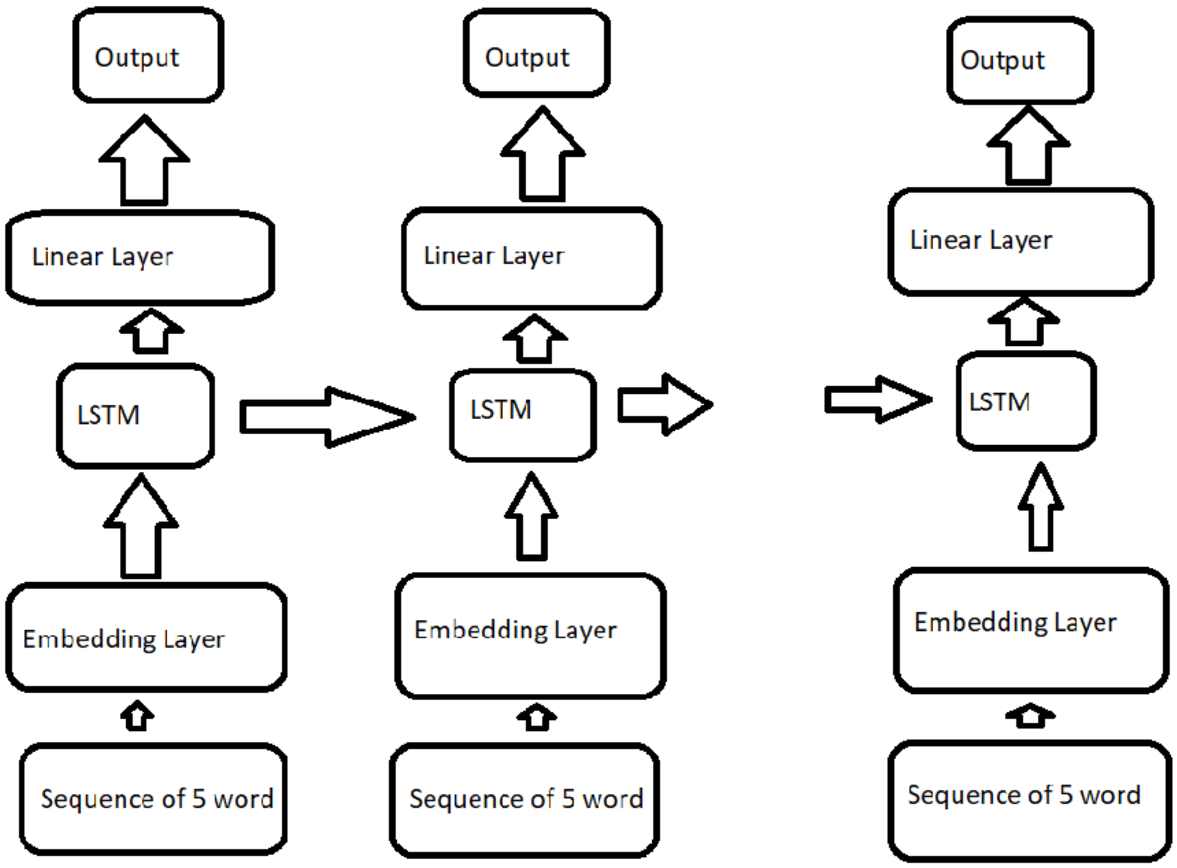
Introduction:

- The overall idea of our project is to take in a sequence of words and generate a different sequence that can follow the input prompt to form sentences that may appear in stories. We created a model that takes a sequence of words as an input and generates a probability distribution as the output, where the distribution is then used in maximum likelihood estimation to predict the next word. We would do this multiple times using the last couple of words in the already existing sentence, until a quota of **m** words have been generated, thus forming a sequence of **m** words that may follow the initial input text.
- We are using the RNN model which takes a sequence of words as input and generates the probability distribution of the next word based on our model's dictionary, which contains words that it recognizes, as output. We would do this multiple times until we satisfy the **m** words quota as mentioned above.
- The model uses a combination of an embedding layer, a LSTM layer instead of a traditional RNN layer, and a linear layer at the end to generate a narrative short story.
- We use a cross entropy function to evaluate the loss of each input and then use MLE to predict the next word using the output of the model.
- All code will be in the CSC413_final.ipynb file.
- To run our file, all the parameters that could be changed are at the top of the file in the second cell. If it is ran on google colab, remember to uncomment the google colab related lines in the first and fourth cell.

