Open-World Story Generation with Sequence-to-Sequence and Hierarchical Recurrent Encoder-Decoder Models

Ryan Zembrodt
University of Kentucky
ryan.zembrodt@uky.edu

April 13, 2019

Introduction

Story generation is an ongoing field of great importance in the future. Machines and AIs immersed in our society will be expected to behave certain ways by humans. They will need to naturally interact with humans in accordance to social conventions, and will require a commonsense and socio-cultural knowledge that humans will expect them to understand. Another reason is that humans regularly use stories in their everyday life to describe events, give context to an issue, or for entertainment. This allows us to convey and transfer complex ideas to one another. As machines are further integrated, they will need to be able to understand these stories to correctly understand humans, and generate these stories to interact with humans. Understanding and telling these stories will further help computers communicate with humans. One area currently used for these problems is that of machine learning, and more specifically, deep learning.

Recurrent neural networks (RNN) have been used recently in various applications for natural language processing and generation. They have been used for language translation [25], query predicting [24], dialogue generation [23], and many other areas. The project discussed here, supervised and guided by Dr. Harrison, had the end-goal of applying these RNNs, using two different existing approaches, for story generation. The approaches used in this project are a sequence-to-sequence network[1] and a hierarchical recurrent encoder-decoder (HRED)[2]. By training these models on an existing naturally occurring story text, in particular *Harry Potter and the Sorcerer's Stone* [22], they are able to predict, or "generate" a sentence given previous sentences as input. This generation of sentences can then be strung together to result in an automatically generated story.

[1] github.com/zembrodt/story-generation
[2] github.com/zembrodt/hred-story-generation

Several topics within machine learning and natural language processing were explored in this project, including recurrent neural networks, sequence-to-sequence networks, hierarchical networks, and word embeddings. A sequence-to-sequence network was chosen to allow deep learning without predetermined input and output sizes. The HRED model was implemented to compare with a sequence-to-sequence model and measure the benefit of providing the model with more context to a prediction than the previous sentence. The addition of this context provided better results and much lower perplexity scores, even with a smaller amount of data points. Finally, a large portion of the project was dedicated to the study of various word embeddings and their performance and effectiveness when training a model on the corpus using these two models. Various word embeddings were used, including a pre-trained *GloVe* embedding from a Wikipedia dataset, and two custom *word2vec* embeddings from a *Harry Potter* dataset. The final result of both models showed that out of all embeddings tested, the *GloVe* embedding provided the best models.

Related Work

Story generation can be split into two types: *closed-world* and *open-world*. While the project discussed in this paper is an implementation of open-world story generation, it is important to understand both types along with their benefits and limitations.

Closed-world Story Generation

A closed-world model is one that has constraints on the story given by the author, whether on the story's plot or its space. Their quality relies on the effectiveness of knowledge engineering by the author and the domain model. This domain model dictates what types of stories are possible, and includes predefined rules, entities, and outcomes of the world.

Considered one of the first story generation systems, *TALE-SPIN* simulates the events of characters, who each have their own unique personalities and goals. The system simulates events that attempt to allow each character to achieve their goal. By doing this, the system is able to create basic stories. The space is mainly authored, but is allowed to be modified by the system so characters' goals may be reached. The main achievement of this system is that it shows how important the quality of a story's space is. A poor quality space will lead to dull or nonsensical stories [14].

Another early system is *UNIVERSE*, aimed at generating melodramatic plots. Characters in the space are provided rankings for different elements and a set of goals. These traits act as constraints for each character with respect to different events. The system's goal was to be able to run indefinitely, continuously creating new plot twists. To achieve this, the overall plot was provided a goal, with different plot fragments provided. These fragments each had their own goals, and are continuously mixed and combined to generate new plots [8].

Mateas and Stern developed a system, *Façade*, to develop an interactive narrative. In this system, constraints are applied to the plot to ensure certain situations occur. The space is also completely authored to provide depth and personality to the characters. *Façade* uses a procedural authorship technique to generate its narrative when users interact with the system [13].

Tomaszewski and Binsted describe an implementation of an interactive drama system with *Demeter*. In this system, the space consists of fixed locations, props, characters, etc. The user dictates actions of their character from a fixed set while the system's manager responds with actions by the story's other characters. The user interacts with objects in the world by selecting an available *verb*, which is then translated into one or more *actions* that give the verb context within the current story. The system's manager uses this to select the next *scene* to play based on a set of rules [26].

In another algorithm, *Fabulist*, by Riedl et al. rules needed to be provided about the world along with the goal of the story. These rules defined different actions that could be taken along with their parameters, constraints, preconditions, and effects. In this algorithm and similar ones, it is difficult to separate the creativity of a story between the algorithm and the author [21].

Closed-world story generators require authored domains for their use, and are unable to generate stories outside of this domain. Planning for stories is also a difficult problem to solve due to being PSPACE-complete. However, due to the authorship of these systems, the stories they are able to generate will be coherent and detailed, and each event generated will be motivated intentionally.

Open-world Story Generation

An open-world model is one that is able to generate a novel story about any number of topics, just like humans.

Li and Riedl attempted to generalize the story generation concept to open-world by creating an interactive generator in *Scheherazade* using plot graph learning. This method of learning requires the input to be pre-processed into events of the story's plot. *Scheherazade* does not rely on any rules of the fictitious world and uses crowdsourcing to automatically learn the domain knowledge. With this knowledge the system is able to construct and understand stories about everyday activity. However, without specific rules for the story to follow, the stories are simpler and shorter than those possible by closed-world models. The use of plot events also limits the type of stories that can be told. This lack of complexity is a trade-off for the system's scalability [9].

Recently, recurrent neural networks have been used to handle the large and growing story corpora. These neural networks also allow a more general approach to their input, requiring less pre-processing than other methods, such as plot graph learning. The difficulty in this is the randomness due to the complexity of learning patterns from the corpora. Pichotta and Mooney describe several systems for predicting both events of a text and the raw sentences of a text. Their technique uses Long Short-Term Memory recurrent neural nets for their models to encode and decode sequences. They compared three systems for

these techniques. First, a system that was trained to encode a sentence's tokens and decode the successor sentence's tokens. Next, a system trained to encode a sentence's events and decode the successor sentence's events. Lastly, a system trained to encode a sentence's events, decode the successor sentence's events, encode these events, and then decode the successor sentence's tokens. These systems are similar to the sequence-to-sequence network implemented within this project, however these implementations just use a single recurrent neural net for both encoding and decoding [20].

Martin et al. developed a system using deep neural networks to generate the events of a story, rather than specific sentences, then generate sentences from each event. This system required a technique of pre-processing existing story corpora into event sequences. The *event2event* was a recurrent multi-layer encoder-decoder network, similar to what is used within this project [12].

Harrison et al. introduces an open story generation approach using Markov Chain Monte Carlo (MCMC) search. An MCMC method attempts to approximate a posterior distribution of an unknown function by performing simulation. This differs from previously discussed neural net approaches since they attempted to generate stories by predicting based on previous observations. Here, the MCMC method wants to learn the unknown distribution of a hypothetical storyteller that is able to tell coherent stories [6].

Open-world story generators significantly reduce the burden of authorship due to the lack of a domain model. Without a domain model, an open-world system has an increased variety in stories it can generate. Due to this generality though, the systems have an increase in nonsensical stories or ones that lose track of context over time. They also rely on a massive amount of data to train on in order to be effective.

Background

To create the models implemented in this project, many concepts within deep learning are utilized and will be discussed below. Beyond deep learning and neural networks, word embeddings are also used to allow the models a jump-start with and potentially more accurate representation of words in a vector-space.

Recurrent Neural Networks

While deep neural networks (DNN) are useful and powerful models, they are limited with the requirement of fixed input and output dimensionality, and therefore cannot perform sequential tasks. The RNN is an improvement upon this model, allowing the neural network to read in a non-fixed number of vectors. An RNN is a neural network that allows information to persist by looping with a persistent state carried over between each iteration. This allows the RNN to be given a sequence of inputs and compute an output at each step, with a recurrent

state passed on to the next step in the sequence. This recurrent state is used as a context for the current input of all previous inputs within the sequence.

If a neural network has input x_i and output y_i, an RNN builds on this by passing the last hidden state h_i to the next network for each input in the input sequence $(x_1, ..., x_t)$. This hidden state is then used within the next network, along with the next input in the sequence x_{i+1}, to compute the next output y_{i+1} in the output sequence. Each value in the input sequence will be given to a separate copy of the neural network, with each network connected as a chain by their last hidden states. As mentioned previously, this hidden state connection allows information to pass between steps in the neural network, and therefore persist across the entire input. With each network computing an output y_i, these outputs can then be combined to form the RNN's output sequence $(y_1, ..., y_t)$ [10].

RNN Overview

A simple implementation of an RNN consists of two equations. The first equation is used to compute the hidden state h_t at each timestep, that will then be needed when the RNN is given the next input in the sequence x_{t+1}.

$$h_t = \sigma(W^{\text{hx}} x_t + W^{\text{hh}} h_{t-1} + b_h)$$

In this equation, σ is the activation function. A common choice for σ is *tanh*. W^{hx} represents the matrix of weights between the input and hidden layer, and W^{hh} is the matrix of weights between the hidden layer and itself. The output at time t is computed as

$$y_t = \text{softmax}(W^{\text{yh}} h_t + b_y)$$

In this case, W^{yh} is the matrix of weights between the hidden layer and output layer. In both equations, the vectors b_h and b_y are bias parameters that allow each node to learn an offset. With these two equations, output y_t will therefore be influenced by input x_t due to h_{t-1}'s use in the first equation. Since y_t is influenced by x_{t-1} by way of the recurrent state via the hidden layers, it will also be influenced by all previous inputs within the sequence. *Figure 1* shows an illustration of an RNN. However, within this illustration, the issue with a basic RNN is shown. As the timesteps in the RNN increase, the contributions of previous inputs decrease exponentially overtime, such that the most recent input has the highest impact on the current output. The structure of various input sequences may cause related inputs to not necessarily be adjacent, and because of this the basic RNN implementation is not ideal [10]. Replacements for the basic activation function have been proposed to remedy this issue, and will be discussed in further detail below.

