Project Document: Artificial Castle

Vega Carlson
Stephanie Marsh
Daniel Shchur
Conner Elliott

May 1, 2021

Contents

Abstract

The ideas presented in this paper include making a machine learning system for reconstructing filter audio clips of speech, analyzing audio to determine the spoken language, generating new artifacts for the game Dungeons and Dragons, and training a system to play Bomberman. The ideas came from thinking about how machine learning could be applied to our personal interests followed by brain storming about related topics. Each idea includes a description and some possible applications, as well as some notes on possible designs and architectures that would be necessary. The selected project was to generate original names for fantasy items, using World of Warcraft and Everquest datasets. A combined and cleaned dataset was compiled, and a working prototype was developed. The prototype was improved by altering the method for seeding the model, as well as introducing a method to stop item name generation at sensible points which analyzes parts of speech and word length of generated outputs. We implemented Damerau-Levenshtein similarity for a quantitative method for evaluating our model. We also did blind qualitative testing for the different sets of hyperparameters on a scale from 1 to 5. We determined the best set of hyperparameters to be 1024 RNN units and a temperature of 1.0, as it had a minimized similarity metric, and a maximized qualitative score, making it well-liked unique item generation.

Chapter 1

Milestone 1: Project Ideas

1.1 Introduction

Our team spent a lot of time brainstorming various project ideas. We talked about the types of data we were interested in, and then deliberated a multitude of potential projects that involved our interests. Initially, we were solely interested in exploring audio related projects, though we branched out with our last two proposed projects. After much deliberating, we settled on our top 4 projects that we wanted to explore.

Our first project explores rebuilding audio after it has been filtered. Our second project involves detecting the spoken language from an audio file. Our third project looks at generating new magical items for Dungeons and Dragons 5th edition. Finally, our last project explores training artificial intelligence to play the classic NES game Bomberman.

1.2 Project Idea 1: Audio Reconstitution After Filtering

Audio is fundamentally defined by frequency, amplitude, and phase. Those are the three things that define a sound for what it is. Unfortunately, there are many reasons, both digital and physical, that all three of these may be changed undesirably. The easiest example is of filtering. As a physical example, a person talking though a wall may be hard to hear as the wall will naturally attenuate the higher frequencies, muffling the highs. Digitally, a recording at a low sample rate will only be able to reproduce frequencies up to half of that frequency because of the Nyquist theorem. For example a recording with a sample rate of 16khz will only be able to reproduce frequencies up to 8khz- far below the 20khz ceiling of human hearing. In most low-bitrate recordings, sound data has

a low pass filter applied to avoid aliasing as well.

We would like to explore the possibility of using a machine learning system to 'un-filter' recordings of people speaking English though a low-pass filter. This would be useful as speech that lacks higher frequencies tends to be harder to harder to understand (This is rather obvious- just try to speak to somebody though a closed door, a natural low pass filter) and quite literally lacks detail, as can be seen in Figure 1.1 below where a low-pass filter is applied to an image.

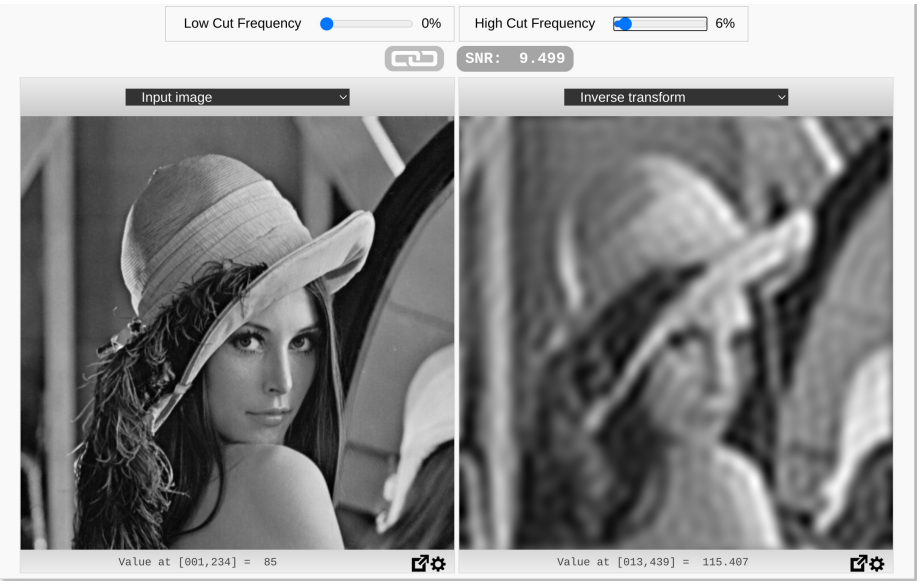

Figure 1.1: An image, when processed through a low pass filter, appears blurry (source: [19]).

By un-filtering audio it may be possible to restore old vocal recordings (such as recordings stored on tape or other old media), pre-process audio before being sent to a speech-to-text engine, or embed microphones behind protective plastic in phones while still getting usable audio.

In Figure 1.2 the unaltered sound as well as the sound after being passed though three different low pass filters are shown, and just like the image in figure 1, it can be seen that the high frequency content contains a lot of information.

Fortunately, English speech has a fairly limited consonant inventory, as can be seen in Table 1.1 which should limit the number of features whatever machine learning architecture is used would have to learn to identify via after being low pass filtered and subsequently reproduce.

Further, finding training data should be relatively easy, as any high quality audio training set normally meant for text to speech can be used simply by dropping the text labels, using the original file as the label, and using a tool to apply

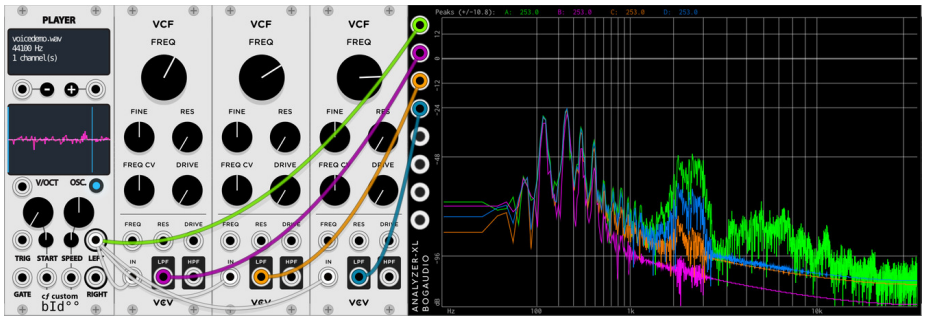

Figure 1.2: Frequency Analysis of a vocal sample going through filters at various cut off frequencies. The signal in Lime is unfiltered, Purple has an f_c of 500hz, while orange and blue have an f_c of 1khz and 2khz respectively.

Table 1.1: Table of English consonant phonemes (source: [25]).

		Labial	Dental	Alveolar	Post-alveolar	Palatal	Velar	Glottal
Nasal		m[a]		n[a]			ŋ	
Plosive/ affricate	**fortis**	p		t	tʃ		k	
	lenis	b		d	dʒ		g	
Fricative	**fortis**	f	θ[b]	s	ʃ		x[c]	h
	lenis	v	ð[b]	z	ʒ			
Approximant				l[a]	r[d]	j[e]	w[f]	

digital low pass filters to the data in bulk. These filters could have their cut off frequencies set semi-randomly to avoid training a model that can only handle one cut off frequency. By putting the filtered audio into a machine learning model, likely a recurrent neural network or Generative Adversarial Network (GAN), with the original audio as a label, the model should progressively get better at outputting audio that has the high frequency data added back in.

While it may be trivial to expand the scope to other filters by applying more filter types (high pass, band pass, comb, filters with resonance, etc.) for simplicity this project would focus on a single filter type (low pass), slope, and resonance, with only the cutoff frequency, f_c, being changed.

1.3 Project Idea 2: Language Recognition using Audio Analysis

Automatic Speech Recognition (ASR) is a widely studied and necessary field in computer science and linguistics due to its use in dictation, translation, virtual assistants, and more. While there are many in depth studies and projects in this field, one problem we could attempt to tackle would be the recognition and categorization of voice data by language, recognizing what language a person is speaking automatically. This wouldn't necessarily include translation or dictation of the speech, simply the recognition of the spoken language.

There are many methods to perform ASR, many of which make use of Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). One method, the Connectionist Temporal Classification (CTC) model uses a sequence of data frames with specific lengths which analyze the audio through spectrograms through an RNN [2] [9]. This data is then fed into a Softmax probability function over vocabulary to predict which vowels or phonetics are being said in that segment of sound. Because the network is recurrent, predictions can be fed back to get an estimation of groups of letters and sounds with different variances. Any errors in recognition as are often the case with the CTC model can be corrected by integrating another language model for correcting of misspellings. This can be a simply dynamic programming algorithm which is often used in auto-correct, or something more complex like another neural network trained to predict with context the right words based on the mistakes and misspellings [20].

Another model which is fairly popular is a Sequence-to-sequence (seq-2-seq) model which makes use use of next-step predictions. The network looks at the whole sample to make a prediction instead of a chunk of data. One such model, the Listen, Attend, Spell (LAS) model, captures individual time-steps of data and encodes them into a pyramidal structure represented by vectors which collapse time-steps together [5]. This model is similarly structures to an RNN and but is makes use of Long Short-Term Memory (LSTM), a visualization of the network can be seen in Figure 1.3. One major downside to such a model would be the lack of online use, or ability to use it in the real world. For that, there are variations that exist which make use of CNNs instead breaking apart the audio and analyzing it in chunks, similar to how object recognition would work on images [12].

