Problem 1:
In the cell which runs both the get_vocab and encode method, we see a portion of the input text which appears to be a script for a shakespeare play, a list of unique characters of letters and symbols under Vocab, and an encoding where the Text “Romeo and Juliet” where each character is mapped onto integers. Since there is a numeric mapping for each character it implies that the model we are building doesn’t really know words, sentences, grammar, etc. Instead its learning patterns within a string of characters. Since each character has an integer mapping we think that the model is predicting which character is going to come next by assigning probabilities to each of the 66 characters in the vocab and choosing the next letter with the greatest probability.

Problem 2:
After running the transformer language model for 20000 steps I’ve noticed for the first thousand steps, the training loss and test loss decreases linearly. During the first thousand steps, it appears that both the losses are approximately the same. Over the next couple thousand steps the decrease in both losses appears to still be linear, however the slopes for each loss becomes much less steep. Beyond 5000 steps there an increase in steps appeals to barely decrease both losses. After the first 1000 steps, the loss in training slowly becomes lower than the test loss as more steps are taken.

Problem 3:
The generate function works by predicting the next character one character at a time by using probabilities for each character in our vocab. These probabilities are calculated by getting the raw scores and softmaxing them. The line logits,= self(tokens[:, −ctxsize :]) is used so that the characters are used so that we only use the most recent characters in order to predict the next character. We limit the amount of characters used to predict the next character so that the context for the prediction is recent and relevant to the characters directly behind it.

Problem 4:
The variable device controls if the model is trained using the GPU or the CPU. When we remove the device = device when we try training our open we get an error and this error happens because there is a type mismatch since pytorch requires all the tensors that are interacting with each other to be on the same device. Our model is on the GPU, but the input tensor is on the CPU.

Problem 5:
In the training loop, the variables X, y, logits, and loss all have specific dimensions and purposes for how the model processes the data. X is the input batch, and its dimensions are [batch_size, ctx_size]. The batch_size refers to how many sequences are being processed at once, and ctx_size is the length of each sequence in the batch. The input sequences in X are made up of tokenized characters, where each character has been mapped to an integer. The variable y has the same dimensions as X because it’s basically the target sequences that the model is learning to predict. It’s just a shifted version of X where each position represents the next character in the sequence, so the model can learn what comes next. The logits variable, which is the output of

the model, has dimensions [batch_size, ctx_size, vocab_size]. Here, vocab_size is the total number of unique characters in the dataset. For each position in the input sequence, logits contains a vector of raw scores that represent the model's prediction for the next character across all characters in the vocab. Finally, loss is a scalar value that measures how far off the model's predictions are from the actual target values in y. This is calculated using cross-entropy loss, which is used to guide the model's training by adjusting its weights to minimize this value over time. Overall, the batch dimension lets multiple sequences to be processed at once, which makes training more efficient.

Problem 6:

1. Collect all word vectors w1,w2,…,wn into matrix X = [w1,w2,…,wn]
2. Compute queries/keys/values for each head
    a. Xqi = [Qiw1, Qiw2,...,Qiwn] = [qi1, qi2,... qin] = QiX
    b. Xki = [ki1, ki2,... kin] = KiX
    c. Xvi = [vi1, vi2,... vin] = ViX
3. Compute pairwise comparisons between words for each head
    a. Similarity matrix Si = XqiT * Xki
4. Scale the similarity matrix Si for each head
    a. Si = Si / sqrt(head_size)
5. Compute attention probs for each head
    a. Ai = softmax(Si,row)
6. Compute new word representations by taking weighted sum of values for each head
    a. Xhi = AiXvi
7. Concatenate all attention heads and project into final matrix
    a. Xh = [Xh1, Xh2,..., Xhh]
    b. Wo = projection weight matrix that is learned during training
    c. Return Wo*Xh

- Conceptually, multi-headed attention differs from the attention mechanism we looked at in lectures. This is because the attention mechanism we looked at in class uses just one head to find relationships between words but in multi-headed attention, we use multiple attention heads. Each attention head computes its own set of attention scores, allowing the model to focus on different aspects of the input sequence simultaneously. This enables the model to capture more complex relationships that our previous model utilizing a single head would miss.

Problem 7:

Matmul works in this case by computing matrix multiplication between q and the transpose of k across the batch dimension. This means that it computes a similarity score between every query and key vector for every batch. The output is the matrix scores with dimensions (batch_size,ctx,ctx). ctx is the sequence length, so each entry in scores[i,j,k] represents the similarity between the i-th batch which contains the j-th query and the k-th key. Broadcasting is important as it is used to multiply the scaling factor ($head\_size^{-0.5}$) across all the values in the matrix.

Problem 8:
This line applies a causal mask to the tensor ‘scores’ and it is necessary for the language model to work correctly as it makes sure that the predictions made by the model don’t include data from our future tokens. The dimensions of ‘scores’ is (batch_size,T,T) and the dimensions of self.tril is (T,T). masked_fill checks the condition if self.tril[:T,:T] == 0, and replaces the True values with negative infinity to ensure that when we calculate the softmax value, we get 0 so that the calculations for attention scores doesn’t use these particular tokens.

Problem 9:
Replacing dim = −1 to dim = 1 changes the dimension that we apply Softmax to from the last dimension (over all keys for each query) to the second dimension (over all queries for each key). Changing dim to equal 1 normalizes the attention scores across the dimension T, treating attention weights for each key across all queries separately. This changes the sum over value vectors because the weighted sum is now influenced by attention distributions across all queries for each key rather than within the context of a single query.

Problem 10:
After we train our model with the updated code for 20,000 steps, we find that the training loss and test loss are basically identical as we can barely differentiate between the two of them. In problem 2, the testing and training loss looked the same until the training loss decreased at a slightly faster rate but changing the model with the updated code causes the loss to be very similar. We also have a lower loss compared to the original model as the loss has a steep drop to 0.12552 in step 9000 and barely decreases until the very end. However, the generations produced by the updated model lack meaningful patterns compared to the coherent outputs from the original model in problem 2. This behavior occurs because applying softmax over the batch dimension breaks the attention mechanism's ability to focus on relationships within the sequence, disrupting the model’s ability to generate text that reflects the input context.