Long Short-Term Memory (LSTM)

An RNN's performance can be improved further by replacing its default unit with a more advanced one. One such unit is the LSTM. An LSTM is effective at

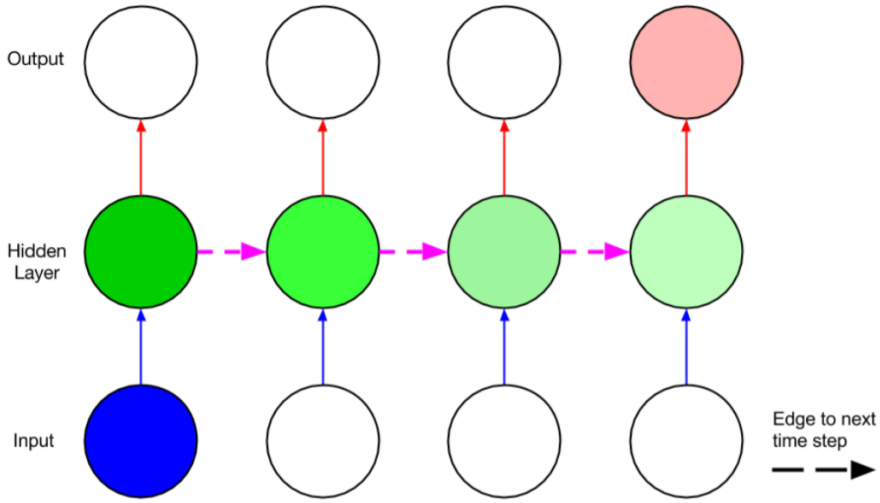


Figure 1: Example illustration of an RNN by Lipton and Berkowitz

being able to capture long-term dependencies. The LSTM's memory cell is able to maintain its state over time, and is able to regulate the flow of information into and out of the cell. Due to the memory cell being able to maintain a state over time, it allows the LSTM to capture these long-term dependencies.

The LSTM replaces the RNN's default unit, such a simple single *tanh* layer. It also adds a further input/output to the node: a cell state c_t. The cell state runs through the entire chain of nodes, with only minor interactions. The LSTM has the ability to add or remove information from this state, and information is allowed to flow between states. This concept is extremely useful for story generation, where it is important for the structure of a sentence, paragraph, chapter, etc to know what information came previously [5].

LSTM Overview

At timestep t, the LSTM first decides what information is removed and forgotten from the cell state. The *forget gate layer*, implemented as a sigmoid layer, looks at the LSTM's inputs x_t and h_{t-1} and outputs a number between 0 and 1 for each number in the previous cell state c_{t-1}, where 0 is to completely forgotten and 1 is to completely kept. The next step is to decide what new information should be stored in the cell state. An *input gate layer*, also implemented as a sigmoid layer, decides what is to be updated. A *tanh* layer then creates a vector of new candidates $\widetilde{c}_t$ that could be added to the new cell state. The output of the *input gate layer* i_t is then combined with $\widetilde{c}_t$ to create the update. Next, the previous cell state c_{t-1} is updated. It is first multiplied with the output of the *forget gate layer* f_t, to forget specific things, then added with the update $i_t \cdot \widetilde{c}_t$, to finally get c_t. Lastly, the output of the LSTM must be calculated. A

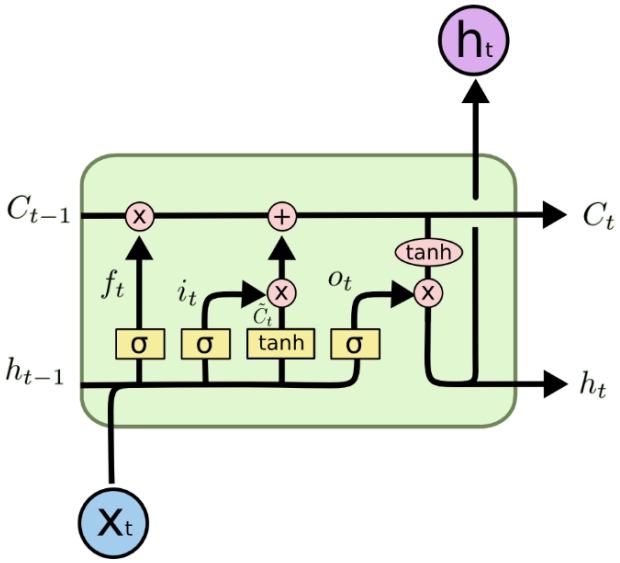


Figure 2: Example illustration of an LSTM unit [17]

sigmoid layer on the inputs x_t and h_{t-1} is used to decide what parts of the cell state should be outputted. The cell state c_t is then put through *tanh* to push its values between -1 and 1 and multiplied with the output of the sigmoid layer, resulting in h_t. Both the LSTM's output and hidden state will be this value of h_t [5].

Gated Recurrent Unit (GRU)

The GRU was proposed by Cho et al. as a new unit that is able to adaptively remember and forget. They were motivated by LSTM, but to develop a unit that is much simpler to compute. This simplicity comes from only needing to compute two gating units, rather than the four LSTM contains. Besides this gate count, a GRU is also able to capture long-term dependencies between time steps. However, the GRU is able to do this without having separate memory cells, unlike an LSTM [3]. It has also been shown that the GRU performs better and is more effective with smaller datasets than an LSTM [4]. Due to the potentially limited datasets available for certain styles of stories, the GRU is extremely important to still allow the model to train effectively and efficiently while capturing long-term dependencies.

GRU Overview

Figure 3 displays an illustration of the GRU's hidden activation function. The GRU consists of an update gate z and a reset gate r. The update gate decides

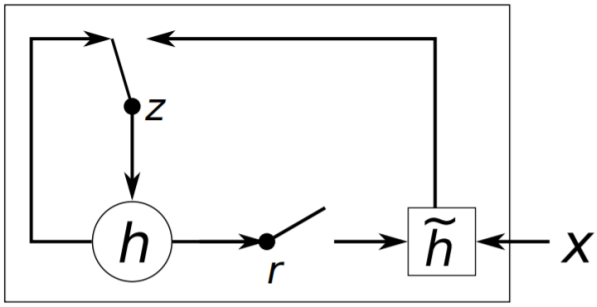


Figure 3: Example illustration of a GRU by Cho et al.

whether the hidden state should be updated with the new hidden state $\widetilde{h}$, while the reset gate decides whether the previous hidden state h is ignored when computed the new hidden state $\widetilde{h}$. The reset gate of the j-th activation unit is computed by

$$r_j = \sigma([W_r x]_j + [U_r h_{t-1}]_j)$$

Where sigma is the logistic sigmoid function, and $[.]_j$ denotes the j-th element of a vector. x and h_{t-1} are the input and previous hidden state. W_r and U_r are weight matrices that are learned. The update gate of the j-th activation unit is computed by

$$z_j = \sigma([W_z x]_j + [U_z h_{t-1}]_j)$$

Similar to r_j definition. The actual activation of the proposed unit h_j is computed by

$$h_j^t = z_j h_{t-1}^j + (1 - z_j)\widetilde{h}_j^t$$

Where $\widetilde{h}_j^t$ is computed by

$$\widetilde{h}_j^t = \phi([W_x]_j + [U(r \odot h_{t-1})]_j)$$

When the rest gate is close to 0, the hidden state is forced to ignore the previous hidden state and reset with the current input. This allows the hidden state to drop any information that's found to be irrelevant, forming a more compact representation. However, the update gate also can control how much information from the previous hidden state will be passed to the current hidden state. This allows the RNN to remember long-term information. Each hidden unit has separate reset and update gates. Therefore, each will learn to capture dependencies in varying amounts of time. The reset gate will be frequently active in units that learn to capture short-term dependencies, while updates will be mostly active in units that capture long-term dependencies [3].

Sequence-to-Sequence Networks

An RNN is able to easily map an input sequence to an output sequence if the size of both is known ahead of time. However, it is not clear how to apply a single RNN to the mapping of an input sequence to an output sequence for sequences in general. The solution to this is the sequence-to-sequence network. Using two RNNs in tandem makes this possible, with on RNN mapping the input sequence into a fixed-size vector, and the other RNN mapping this vector into an output sequence. With this method, the size of both of these sequences does not need to be known and can vary. To allow this variation, a technique is employed where a token is defined in the sequence vocabulary to designate the end of the sequence. With this method, the decoder is able to generate the output sequence until the token is generated, or a max output size is reached. These two RNNs are known as the encoder and decoder. The encoder RNN maps the input sequence, therefore encoding the "meaning" or "concept" of the input. The decoder will then take the encoder output, decoding the "meaning" back into a target sequence. The goal of the network is to estimate the conditional probability $p(y_1, ..., y_{T'}|x_1, ..., x_T)$, where $(x_1, ..., x_T)$ is an input sequence, and $(y_1, ..., y_{T'})$ is the corresponding output sequence. The lengths of both of these sequences, T and T', may differ. This conditional probability is first computed by obtaining the fixed-dimension vector v that represents the input sequence. This vector is given as the last hidden state of the encoder. Then the probability of $y_1, ..., y_{T'}$ is computed by the decoder whose initial hidden state is set to v.

$$p(y_1, ..., y_{T'}|x_1, ..., x_T) = \prod_{t=1}^{T'} p(y_t|v, y_1, ..., y_{t-1})$$

Each $p(y_t|v, y_1, ..., y_{t-1})$ distribution is represented as a softmax over all the items in the vocabulary. This concept is very important for generating sentences in a story as their lengths will vary, and this allows the model to generate a sentence of any length [25].

Attention Decoders

A simple decoder uses only the *context vector*, or last output of the encoder, as its initial hidden state. According to Bahdanau et al., a potential issue with this approach is that because the encoder must compress the input sequence into a fixed length vector, the network may have trouble handling long sequences. An attention decoder, however, allows the decoder RNN to "focus" on a different part of the encoder's outputs for every step of the decoder's own outputs. Meaning the model is able to learn which parts of the input sequence are relevant to which part of the output sequence without relying only on the context vector.