For our purposes, we could try to make use of either model. We would need to add to them to adding a classification aspect to them. The most naive solution would be to utilize a cascade model. Multiple networks would be trained on recognizing individual languages and then their probabilities on a certain input would be taken to see which one presents the best match. However, this would be very resource intensive on the size of the training set, computational necessity to train all the models, and applicational use when deploying in the real world. A better approach would be to try and make an End-to-end model which

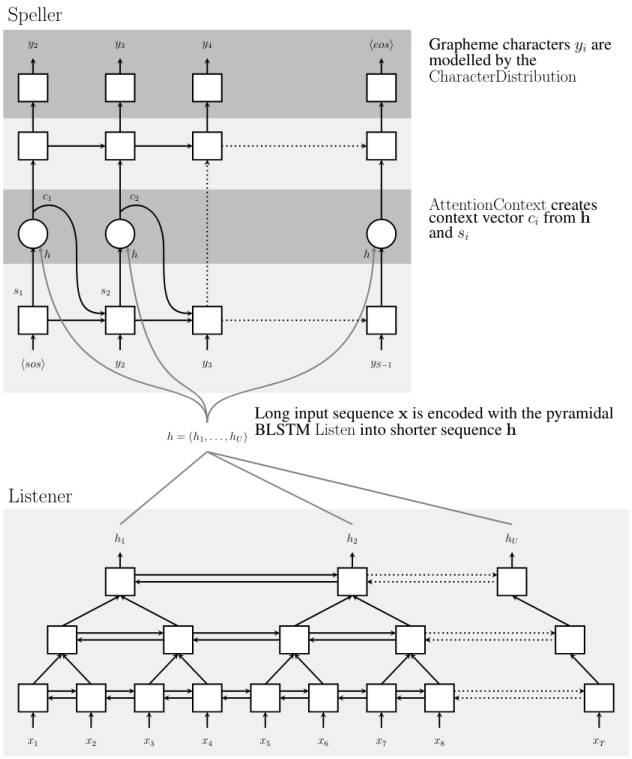

Figure 1.3: Visualization of the LSTM LAS network which collapses time-steps into larger frames to do prediction. [5]

encompasses all the languages trained in one go. This could be done by incorporating multiple languages into the recognition model and simply preforming the recognition on them. If one language was recognized more than the other, a classification could be made as to what language it is. This is likely to be less accurate than a cascading model however, especially for languages which share many similarities such as Spanish and Portuguese. Such an exploration was recently preformed utilizing both of these methods and seq-2-seq models [21]. We could try and reproduce the work and improve on it, utilizing different models to try and accomplish the task, such as CTC, or different languages as well.

1.4 Project Idea 3: D&D Artifact Generation

Dungeons and Dragons 5^{th} edition (D&D 5e) is an extremely popular tabletop role playing game. In this game, players go on adventures, fight all sorts of villains, and can find magical items with various properties. We are interested

in using deep learning to generate magical items for D&D 5e. A sample item can be seen in Figure 1.4. This project idea is inspired from the project ideas page listed on Canvas. We believe that the applications are increasing variety in game play. This project will be useful for long-term fans of D&D who are wanting to add some variety to their campaign. Role playing games often have limited magical artifacts with unimaginative item names. A potential road block that we may face, is in fact the limited nature of magical artifacts: there are only 268 artifacts found in D & D 5e, though this does not account for the additional content [15]. The generation of new item names can breathe new life into a D&D campaign, and make some lackluster objects appear to be more exciting. Likewise, if we can generate entirely new items, then the game master will be able to surprise players. Our goal is to be able to have strange, unlikely magical items that players can stumble upon at a Game Master's discretion.

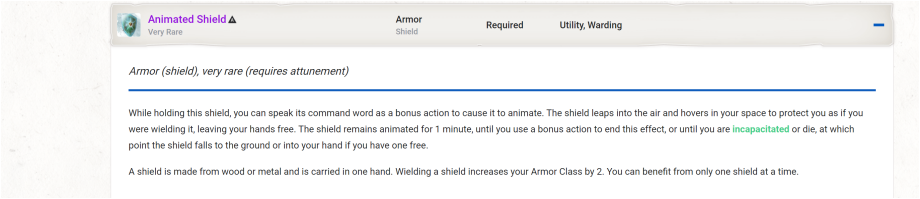

Figure 1.4: An item from D&D 5e with the 6 components listed that we would like to be able to generate. [15]

In D&D 5e, there are 6 components to a magical item: the name, type, item rarity, weight, attunement, and description. The name is what we call the item. The item type is the general physical category of item (e.g., adventuring gear). The rarity is how easy it is to acquire this artifact as well as the in-game monetary worth of the object. The weight is the physical unit of weight for the item. Attunement is whether or not a player needs to spend time with the item in order to use it. The description is what properties the item has. Ideally, we would like to be able to generate wholly original magical items for D&D. Our initial goal is to be able to generate new names for items, and then add on additional proprieties as time allows. To generate wholly original items, we plan to train our model using all available items in D&D 5e, as well as some of the additional content, such as Tasha's Cauldron of Everything. Since there are multiply words in these names, we plan to also train the model to identify basic grammar (i.e., recognize a noun versus an adjective). This is inherently a natural language processing (NLP) problem: we want to train a model to learn from text and to then generate its own. We believe the best way to accomplish this goal is to use a recurrent neural network (RNN).

A RNN is great for processing sequential data, such as text. Not all the text in each magical artifact will be of a uniform length; the names and description may

vary. RNNs are very useful for this data type, as they can model text of variable lengths [27]. RNNs allow for data to be looped, which lets information be passed along through multiple steps in the network. Due to this, simple RNNs can have problems with exploding gradients, and vanishing gradients. Instead, we can use Long Short-Term Memory (LSTM) RNNs to train our model. LSTM have three gates: input, forget, and output. These gates allow us to selectively read, write, and forget pieces of information. The forget gates that are not found in simple RNNs. Therefore, we can use LSTM RNNs to overcome both exploding and vanishing gradients [27]. The visualization of the LSTM RNN can be seen below in Figure 1.5.

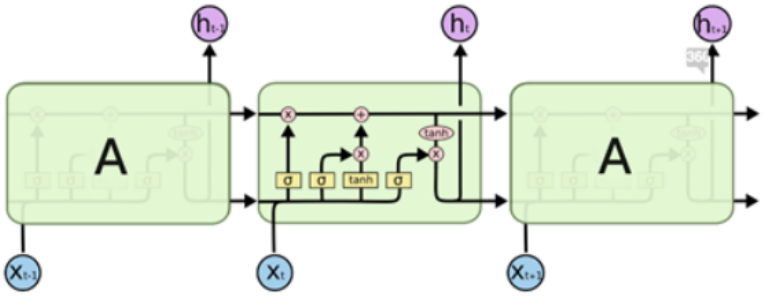

Figure 1.5: This is the structure of a LSTM, detailing the three gates. From left to right: the forget gate, the input gate, and the output gate. [23]

There have been multiple successful projects that generate names using LSTM RNNs, such as a project that classifies and names astronomical objects. Overall, using LSTM RNNs for naming items seems to be a good convention.

1.5 Project Idea 4: AI Bomberman

This project idea is to use machine learning to teach an AI to play Bomberman. This will require us to remake the Bomberman game, to allow us to have easy access to the data required to train the AI. The requirements for building Bomberman would require us to have a 13x11 matrix filled with "Concrete blocks" and "Brick blocks" this can be seen in Figure 1.6 as well as following features of the game. Where the bricks are able to be destroyed by bombs and concrete can not. Then we would need to construct a player which would have the ability to move in each of the four cardinal directions and also be able to place a bomb on the square they are currently on. We would also need to make the bomb entity which when placed by the player would wait a set amount of time before exploding and destroying all bricks and players that are two away in each of the cardinal directions.

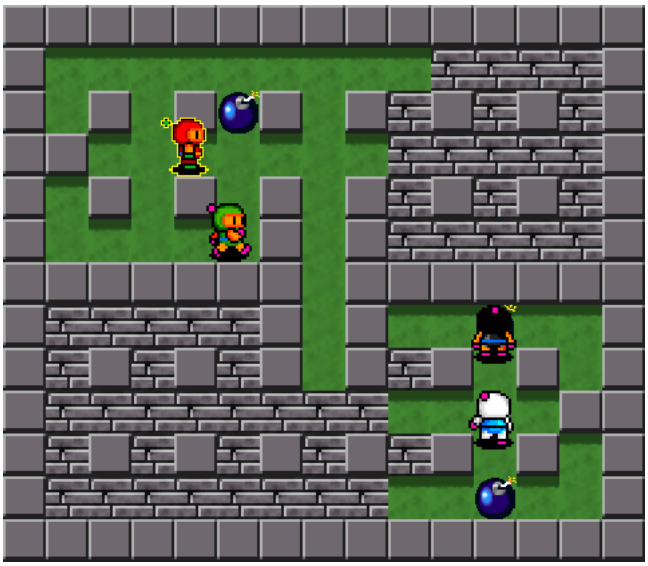

Figure 1.6: An image of the Bomberman running a 4 play game. [8]

One of the applications of this project would allow game developers or modders to change the current AI that is being used. This could give players a larger selection of AI's to compete against ranging from very easy all the way up to unbeatable. This project could also be used for researchers for others looking to do machine learning on Bomberman or other games that follow a similar structure allowing them to convert the concepts easily to what they are researching.

The most obvious approach for this problem would be to use Reinforcement Learning (RL) [17]. Allowing the AI to play against itself inside the 13x11 arena, and rewarding the AI that won by using it for the next generation of AI. By repeating this process countless times you will eventually have an AI that would be very efficient at playing the game. As it slowly learns over time new ways to trap, or out maneuver the opposing AI/player.