Model Figure:



- The nn.embedding layer that takes in indices is identical as a linear layer used to compute features that take in one-hot vectors. Instead of passing in one-hot vectors, passing in an index to the embedding layer will do the same thing as passing a one-hot vector with said index being set to 1 into a linear layer since the embedding layer is a lookup table that returns the features of the word with the indices passed in.
- Our LSTM model is a repeating RNN, for each input we take in the last 5 words of the current sequence, and feed said sequence through an embedding layer before it is fed to the LSTM layer. Afterwards, the output is put through a linear layer to be transformed to the actual output. In the next sequence, the model uses both the last words of the existing sentence and the hidden state from the previous input as input for the current input sequence, and the calculation is repeated until a total sequence of **m** words is generated.

Model Parameters

- Our Story Generator model accepts input_size, which is the amount of unique words in the vocabulary of a dataset.

- The model contains the following hyperparameters that are involved in the number of parameters in the model.
    - hidden_size is the size of the vector output of the embedding layer for one given word. This represents the size of the features that we extract out of each word before passing it into the LSTM layer.
    - n_layers represents the amount of layers of LSTM.
    - Finally, the n_gram_size parameter allows for setting the size of n_grams in our model.
- The vocab_length is a parameter that depends on the number of unique words in the vocabulary of the dataset.
- The model parameter comes from 3 layers, weight and bias of Embedding layer, weight and bias of LSTM layer and weight and bias of Output layer. The number of parameter in Embedding layer has input_size * hidden_size parameters. For Output layer we have hidden_size * output_size (weight) + output_size (bias) parameters. For LSTM layer we have n_layer * 4 * (hidden_size * n_gram_size) + (hidden_size * hidden_size) + hidden_size parameters. 4 represent (4 gates: input gate, forgot gate, cell gate, and output gate), (hidden_size * n_gram_size) is the input size, (hidden_size * hidden_size) is the weight matrix, and last hidden_size is the bias.

Model examples:

The following prompts were taken randomly from our test set.

Input prompt: ‘one day police decided implement’, length of generation is set to 20
Output: ‘one day police decided implement yadda explanation enchantress left help would sorry scared front two inside bar planet chair like peter hammer extinction thought metal’

Input prompt: ‘family decided spend time’, length of generation is set to 20
Output: ‘family decided spend time piece intention even wall alien holiday receiving farmer cry plastered shook top sub come filled red laughing soon listen gave’

Our model does not really have fail examples, even though the output sometimes looks like it does not make sense. This is due to the absence of stop words and punctuations. There can be multiple stopwords in between 2 words in the sequence that our model outputs. There could also be different punctuations, such as a period, between the word that our model does not include. More details on that will be explained in the justification part of this file.

Data source, summary, transformation, and data split:

- We found a “Popular Reddit Short Stories” dataset (https://www.kaggle.com/datasets/trevordu/reddit-short-stories). The creator of the dataset collected short stories which were written by real humans using Reddit API.

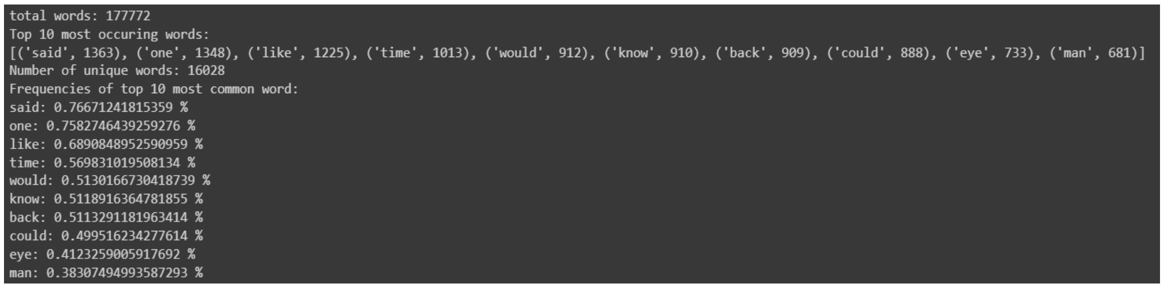
- The data consists of 4308 short stories, saved in a txt file that can be easily accessed. Each story in the dataset was written by an actual human: starting with <sos> to indicate the start of the story and ending with <eos> to indicate the end of the story. The data required filtering, since the dataset contained unnecessary comments as well as links.

