_h (to maintain the same dimensionality between the input and output). The algorithm, for each head, is the following:

1. Compute the queries $Q = XW_q^h$, keys $K = XW_k^h$, and values $V = XW_v^h$. Note that the linear maps are applied independently for each token across the embedding dimension (not sequence dimension), such that $Q, K, V \in \mathbb{R}^{n \times d_h}$.
2. Compare the queries and keys via inner products to get an $n \times n$ attention matrix $A = \text{Softmax}(QK^\intercal/\sqrt{d_h}) \in \mathbb{R}^{n \times n}$.
3. Use the attention scores A to select values, producing the output of the self-attention head: $\text{head}_h = AV \in \mathbb{R}^{n \times d_h}$. We then concatenate the retrieved values across all heads, and apply a final linear map. Putting this all together yields:

$$\text{head}_h = \text{Softmax}((XW_q^h)(XW_k^h)^\intercal/\sqrt{d_h})XW_v^h \quad (1)$$

$$\text{MultiHeadAttention}(X) = \text{concat}(\text{head}_1, \ldots, \text{head}_{n_h})W_o \quad (2)$$

   Note that the matrices W_q^h, W_k^h, W_v^h are "packed together" across heads when you read in the parameters of the model.

Problem 2.3: Implement multi-head attention

Complete the implementation below.

In [13]:
```python
# first, we provide a couple of utility functions

def softmax(x, axis=-1):
    # a stable implementation of the softmax function
    exp_x = np.exp(x - np.max(x, axis=axis, keepdims=True))
    return exp_x / np.sum(exp_x, axis=axis, keepdims=True)

def apply_presoftmax_causal_mask(attn_scores):
    # apply a causal mask to the attention scores (set the entries above the diagon
    n = attn_scores.shape[-1]
    mask = np.triu(np.ones((n, n)), k=1)
    masked_scores = attn_scores - 1e9 * mask
    return masked_scores

def multi_head_attention(x, params, layer_prefix='layers.0'):
    """
    Compute multi-head self-attention.

    Args:
        x (np.array): input tensor, shape (n, d_model)
        params (dict): dictionary containing the model parameters
```

```
    layer_prefix (str): prefix of parameter names corresponding to the layer
    verbose (bool): whether to print intermediate shapes
"""

# get parameters of multi-head attention layer
wq = params[f'{layer_prefix}.attention.wq.weight'].T # (d_model, d_model)
wk = params[f'{layer_prefix}.attention.wk.weight'].T # (d_model, d_model)
wv = params[f'{layer_prefix}.attention.wv.weight'].T # (d_model, d_model)
wo = params[f'{layer_prefix}.attention.wo.weight'].T # (d_model, d_model)

head_dim = d_model // n_heads # dimension of each head
attn_scale = 1 / math.sqrt(head_dim) # scaling factor for attention scores

# the wq, wk, wv, wo matrices contain weights for all heads, concatenated
# first, we split wq, wk, wv, wo into heads
# note: there are more efficient implementations, but this is more verbose/peda
wq = wq.reshape(d_model, n_heads, head_dim).transpose(1, 0, 2) # (n_heads, d_mo
wk = wk.reshape(d_model, n_heads, head_dim).transpose(1, 0, 2) # (n_heads, d_mo
wv = wv.reshape(d_model, n_heads, head_dim).transpose(1, 0, 2) # (n_heads, d_mo

head_outputs = []
for head in range(n_heads):

    # get head-specific parameters (these are the query/key/value projections f
    wqh = wq[head] # (d_model, head_dim)
    wkh = wk[head] # (d_model, head_dim)
    wvh = wv[head] # (d_model, head_dim)

    # compute queries, keys, values
    q = np.dot(x, wqh) # (n, head_dim)
    k = np.dot(x, wkh) # (n, head_dim)
    v = np.dot(x, wvh) # (n, head_dim)

    # compute attention scores
    attn_scores = np.dot(q, k.T)

    attn_scores = apply_presoftmax_causal_mask(attn_scores)
    attn_scores *= attn_scale

    # apply attention scores to values
    head_out = np.dot(softmax(attn_scores), v) # (n, head_dim)

    # store the head output
    head_outputs.append(head_out)

# concatenate all head outputs
head_outputs = np.concatenate(head_outputs, axis=-1) # (n, d_model)

# apply output linear map W_o to concatenated head outputs
output = np.dot(head_outputs, wo) # (n, d_model)

return output
```

Problem 2.4: Test your Attention implementation

To test if you have the correct implementation, you can run the following test line. We show the expected output if your implementation is correct.