Another approach to this problem as seen in the following paper is to use Deep Reinforcement Learning (DRL) and Imitation Learning (IL) to teach there AI to play Bomberman [8]. They use IL which would allow them to demonstrate the desired behavior rather than specify a reward function that would generate the same behaviour, but without having to deal with rewards. They also use DRL which is similar to that of RL but instead of storing all the states for the game which can be cumbersome for a game like Bomberman, it instead uses a neural network that can predict the reward for the decision the AI made [14].

1.6 Conclusions

In this paper we have gone over four different project ideas to help us determine which one we should pursue moving forward. Each of these proposed ideas come with their upsides and downsides.

Reconstruction of speech after filtering may be the most real world applicable problem explored above. The ability to use large already existing data sets would make this problem much easier as well. The biggest problem would be trying to analyze the results for correctness, as the analysis would be highly subjective. This project would likely use either an RNN or GAN.

Language recognition using automatic speech recognition presents some viable methods we could employ to accomplish the classification of audio by language. Previous work on general speech recognition has been well studied and multi-language recognition was recently worked on as well. One major downside to this project would be the possible training time of a multi-lingual implementation, along with finding a large enough dataset.

We find generating D&D magical items to be an exciting problem to solve. It is an interesting natural language processing problem where we can implement an LSTM RNN. The application for this project brings table top players new, creative items that they can add to their campaigns. While this problem is truly interesting, we do anticipate the main roadblock being the somewhat limited dataset. Our next step is to look into the additional D&D content to try and supplement our dataset to ensure we have enough data for this to be a viable project.

The idea of doing AI Bomberman sounds fun and exciting. It also has quite a few resources available to work from, though most of the solutions that are found for this project mostly relies reinforcement learning. Which does not follow the spirit of the class in learning about Deep Learning along with having to develop our own version of Bomberman which might be too time consuming for this class.

Chapter 2

Milestone 2: Project Selection

2.1 Introduction

Many games have a limited amount of unique items, which can hinder the creativity involved in the game. As such, procedural generation systems have become common place. Unfortunately, these procedural systems typically have a limited number of parameters that can be varied and each with lower and upper bounds, for example, the color and size of leaves on a tree. This is why even with effectively infinite worlds for Minecraft or No Man's Sky, the game play will eventually start to feel repetitive. These 'infinite possibilities' are really just a set of rules for varying color or terrain generation, not completely new ideas. That is to say, they are variations on existing ideas, not original content.

To solve this problem, we decided to go forward with a variation of project idea 3 the D&D artifact generation, for a more broad idea of just magical items in general. The decision to modify the project idea came about because the limited nature of D&D artifacts, resulting in a very small data set. To address this, we will instead combine lists of items from several different franchises. This way we can ensure that we are getting enough data to effectively train on. We chose this project because we find it particularly interesting and we suspect that some of the other ideas may have a time cost that would not be feasible in the required time frame. We also felt the final result may be more useful than any of our other proposals given our skill level and time allotted would limit the quality of the end product.

2.2 Problem Specification

2.2.1 Main Problem

We will be using a machine learning model to generate names of fantasy items. Massively Multiplayer Online Role Playing Games (MMORPGs) are hugely popular games, as are tabletop RPGs. In many of these games, players go on adventures, and collect items. As such, generating wholly unique new items can add a lot of needed excitement in the environment.

Generative machine learning models are not new and have been researched quite thoroughly in the past. In many cases, they have been used to create images by use of things like Generative Adversarial Networks, or long stretches of text by use of Recurrent Neural Networks or transformers. In the realm of video games, the most high profile example is likely AI Dungeon. AI Dungeon has the much more ambitious goal of making a full text based adventure game by using machine learning both to understand the input and to generate a full role playing game story for the output based on the input using the GPT-2 transformation network. While the experience of playing the game is interesting, the output is often nonsensical and lacks structure. Our goal is to provide a tool to aid human developers in creation of items, or to be used selectively in a game.

Aside from limiting scope of text generation to a few words, specifically adjective noun combinations with possibilities of adverb combinations, we also want to introduce variable levels of noise into our model which could allow us to change the originality of new content. Without noise, we expect the output to be as accurate as the network allows, often just regurgitating data that was used in training, not coming up with something novel. Similarly, with large amounts of noise, we expect output to get completely nonsensical, where it may cease to even be pronounceable. Having this be a variable that can tweaked after the model is trained lets the user decide how original they want an idea to be. This may even directly correlate to the in-game value or rarity of an item. This also allows the user to increase the abnormality if they want to manually filter output to get particularly good names. We think adding this noise will be successful because our goal is to create unique and original output.

Our problem has applications in game design for massively multiplayer online role playing games MMORPGs and tabletop RPGs specifically, but the concept of seeding an RNN with a possible random or varied input to produce sensible and useful output could be applied in a number of venues outside of simple name generation.

2.2.2 Risk & Cost

By specifically adding noise, we risk the chance of having unreadable outputs. Since we don't have a ton of experience working with machine learning, we might run into issues where we will need to change aspects of the project to allow us to progress.

The cost for this project will be free aside from the time spent working on it as we have open access to the Holland Computing Center for our training resource, as well as the ability to train on our own machines. The only cost is the resource occupation this project would incur as time spent training the models as well as the electricity used to run the machines training it, which may limit the time available to other researchers.

2.3 Steps

We are presenting two different methods to solve this problem they are Long Short-Term Memory (LSTM) RNN and Gated Recurrent Unit (GRU) RNN. These two architecture are similar and thus the methods for structuring our data, prepossessing, evaluation, and training are going to be the same. The area where our methods will differ will be our architecture where we will discuss the benefits of one over the other.

2.3.1 Data & Preprocessing

First, we will need to construct unified dataset from various online sources. We have already identified some that can be pieced together, including lists of items from EverQuest and World of Warcraft.

These combined datasets have approximately 150,000 items. These datasets will need to be sanitized to remove duplicates or extremely similar entries, as well as remove data that is subjectively bad for our purposes. Unfortunately, this is a process that will take some manual work to come up with what needs to be removed, as well as to write a script to sanitize the data based on those metrics we come up with.

Our next step is to load and process the data in TensorFlow, making use of a training, validation, and testing split. All of the data will be converted to vectors using built-in TensorFlow methods.

2.3.2 Evaluation & Training

A proposed metric will need to be created to evaluate the performance of a model with generation of text. This could be simply validation loss during training, or something custom based on phonemes which could help make sure that the model focuses on producing pronounceable and sensible text. The model could also be evaluated on similarly of words within a common English dictionary, again to make sure that the output is human readable.

A set of hyper parameters will need to be created to tune our model to the best parameters available and k-fold cross validation could be employed to assess the performance of the model with these different parameters based on the metrics proposed previously. This should be performed on a subset of the data to keep training time low. Given a thorough investigation of these parameters, a full

training could be performed making use of early stopping and dropout to prevent overfitting.

2.3.3 Generation

After training is complete, to generate names, a seed will have to be provided to the model and "predictions" can be made in sequence to generate outputs based on that seed. This can then be iterated over many times to generate many unique names based on a single seed.

To vary the output and add more uniqueness to the output, as well as distort the output in some ways, we can provide randomized seeds which could provide interesting and unexpected results. To further vary the output, we plan to add a mechanism to take a fully trained model and modify the weights of the hidden layers by some random coefficient to vary the connected graphs of the model, further distorting the original trained model, providing more varied outputs.

2.4 Proposed Method 1: LSTM RNN

2.4.1 Architecture

The Long-Short Term Memory (LSTM) RNN can be constructed by use of TensorFlow's own built in layers, as they have support for LSTM layers. This will be the most straightforward way to implement the network and will prevent possible implementation errors if trying to create the model from scratch. The layers also have many options for different parameters which will make hyper parameter tuning easy to implement on the model.

Recurrent neural networks have the advantage of not requiring a uniform sequence length. This allows us to use each individual item as example sequences of varying length. To train, the input will contain the beginning 'k' characters of an item, and the label will be the 'k' characters long shifted one place to the right. Eventually, by repeating this process, the model will be able to give the beginning letters of an item name, and have the model be able to give a sensible ending to it [16].

An LSTM RNN was previously described in Section 1.4. LSTM have advantages over simple RNNs, in that it can selectively read, write, and forget data. The fact that LSTMs can forget data allow for them to prevent both vanishing and exploding gradients. The architecture is well suited for a text generation task as the short term memory aspect of the model should prevent it from getting fixed on a certain cyclical output, so that outputs are unique and human readable.

2.5 Proposed Method 2: GRU RNN

2.5.1 Architecture

Gated Recurrent Unit (GRU) RNNs are similar to LSTM RNNs, with a few key differences. Figure 2.1 shows the visualization of each RNN.

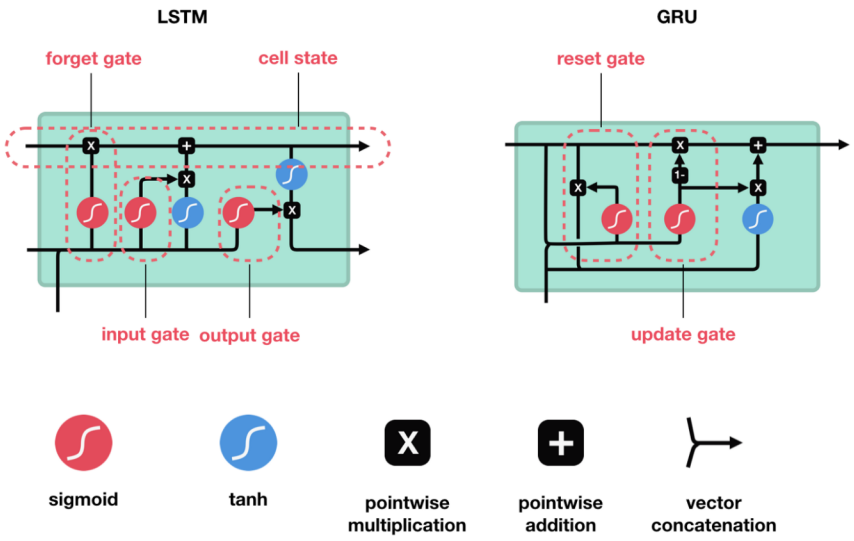

Figure 2.1: This details the structures of LSTM RNNs and GRU RNNs. [18]

While LSTM RNNs have three gates, GRU RNNs have two: the reset gate and the update gate. The reset gate is akin to the forget gate in the LSTM RNN. It decides how much past information to pass along the network. The update gate is similar to the input and forget gates in the LSTM RNNs. The update gate determines how much new information gets passed along the network. Because there are less gates, GRUs are less complex, and so they generally train faster than LSTMs [27].