- The stories in the dataset were written by actual humans. Each story in the dataset starts with <sos> indicating the start of the story and ends with <eos> indicating the end of the story.

- Our data required some filtering because the dataset contained actual comments by users as well as various links. Data filtering was performed but not perfectly, as it was hard to determine whether certain texts were comments from users or part of the story.

- Data cleaning was performed using multiple following steps:

    1) We reduced our dataset to 500 stories to reduce computation times in training/model fitting.
    2) We identified ‘<sos>’ and ‘<eos>’ as each story was between these tags.
    3) We removed ‘<sos>’ and ‘<eos>’ tags to make sure that they don’t influence our predictions in the future.
    4) We tried to clean HTML tags
    5) We removed punctuation to ensure that it does not influence our predictions
    6) We removed all stop words using the NLKT library (https://pythonspot.com/nltk-stop-words/) and made all words lowercase. Stop words are common words that are often considered insignificant in the analysis of natural language text so we decided to remove them to improve training and validation.
    7) We lemmatized words to their original form to simplify pattern detection for training using same NLKT.
    8) We stripped spaces.
    9) We converted short stories to ngrams and made sure that ngrams do not overlap from one story to another. This was done to make sure that ngrams are making sense.

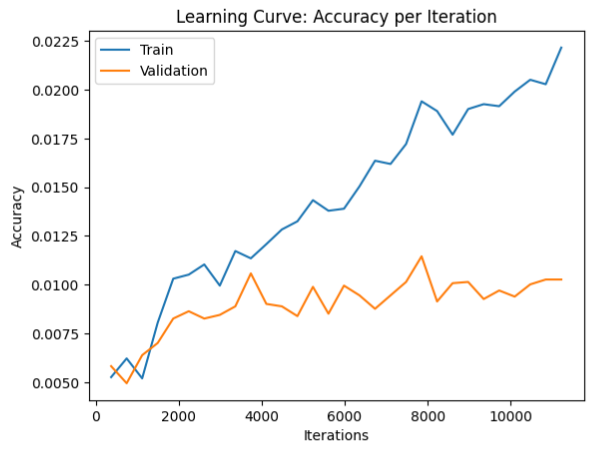


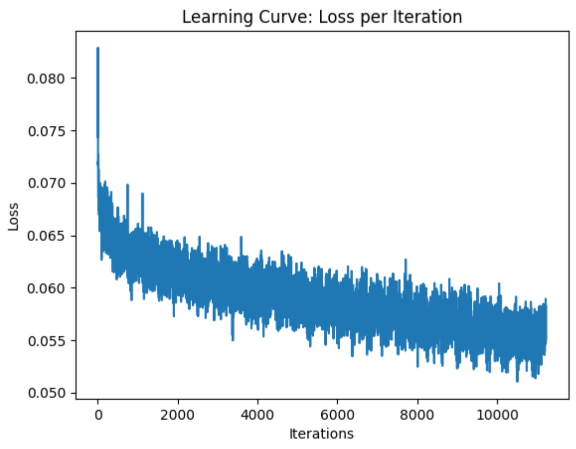
```
total words: 177772
Top 10 most occuring words:
[('said', 1363), ('one', 1348), ('like', 1225), ('time', 1013), ('would', 912), ('know', 910), ('back', 909), ('could', 888), ('eye', 733), ('man', 681)]
Number of unique words: 16028
Frequencies of top 10 most common word:
said: 0.76671241815359 %
one: 0.7582746439259276 %
like: 0.6890848952590959 %
time: 0.569831019508134 %
would: 0.5130166730418739 %
know: 0.5118916364781855 %
back: 0.5113291181963414 %
could: 0.499516234277614 %
eye: 0.4123259005917692 %
man: 0.38307494993587293 %
```

Image above shows the data summary of total number of words, number of unique words, top 10 most occurring words, amount of unique words, and the frequencies of the top 10 most common words.