In [14]:
```
multi_head_attention(np.ones((block_size, d_model)), transformer_model_weights)[:3,
```

Out[14]:
```
array([[ 0.48334133,  0.19740422, -0.39514927, -0.40647455,  0.4831646 ],
       [ 0.48334133,  0.19740422, -0.39514927, -0.40647455,  0.4831646 ],
       [ 0.48334133,  0.19740422, -0.39514927, -0.40647455,  0.4831646 ]])
```

Expected output:

```
array([[ 0.48334133,  0.19740422, -0.39514927, -0.40647455,
0.4831646 ],
       [ 0.48334133,  0.19740422, -0.39514927, -0.40647455,
0.4831646 ],
       [ 0.48334133,  0.19740422, -0.39514927, -0.40647455,
0.4831646 ]])
```

MLP

Each Transformer layer (i.e., block) consists of two operations: 1) (multi-head) self-attention, which enables exchange of information between tokens, and 2) a multi-layer perceptron, which processes each token independently. A Transformer model is essentially just alternating between these two operations. In this problem, we will implement the multi-layer perceptron step. Typically, the MLP at each layer is simply a two-layer (one hidden layer) MLP or Feed Forward Network. In our model, we use a ReLU activation in the hidden layer, though other activations are possible. The same MLP network is applied to each token embedding in the sequence independently.

Given $X = (x_1, \ldots, x_n) \in \mathbb{R}^{n \times d_{model}}$, we apply the MLP as follows:

$$\mathrm{MLP}(X) = \mathrm{ReLU}(XW_1)W_2$$

Note that we don't use biases for simplicity.

Problem 2.5: Implement the MLP

Next, we need to apply the multi-layer perceptron in each layer. Complete the implementation below.

In [15]:
```
def relu(x):
    return np.maximum(x, 0)

def mlp(x, params, layer_prefix='layers.0'):
    # get MLP parameters
    w1 = params[f'{layer_prefix}.feed_forward.0.weight'].T # (d_model, d_ff)
    w2 = params[f'{layer_prefix}.feed_forward.2.weight'].T # (d_ff, d_model)
```

```
    # Your code here
    o = np.dot(relu(np.dot(x, w1)), w2) # (n, d_model)

    return o
```

Problem 2.6: Test your MLP implementation

To test if you have the correct MLP implementation, you can run the following test line. We show the expected output if your implementation is correct.

In [16]:
```
mlp(np.ones((block_size, d_model)), transformer_model_weights)[:3, :5]
```

Out[16]:
```
array([[0.06068574, 0.6727857 , 0.20872724, 0.42208509, 0.29517956],
       [0.06068574, 0.6727857 , 0.20872724, 0.42208509, 0.29517956],
       [0.06068574, 0.6727857 , 0.20872724, 0.42208509, 0.29517956]])
```

Expected output:

```
array([[0.06068574, 0.6727857 , 0.20872724, 0.42208509,
0.29517956],
       [0.06068574, 0.6727857 , 0.20872724, 0.42208509,
0.29517956],
       [0.06068574, 0.6727857 , 0.20872724, 0.42208509,
0.29517956]])
```

Final Prediction Layer

A Transformer model iteratively applies multi-head attention and MLP layers to process the input. This produces a processed representation of shape $n \times d_{model}$. To make the final prediction (e.g., predict the next token), we need to map the d_{model}-dimensional embedding vectors to logits over the output vocabulary. To do this, we simply apply a linear map that maps from d_{model} to **`vocab_size`**.

The starter code for this is given below; you need to complete it.

