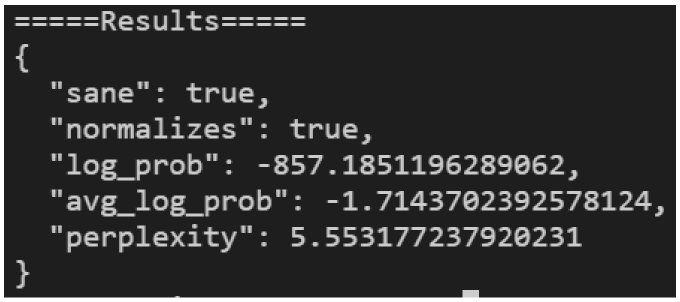
Part 1: Implementing an RNN Language Model
The RNN language model should pass sanity and normalization checks, get a perplexity less than or equal to 7, and train in under 10 minutes. This model is able to achieve these requirements. The output is shown below:

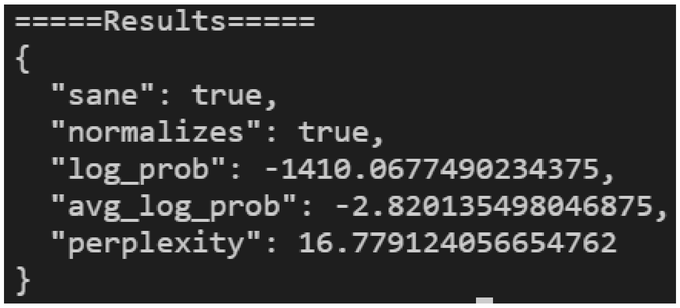


```
=====Results=====
{
  "sane": true,
  "normalizes": true,
  "log_prob": -857.1851196289062,
  "avg_log_prob": -1.7143702392578124,
  "perplexity": 5.553177237920231
}
```

The implementation of the RNN language model was very similar to the implementation in homework 3. It includes a recurrent module (such as LSTM), then a linear and softmax layer to produce an output. The training loop is also very similar, looping through chunks of text and repeatedly making predictions with the decoder, then comparing to the expected values and optimizing parameters with gradient descent. The RNN represents words by maintaining a hidden state that is passed along the entire sequence, which differs from transformer models.

During training, I experimented with multiple hyperparameters, mostly the number of epochs, hidden size, and embedding size. The final parameters used 5 epochs with a hidden size of 20 and an embedding size of 50. I found that increasing the hidden sizes and embedding sizes did not have too large of an effect on model performance; however this is likely due to the 10-minute time limit on training and the model would likely perform better given more training epochs.

Part 2: Implementing a Transformer Language Model
Due to the resource limits of this homework, there was no hard performance threshold for the transformer language models. This model was able to pass the sanity and normalization checks, and received a perplexity of about 17. The output is shown below:



```
=====Results=====
{
  "sane": true,
  "normalizes": true,
  "log_prob": -1410.0677490234375,
  "avg_log_prob": -2.820135498046875,
  "perplexity": 16.779124056654762
}
```

Implementation of the Transformer Decoder module required implementing multiple sub-components individually. First was the self-attention module, which consisted of multiple linear layers to transform the input vector into keys, queries, and values. These values are then combined, masked, and finally compressed back down into a smaller dimension with another linear layer. Second was the transformer block, which consists of one self-attention module, two layer normalizations, and one final feed forward sequence. The input is first fed into self-attention, normalized, fed through the feed forward sequence, and finally normalized again. Finally, the transformer decoder was implemented, which uses a token embedding, position embedding, transformer blocks, and a final linear output layer. This module first creates token and position embeddings of the input, passes them through the transformer blocks, and finally goes through a linear and softmax layer.

The training of the transformer model was slightly different compared to the RNN. This time, instead of looping through each character in the chunk and making a prediction, the entire chunk is fed into the transformer at once and used to predict the next token. This shows a fundamental difference between Transformers and RNNs, since with RNN you would have to input the chunk one character at a time. Instead, dependencies between tokens are calculated using self-attention, and is a reason why the transformer requires a positional embedding on top of a token embedding.

Tuning the hyperparameters was more work compared to the RNN, as there were many more parameters to adjust. During this stage, I experimented with different embedding size, hidden size, depth, chunk length, and learning rate. The final parameters used 5 epochs with a hidden and embedding size of 50, depth of 2, chunk length of 20, and learning rate of 0.001. I was not able to match the RNN performance with the transformer model, but this is expected due to the larger computational requirement of training these models.