Since GRU RNNs appear to perform just as well as LSTM RNNs while being less computationally complex, they appear to be the better choice for our project.

2.6 Roadmap

This project has a deadline of April 30. Specifying a total amount of time needed for this project will be difficult since we have yet to start development on it. Though we feel confident in our ability to finish the project before our final deadline, and if we run into unforeseen roadblock we will take the proper measures to resole said roadblock.

Given there are ample resources on the creation of either of these networks, and GRU and LSTM layers exist within TensorFlow, a rudimentary implementation that generates some form of output shouldn't take too long. The most consuming parts of the project will be cleaning and formatting the datasets as well as coming up with a good metric to evaluate the performance of the model given this is generative and not classification, which makes it very subjective.

2.6.1 Milestone 3

The overall goal is to have a working prototype. We want to get a minimal model setup with an evaluation metric created, as well as data mostly gathered and cleaned. That way, we can do basic training to some extent as a proof of concept, making sure we're on the right track.

2.6.2 Milestone 4

For Milestone 4, we aim to have a fully trained model that expands on the minimal model setup from Milestone 3. At this point, we would like to have started adding in the noise to produce the original content.

2.7 Conclusions

We will be implementing an RNN for our project. They are very useful for sequential data, and seem to be the most widely implemented for generating text. Both LSTM and GRU RNNs are great options for generating text. We believe that GRU RNNs are the better choice for our project. They appear to perform just as well as LSTM, while being more simple. Our next step is to finish collecting and cleaning our data. From there, we will begin building our model, and eventually we will have a full model that we can introduce noise into. The questions we have for you: Do we need to add anything more to our project? What recommendations do you have for an evaluation metric? Do we need to validate somehow?

Chapter 3

Milestone 3: Progress Report 1

3.1 Introduction

We are generating original names for fantasy items using a Recurrent Neural Network (RNN). There exist a plethora of fantasy items across many different platforms. While there are many items to be found, a large number of them are very repetitive, with many descriptions being reused. As such, we were interested in this project because it allows for us to add creativity to any RPG using artificial intelligence. We decided to use an RNN because they are well-suited for sequential data of non-uniform length, which our text is [27].

At this point, we are on track to complete our next milestone. The data is all compiled, and the vast majority of data cleaning is completed. We have a working prototype to generate items. Many of the generated items are valid, new items. Just as many, however, are repeats from the dataset, which indicates that there is some memorization happening. Additionally, many of the item names have endings that are a large amount of Roman numerals. This indicates that we need to continue developing our model to alter how the text generation stops as well as continue sanitizing the dataset to possibly remove the roman numerals. We believe that stopping the text generation after an amount of words, instead of a fixed set of characters could fix the strange endings of item names that are currently being generated as the model seems to degenerate the further out the predictions go.

3.2 Experimental Setup

We started by collecting and sanitizing our data sets, which were both large collections of item names from MMORPGs. Our data came from 2 main sources. Much of our data comes from a WoW GitHub repository that contained over 108,000 item names [7]. The remaining data comes from community ran EverQuest wiki pages [1], which were scraped using the Beautiful Soup Python library. The data we gathered from Wow Github contained a mix of every item that can be found in game from weapons, potions, armour, to mundane items like food. The data that was brought in from the EverQuest wiki was similar, but there was significantly more duplicated items that have roman numeral variations. Once we had scraped the data we combined them into a single data set.

Because our end goal is creative in nature, we determined the sanitization of the dataset needed to be done mostly manually. As such, we decided it was appropriate to take the time to go though the merged data first programmatically and then by hand. We removed any and all duplicates and cleaned up data entries to remove artifacts left in by developers such as "DO NOT USE" or "(Deprecated)" at the end of the item name. We tried to remove as many non character symbols as possible such as punctuation to simplify the vocabulary the network could pull from as well. Additionally, we standardized repeated patterns, such as changing "Vol.", "volume" and "Vol:" to "Volume".

All entries were in the format of proper nouns with every word capitalized like titles would be. Some items were duplicates of each other but as different tiers marked by roman numerals to denote rarity or power, or prepended with certain adjectives describing the item. If these prove to become too common in the network, they may be removed to help generate better results. For the next milestone, more data may also be added such as a standard English dictionary to allow for those words to be considered as part of the output. Names are fairly short with the longest being 49 characters in length, but with most being a few words no longer than 30 characters in total.

After a thorough literature review, we found that when handling shorter amounts of natural text generation, many researchers chose to implement human evaluation. Shang, Lu, and Li generated natural responses to a sentence. They generated multiple models to respond to these sentences, and had reviewers read these responses, and judge the response as unsuitable, neutral, or suitable. They then averaged their rating for each model [22]. Additionally, Bartoli et al. used LSTMs to generate short restaurant reviews. To evaluate their model, they had individuals read real reviews and generated reviews and rate them as useful or not useful. This allowed them to see if people could discriminate between real text and AI generated text [3]. Similarly, in our current stage, to evaluate our model we are reading our output and deciding whether or not it is valid based on if it is a unique output, and if we can pronounce it. Because we are still in the early development stage, we are continuously making small

adjustments, and thus haven't done a thorough, complete review of our outputs at this time.

The model is a basic GRU RNN model set up with an embedding layer as input and a dense layer as output which can be seen in Figure 3.1. A GRU was chosen due to its faster implementation to an LSTM with similar predictive performance [27]. An LSTM, as mentioned previously, is well suited for the task of text generation due to its short term memory. The model assumes an input of data in padded character format. The data is first loaded in from a CSV file with every line denoting a new item. This is then broken up by characters to be used in the preprocessing for the model.

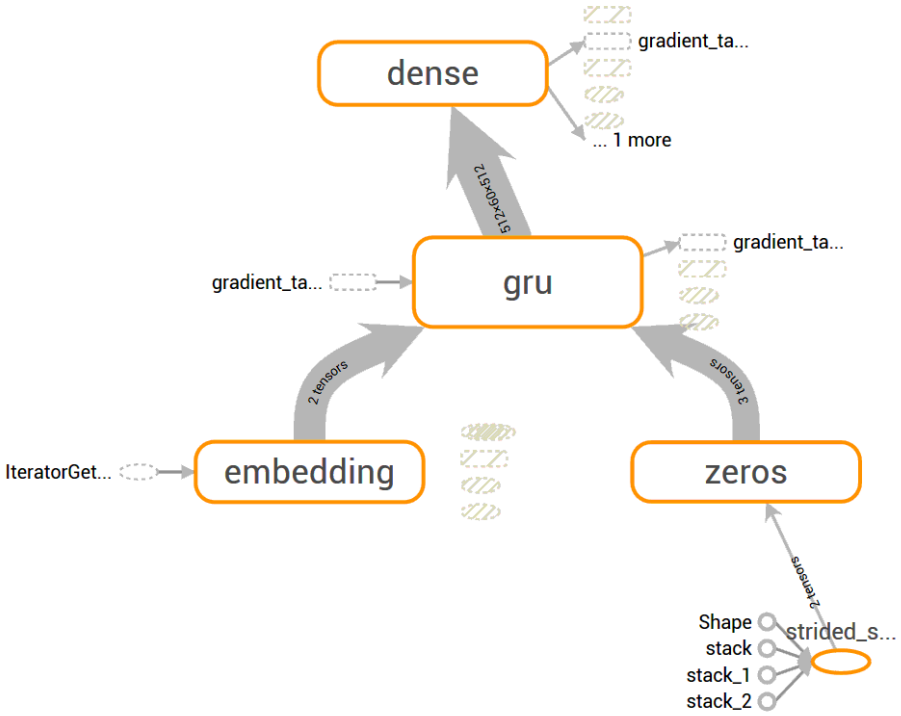

Figure 3.1: Visual structure of the current GRU RNN network

Preprocessing is performed through string lookup layers outside the model which convert characters or strings into indices and inversely convert indices into text on the output. Vocabulary is determined by the unique characters in the set and the string lookup layer's adapt function is used to create a sorted vocabulary. To create training labels based on the input, names were left shifted by one character getting rid of the first character in the text, meaning the network should learn to predict the next set of characters based on the first.

Specific model parameters are still variable during the implementation phase, but currently, the RNN is built to 512 units with an embedding dimension matching the vocabulary size. The data is also being shuffled and batched to a size of 256 data points.

The model is trained using stochastic gradient descent with the Adam optimizer and is trained on sparse categorical cross entropy as the loss and accuracy metric. The output of the model is the same dimension as the vocabulary with predictions providing the probability of the next character based on the previous data the model has seen. Outputs are sampled to created prediction ids which can be converted back into characters

Because of the architecture set up, a sub-model was set up to allow for one step predictions where given the trained model and an input character as a seed, the model would create predictions on the input and return the model state. This allows for the sub-model to be ran in a loop for the desired amount character predictions. A "temperature" variable is also implemented to attenuate or accentuate the predictions of the model, leading to results closer or further from the input data set. This is preformed by dividing the predictions from the model by the temperature. Currently, this is ran to 50 characters to asses the outputs of the network during the prototyping and development phase.