- After data cleaning we created a vocabulary of words based on the ngrams. Additionally, we converted the words in vocabulary to indices which are used for training.
- Our data was split into the following: 60% training data, 20% validation data, and 20% test data. This split ratio is a common choice, as it provides a good balance between the amount of data used for training and the amount reserved for evaluation. A 60% Training set ensures that the model can learn patterns and relationships within the data effectively. A 20% Validation set, provides a sufficient amount of data for assessing the model's performance and helps to identify any overfitting or underfitting. It is also important for hyperparameter tuning to make the necessary adjustment for better accuracy. A 20% test provides a large enough sample of unseen data for evaluating the model's final performance.
- Our Training vocabulary contains 16028 unique words, with “said” being the most common word, appearing about 0.767% of the time.

Training curve:





- Note: the curve is not in percentage.
- As you can see from the accuracy curve graph, our model overfits with some time, especially after iteration 4000 where we see the validation accuracy start to stagnate and the training accuracy rise.
- Our loss falls, showing that the model is improving. It eventually slows down, continuing to improve, albeit at a much slower pace.

Hyperparameter tuning:

- The hyperparameters we experimented with include num_epochs, hidden_size, n_layers, learning_rate, batch_size, and n_grams_size. The final hyperparameter choices are as follows:
    - num_epochs: 15.This is the number of epochs that the model is being trained for.

- hidden_size: 128. This is explained in the section “Model Parameters” as it is relevant there.
- n_layers: 3 This is explained in the section “Model Parameters” as it is relevant there.
- learning_rate: 0.015. This is the learning rate of our model with each iteration.
- batch_size: 128. This is the number of n_grams that are used in each iteration.
- n_grams_size: 5. This is explained in the section “Model Parameters” as it is relevant there.

- For num_epochs, we tested multiple different values, including but not limited to 5, 10, 15, 30, 50, and 100. As we increased the num_epochs, our training accuracy increased, but the validation accuracy stabilized around 15 epochs.
- For hidden_size, we found a hidden_size of 128 yielded the best model performance out of the options we tried (64, 128, 256).
- For n_layers, stacking multiple layers can help the model capture more complex patterns in the input data. However, more stacked layers also increases the model’s complexity and computational time. After testing n_layers of 1, 2, and 3, we found n_layer of 1 quickly overfits the model, giving results of 1% validation and high % of training.
  n_layers of 2 also shows somewhat of a similar behavior but less extreme. In the end we choose n_layer of 3 because it best captures our model and still allows relatively fast computational time.
- For learning_rate, we tested learning rates of 0.5, 0.1, 0.05, 0.01, and 0.001. Learning rates greater than 0.015 resulted in some unstable performance when converging while a value less than 0.015 converges far too slowly.
- For batch_size, we experimented with multiple batch sizes, including 256, 128, 64, and 32, and found a batch size of 128 to be the best size for our model. Larger batch sizes take longer to converge, and considering we have a large dataset, selecting a smaller batch size is both time-efficient and better for generalization.
- For n_grams_size, we set the value to 5. We didn’t see major differences between 5 to 10 n_grams_size so we decided to go with 5 to make sure that vanishing gradient in the model doesn’t affect the results too much. On top of that, we could end up not using too many sentences in our data if the n_grams_size is too large. Since we cut off our story entries by sentences, if a sentence has a length < n_grams_size, then we decide to not use it since it cannot form a n_gram consisting of a sequence that originates from a single sentence.

Quantitative Measures:

- In our evaluation of the text-generation model, we chose to use cross-entropy loss and BLEU score as a quantitative measure rather than accuracy. This decision is based on the inherent nature of language generation tasks, where multiple valid outputs can coexist for a given input. Thus, relying on accuracy as a metric might not truly reflect the performance of an RNN model, as it could generate an appropriate sentence that deviates from the target but is still considered valid. Instead, cross-entropy loss and

BLEU score are more suitable for evaluating language models, although they come with their own drawbacks. Both cross-entropy loss and bleu score are sensitive to subtle differences between generated and reference text. Therefore, we might expect a low Bleu score.

Quantitative and Qualitative Results:

Quantitative:

- We will refer to the graphs in the Training Curve section regarding loss and accuracy.
- Through our graph, we can see that our lowest cross-entropy loss hangs at around 0.052 after iteration 10000.
- Referring back to our training vocabulary, the most likely word "said" appears 0.767% of the time, and since our validation accuracy at its best is around 1.15%, and training accuracy can go above 2%, we can safely say that our model performs better than repeating the most common word in the vocabulary to create a story.
- We also tried to calculate the BLEU score using our test set. Due to the size of the n_grams in the test set (roughly 10000), we decided to pass calculate the score of every 50 n_grams in the test set, up to a total of 100 entries, and average them. After passing 100 generated results using test data as inputs through BLEU evaluation, we got an average score of 0.39 out of 1, which by general standard is a decent score.