To accomplish this, instead of using a single context vector, an attention decoder develops a vector c_i that is filtered specifically for each output of the encoder. The context vector c_i for the i-th output depends on a sequence of *annotations* $(h_1, ..., h_{N_x})$ that are mapped by the encoder from the input sequence

x. Each annotation h_i contains information on the whole input sequence, but with a strong focus on parts surrounding the i-th element. The context vector c_i is computed as a weighted sum of these annotations

$$c_i = \sum_{j=1}^{N_x} \alpha_{ij} a_j$$

where α_{ij} is the weight of the annotation, computed as

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{N_x} \exp(e_{ik})}$$

e_{ij} is computed as

$$e_{ij} = a(h_{i-1}, a_j)$$

which is the *alignment model* that scores how well the inputs around position j and output at position i match. It is dependent on the decoder's previous hidden state and the j-th annotation of the input sequence. This *alignment model* a is a feedforward neural network and is jointly trained with the other components within the system. Finally, with this information we can compute the conditional probability of output y_i using

$$p(y_i|y_1, ..., y_{i-1}, x) = g(y_{i-1}, s_i, c_i)$$

where g is a nonlinear, potentially multi-layered, function that outputs the probability of y_i and s_i is the hidden state at time i. This hidden state is computed by

$$s_i = f(s_{i-1}, y_{i-1}, c_i)$$

The output at position i is therefore based on the RNN's previous hidden state and the j-th annotation h_j of the input sequence. *Figure 4* shows how each output y_t is computed using the previous hidden state s_{t-1} and a sum of all the weighted *annotation* as the context vector after encoding the sequence x [1].

Hierarchical Recurrent Encoder-Decoder

These solutions, however, do not allow context to persist between sentences. A sequence-to-sequence network is only able to learn the relationship between two sentences, but has no way of ensuring that long-term dependencies are kept over many sentences. Essentially, this becomes a system of chaining bigrams of sentences together to form a story. The idea of a story has an overarching narrative and ideas between sentences, such as for paragraphs, chapters, and an entire book. For a model to effectively generate these, it must consider more than just the most recent sentence. The idea of storing context over time for deep learning has been explored in several similar problems recently, such as context-aware query suggestion and dialogue systems.

Sordoni et al. improves upon the sequence-to-sequence encoder-decoder concept by developing a hierarchical recurrent encoder-decoder for generating

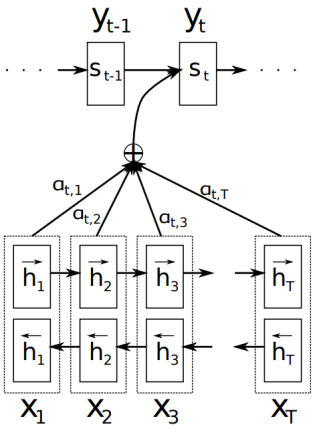


Figure 4: Example illustration of an attention decoder by Bahdanau et al.

context-aware query suggestions. The idea is to not only have an RNN for encoding the input sequence, and an RNN for decoding the output sequence, but also a context encoder RNN to encode a sequence of encoded sequences. This context encoder's hidden state is then used within the decoder RNN to allow it to have a context of all input sequences encoded up to that point when it decodes the output sequence, and therefore is able to know beyond the most recent sequence. This implementation splits this up into *query-level* encoding, *session-level* encoding, and *next-query* decoding.

The *query-level* RNN reads in each word from a query $Q_m = \{w_{m,1}, ..., w_{m,N_m}\}$ for each training session, or sequence of queries. It updates its recurrent state $h_{m,n}$ using each word $w_{m,n}$ and the previous hidden state $h_{m,n-1}$ as inputs for the GRU such that

$$h_{m,n} = GRU_{enc}(h_{m,n-1}, w_{m,n}), n = 1, ..., N_m$$

where $h_{m,n} \in \mathbb{R}^{d_h}$ and $h_{m,0}$ is the null vector. Therefore, the last recurrent state h_{m,N_m} is the fixed-length vector that stores information on all words within the query Q_m, and is denoted as q_m. This vector will be used within the *session-level* encoder and is a general, acontextual representation of the query. It is also possible to compute all these vectors $q_1, ..., q_M$ in parallel.

Next, the *session-level* encoder takes the input of all final *query-level* encoder recurrent states and computes a sequence of session-level recurrent states $s_1, ..., s_M$. It uses its own GRU function, where its recurrent state is of the *session-level* dimension, potentially different than the *query-level* dimensionality. It's recurrent state s_m is updated according to

$$s_m = GRU_{ses}(s_{m-1}, q_m), m = 1, ..., M$$

where $s_m \in \mathbb{R}^{d_s}$ and s_0 is also initialized as the null vector. The number of *session-level* recurrent states is the number of queries in the session. Each

session-level recurrent state summarizes the queries that have been processed up to position m within the session and is sensitive to the order of previous queries. It also inherits the sensitivity of the order of words in a query from the query vectors q_m.

Lastly, *next-query* decoding is an RNN decoder responsible to predict the next query Q_m given the previous queries $Q_{1:m-1}$. By doing this, the decoder estimates the probability

$$P(Q_m|Q_{1:m-1}) = \prod_{n=1}^{N_m} P(w_n|w_{1:n-1}, Q_{1:m-1})$$

In this instance, the decoder's recurrent state is initialized with a non-linear transformation of the *session-level* encoder's last recurrent state s_{m-1} according to

$$d_{m,0} = \tanh(D_0 s_{m-1} + b_0)$$

where $d_{m,0} \in \mathbb{R}^{d_h}$, $D_0 \in \mathbb{R}^{d_h \times d_s}$ projects the *query-level* vector s_{m-1} into the decoder's space, and $b_0 \in \mathbb{R}^{d_h}$. This allows the information of the previous queries to be transferred to the decoder RNN. The decoder recurrent state $d_{m,n}$ is updated with its own GRU function, and each recurrent state is used to compute the probability of the next word $w_{m,n+1}$ given both the previous words $w_{m,1:n}$ and previous queries $Q_{1:m-1}$. The probability of word $w_{m,n}$ is computed as

$$P(w_{m,n} = v|w_{m,1:n-1}, Q_{1:m-1}) = \frac{\exp\ o_v^\top \omega(d_{m,n-1}, w_{m,n-1})}{\sum_k \exp\ o_k^\top \omega(d_{m,n-1}, w_{m,n-1})}$$

where $o_v \in \mathbb{R}^{d_e}$ is the output embedding of word v and ω is a linear transformation layer that's used instead of computing the decoder's recurrent state directly. ω is computed as

$$\omega(d_{m,n-1}, w_{m,n-1}) = H_0 d_{m,n-1} + E_0 w_{m,n-1} + b_0$$

where $H_0 \in \mathbb{R}^{d_e d_h}$, $E_0 \in \mathbb{R}^{d_e V}$, and $b_0 \in \mathbb{R}^{d_e}$. When predicting the first word in the query Q_m, $w_{m,0}$ is set to the null vector. The parameter E_0 modifies how much the previous word is responsible for predicting the next word. Therefore, the word v has a high probability when o_v is "near" the vector $\omega(d_{m,n-1}, w_{m,n-1})$ *Figure 5* shows an illustration of the model with example queries.

Sordoni et al. note the impact of session length and of context length in an HRED model. The HRED model's performance increases significantly as the session's length, or number of queries, increases. For shorter sessions the improvement over pairwise models is marginal, but consistent with that it is slightly improved by the short context [24].

Word Embeddings

Input for recurrent neural networks is always in the form of vectors, but they are applied in many natural language problems. For the networks to take an

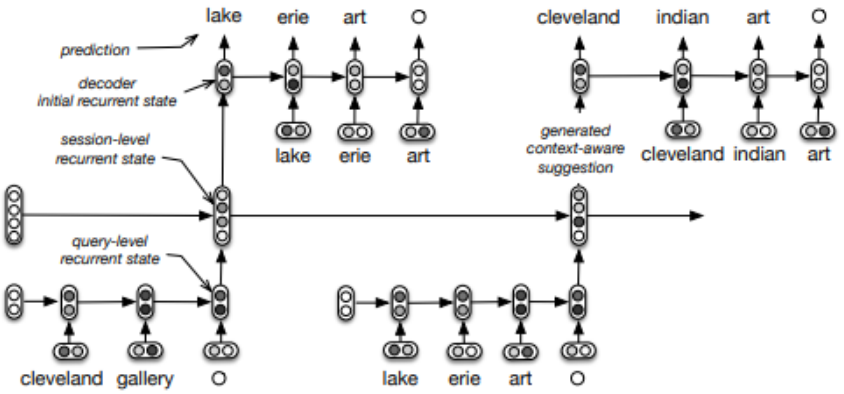


Figure 5: Example of an HRED illustrated by Sordoni et al.

input sequence of natural language, vector representations of words must be developed. One such way to do this is creating one-hot vectors for words, with every index in the vector representing a different word from the vocabulary. This approach, however, is not ideal, as it leads to very large and sparse vectors, and will likely not provide a semantic relation from one word to another. Word embeddings provide such relations between words in a vocab, creating a vector space representation of the words [19].

Within the experiments for this project, two different models for creating word embeddings are used: *GloVe* [19] and *Word2vec* [16]. These models are able to map various statistics of words down to smaller and denser vectors than the previously mentioned one-hot vectors. The resulting vector space also captures the semantic meaning of words within the corpus the model was trained on, where semantically similar words are mapped to vectors near each other [19].

GloVe

Proposed by Pennington et al., *GloVe* is a count-based model that uses the statistics of word occurrences within a corpus. It is able to use word-word co-occurrence counts and global corpus statistics to develop its word representations. The model uses a method to find the relationship between two words, i and j, and studies their co-occurrence probabilities with various probe words k. The probability of word j appearing within the context of word i is denoted as P_{ij}, calculated by

$$P_{ij} = P(j|i) = \frac{X_{ij}}{X_i}$$

where X_{ij} denotes how many times j appears in the context of i and $X_i = \sum_k X_{ik}$.

As an example, given $i = ice$ and $j = steam$, the ratio P_{ik}/P_{jk} would be large for a word unrelated to j, such as $k = solid$. For words related to j and

not i, such as $k = gas$, the ratio would be small. Finally, for words either related to both i and j, or to neither, such as $k = water$ or $k = fashion$, the ratios will be close to one.