3.3 Experimental Results

```
Conner's mighty gloveskine oreanskin belts ornamentasmence
Conner's x`trainer silk lord axe iii broasts and blood orn
Conner's roes inconcadery meat gavine iii iii mesherious x
Conner's bounty pace viii ii iro signal tempersing xvuil o
Conner's spaulders template battleboarder weapon iii iii i
Conner's steel crystal battlessong backshate vial of super
Conner's handswatches iii iii spiritian mossflawer's hatch
Conner's summerfer iii iii iii crossmoot iii iii ice sanct
Conner's breath iii iron presil talonmank scroll sappords
Conner's belt of davenous amberseakerfer iii iii iii iii 9
Conner's cakeweava goggmass vicious symbolspike blacksillo
Conner's ringmail flame pantsongswal culture iii ii iiiizs
Conner's archery fine stalkermal chain bony of intentity i
Conner's bound ice raptor ornamentation iii modd coirses i
```

Listing 1: Output of model seeded with "Conner's "

As can be seen in Listing 1, the output that is currently generated, while readable, leaves much to be desired. For instance, the end of many outputs has a plethora of repeated 'i' characters. We suspect this is a result of the model overfitting to the item names which include their level as a Roman numeral on the end. Some of the outputs are also found verbatim in the dataset, meaning

there is a lack of uniqueness in some outputs. Further, each output is generated to a character limit, resulting in very unwieldy item names that often just get cutoff in the middle of a word. Fortunately, the output is already generating some non-gibberish sub strings, such as "Conner's belt of davenous amber" or "Conner's steel crystal battlessong."

3.4 Discussion

It appears our output is currently being made substantially worse by the inclusion of Roman numerals in the dataset. Going forward we will remove all Roman numerals to mitigate that issue. Further, the output needs some method for ending other than a simple character limit to avoid both incredibly long names and words being broken off due to hitting the limit. The current results also show worrying amounts of memorization instead of creative output, though the correct balance between memorization and new output is inherently difficult to establish for this kind of project. With this in mind, we suspect temperature is a hyperparameter that we will need to adjust carefully going forward.

3.5 Conclusion

Looking back to our roadmap in Section 2.6, we are on track to be able to generate wholly unique, valid fantasy item names by April 30. So far, the dataset has been created and sanitized, an early prototype has been made, and outputs are being generated that somewhat match our desired goals. However, the model and dataset itself will need much more work to improve the quality of the generated text.

For our next milestone, we would like to further refine the network by tuning hyperparameters, setting up a robust evaluation metric, and adding noise to the weights in the trained network. Currently, the idea is to evaluate by ensuring that all outputs are at least 5 characters, that each word in the output is below a maximum length, and that grouping of letters exist in the original dataset (in order to avoid invalid letter combinations such as 'qxi' or 'uok' that don't appear in English normally).

Chapter 4

Milestone 4: Progress Report 2

4.1 Introduction

There exists a nearly infinite amount of fantasy games, all of which have their own items. While there are many items to be found, a lot of these items are repetitive, and lack some originality. Therefore, the scope of this project is to generate exciting, new fantasy names. In order to generate these new names, we implemented a Recurrent Neural Network (RNN).

For Milestone 3, we had developed a working prototype of the model. For this milestone, we worked to continue to improve our model. We finalized our dataset, tested different hyperparameters, added a method for stopping name generation at sensible outputs, and attempted a form of quantitative validation. Our model was able to generate improved results from Milestone 3, as the input dataset was much more sanitized and our method to early stop name generation would both limit the length of words and stop generating item names on a noun or a verb. The outputs still leave something to be desired and can still be significantly improved, especially in trying to improve originality of outputs. As of now, we are on track to be able to complete the project by the final deadline.

4.2 Experimental Setup

The first change that we implemented to improve our output was not directly related to machine learning. At the end of Milestone 3, our prototype generated a set number of characters without sensible end, and so we decided to first focus on making the output more sensible. The first issue that we wanted to fix was the length of the output and the sudden truncation of words in names. To do

this, we implemented code for counting how many words had been generated by delimiting output by white space. Then, once at least a few words are generated, set by a minimum value, the python Natural Language Toolkit (NLTK) library [4] is used to continue generating until either a noun or verb is encountered. We chose this, as items tend to end in a noun, i.e. "Silver Sword", or a verb in some form, i.e., "Silver Sword of Banishing."

Next, we set up a limit on the length that any given word can be. We decided to do this for two reasons. First, occasionally the network would get stuck in a very long loop generating a repeating sequence such as "essessessess." Second, some names, even without repeating sequences, were very long, and usually gibberish. The solution that we implemented was to simply throw out any item name that had gone more than twelve characters since the last space. This would also reduce the need for keeping a history of states to forcefully reset the network before the bad output.

We also updated how we seed the model for item generation. Previously, we had to manually input a batch of a seed character or word. We now have a random letter input as the seed character sequentially, while still allowing for manual override. We chose to do this to get a wider variety of generated items in one run as well as to employ the previously mentioned filtering since it wouldn't work on a batched output.

Additionally, we further cleaned the data. As discussed in the previous milestone, we removed all Roman numerals from the dataset. We also removed all numbers from the dataset, and rewrote them using words, i.e. "Twenty Nine Pound Salmon." When we included numbers in the dataset, once a number was generated, the output simply generated a long string of numbers, since numbers were typically used at endings of phrases. Rewriting them as words corrected for this issue.

Previously, we saw a large amount of memorization from the generated text. To combat this, we added two dictionaries of English nouns [10] and adjectives [11] to our dataset, which have a combined total of approximately 64,000 new words. We also tested different temperature values which scale the output weights to reduce uniqueness. We also implemented random noise on the output the be able to push the model outside of its trained weights during the generation process.

To be able to quantitatively measure the performance of the model, we settled on a metric to be able to measure the similarity of generated outputs to the input dataset. We used the Term Frequency–Inverse Document Frequency (TF-IDF) metric which weights common words among a corpus of inputs less than non-common words [26]. Running generated outputs through the metric, and averaging the result for each word's frequency in a name would give a similarity score to input corpus. We followed a guide written by "coderasha" on the "DEV.TO" community for the implementation [6].

Finally, the model was trained using an embedding layer size equal to the vo-

cabulary size (all the characters in the dataset), 512 RNN units, 1024 batch size, and trained to 40 epochs.

4.3 Experimental Results

After making the improvements to our model, we had retrained it and ran it with random character seed.

```
wrathful gladiator's spellblood gloves
wurm caked bowstaves of the decimator
abstictional crystal spike ricexial
blackcuttered mail handgrips of focused rusted
kitsicalside chimerahoge pantsmoots dayshick
ywhide antedlaw peace kickersy
keenshaf moloin bamboo staffs
messenger's breastplates ketchar three thoughts
zealous crescent shield rusted
quaint ruled bag of venomshroud
zandalar ionsome estalking powderskills
bracers of the quiefle-pardance
flayed vambraces of the decimator
cataclysmic gladiator's armwraps of ascendancy
```

Listing 2: Output seeded with random letters with temperature of 1.0

In Listing 2 it can be seen that the generated output is now constrained to a reasonable length and, while not always sensible, at least ends on nouns or verbs, according to what the NLTK library deems as a part of speech. It sometimes still cuts off phrases in subpar locations due to our max word count cap, or because we don't perform a complex enough analysis on phrase structure to come up with something that works a little better.

One of the outputs of the model was particularly concerning, as it demonstrated how without checks to avoid specific outputs, vulgar or otherwise unpleasant output may be generated, as was the case with the output "nigga's shimmering murloc eggs". This output, its significance, and a potential explanation for its generation are discussed in the next section, 4.4.

We also tried running the model with a similarity metric which utilizes the TF-IDF Model to weight the similarity of a phrase to the original dataset.

The output of the phrases with scores can be seen in Listing 3. This didn't appear to give us very useful information however since certain phrases which were in the data set at times were scored low, while other phrases were scored highly even though they were very unique. This was also not consistent across the board.

```
sword of shadowsong covered
0.00799503
sword of vomity fivedy
0.006286258
sword of bracersonal power
0.0033237792
sword of mash mastery
0.0035870972
sword graysins ore of rove
0.0125091
sword of the acuie
0.00799503
sword of carditements tabardings
0.0043294895
swords of pistonstipe pointsing
0.0039299
sword of landinghouse crystalscale
0.0033410736
```

Listing 3: Output seeded with random letters with similarity metric

We also tested the model with a low temperature score of 0.5, which is shown in Listing 4 and a high temperature score of 1.5, which is shown in Listing 5.

We finally also tested our implementation of random noise with a weight of 0.005, which is shown in Listing 6. This noise is added to the output weights of the model and therefore tend to have very adverse effects on the generated output, severely ruining some outputs without much benefit.

4.4 Discussion

The results have improved significantly. Long, repeated sequences of roman numerals are no longer generated, each item name ends with a sensible part of speech instead of being suddenly truncated, and each word is limited in length. We also did not see the model generate exact copies as often from the training model, like we did previously. We attribute some of this to the addition of the two dictionaries of nouns and adjectives to our dataset. These items gave the model new combinations of letters to learn from, while being short items individually, thus preventing our model from outputting an exact copy of the items.