Problem 2.7: Implement the prediction layer as logits.

In [17]:
```
def prediction_head(x, params):
    # get needed parameters
    w = params['fc_out.weight'].T # (d_model, vocab_size)
    b = params['fc_out.bias'] # (vocab_size,)

    # Your code here
    logits = np.dot(x, w) + b # (n, vocab_size)

    return logits
```

Problem 2.8: Test the prediction head

In [18]:
```
prediction_head(np.ones((block_size, d_model)), transformer_model_weights)[:3, :5]
```

Out[18]:
```
array([[ 0.88448969,  0.07200013, -0.67318526, -1.11558351,  0.14410351],
       [ 0.88448969,  0.07200013, -0.67318526, -1.11558351,  0.14410351],
       [ 0.88448969,  0.07200013, -0.67318526, -1.11558351,  0.14410351]])
```

```
array([[ 0.88448969,  0.07200013, -0.67318526, -1.11558351,
0.14410351],
       [ 0.88448969,  0.07200013, -0.67318526, -1.11558351,
0.14410351],
       [ 0.88448969,  0.07200013, -0.67318526, -1.11558351,
0.14410351]])
```

Putting it all together: A Full Transformer Language Model

We are now ready to put this all together to assemble our Transformer Language Model. Recall that the Transformer architecture consists of iteratively applying multi-head attention and MLPs. Each time we apply attention or the MLP, we also apply a *residual connection*: $X^{(\ell+1)} = X^{(\ell)} + F(X^{(\ell)})$. This can be interpreted as a mechanism to enable easy communication between different layers (some people call refer to this idea as the "residual stream"). Real Transformers also include layer normalization in each layer, but we omit this for simplicity in this problem.

The full algorithm is given below:

1. Embed the tokens using the embedding lookup table: $(t_1, \ldots, t_n) \mapsto (E_{t_1}, \ldots, E_{t_n}) =: X^{(0)}$
2. Add the positional embeddings: $X^{(0)} \leftarrow X^{(0)} + (PE_1, \ldots, PE_n)$
3. For each layer $\ell = 1, \ldots, L$:
   A. Apply Multi-Head Attention: $\tilde{X}^{(\ell)} \leftarrow X^{(\ell-1)} + \mathrm{MultiHeadAttention}(X^{(\ell-1)})$.
   B. Apply the MLP: $X^{(\ell)} \leftarrow \tilde{X}^{(\ell)} + \mathrm{MLP}(\tilde{X}^{(\ell)})$.
4. Compute the logits

Problem 2.9: Complete the implementation

Complete the starter code below, which takes embeddings, adds positional encoding, and then adds the attention and MLP components to each layer. Remember that everything is added together, with the computations in one layer added to the outputs of the previous layer, forming the "residual stream".

In [19]:
```
def transformer(tokens, params):
    # tokens: (n,) integer array
    # params: dictionary of parameters

    # map tokens to embeddings using embed_tokens
```

```
    x = embed_tokens(tokens, params) # (n, d_model)

    # add positional embeddings
    pe = get_sinusoidal_positional_embeddings(x.shape[0], x.shape[1]) # (n, d_model
    x = x + pe

    # transformer blocks
    for i in range(n_layers):

        # compute multi-head self-attention and add residual
        attn_out = multi_head_attention(x, params, layer_prefix=f'layers.{i}') # (n
        x = x + attn_out

        # compute MLP and add residual
        mlp_out = mlp(x, params, layer_prefix=f'layers.{i}') # (n, d_model)
        x = x + mlp_out

    # compute logits via the prediction_head
    logits = prediction_head(x, params) # (n, vocab_size)

    return logits
```

Problem 2.10: Test your implementation

You can check your implementation against the expected output below.