Using these co-occurrence probabilities, the authors propose a model with a weighting function to filter out rare co-occurrences as noise.

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \widetilde{w}_j + b_i + \widetilde{b}_j - log(Xij))^2$$

Where V is the size of the vocabulary, $w \in \mathbb{R}^d$ are word vectors and $\widetilde{w} \in \mathbb{R}^d$ are separate context vectors. These vectors are equivalent when the co-occurrence matrix X is symmetric, and only differ as a result of their random initializations. b_i and $\widetilde{b}_j$ are biases, with b_i used as a bias for w_i, and $\widetilde{b}_j$ an additional bias for $\widetilde{w}_j$ to restore symmetry. Finally, $f(x)$ is the weighting function parameterized as

$$f(x) = \begin{cases} (x/x_{max})^\alpha & \text{if } x < x_{max} \\ 1 & \text{otherwise .} \end{cases}$$

where x_{max} and α are parameters, set to $x_{max} = 100$ and $\alpha = 3/4$ by Pennington et al.

The model produces a set of word vectors W and $\widetilde{W}$, and the sum of the two, $W + \widetilde{W}$, is the final resulting word vectors for the corpus [19].

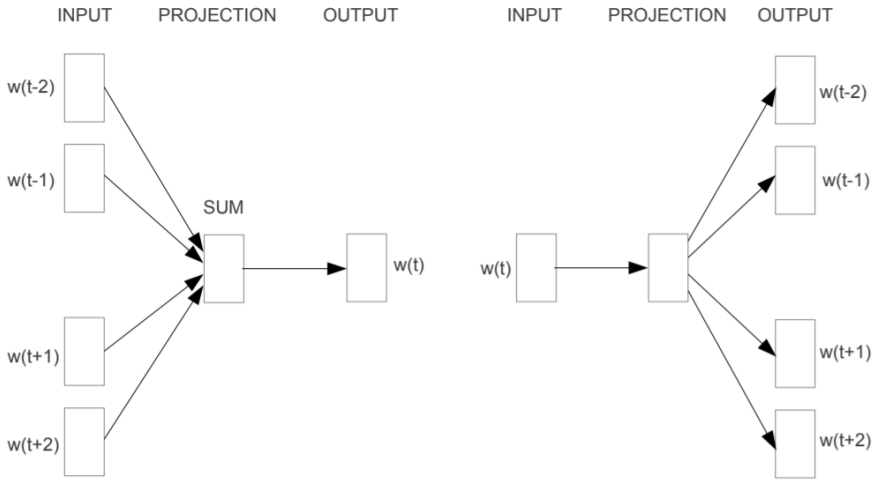
Word2vec

On the other hand, *word2vec* is a predictive-based approach. It attempts to learn its word vectors via the loss of predicting the target word from context words given a word vector representations. Mikolov et al. detail two approaches for this model: continuous bag-of-words (CBOW) and continuous skip-gram.

Continuous bag-of-words: The architecture for CBOW is similar to a feedforward neural net language model with the non-linear hidden layer removed and projection layer is shared for all words. Therefore, all words are projected into the same position by averaging their vectors. In this architecture, the order of words in the history does not influence this projection, and words from the future are obtained. For example, shown in *Figure 6a*, computing the projection for some word w_t takes two words from the history and two words from the future. Therefore, the current word is predicted based on the sum of vectors from its surrounding words within the context window [15].

Continuous skip-gram: The continuous skip-gram architecture is similar to that of CBOW. Instead of predicting the current word based on the context, it attempts to maximize the classification of a word based on another word in the sentence. Each current word is used as an input to a log-linear classifier with a continuous projection layer, and predicts words within a certain range, or window, before and after the current word. In this case, increasing the window size improves the quality of the word vectors while increasing the computation complexity of *word2vec*. Since more distant words are likely to be less relevant

to the current word, they are weighted less than closer words. *Figure 6b* shows computing the four words in the window surrounding some word w_t [15].



(a) Continous Bag-of-Words (b) Continuous Skip-gram

Figure 6: Illustrations of both *word2vec* implementations by Mikolov et al.

Comparison

Baroni et al. provide an extensive comparison of count-based and predictive-based models, resulting in a superior performance for predictive-based methods [2]. The main difference between the two methods listed above is that *GloVe* attempts to capture semantic similarity by capturing the global statistics of how words appear. *Word2vec* uses local windows to capture this similarity, and words only have the same semantics when words in these windows are similar. *GloVe* has the ability to be parrallelized, and therefore can quickly train on a large corpus. It is also able to incorporate global statistical properties of the corpus into its resulting word vectors. However, words that co-occur with other words frequently, such as *water* with *steam* and *ice* may be filtered out as noise [19]. *Word2vec* is able to preserve arithmetic functionality for semantic analogies (such as $king - man + woman = queen$) using the distance between words. Since this model only takes into account relations between words in a pre-defined window, some semantic similarity between words may be lost [15]. In practice, both types of word embedding models may perform better on different datasets [2], and in this paper both types were used to measure their effectiveness on the corpus.

Architecture

The goal of this project was to implement both a sequence-to-sequence network and hierarchical recurrent encoder-decoder and explore the applicability for them to open story generation using naturally occurring story text. These models are trained on the corpus of a novel, a small corpus in comparison to other natural language datasets, and attempt to predict sentences. The sequence-to-sequence network was implemented using *PyTorch*. The HRED model was developed by implementing a context encoder into this existing sequence-to-sequence network.

Beyond the models, various word embeddings were implemented, both pre-trained ones on a Wikipedia dataset, and custom ones trained on the corpus used in this project. The use of various embeddings were important for two reasons: to reduce the loss value of the training model, as well as getting a head start on the accuracy of the initial vector space for word representation.

Explanations of each implementation of these models, as well as the integration of word embeddings, can be found in further detail below.

Sequence-to-Sequence Network

The sequence-to-sequence network was implemented by creating an encoder RNN and an attention decoder RNN. Both RNNs are multi-layer GRUs, implemented by *PyTorch*. GRU was chosen, as opposed to LSTM, for its increased performance when training on smaller datasets. An embedding vector dimension of 300 was chosen to correspond with custom embeddings, and a hidden state dimension of 256.

Data for the model is expected to be in the form of a list of sentence pairs, where each pair contains two sentences in the same order they would be found in within the corpora.

Sentence encoder

The encoder for the sequence-to-sequence network contains the GRU and an Embedding. The GRU was chosen over LSTM for several reasons mentioned previously. It's been shown to be more effective on smaller datasets [4] and is simpler to compute, potentially speeding up training time [3]. It is able to provide these benefits while still being able to capture long-term dependencies. The input for our GRU was the embedding's output $w_i \in \mathbb{R}^{300}$ and the previous hidden state $h_{i-1} \in \mathbb{R}^{256}$. The Embedding is used to translate the input word x_i into a vector with the word's corresponding values within the embedding w_x.

For each word in the input sentence, the GRU is given these parameters and used to encode the sentence. Therefore, for sentence $S_m = \{x_{m,1}, \ldots, x_{m,n}\}$, the hidden state for each word $n = 1, \ldots N_m$ is updated as

$$h_{m,i} = GRU_{enc}(h_{m,i-1}, w_{m,})$$

where $h_0 = 0$ and $w_{m,i}$ denotes the output of the embedding given the input word $x_{m,i}$.

Attention decoder

Like the encoder RNN, the decoder also contains a GRU and Embedding. However, because this is an attention decoder, extra steps must be taken to compute the context vector at each output y_i. First, a set of *attention weights* is calculated with a feedforward layer with the decoder's input and hidden state. These weights are then multiplied by all of the encoder's output vectors to create the weighted combination. After the attention weights are applied, we have the decoder's input for the GRU $w_i \in \mathbb{R}^{300}$ and the previous hidden state h_{i-1}. The decoder's next hidden state is calculated with

$$h_i = GRU_{dec}(h_{i-1}, y_{i-1}, c_i)$$

The GRU also gives the decoder's output, $\widetilde{y}_i$, that is then put through a linear transformation to make $\widetilde{y}_i \in \mathbb{R}^X$ where X is the max sentence length. The final decoder output is then computed with

$$y_i = \log(\text{softmax}(\widetilde{y}_i))$$

Hierarchical Recurrent Encoder-Decoder

As discussed earlier, a hierarchical recurrent encoder-decoder is similar to the sequence-to-sequence network, with the addition of context-awareness. Between the encoding and decoding phases, a context encoding phases is implemented. This was implemented by running the last hidden state of the sentence encoder RNN for each sentence in the paragraph into the context RNN. After the second-to-last sentence of the paragraph is encoded, and its encoded vector passed through the context RNN, the decoding phase begins. The decoder's hidden state is now initialized on the context RNN's output, with the last context RNN hidden state used during each state in the decoder RNN. A GRU was used once again, to keep consistent with both the sequence-to-sequence network and the HRED model described by Sordoni et al. As stated previously, GRUs also provide increased performance when training on smaller datasets. An embedding dimensionilty of 300 was chosen for use with custom embeddings, and a hidden state dimensionality of 256 for consistency with the sequence-to-sequence model.

The data for the model is expected to be in the form of a list of paragraphs, where each paragraph is a list of five sentences. These paragraphs are expected to contain sentences in the same order they would be found in in the corpora.

Sentence encoder

The encoder RNN used for HRED is the exact same as the one describe for a sequence-to-sequence network. It uses a GRU and Embedding and its outputs and hidden state is passed along to the context encoder and decoder.

Context encoder

The context encoder RNN is the main deviation of the sequence-to-sequence network. Since it does not need to transform words into natural language, it makes no use of an Embedding. It is also very similar to the GRU, in that it encodes a sequence of vectors into a single vector of potentially different dimensions. It takes the final hidden state of each sentence encoder: $h_{m,1}, \ldots, h_{m,N_m}$ for paragraph m as its input. The GRU updates each hidden state of the encoder according to: $c_{m,n} = GRU_{ctx}(c_{m-1}, e_m)$ where $e_m = h_{m,N_m}$. The context hidden state at c_m therefore summarizes all input sentences up to point m. For simplicity, the dimensionality of the context states was left at the size of the embeddings, such that $c_m \in \mathbb{R}^{300}$.