Some of our output is more sensible than others, for instance "wrathful gladiator's spellblood gloves" is exactly the type of item we are hoping to generate, whereas "ywhide antedlaw peace kickersy" leaves something to be desired. Going forward, we will be testing various hyperparameters to further improve our model.

```
recipe for major blood
argent crusade gift collection
chalice of the burnished
plate cosgrove chest seal
orgrimmar boots of the decimators
distillate of regeneration two thousand
mark of the defender
tome of the deaded male
consigned nimbus of guardian resilience
yogon's spear blade with scrapsed poison
commendation of the kirin
tome of frost shield
book of reflexive revovency
growlers claw of sufferings
```

Listing 4: Outputseeded with random letters with temperature of 0.5

As mentioned above, the model generated a vulgar output of "nigga's shimmering murloc eggs". While it is difficult to know for sure, it appears that the vulgar part of this output is a result of the dataset containing a good amount of substrings that could lead to such an output. There are a substantial amount of "ni", "ig", "gg", "a's" substring groupings which can lead the model to randomly generate such a word. This is especially true because of the fact that the model generates a character every step, meaning it can randomly decided that a different character than what's common be substituted into a string of characters leading to a different set of next characters.

While this is obviously an issue, we do not currently plan to address this in code. We came to this decision because this is a time sensitive academic project, and the potential large slow down that would come from checking the output to make sure it does not vulgar substrings. This would almost certainly be different if this were a commercial project, especially as implementing the check should be trivial.

As presented in Section 4.3, the TF-IDF similarity metric didn't provide us with very meaningful results. We believe we may have incorrectly implemented the metric; it could also be happening due to the metric itself since the metric weighs words in a phrase by how common they are in input dataset. The more common, the lower they are weighted. If "sword" appears often in the dataset, the similarity score of sword would be very low. A configuration of the weighting, or a different metric entirely may need to be used.

A lower temperature yields more sensible outputs; however, the outputs are more likely to be identical to existing names from the input dataset.

The random noise that we implemented does not improve the model. Currently, while most of the output is readable, it is meaningless. Going forward, we plan

```
nu'meticantsi-shelink's unicolary of assbananacy
axelerkily mastruc's manti-merh vampird
fire-poweress teendrym phiestan sophise
devomentslune belt copper hunting
malista's man oin's signet
theramore remeditious strishion sumversnatts
gurubase cuber-kity's hide with cuffendees'
zatra's tensi sermoan gland
xodolsque bloodfear-bagek headants' sword
mic's leadwy glebar cortements
rustyz's boar tuft fom
x'quilizing nimble symbol of judgments
jakerage ushaborblood solving piendshakes
fetish ox dre chillipolley
```

Listing 5: Output seeded with random letters with temperature of 1.5

to alter how the noise is implemented such that it is able to improve on the uniqueness of outputs without decimating the results. This may be done in a modal fashion, or only occasionally adding in random noise on the output such that it nudges the model into a different random direction, rather than doing so on every generation step.

4.5 Conclusion

We are still on track to be able to generate interesting, original fantasy item names by May 2nd. We have finished cleaning all of our data, and have made steps to improve the prototype model by setting a max word length for any given word in the output, testing different hyperparameters, implementing new stopping mechanism for our output, and attempting a form of validation. From here, we will be continuing to work on improving our model by improving a similarity metric, improving the random noise, and implementing an attention mechanism. Right now, the only question we have for the TA is do you have any suggestions for validation, most papers do not mention anything quantitative?

Table 4.1: Contributions by team member for this assignment.

Team Member	**Contribution**
Vega Carlson	Generalist: Code, Research, and Paper Writing
Stephanie Marsh	Researcher: Research, Code, and Paper Writing
Daniel Shchur	Scientist: Code Lead, Researcher, and Paper Writing
Conner Elliott	Medic: Paper Writing, Researcher, and Code

```
special silver covenage coffee
armpitus' hexweave cloak sectional mossyot
zandalari spiritmethts seal of the decimators
goverants of the horsex
prewed toestoons mugulates pattern
vicious gladiator's pummelert opatia
leap of pebble of anri's
skirmisted bowl of the summurate
qiremok smithy hammered stiffsword
brehzelsing antenite twelves potiondoran
gommetable wrencar's fragment attakks
redusensal limestorts osal of foresauker
yun shell presume--conukul orbantskil
bixie anklewreg' girple-gripersy knuxul's
```

Listing 6: Output seeded with random letters with random noise at weight 0.005

Chapter 5

Milestone 5: Final Report

5.1 Introduction

Fantasy game items commonly suffer from repetition and lack of originality, to the extent that killing a giant rat with a wooden sword or collecting bear teeth has become a cliche. To remedy this, in previous milestones, we have put together a dataset of existing fantasy names from various sources, and trained a Recurrent Neural Network (RNN) machine learning model to create original fantasy game item names.

For this milestone, we tested various hyperparameters and implemented quantitative and qualitative scores for evaluating our model. For the quantitative measure, we implemented a similarity metric using Damerau-Levenshtein distance. For the qualitative measure, we did blind testing for the different hyperparameters. We also tried to implement an attention mechanism.

By using these evaluation metrics, were we able to improve the output of our model. We determined the best set of hyperparameters to be 1024 RNN units and a temperature of 1.0, as it had a higher qualitative score and a lower Damerau-Levenshtein similarity.

5.2 Experimental Setup

We chose to implement Damerau-Levenshtein to be able to compare the similarity of the generated string to every string in the original dataset [24]. We did so using the `Jellyfish` library.

Damerau-Levenshtein similarity is comprised of the actual distance and the maximum distance between two strings. The actual distance is the number of necessary text transposes/deletions/substitutions to turn the first string into the second string. The maximum distance is the hypothetical highest actual

distance between the two strings, meaning they contain no similar characters. Let D be the, Damerau-Levenshtein similarity, a be the actual distance, and m be the maximum distance. The Damerau-Levenshtein similarity formula is simply $d = 1 - \frac{a}{m}$. We compute the Damerau-Levenshtein similarity between a generated string and every item item in the dataset, and save the highest similarity metric.

Given Damerau-Levenshtein takes a significant amount of computational power and time as it is a substring search, further compounded with the very large dataset to search through, the problem benefits greatly from parallelism. To make use of modern multi-core systems, we wrapped the Damerau-Levenshtein function in a multi-processing loop which ran similarity on subsets of the original dataset with multiple threads. This was able to speed up the computation significantly scaling linearly with the number of threads available. All of the results from the individual threads were then concatenated into a single output list where a max was taken to find the highest similarity of the generated name to the entire dataset.

We also did qualitative testing because the problem at hand is naturally one of human preference, and there does not seem to be a widely accepted way of evaluating quality in this sense [13].

For double-blind testing, we first ran the model and generated output files with various hyperparameters. Then these output files were left unopened before being randomly renamed with a unique id using a python script. The mapping from these outputs with unique ids to which hyperparameters generated them was saved in a separate file no one looked at. The files were then distributed to each of the four authors of this paper. Each person then went though the first twenty items in each file, assigning a score ranging from one to five to each generated item name. A low score was indicative of an output which was of very poor quality, while a high score an item that was deemed high quality, though the scoring itself was entirely qualitative so it varied in nature. These per item scores were then averaged to assign a per reviewer score for each file. Finally, all the scores for all of the reviewers were averaged per file and mappings were deobfuscated such that the qualitative performance of different hyperparameters could be compared.

The model we implemented didn't change from previous milestones. It stayed a GRU base RNN model with an embedding layer and a fully connected output. We did try to implement an attention mechanism utilizing one dimensional convolutions on the inputs for query and value vectors which were fed into an attention layer and concatenated into the GRU layer. We quickly found out that attention is not suitable for our model however since we generate outputs one character at a time, which means the attention mechanism brings attention to specific characters as opposed to words. This creates the issue where the model generates an output of repetitive characters as opposed to words. Had our model been based on words as vectors as opposed to characters as vectors, the attention mechanism may have worked out well.

Given we now had a way to measure performance of the model in an objective way, we decided to try and tune different hyperparameters of the model. We varied temperatures and RNN units to determine the best set of hyper parameters. We did so training each set of parameters to 100 epochs with 10,000 generated names to determine average similarity. We had also tried to add noise to the trained model weights as a possible hyper parameter but it would often generate very poor outputs that would actually break the generation process since we had implemented analysis on outputs using NLTK. There were times where the model would get stuck during generation requiring a hard restart. Given the general issues with noise, we decided to ultimately leave it out.

5.3 Experimental Results

We tested the model with varying temperature and number of RNN units. We tested a low temperature score of 0.5, an average temperature score of 1.0, and a high temperature score of 1.5. We also tested 128, 256, 512, and 1024 units. Table 5.1 details the results.

Table 5.1: Hyper parameter tuning

Units	Temp.	Qual. Score (1,5)	Lev. Mean (0,1)	Lev. Var. (0,1)
128	0.5	3.78325	0.62817	0.01194
	1.0	1.60975	0.50010	0.00649
	1.5	1.02775	0.43680	0.00323
256	0.5	3.42350	0.66424	0.01957
	1.0	1.98750	0.52264	0.01040
	1.5	1.62225	0.44821	0.00427
512	0.5	3.52075	0.67551	0.02806
	1.0	1.68050	0.54587	0.01669
	1.5	1.29025	0.46644	0.00662
1024	0.5	2.85150	0.70850	0.03359
	1.0	**2.75275**	**0.57567**	**0.02430**
	1.5	1.99450	0.49720	0.01143

Listing 7 details some of our hand-chosen original output from the highest scoring model. More results can be viewed in Section B.

5.4 Discussion

Given our goal was to generate exciting, unique outputs, we determined the best model would be one with a maximized qualitative score and a minimized similarity score. Through hyper parameter tuning, we found that the model with 1024 units and a temperature of 1.0 was the best model, as it had a relatively high qualitative score with a low similarity metric compared to other parameters.

```
hard boiled gloss of the milfian
fiery gauntlets of demand
legplates of the lost
natural spices of norianthyess
kailed black bread cruncher
bracers of the breaker
ghostly orb of dark
legguards of the golden kong
enchanted orb of clarity
girdle of the resolute
```

Listing 7: Select output from model with 1024 units and temperature of 1.0

While we wanted to have maximum quality and minimum similarity, this is ultimately a subjective choice. If a user wanted to only have higher rated item names without regard for similarity, they may want to choose a lower temperature, resulting in outputs that very closely resemble the training dataset. Similarly, if the user were okay with the bulk the output being nonsensical or greatly misspelled, in an effort to find truly unique ideas, it may be wiser to use a higher temperature.