Qualitative:

- After the model was trained, we passed 1 standard example from the test set, and 1 self-written example with no influence from our dataset.
- When prompted with 'sliding across wet floor hit police' and told to generate a story of 20 words long, our model returned: "sliding across wet floor hit police conscious think well fist hear sword address followed door heard man said clearly pressed hand tried smile lamp putting taunted".

- When prompted with "he woke every single day like" and told to generate a story of 20 words long, our model returned: "he woke every single day like quest breeze door shut felt anyone inside already swat nosedive catholic whatever hand help sight reacting blue captain pouch tommy".

Justification of Results:

- On the surface, our model has a relatively low accuracy when comparing the generated word to the expected next word. This is because the problem we are trying to solve is complicated: sentences and stories do not have a fixed structure or method of being said or told. While a story may be written in 1 way by our model, there exists infinite other possibilities the same story can be told, whether through different words, expressions or sentence structures. Furthermore, many words, though not the same, can be used for similar meanings. This results in our model having a low accuracy because the story could use a different word from the expected target but still retain similar meanings due to the nature of the language having synonyms.

- Our model has a relatively low loss, meaning that it will generate a sequence of words with meanings close to what the intended result is, meaning that there will be grammatical issues, but the words will have similar meanings. It also shows that the model has and will be continuing to improve over time.
- During data augmentation, stopwords and punctuations were removed from the whole dataset to prevent the model from over-predicting them in the outputs as they are much more frequent than the other words. As a result, the output would be grammatically incorrect and incoherent at first glance but, if we add stopwords and punctuations in the correct spots, the returned stories should make more sense. If we look at the example of the results described in the previous section using "<u>he woke every single day like</u>" as the beginning of the story, there are many parts in the given example where connecting words made sense, one of such is "breeze door shut". With the addition of stopwords, it could be augmented into "the breeze made the door shut." which is a sentence that makes sense. Another example is "felt anyone inside already". It doesn't need too many augmentations with stopwords to make sense, simply commas and periods to separate it from previous words, for example "the breeze made the door shut and he didn't felt anyone inside already".
- Another glaring issue is that although small phrases may make sense together, the entire generated sequence may not. This is because the model is primarily an LSTM model and we are using n_grams to generate the input and targets. Not only does LSTM look more at the local words and features and does not remember the features globally due to vanishing gradient, but n_grams limit it even more because only the previous n_grams_size - 1 words will affect the next word generated since that is all we look at for the generation of the next word.
- Our BLEU score for the test set may have been sitting at around 0.39, but when looked at further it really doesn't mean too much. It's important to note that due to how bleu works, it will attempt to match the similarity of our given text against a set of references, which we have chosen to be our entire story set. When we calculated our BLEU score, the prompts used to generate the sequences are a sequence from the set of all possible sequences of some story from the entire story set since it is from our test set, which originates from entire story set. Due to this, the BLEU evaluation method will be guaranteed to find some similarity between our output story and list of every story.
- Therefore, due to the difficulty of the problem we are trying to solve, the expansiveness of the english language and infinite permutations of any story or sentence, the relatively low loss and example outputs being capable of making sense: we believe that our model at the very least shows potential of being able to perform the work intended with decent results.

Ethical Consideration:

- Positives:

    - The model could generate interesting stories that can give other authors some inspiration for their future work ideas.
- Negatives:
    - The model could potentially generate some texts that are very offensive towards a certain demographic.
    - The model could imply/directly mention actions that go against the social norms.
    - The people that collaborated in the story might feel that their IP rights are being violated since we are taking their work for other purposes that they did not know/plan of.

Authors:

Overall, we balanced and distributed the work equally between the members. Everyone worked on the report together, each pitching in on the parts that they are responsible for as described below.

- Eric: conceptualized, assisted in implementing the initial RNN LSTM sequence to sequence model.
- Ben (Shiang Zhi): wrote the training code, implemented the model code with Stepan, and also assisted with training the model and deciding the hyperparameters.
- Stepan: implemented and trained the final model, as well as helped with augmenting the data.
- Lida: wrote the code for augmenting the data along with Stepan, wrote the accuracy evaluation function, and also worked on splitting the data and preparing the batches for training.