Attention decoder

The attention decoder RNN is also similar to the RNN used for the sequence-to-sequence network. In this case, a third feedforward layer is added with its inputs as the previous attention outputs and the most recent context hidden state. The output of this transformation is then given to the GRU along with the decoder's last hidden state, which is initialized as the context encoder's output.

Experiments and Results

Similar experiments were conducted on both architectures in their final states. Models for each architecture, with various word embeddings, that had the lowest training loss values were used for comparison. In doing this, we hoped to find the best architecture and embedding combination, and test the viability of these architectures for storytelling. A dataset from a *Harry Potter* story text was used to create the training and test sets, and was kept consistent for each model in both architectures. Perplexity was used as the evaluation metric to score each of these models, since evaluating the predicted sentence against the target sentence does not provide significant differences. These translation metric experiments will be detailed for the sequence-to-sequence model below. Perplexity scores are calculated in four different ways:

- *Actual sentences*: The perplexity of the model when given the actual target sentence it should predict
- *Random sentences*: The perplexity of the model when given another random sentence from the testing dataset rather than the target sentence
- *Exact random sentence*: The perplexity of the model when given another random sentence that is of the same length as the target sentence
- *Random words*: The perplexity of the model when given a sentence of the same length as the target sentence, but composed of random words taken from the corpus.

Of these four tests, *actual sentences* are the ones we hope to score the best, i.e. have the lowest perplexity scores, and *random words* should score the worst. The reasoning is we need a model that is trained to most likely predict the correct next sentence, and therefore a sentence that makes sense in the current context, rather than a sentence composed of random words or one with no connection to the context. The two random sentence scores are calculated to determine if the model is able to recognize a sentence structure over words taken at random.

For both models, when the predicting of a sentence is done, *beam search* has been implemented. This allows the models to search for the top-k words at each timestep in the hopes of finding a better prediction. The idea behind this is that since the output of the decoder at each time step is used in the next step, a potentially lower overall score could be found by taking a word in the k first words at the previous step, rather than the actual first word. This evaluates the top-k words at a timestep t, and for each word in the next timestep $t+1$, their top-k words, resulting in k^2 scores and partial sentences. Then, the top-k of these k^2 combinations is taken and used in the next timestep $t+2$. Since these sentences could vary in length, and sentences are generated until the *"end-of-sentence"* token is generated, a sentence that has ended is saved unless it drops out of the top-k from a better scoring sentence. The resulting final sentence is potentially different, and if so, a lower score, than if just the top words had been taken at each timestep.

Data and Parsing

To train the models, a corpus of the first *Harry Potter* novel was used: *Harry Potter and the Sorcerer's Stone* [22]. The novel was parsed into a list of sentences, where the order of the sentences in the list is the order they appear within the novel. A sentence was determined as a series of words, potentially over multiple lines in the novel, and were split with various methods. Firstly, they are split on a sentence terminator: a period, exclamation point, or question mark. Sentences were also potentially split on ellipses, or multiple periods, in a naïve method of if the word following the ellipsis began with a capital letter. Lastly, dialogue, or a word or series of words surrounded by single or double quotes, were also split into their own sentences. Along with this sentence splitting, all text was lowercased and all punctuation was removed, except select punctuation such as exclamation points and question marks, kept for their potential semantic usefulness within the sentence. Contractions were also tokenized out, such as *"would've"* to {*would, 've*} or *"Harry's"* to {*Harry, 's*}. In this case, single quotes were kept for their use with pre-trained word embeddings. Various other parsing methods were implemented that specifically relate to the given text, such as removal of chapter titles, "stuttering" dialogue, changing of slang words to their actual words, and others, in an attempt to create a smaller vocabulary and more accurate sentence representation of the novel. The accuracy of these parsing methods was never calculated, but the amount of sentences that may be incorrect or contain unwanted text, such as chapter titles or page numbers,

is negligible with respect to the rest of the data that is correct. These inconsistencies should not have had much of an effect on the models trained with this data.

This list of parsed sentences was then used to generate the train, validation, and test datasets for both models. A total of 7,093 sentences were parsed from the story text, with a vocabulary size of 2,682.

Sequence-to-Sequence data

The sequence-to-sequence data was developed by creating pairs of all the sentences, where each pair is a sentence with the sentence that follows it. For example, a group of sentences $\{s_0, s_1, ..., s_{k-1}, s_k\}$ would create the pairs $\{(s_0, s_1), (s_1, s_2), ..., (s_{k-2}, s_{k-1}), (s_{k-1}, s_k)\}$. Since sequence-to-sequence is a pairwise method, a list of sentence pairs is enough for its dataset. A total of 7,092 sentence pairs were created, being further split into training, validation, and test sets. The training set contained 5,106 pairs, the validation set had 567 pairs, and the test set had 1,419 pairs.

Hierarchical Recurrent Encoder-Decoder data

As for the HRED model, the list of original lines, before the sentence parsing process, were divided into paragraphs, resulting in a list of paragraphs. This was done in a naïve method of how the provided text document was organized: each paragraph was separated by a blank newline. The list of paragraphs $\{p_0, ..., p_k\}$ is now created, where each paragraph $p_i = \{l_0, ..., l_n\}$ is a list of n lines. Each paragraph was then parsed into its actual list of sentences to get the exact list of m sentences: $p_i = \{s_0, ..., s_m\}$. Then, to filter out short paragraphs and single sentences, only paragraphs of five sentences or larger were used. To be consistent with previous work [23] on using an HRED, these paragraphs were then truncated to length five so that every data point had five sentences. Due to this, during the evaluation phase, the fifth sentence is always the one that is predicted. We fixed the paragraph size for two reasons: firstly, having a minimum of five paragraphs ensures that a significant amount of history is provided when predicting each sentence. Secondly, predicting the sentence in the same position of a paragraph keeps a consistent amount of history for each data point. A total of 260 paragraphs were created, being further split into training, validation and test sets. The training set contained 166 paragraphs, with 42 in the validation set, and 52 in the test set.

Word Embeddings

The model learns word embeddings naturally while it trains, but their initial values may help or affect how the model trains or how long the training takes. Three types of word embeddings were used within the project for various experiments. First, is a default random embedding, where the vector for each word is random. Next, a pre-trained *GloVe* 300-dimension embedding, trained on a

2014 Wikipedia dump and Gigaword 5, was used. The corpus for this embedding contained 6 billion total tokens, and a vocab of 400,000 words. Lastly, two custom embeddings were created from the entire *Harry Potter* corpus (all 7 novels) using *word2vec* [16], one with continuous bag-of-words, the other with skip-gram. Both embeddings were 300-dimensions and trained with a window size of 5 tokens. The theory of the two non-random embedding types was to provide a head start with the accuracy of the vector space for word representation. The custom embeddings, in comparison to the *GloVe* embedding, were developed to hopefully have the words' semantic relations be based off how they are used in *Harry Potter*, which may differ to their use in other texts, as well as include *Harry Potter*-specific words in the vector space. The main issue with these custom embeddings, however, is the size of both the corpus and the vocab. In comparison to the *GloVe* embedding's token count and vocab size, the *Harry Potter* corpus that was created has only 2,253,370 tokens. Of those tokens, only 1,220,874 remained after stopwords were filtered, which was the corpus that was used. Along with the token count, its vocabulary size had only 11,834 words. This lack of data, either token count or vocabulary size, most likely hindered this type of embedding.

Model Training

Sequence-to-Sequence Training

To train the model, an input sentence is given to the encoder, with every output and last hidden state stored. The decoder is then given a special *"start-of-sentence"* token as its first input and the previously stored last hidden state of the encoder as its first hidden state, with subsequent encoder outputs given to the decoder as inputs to compute the context vector for attention. Occasionally, for about half the data trained, *teacher forcing* is used. In this case, the real target output is used as input rather than the previous output of the decoder. The loss value for each decoder output is calculated as the negative log likelihood loss between the decoder output, and the target output within the target sequence. The model is trained for a specified amount of epochs, or a complete iteration through the training data, while calculating the loss value at each epoch. Validation values were calculated by creating a validation set, made up of 10% of the training pairs selected at random, and calculating the loss for each validation pair with the model at its current epoch after freezing learning

Hierarchical Recurrent Encoder-Decoder Training

Training of the HRED model is similar to the sequence-to-sequence network. The decoder is given the *"start-of-sequence"* token as its initial input, but now its initial hidden state is initialized with the context RNN's output, rather than the null vector. Teacher forcing is used for about half the dataset, at random. The loss value for each paragraph was calculating by taking the negative log

likelihood loss between the decoder output and the target output only for the final sentence in the paragraph. The models are similarly trained for a specified number of epochs, with 10% of the training data being used for validation to validate loss values after freezing training.

Model Evaluation

Once trained, the models can then be evaluated. This step is similar to the training step, however, after the inputs have been encoded, the top-1 (or if using *beam search*, top-k) prediction is generated by the decoder to form the output sentence, until the *"end-of-sentence"* token is generated, or max sentence length is reached. These predicted sentences can then be used with translation metrics, such as *BLEU* or *METEOR*, to score the prediction compared to the real target sentence.

Translation metrics

Two translation metrics were used in the initial experiments for the sequence-to-sequence network. These metrics, *BLEU* and *METEOR* were used to score predicted sentences against target sentences.

BLEU is a method for automatic machine translation that is language independent and is quick and inexpensive to compute. It is used to compare translations produced by a machine with a professional translation or translations. Its score then provides an assessment of the quality of the machine translation. The baseline of *BLEU* uses n-gram precision to compare the reference and candidate translations. It counts up the number of matching n-grams between the translations, with more matches meaning a better candidate translation. *BLEU* then modifies this idea by truncating the total amount of matches of a specific n-gram by its total amount of occurrences of this n-gram in the reference text. A summation of all n-gram matches from the candidate translation, truncated by the amount found in the reference, and divided by the total count in the candidate provides the score for a single translation. The equation for this is as follows

$$\text{BLEU} = \frac{\sum_{n\text{-gram} \in C} Count_{clip}(n\text{-gram})}{\sum_{n\text{-gram}' \in C'} Count(n\text{-gram}')}$$

where C and C' are the candidate and reference translations, respectively. *BLEU*'s resulting score will be between 0 and 1, with 0 meaning no match, and 1 being an exact match [18]. In this project, we use an implementation provided by *NLTK* [11].