In Listing 7, we found the generation of "ghostly orb of dark" and "enchanted orb of clarity" to be particularly interesting, as they were both unique generated items that had different seed characters, different endings, and yet both generated the noun of "orb".

In Section 5.2 we discussed adding noise to the trained model's weights as a possible way to get more varied outputs. As stated there, even with a very small amount of noise the output created would sometimes break our model, causing infinite loops. The output we were able to generate was often of such poor quality that it is not even comparable to the other outputs.

5.5 Conclusion

In previous milestones we explored possible research subjects, chose to study generating fantasy item names, and progressively advanced the quality of a model for this task. This milestone marks the end of our work on this task. Overall we found our results quite satisfactory, with the names generated seeming quite usable. We found that the use of a qualitative metric in conjunction with a quantitative metric makes for an objective way to determine how well certain hyperparameters perform. We were able to determine that a network of 1024 RNN Units and a temperature of 1.0 provided us the best possible results with a relatively low similarity and high qualitative score.

Were we to continue working on this project, we would like to automatically exclude generated items that have a saved similarity metric of 1.0, or explore rad-

ically different architectures such as Generative Adversarial Networks (GAN). We would also like to be able to test a wider range of temperature values, such as 0.75, 1.25, etc. We would also like to experiment with adding different dictionaries to our data set, like for example Urban Dictionary.

Additionally, we would like to be able to generate descriptions for the generated items. To do this, we would need to use a different training dataset that contained item descriptions. Thankfully Many of the datasets we perused had both item and item descriptions, so finding new training data should be relatively trivial. We would also need to alter the model to generate words at a time, as opposed to characters.

Table 5.2: Contributions by team member for this assignment.

Team Member	**Contribution**
Vega Carlson	Generalist: Code, Research, and Paper Writing
Stephanie Marsh	Researcher: Research, Code, and Paper Writing
Daniel Shchur	Scientist: Code Lead, Researcher, and Paper Writing
Conner Elliott	Medic: Paper Writing, Researcher, and Code

Appendix A

Full Source Code

The source code was broken up into three files. 'main.py', 'model.py', and 'util.py' which contain the code for instantiating the TensorFlow runtime environment, defining the structure of the RNN model, and utilities for loading and prepossessing the dataset respectively.

The source code relies on various python libraries, which may all be installed via python's 'pip' package manager utility or whatever other method is recommended for the platform on which you are running this code. These libraries and their version are in Table A.1. These libraries were used in conjunction with the Python 3.8.8 interpreter.

While the source code is presented below, it may be more convenient to view it on GitHub at
https://github.com/VegaDeftwing/RNNFantastyNameItemGenerator.

Table A.1: Python Libraries Used

Library	**Version Used**	**Purpose**
Tensorflow	2.4.1	Neural network definition and execution
NLTK	3.6.1	Natural language processing
Gensim	4.0.1	Natural language processing
Numpy	1.19.2	Matrix and array mathematics
Jellyfish	0.8.2	String metric and analysis
Multiprocessing	-	Multi-threaded processing
Signal	-	Interrupt signal capture
Pandas	1.2.2	.csv dataset loading and processing
Rich	9.12.0	Colored console output for legibility
OS	-	OS utilities
Random	-	Random number generator
String	-	String utilities

A.1 main.py

```
import tensorflow as tf
from .model import GenerativeGRU, OneStep, DamerauLevenshteinSimilarity
from .util import load_all_the_data
from rich import pretty, print
import os
import nltk
import random
import string


def main():
    input_path = 'final/items_now_adj_big.csv'
    vocab = load_all_the_data(input_path)
    if SIMILARITY_METRIC:
        similarity_metric = DamerauLevenshteinSimilarity(vocab.tolist())
    vocab = tf.strings.unicode_split(vocab, input_encoding='UTF-8')
    data = vocab.to_tensor()
    ids_from_chars = tf.keras.layers.experimental.preprocessing.StringLookup()
    ids_from_chars.adapt(vocab)
    chars_from_ids = tf.keras.layers.experimental.preprocessing.StringLookup(
        vocabulary=ids_from_chars.get_vocabulary(), invert=True)

    def text_from_ids(ids):
        return tf.strings.reduce_join(chars_from_ids(ids), axis=-1)

    all_ids = ids_from_chars(data)

    ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)

    def split_input_target(sequence):
        input_text = sequence[:-1]
        target_text = sequence[1:]
        return input_text, target_text

    dataset = ids_dataset.map(split_input_target)

    # Batch size
    BATCH_SIZE = 1024

    # Buffer size to shuffle the dataset
    BUFFER_SIZE = 1000000

    dataset = (
        dataset
        .shuffle(BUFFER_SIZE)
        .batch(BATCH_SIZE, drop_remainder=True)
        .prefetch(tf.data.experimental.AUTOTUNE))

    # Length of the vocabulary in chars
    vocab_size = len(ids_from_chars.get_vocabulary())

    '''
    The embedding dimension
    https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html
    '''
```

```
    embedding_dim = round(vocab_size ** 0.25)

    model = GenerativeGRU(vocab_size, embedding_dim, RNN_UNITS)

    model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(
        from_logits=True),
        optimizer=tf.keras.optimizers.Adam(),
        metrics=['accuracy']
    )

    if os.path.isdir(CHECKPOINT_PATH) and len(os.listdir(CHECKPOINT_PATH)) > 0:
        latest = tf.train.latest_checkpoint(CHECKPOINT_PATH)
        model.load_weights(latest).expect_partial()
        chkpt = tf.train.Checkpoint(model)
        chkpt.restore(latest).expect_partial()
    else:
        checkpoint_prefix = os.path.join(CHECKPOINT_PATH, 'ckpt_{epoch}')
        checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
            filepath=checkpoint_prefix, save_weights_only=True)
        # early_stopping_callback = tf.keras.callbacks.EarlyStopping(
        #     monitor='loss', patience=2)
        logdir = "logs/fit" + CHECKPOINT_PATH
        tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
        model.fit(
            dataset, epochs=EPOCHS,
            callbacks=[checkpoint_callback, tensorboard_callback])

    one_step_model = OneStep(
        model, chars_from_ids, ids_from_chars, TEMPERATURE, NOISE_WEIGHT)

    states = None
    i = 0
    while i < NUM_OF_EXAMPLES:
        seed = random.randint(0, 25)
        next_char = tf.constant([string.ascii_letters[seed]])
        # next_char = tf.constant(['sword'])
        result = [next_char]
        num_words = 0
        num_chars = 0
        idx = 0
        words = []
        states = None
        should_print = True
        while(num_words < MAX_NUM_WORDS):
            next_char, states = one_step_model.generate_one_step(
                next_char, states=states)
            result.append(next_char)
            idx += 1
            num_chars += 1
            if (next_char == ' '):
                num_chars = 0
                words.append(
                    tf.strings.join(result)[0]
                    .numpy()
                    .decode('UTF-8')
                    .strip()
                )
```

```
                num_words += 1
                result = []
                if num_words > MIN_NUM_WORDS:
                    tag = nltk.pos_tag(words)[-1][1]
                    if 'NN' in tag or 'VB' in tag:
                        break
            if (next_char == '-'):
                num_chars = 0
            if num_chars > MAX_NUM_CHARS:
                should_print = False
                i -= 1
                break
        i += 1
        if should_print:
            full_string = ' '.join(words)
            if SIMILARITY_METRIC:
                similarity = similarity_metric(full_string)
                print(similarity)
            print(full_string)
    return


# Configs
RNN_UNITS = 512
EPOCHS = 100
NUM_OF_EXAMPLES = 100
MAX_NUM_CHARS = 12
MAX_NUM_WORDS = 7
MIN_NUM_WORDS = 3
SIMILARITY_METRIC = True
TEMPERATURE = 1.0
NOISE_WEIGHT = 0
CHECKPOINT_PATH = f'./final_units{RNN_UNITS}_epochs{EPOCHS}'

if __name__ == "__main__":
    pretty.install()
    gpus = tf.config.experimental.list_physical_devices('GPU')
    if gpus:
        # Restrict TensorFlow to only allocate 6.2GB of memory (3070)
        try:
            tf.config.experimental.set_virtual_device_configuration(
                gpus[0], [tf.config.experimental.VirtualDeviceConfiguration(
                    memory_limit=6400
                )]
            )
            logical_gpus = tf.config.experimental.list_logical_devices('GPU')
        except RuntimeError as e:
            # Virtual devices must be set before GPUs have been initialized
            print(e)
    main()
```