In [20]:
```
transformer([0, 1, 2], params=transformer_model_weights)[:3, :5]
```

Out[20]:
```
array([[-1.9641349 , -5.12566872, -5.90677718, -5.57889839, -3.85043564],
       [-2.31297379, -4.9703405 , -3.49086668, -5.3996587 , -3.99684942],
       [-2.97001726, -4.84049568, -4.04949194, -4.01892107, -6.03805991]])
```

Expected Output:

```
array([[-1.96413494, -5.12566872, -5.90677725, -5.57889829,
-3.85043572],
       [-2.31297382, -4.97034061, -3.49086669, -5.39965866,
-3.99684957],
       [-2.97001738, -4.8404957 , -4.04949199, -4.01892112,
-6.03806011]])
```

Generate some text

Below, we provide some code for generating text from a Transformer language model. The sampling procedure is *autoregressive*. This means that we input some text to the model and it outputs a distribution over next tokens. We sample the next token and append it to the text, then repeat the procedure.

Problem 2.11: Complete the next token generator

Complete the next token generator, but filling in the missing code below. This uses

"temperature" to focus on the more probable tokens in a given context (as the temperature decreases). This results in sampling according to $\mathrm{Softmax}(\mathrm{logits}/T)$ where $T \geq 0$ is the temperature; lower temperature places higher probability on tokens having larger logits. Greedy sampling corresponds to $T = 0$, and selects the token with the largest logit.

In [23]:
```
def generate_with_transformer(prefix_text, params, max_len=128, greedy=False, tempe
    # encode seed text
    prefix_tokens = list(enc.encode(prefix_text))

    # initialize generated tokens
    generated_tokens = prefix_tokens

    # generate new tokens
    for i in range(max_len):
        # predict next token
        logits = transformer(generated_tokens, params)
        # logits[-1] corresponds to prediction of the next token
        if greedy:
            logit_predict = logits[-1]
            next_token = np.argmax(logit_predict)
        else:
            logit_predict = logits[-1] / temperature
            probs = np.exp(logit_predict) / np.sum(np.exp(logit_predict))
            next_token = np.random.choice(range(vocab_size), p=probs)

        # add next token to generated tokens
        generated_tokens.append(next_token)

    # This converts the tokens to text, using the tiktoken decoder:
    generated_text = enc.decode(generated_tokens)

    return generated_text
```

Problem 2.12: Test your implementation by generating text. You're the Bard!

Use your implementation to generate text according to the model. Generate text at different temperatures. Do the results make sense? Comment on the quality of the model. What changes to the model would lead to better results? Comment in the Markdown cell below.

In [24]:
```
prefix_text = """ANTONIO:
Do you not hear me speak?"""

generated_text = generate_with_transformer(prefix_text, transformer_model_weights,
print(generated_text)
```

```
ANTONIO:
Do you not hear me speak?

First Citizen:
We cannot

First Citizen:
He that hath done fell, sir, sir, sir, sir, sir, sir, sir, sir,
Which ne'er could the belly answer'd--
Which you,
Which you shall fell'd:
Which you shall tell you'll hear it smile, Iliber,
Which ne'--it it belly taunted head the belly,
MENENIUS:
There answer'd, you'll have lusedup
'
There
```

In [26]:
```
prefix_text = """ANTONIO:
Do you not hear me speak?"""

generated_text = generate_with_transformer(prefix_text, transformer_model_weights,
    greedy=False, temperature=0.8, max_len=128)
print(generated_text)
```

```
ANTONIO:
Do you not hear me speak?


First Citizen:
We cannot is a very do live upon the one o' the back, deed
The counse midle and the smpetering state,
Theirtill the whole, if that ner carein up, and thus--
Forembers
Thou rash like a taunting the restrainous parts of leads and unaily
No pat us: and cl, indeed
in
inion, the point yourselves while
F
```

Woah! That's awesome! It's like magic that it generates anything coherant at all. It labels each passages with an actor, which is characteristic shakespeare. In the greedy temperature run, the First Citizen actually responds to Antonio's question in a way that makes sense. With that said, the model is not perfect. In the low temperature model, almost all of the word are real words, but the phrases they form don't make sense. The higher temperature run shows greater variety of words and ideas but at the expense of some words not being real english. This greater variety make sense because the high temperature relatively privalidges lower probabilities, increasing variability. To improve the model, it could be trained further and on more shakespeare. More transformer layers and heads could also be added.