METEOR another method for automatic translation evaluation, and is shown to significantly outperform the *BLEU* metric. One improvement, noted by Lavie and Agarwal, is that BLEU does not provide reliable sentence-level scores, as it was originally designed to compare the translations of an entire corpus. *METEOR* was designed specifically for evaluation of translation at the segment level. The score it computes is this metric is based on explicit word-to-word

matches between a candidate and reference translation. This is done by creating a *word alignment* between the two strings. This *alignment* is a mapping between words in the strings such that every word in each string maps to at most one word in the other string. Initially, all possible word matches is computed, and the largest subset of these matches is used as the *word alignment*. If more than one subset are of this largest value, the metric selects the one who's word order is most similar as the *alignment*. This is done by picking the mapping that has the least amount of intersections between words. With this final *word alignment*, the score can be computed. First, the harmonic mean, F_{mean}, between the unigram precision P and the unigram recall R must be computed

$$F_{mean} = \frac{P \cdot R}{\alpha \cdot P + (1 - \alpha) \cdot R}$$

where the unigram precision is calculated as $P = m/t$ and the unigram recall is calculated as $R = m/r$. In these equations, m is the number of mapped unigrams found between the two strings, t is the total number of unigrams in the candidate translation, and r is the total number of unigrams in the reference translation. These equations are all based on single-word matches. In order to take into account the order of matched unigrams are between the two strings, a penalty Pen is computed. The sequence of matched unigrams between the strings is divided into the fewest number of *chunks* where the matched unigrams in each *chunk* are adjacent and in identical order in both strings. The number of chunks ch and number of matches m are used to calculate the fragmentation fraction $frag = ch/m$. The penalty is comptued as

$$Pen = \gamma \cdot frag^{\beta}$$

γ and β are parameters, where $(0 <= \gamma <= 1)$ determines the maximum penalty and β determines the functional relation between fragmentation and the penalty. Finally, the *METEOR* score can be computed as

$$\text{METEOR} = (1 - Pen) \cdot F_{mean}$$

[7]

The *METEOR* metric for this project was provided by a custom *Python* implementation[3] based on the algorithm described by Banerjee and Lavie [7].

Perplexity

The main evaluation method used in these experiments is perplexity. The perplexity of a probability model is a metric used to evaluate trained models. In a sense, it is used to determine how "perplexed" a model is with target output after given its associated input. A lower perplexity will indicate that the model performs well at predicting the target output. It is calculated with the equation

$$perplexity = b^{-\frac{1}{N}\sum_{i=1}^{N} \log_b p(x_i)}$$

[3] github.com/zembrodt/pymeteor

Where p is a proposed probability model, and $(x_1, \ldots, x_N)$ is the test sample. Specifically for these models, the perplexity is calculated by forcing the models to output the target sentence after given the input sentence(s). A value of $b = 2$ was used as a portion of this equation, specifically because $\log_2 p(x_i)$ was already calculated by *PyTorch*.

Sequence-to-Sequence Results

The experiments conducted for the sequence-to-sequence model can be broken down into several sections as various concepts were implemented or tested: baseline, beam search, and custom embeddings. Since these were implemented in various stages of the project, the code used to parse the natural language in the novel may not be consistent through all experiments, but the results from each experiment should not be greatly affected by this. The following sections will discuss the process of experimenting with this implementation and its results. In the final section, custom embeddings, the end results of the sequence-to-sequence model will be provided and discussed. These results will be what is used to compare against the HRED model.

Baseline implementation

The baseline implementation is the bare-bones implementation of the sequence-to-sequence network. Beam search had not yet been implemented, only evaluating with the top-1 result, and random embeddings were used. Unfortunately, at this point in time, perplexity was not being calculated correctly, and would not be fixed until beam search was implemented. At this point, however, we have results from the translation studies.

Epochs	*BLEU* score	*METEOR* score
10	0.0	0.0157
40	0.0	0.0389

Table 1: Baseline implementation *BLEU* and *METEOR* scores

The initial results in *Table 1* had poor analysis scores, both at 0 or near 0. For both *BLEU* and *METEOR*, scores range from 0 to 1, with 1 meaning an exact match. However, the sentences they predicted were promising at the time.

Beam search

At this point, the evaluation of the trained model was updated to include beam searching of the top-k results, rather than the default top-1. The implementation of this was to check if the best scoring output sentence was perhaps not found by taking the top-1 output at each stage of the decoder, but somewhere else. This model was once again trained for 40 epochs.

Epochs	*BLEU* score	*METEOR* score
40	0.0	0.0346

Table 2: Beam search *BLEU* and *METEOR* scores

This model showed no real improvement with the translation metric scores, as shown in *Table 2*, but the first perplexity study was also done during this stage.

Epochs	*Actual sentences*	*Random words*
25	**85241.0183**	306539.759

Table 3: Perplexity scores for *"Actual sentences"* and *"Random words"*

While the perplexity scores in *Table 3* were not very good, particularly due to the small amount of epochs the model was trained for, the *actual sentences* score being fairly smaller than the *random words* score was promising, meaning the model had learned the structure of a sentence as more than a random assortment of words.

Custom word embeddings

The bulk of experiments were conducted with custom word embeddings. At this point in the project, models started to be trained for a larger amount of epochs, ranging from 100 to 500, and the under-fitting of the model to the data became apparent. The value of loss while training was also calculated and recorded for comparison and analysis of each model. It was also determined that translation metrics such as *BLEU* and *METEOR* may not be ideal to measure the effectiveness of the model, at least in its current state, and were no longer calculated.

Figure 7 shows an initial training of the model using the *GloVe* word embedding discussed previously, trained to 100 epochs. While its final loss value of around 6.72 is not an ideal value, this model seemed to still be learning, and the loss decreasing, when it finished training at 100 epochs. Nevertheless, it was decided to train two custom word embeddings with *word2vec* using the entire *Harry Potter* corpus, as discussed previously. *Figures 8a* and *8b* show results for both skip-gram (SG) and continuous bag of words (CBOW) implementations.

At this point, the SG implementation seemed the most promising, having finished training for 100 epochs with a loss value of 4.807. It was decided then to do a full study on all word embedding implementations for a full comparison between them, and for a larger amount of epochs.

Figure 9 shows a comparison between random (called *none*), *GloVe*, *word2vec* with SG, and *word2vec* with CBOW embeddings. The result shows that out of all the models tested, only *GloVe* had any improvement, with a decreasing loss value after 500 epochs. However, it became apparent that the project developed

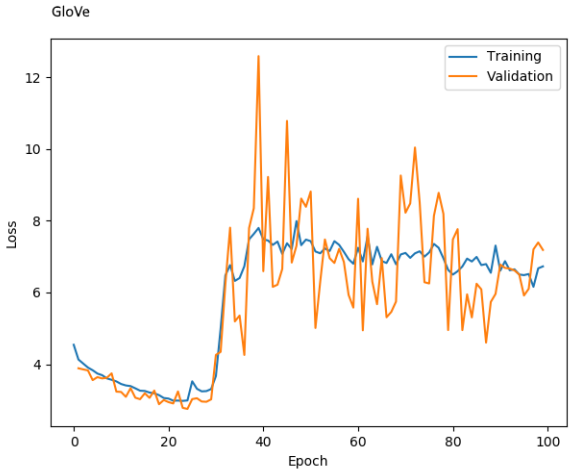


Figure 7: Initial training with *GloVe*

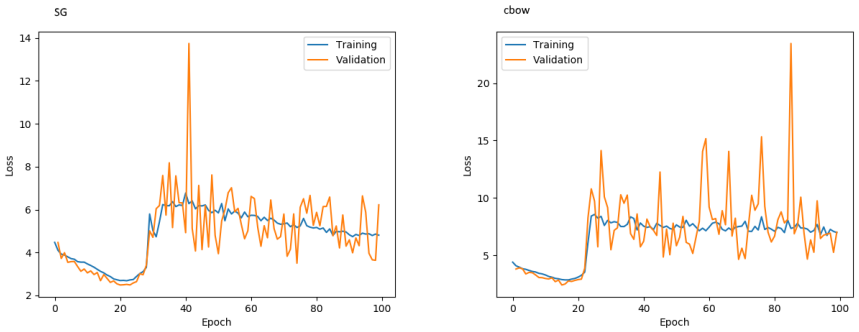
was incorrectly saving and loading models, causing a large spike in the loss value when a checkpoint was loaded (see emphasis of *GloVe* value in *Figure 9*).

After conducting this comparison, however, it was determined that the custom *word2vec* embeddings should be trained on the *Harry Potter* corpus for longer. At this point, both were trained for 15 epochs over the corpus, taking a little over one minute. CBOW also seemed to be the better implementation to move forward with, rather than SG.

Figure 10 shows a comparison of two models trained for 500 epochs. One with the 15-epoch CBOW embedding, the other with a 300-epoch CBOW embedding. This result was once again corrupted by the loss spiking of loading a checkpoint (emphasized) for the 300-epoch CBOW embedding. However, before the checkpoint, the 300-epoch CBOW embedding was performing better, so it could be argued that training the word embedding for a larger amount of epochs positively affected the model's loss value while training.

Now it was determined to do a comparison of the custom CBOW embedding with the *GloVe* embedding, for 500 epochs, without any checkpoint loading, as this issue still hadn't been corrected.

Figures 11a and *11b* compare these two embeddings with one another. Training with the CBOW embedding was not beneficial for the model's training process, and overall was not decreasing the loss value. *GloVe*, however, performed well. Without having to checkpoint, the loss value never spiked (after the initial spike), reached a minimum value of 2.983, and a final value of 3.083 at the 500th epoch. This was a much better epoch score than the 4.0 to 7.0 and greater values calculated earlier. Perplexity studies were once again conducted on this model, giving even more interesting results.



(a) Initial training with *word2vec* SG (b) Initial training with *word2vec* CBOW

Figure 8: Comparing implementations of *word2vec* embeddings

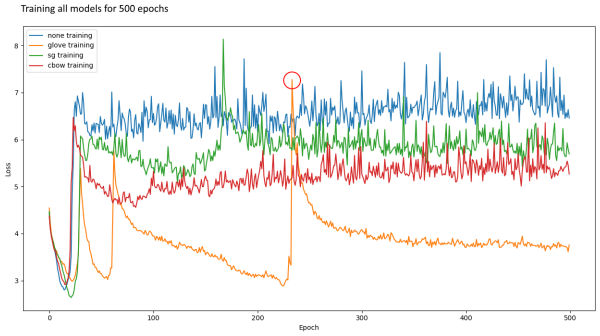


Figure 9: Comparing all embeddings

Perplexity results

In *Table 4*, a further perplexity study was developed for *actual sentences*, *random sentences*, and *random words*. Out of the three categories of studies, *actual sentences* is the one that should ideally perform the best, as they are the real sentences to follow the input sentences. However, this was not the case. As shown in *Table 4*, the *random sentences* study actually performed best. This is not ideal, but the scores for *actual sentences* are relatively close, within the same decimal place. On the other hand, both scores are much lower than those of *random words*, which shows that the model has learned the formation of sentences as more than just random words. The model trained for 500 epochs, however, gave higher perplexity scores than the model trained for only 250 epochs. This could be as a result of the model overfitting more on the training data, and therefore performs worse on data it has not seen. The perplexity scores of both models on the training data seems to confirm this, with the value on *actual sentences* being reduced from 42.989 to 37.041 for the 500 epochs

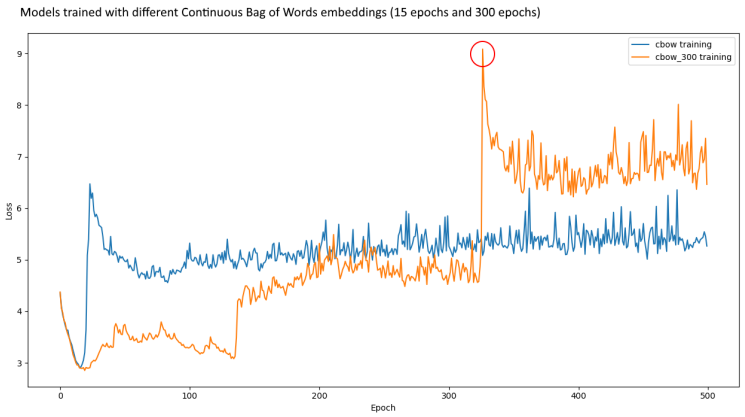
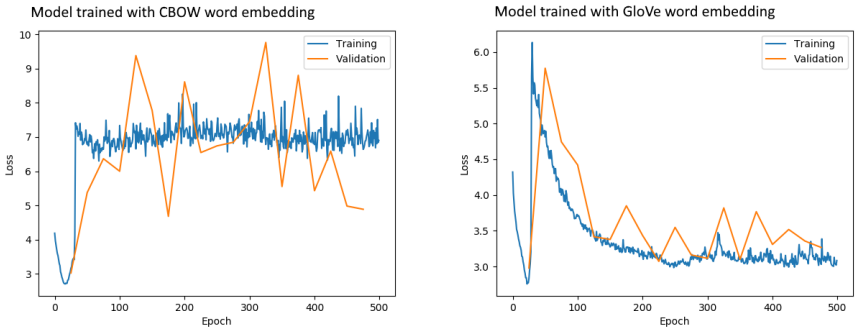


Figure 10: Comparing different *word2vec* CBOW embeddings



(a) Initial training with CBOW

(b) Initial training with *GloVe*

Figure 11: Comparing models trained with CBOW and *GloVe*

model. Finally, the actual perplexity themselves were extremely high for the testing data, further showing that perhaps the model hasn't learned enough to handle unseen data. One cause of this could be due to the lack for training data, with just over 5,000 sentence pairs.

Hierarchical Recurrent Encoder-Decoder

The experiments for the updated HRED model were conducted in a similar manner to the custom word embedding experiments of the sequence-to-sequence network. The model was trained using three different embeddings: random, *GloVe*, and the custom *word2vec* CBOW embeddings. All models were trained for 1000 epochs with their loss values compared. However, for this model two different optimizers were compared: Stochastic gradient descent (SGD) and Adam. The

Epochs		Perplexity
250	*Actual sentences*	736176.1818
	Random sentences	**299692.5997**
	Random words	147674868.5
500	*Actual sentences*	4872462.827
	Random sentences	**2313469.597**
	Random words	6201033178

Table 4: Perplexity scores for model in *Figure 11b*

results of these experiments showed that the SGD optimizer provided better training loss values with congruent validation loss values, while the Adam optimizer's validation loss values dramatically increased and deviated from the training loss values. This is good for consistency as the sequence-to-sequence models all used SGD for their optimizer. Comparisons were also done for the models using two different values for the learning rate: 0.01 and 0.0001. In this case, 0.0001 was found to provide better results, which is also consistent with the value used for the sequence-to-sequence models.

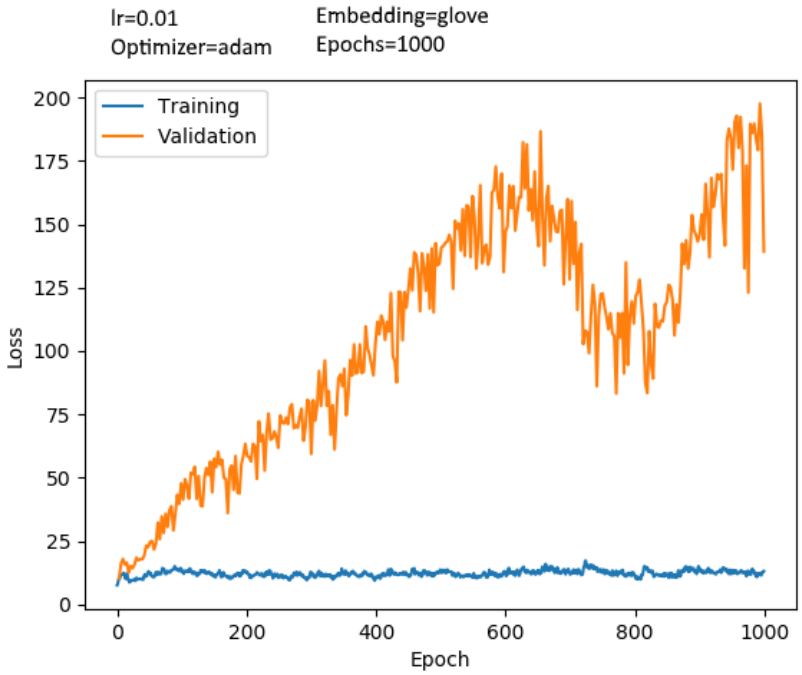


Figure 12: Model trained with Adam optmizer and *GloVe* embedding

Figure 12 shows the initial loss values of an example model using the Adam optimization algorithm and a learning rate of 0.01. As shown, the validation loss

value dramatically diverges away from the training loss value, with no level-off after 1000 epochs. The cause of this was unknown, but shows that the model does not perform well when working with data it has not seen. Models trained with the random and custom CBOW embeddings also resulted in similar loss patterns.

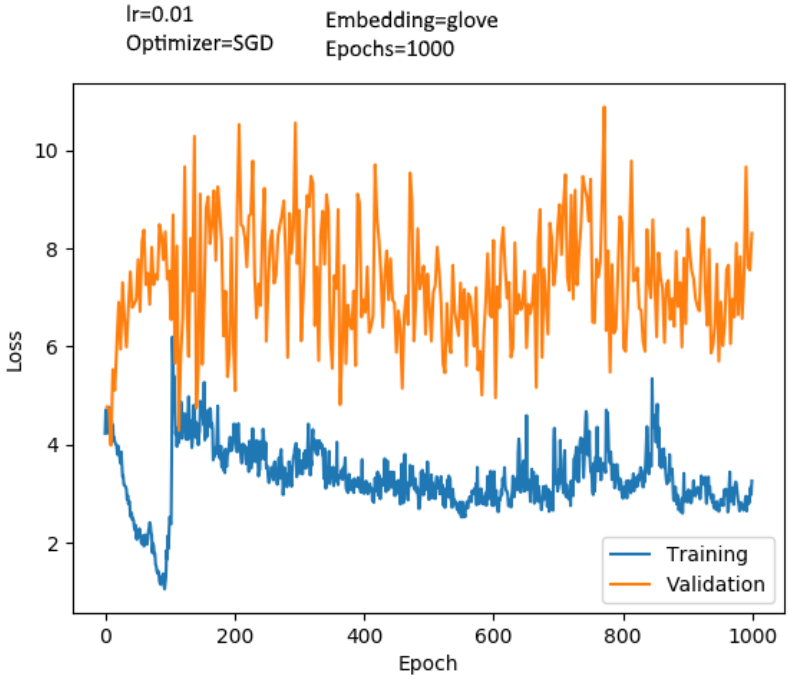
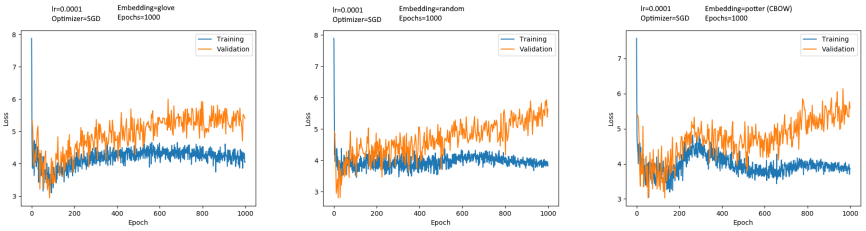


Figure 13: Model trained with SGD optimizer and *GloVe* embedding

Figure 13 shows an example model trained with the SGD optimizer, and also with the *GloVe* embedding and a learning rate of 0.01. These validation loss values are more congruent with the training ones, and follow a similar pattern to the loss results from the sequence-to-sequence models. Results from the random and custom CBOW embeddings provided similar loss patterns. However, since the learning rate for this model used is still 0.01, the model could still be improved.

Figures 14a, 14b, and 14c are the final models trained. They used the SGD optimizer with a learning rate of 0.0001 and were trained for 1000 epochs. With the smaller learning rate, the loss values for each model improved with respect to the previous values from the larger learning rate of 0.01. The values also are much similar to those of the sequence-to-sequence model. The models trained with the *GloVe* embedding had a final loss value of 4.07 for the HRED model, compared to sequence-to-sequence's value of 3.083. For the custom CBOW embedding, the final loss value was halved with the HRED model.



(a) Model trained with a random embedding

(b) Model trained with the *GloVe* embedding

(c) Model trained with the custom CBOW embedding

Figure 14: Models trained with all embeddings and the SGD optimizer

Embedding		Perplexity
Random	*Actual sentences*	**2826.049**
	Random sentences	3125.4459
	Exact random sentences	**2812.2747**
	Random words	23585.7831
GloVe	*Actual sentences*	**1174.7250**
	Random sentences	**1114.8894**
	Exact random sentences	1033.2969
	Random words	18572.4679
Potter	*Actual sentences*	1592.9276
	Random sentences	1447.1607
	Exact random sentences	**1361.7506**
	Random words	24280.3667

Table 5: Perplexity scores for models in *Figure 14*

Perplexity results

A similar perplexity study to the one done for the sequence-to-sequence models was done for these three HRED models as well. This study compared perplexity values for each model in four parts: *actual sentences*, *random sentences*, *exact random sentences*, and *random words*. To find the perplexity score for each of these, the model was given the first four sentences of a paragraph, and then a sentence corresponding with each test, as described previously. The perplexity score was calculated for this sentence and recorded. The sentences were all created from the testing data to see its performance with data it has not seen before. The results of the testing data is also used to compare with the testing data results from the sequence-to-sequence models to see if the addition of context has made a significant improvement of the model.

The perplexity study was run 100 times for all three embeddings, and the average perplexity scores taken.

Table 5 shows the results of this study. It was found that the perplexity scores of *actual sentences*, *random sentences*, and *exact random sentences* significantly

outperform the perplexity scores of *random words*. For each embedding, the scores for the *actual* and *random sentence* in the testing data is very similar, such as *GloVe* having 1174.725 and 1114.8894 respectively. For this embedding, the random words score was extremely high at 18572.4679. This is because, in both cases, the model was able to similarly recognize that the sentence it was given was a properly constructed sentence with respect to the training data. In all three embeddings, the values for *exact random sentences* scored the best. This potentially shows that the length of the target sentence had some affect on the perplexity value, as all three scored better than random sentences of varying length. Besides *random words*, *actual sentences* scored the worst in both custom embeddings. This could be due to the fact that some sentences taken at random from the corpus will potentially score better than the single *actual* target sentence. If this is the case, and over half the sentences score better than the target, then on average both *random sentences* and *exact random sentences* will score better than the *actual sentence*. However, the perplexity scores for all three are on relatively close, and much smaller than *random words*, proving the model has learned the structure of a sentence with data it has not seen before. Finally, once again, as shown by the sequence-to-sequence model's results, the *GloVe* embedding created models that scored the best, with the custom *word2vec* CBOW embedding scoring close behind it.

Model	Epochs		Perplexity
Seq2Seq	500	*Actual sentences*	4872462.827
		Random sentences	2313469.597
		Random words	6201033178
HRED	1000	*Actual sentences*	1174.725
		Random sentences	**1114.8894**
		Random words	18572.4679

Table 6: Perplexity scores of the sequence-to-sequence model in *Figure 11b* and the HRED model in *Figure 14b*, both using *GloVe* embeddings

The HRED model follows a similar trend to the perplexity scores of the sequence-to-sequence model, as shown in *Table 6*. However, even with the HRED model being trained for twice as long, this comparison is significant. The testing scores for the sequence-to-sequence model were all extremely large, and proved the model could not recognize data it has not seen before. The HRED model, on the other hand, proves that the model has begun being able to recognize sentences, and is able to handle data it hasn't seen yet. Even though *random sentences* still performed better than *actual sentences* in the HRED model, the difference compared to the sequence-to-sequence model is dramatic. Whereas before, the value for *actual sentences* was increased by a factor of two, now there was only about a 5% increase. The model, however, is still not able to accurately use the context from the previous four sentences to more accurately predict the correct target sentence versus any sentence at random.

Drawbacks

The project currently has several drawbacks. For one, due to training on such a small corpus, the models also have struggle training and not overfitting on the data. The dataset used for the sequence-to-sequence model contained over 5,000 pairs, while the HRED model's dataset had just 166 paragraphs. Due to this, both models did not have an ideal training loss value or significantly low perplexity scores. The models also incorrectly load checkpoints, and it was not determined if this was user-error or an error with *PyTorch* itself. Finally, the final sentences predicted by the model are in many cases still incoherent and not ideal to generate a story yet. This problem could still be attributed to the lack of data, or potentially for not training for enough epochs.

Future work

Beyond correcting the drawbacks listed above, future work could include training and testing a working model on a larger corpus of stories, such as an entire author's work, thousands of novels of a similar genre, poems, etc. The HRED model in this project was only trained and evaluating using paragraphs of length five. In the future, varying sized paragraphs could be trained with. Another change for this model would be varying the scope of the context. In this project the scope was kept to paragraphs, but this could be increased to a larger scope such as chapters or acts. Training more custom embeddings could also prove beneficial, either training them for much longer than the experiments done for this project, or using *GloVe* to train custom word embeddings rather than *word2vec*.

Conclusion

Telling stories is an important aspect of how humans communicate with one another. Since machines and AI are becoming more and more integrated in our society, it has become increasingly important for them to not only understand these stories, but also generate them. Recurrent neural nets are a method currently being explored on how to train models to generate these stories, and was the focus of this project. Both sequence-to-sequence and hierarchical recurrent encoder-decoder models were trained in this project to compare their effectiveness with one another, and their overall effectiveness at the end-goal of generating stories themselves. The first *Harry Potter* novel was used as their corpus in an attempt for the models to be able to generate sentences within a similar manner to the ones found within this book. A large portion of the project, however, became a study of various word embeddings, and how they affect the training of the models. The results show that even with such a small corpus, these two implemented models are able to train and learn sentence structure based on the training data. The implementation of pre-trained word embeddings also positively affect the metrics of how to evaluate these models, and increased their performance. Finally, the inclusion of a context encoder within

the HRED model shows the significance of a paragraph's context along its individual sentences, greatly decreasing the perplexity of the model when given testing data in comparison to the sequence-to-sequence network only training on sentence pairs.

Before undertaking this project I had a brief knowledge of neural networks, but had never been exposed to these different architectures. I have learned a great amount on how these techniques work, and how their equations are translated into code implementation. This project was also exclusively written in *Python*, a language I did not have much practical experience with beforehand. After implementing these models and writing code for various text pre-processing techniques, I believe my knowledge of the language and many different third-party libraries to have been greatly increased. I am grateful for the opportunity to work on this project, and believe that many of the things I have learned on it can be applied to my future work in the field.

References

[1] Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate, 2014.

[2] Baroni, M., Dinu, G., and Kruszewski, G. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Baltimore, Maryland, June 2014), Association for Computational Linguistics, pp. 238–247.

[3] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder–decoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2014).

[4] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[5] Greff, K., Srivastava, R. K., Koutnik, J., Steunebrink, B. R., and Schmidhuber, J. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems 28*, 10 (Oct 2017), 2222–2232.

[6] Harrison, B., Purdy, C., and Riedl, M. O. Toward automated story generation with markov chain monte carlo methods and deep neural networks.

[7] Lavie, A., and Agarwal, A. Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments. In *Proceedings of the Second Workshop on Statistical Machine Translation* (Stroudsburg, PA, USA, 2007), StatMT '07, Association for Computational Linguistics, pp. 228–231.

[8] Lebowitz, M. Story-telling as planning and learning. *Poetics 14* (12 1985), 483–502.

[9] Li, B., and Riedl, M. O. Scheherazade: Crowd-powered interactive narrative generation. In *AAAI* (2015).

[10] Lipton, Z. C., Berkowitz, J., and Elkan, C. A critical review of recurrent neural networks for sequence learning, 2015.

[11] Loper, E., and Bird, S. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1* (Stroudsburg, PA, USA, 2002), ETMTNLP '02, Association for Computational Linguistics, pp. 63–70.

[12] Martin, L. J., Ammanabrolu, P., Wang, X., Hancock, W., Singh, S., Harrison, B., and Riedl, M. O. Event representations for automated story generation with deep neural nets, 2017.

[13] Mateas, M., and Stern, A. Procedural authorship: A case-study of the interactive drama facade. In *In Digital Arts and Culture (DAC* (2005).

[14] Meehan, J. R. Tale-spin, an interactive program that writes stories. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 1* (San Francisco, CA, USA, 1977), IJCAI'77, Morgan Kaufmann Publishers Inc., pp. 91–98.

[15] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space, 2013.

[16] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119.

[17] Olah, C. Understanding lstm networks. *Github blog* (August 2015). http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[18] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Stroudsburg, PA, USA, 2002), ACL '02, Association for Computational Linguistics, pp. 311–318.

[19] Pennington, J., Socher, R., and Manning, C. D. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)* (2014), pp. 1532–1543.

[20] Pichotta, K., and Mooney, R. J. Using sentence-level lstm language models for script inference. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2016).

[21] Riedl, M. O., and Young, R. M. Narrative planning: Balancing plot and character. *Journal of Artificial Intelligence Research 39* (Sep 2010), 217–268.

[22] Rowling, J. K. *Harry Potter And the Sorcerer's Stone*. Arthur A. Levine Books, 1998.

[23] Serban, I. V., Sordoni, A., Bengio, Y., Courville, A., and Pineau, J. Building end-to-end dialogue systems using generative hierarchical neural network models, 2015.

[24] Sordoni, A., Bengio, Y., Vahabi, H., Lioma, C., Grue Simonsen, J., and Nie, J.-Y. A hierarchical recurrent encoder-decoder for generative context-aware query suggestion. *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management - CIKM '15* (2015).

[25] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 3104–3112.

[26] Tomaszewski, Z., and Binsted, K. Demeter: An implementation of the marlinspike interactive drama system. pp. 133–.