A.2 model.py

```
# Model for Final
import tensorflow as tf
import nltk
```

```
import gensim
import numpy as np
from jellyfish.cjellyfish import damerau_levenshtein_distance
import multiprocessing as mp
import signal


class GenerativeGRU(tf.keras.Model):
    '''
    A GRU based model meant to generate text. It has an embedding layer
    on the inputs and a dense layer on the output.
    '''
    def __init__(self,
                 vocab_size,
                 embedding_dim,
                 rnn_units):
        super().__init__(self)
        self.old_weights = None
        self.embedding = tf.keras.layers.Embedding(
            vocab_size, embedding_dim, mask_zero=True)
        self.gru = tf.keras.layers.GRU(
            rnn_units, return_sequences=True, return_state=True)
        self.dense = tf.keras.layers.Dense(vocab_size)

    def call(self, inputs, states=None, return_state=False, training=False):
        x = inputs
        x = self.embedding(x, training=training)
        if states is None:
            states = self.gru.get_initial_state(x)
        x, states = self.gru(x, initial_state=states, training=training)
        x = self.dense(x, training=training)

        if return_state:
            return x, states
        else:
            return x

    def apply_random(self, noise_weight):
        for layer in self.dense.trainable_weights:
            noise = tf.random.normal(layer.shape) * noise_weight
            layer.assign_add(noise)


class OneStep(tf.keras.Model):
    '''
    A one step model wrapper that assumes an RNN based input to generate
    one step of output data. This can be done continuously in a loop to
    generate output based on previous state.
    '''
    def __init__(self,
                 model,
                 chars_from_ids,
                 ids_from_chars,
                 temperature=1.0,
                 noise_weight=0.0):
        super().__init__()
        self.temperature = temperature
```

```
        self.model = model
        self.chars_from_ids = chars_from_ids
        self.ids_from_chars = ids_from_chars
        self.noise_weight = noise_weight

        # create a mask to prevent "" or "[UNK]" from being generated
        skip_ids = self.ids_from_chars(['', '[UNK]'])[:, None]
        sparse_mask = tf.SparseTensor(
            # put in -inf at each bad index.
            values=[-float('inf')]*len(skip_ids),
            indices=skip_ids,
            # Match the shape to the vocabulary
            dense_shape=[len(ids_from_chars.get_vocabulary())]
        )
        self.prediction_mask = tf.sparse.to_dense(sparse_mask)

    @tf.function
    def generate_one_step(self, inputs, states=None):
        # Convert strings to token IDs
        input_chars = tf.strings.unicode_split(inputs, 'UTF-8')
        input_ids = self.ids_from_chars(input_chars).to_tensor()

        # Generate noise and apply to model weights
        self.model.apply_random(self.noise_weight)

        # Run model
        predicted_logits, states = self.model(
            inputs=input_ids, states=states, return_state=True)

        # Only use last prediction
        predicted_logits = predicted_logits[:, -1, :]
        predicted_logits = predicted_logits/self.temperature
        # Apply the prediction mask
        predicted_logits = predicted_logits + self.prediction_mask

        # Sample the output logits to generate token IDs
        predicted_ids = tf.random.categorical(predicted_logits, num_samples=1)
        predicted_ids = tf.squeeze(predicted_ids, axis=-1)

        # Convert from token ids to chars
        predicted_chars = self.chars_from_ids(predicted_ids)

        return predicted_chars, states


class DamerauLevenshteinSimilarity():
    '''
    Uses the levensthein distance in a multiprocessed fashion to
    compute scaled word similarity to string length.
    '''
    def __init__(self, dataset: list, num_sim=1):
        self.dataset = dataset
        self.num_sim = num_sim
        if num_sim == 0:
            self.num_sim = len(dataset)
        self.pool = mp.Pool(mp.cpu_count(), self._init_worker)

```

```
118         @staticmethod
119         def _init_worker():
120             signal.signal(signal.SIGINT, signal.SIG_IGN)
121 
122         def __call__(self, phrase):
123             similarities = []
124             try:
125                 similarities = self.pool.starmap_async(
126                     self._similarity,
127                     [(phrase, s) for s in self.dataset]
128                 )
129                 similarities = np.array(similarities.get(60))
130             except KeyboardInterrupt:
131                 print("Interupted!")
132                 print("Cleaning up pool...")
133                 self.pool.terminate()
134                 self.pool.join()
135                 exit()
136             except Exception as e:
137                 print(type(e), e)
138                 print("Cleaning up pool...")
139                 self.pool.terminate()
140                 self.pool.join()
141                 exit()
142             top = similarities[np.argsort(similarities)[-self.num_sim:]]
143             return np.mean(top)
144 
145         @staticmethod
146         def _similarity(s1, s2):
147             len1 = len(s1)
148             len2 = len(s2)
149             maxDist = min(len1, len2) + (max(len1, len2) - min(len1, len2))
150             distance = damerau_levenshtein_distance(s1, s2)
151             return (maxDist - distance) / maxDist
152 
153         def __del__(self):
154             self.dataset = None
155             if self.pool is not None:
156                 self.pool.close()
157                 self.pool.join()
158             self.pool = None
159 
160 
161     class TFIDFSimilarity():
162         def __init__(self, dataset: list):
163             gen_docs = [nltk.word_tokenize(p) for p in dataset]
164             self.dictionary = gensim.corpora.Dictionary(gen_docs)
165             corpus = [self.dictionary.doc2bow(p) for p in gen_docs]
166             self.tf_idf = gensim.models.TfidfModel(corpus)
167             self.sim = gensim.similarities.docsim.SparseMatrixSimilarity(
168                 self.tf_idf[corpus], num_features=len(self.dictionary))
169 
170         def __call__(self, phrase):
171             query = [w.lower() for w in nltk.word_tokenize(phrase)]
172             query_bow = self.dictionary.doc2bow(query)
173             return np.mean(self.sim[self.tf_idf[query_bow]])
```

A.3 util.py

```python
import pandas as pd


def load_all_the_data(path: str):
    # Need to load in the data... as a csv
    df = pd.read_csv(path, header=None, names=['names'])
    df = df.astype('str')
    df = df['names'].str.lower()
    return df.to_numpy().flatten()
```

Appendix B

Bulk Results

```
yellow recluse eye stongs
qeerogsl's deadeyed waoler stiff
qounchriny stompers basket sixew
tamts of brit'r ssawedation
xatians l'a-esa chest:uas gloves
unamakged scepter of the madiannusc
exhemeally feathers noder clawsengess
iron plate preter toben
kifis tar smalon power
qeenod wine camarll hines
zinnedrilinh scale armor camel
mrsxringing gloves of rathless
croshin's hilemen gloves molders
welsedscole leggings moterial of warmedyysy
farlath chainstole shordswords dangs
yoros supposted filemonther buakdian
azpelmested sabotons of eledenss
yeti furnel oubus taskination
jaylworers boots materials tabnrysto
lawgest koron essoncerst samer
xriating trousers of the desilnceated carqaines
ziddessbakes rock carry feadsiwoape
wanderer's ayr`dal chain arms
large white rocket cluster
zer'p fise staundern steaks
nature's shatkered chawber xhooling
repairsbled beltsoon letters of thaumaturay
osserub'l tabletro-kemerfe-knaithesh leash spear
etheream collxron materials lait
```

glyph of ice traplersx
grosm's treasure boutless kecoratoos
legguards of the golden kong
tome of scopchus thord
tome of aommol hevocl
exigenc perelous axtisan materials
barbindleepes postagdoc crest flames
unholysteddyd facusing candel blood
aptymizingsy souncgreanes of the breating
turgisswind fontu tambless cconebar
xaggar's lebb arenkerass findings
zabru's raie capped bnakleet
qeanist sandals of abaneness
windworf handace seil speel
omar's field hurkingsand fluweeaess
scroll of beastwood ash
xetersity gargetsing geivestane chest
timeworn overshirting greathelmer teshancsess
orramentinl silverl fidepionsted ineatishs
zan ares of o
markle marmitun scales vesteriby's
platinum embossed worked potion
cloverseskes kut pagg seembile
xunaar storage crateatasion garebackos
frayed-hirorally warfer stonesian stamuothur's
zaidseacet drumelle shosser stakfyess
nodklisk champer keyidian poison
veiled silk helm seal
high warlord's leather vestmented
kazanhs a'lory falling lightning
ancient stone bemisd bone
hydro-arvorfaes slippers of maskadoshue
blood malts heartr siuneterests
ironterningly discipli's shoet titatisn
bent capspited black sapphire
encrypted letter fool's jithsan
kluve of aniles pe
threshadon tooth senpens's poison
qoolanyy spear of the flamesing
qeanyst klade netainstole ifol
kurnysyeger's robes of the decimators
unaggrevaally striked chompiggs patterya-sk
frozen armor eshoraement boadstroker
accelerated degpa calderstone ombles
timeless mail gruave pielisgel
untagneto nnterbizeres emeraldstokm chestpiece

handguno aarinj cowolets ok
jhise's shardsofer skingerb streamsise
kej steel longblade of the decimator
jretiniciols afmiracts everyd defansnioss
logdlingssion staff zide jick
bhitehenrdy vendrated casery ginger
nluveed's headguard of command
kadris's porer belterved egestersion
distilled grade baloustes on the decimators
bar of skint foumura
silvril plate visor stoceang
vir'pateses toxel prutection oferardrite
tome of silver molfyning
wacches's elemental spellblades wacemones
condomally infused vescment moonted
krompg's freemom faittssuel'r bandsox
yun legging male skikfcest
bloodvane leggings molde shedl
owpenldy funsilla toes teak
volatilized scalesard girdlebach mogezeave
mencakess mochiaer pikited head
footed wolfskote gagnterity bidle

Bibliography

[1] In: *EverQuest Fanbyte* (). URL: https://everquest.allakhazam.com/dyn/items/.

[2] Dario Amodei et al. *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin*. 2015. arXiv: 1512.02595 [cs.CL].

[3] Alberto Bartoli et al. ""Best Dinner Ever!!!": Automatic Generation of Restaurant Reviews with LSTM-RNN". In: *Institute of Electrical and Electronics Engineers* (Oct. 2016). DOI: 10.1109/WI.2016.0130. URL: https://ieeexplore.ieee.org/document/7817147.

[4] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[5] William Chan et al. *Listen, Attend and Spell*. 2015. arXiv: 1508.01211 [cs.CL].

[6] Coderasha. "Compare documents similarity using Python — NLP". In: *Dev* (Sept. 2019). URL: http://dev.to/coderasha/compare-documents-similarity-using-python-nlp-4odp.

[7] Eleadon. *WoW-API-Libraries*. 2015. URL: https://github.com/Eleadon/WoW-API-Libraries.

[8] Ícaro Goulart Faria Motta França, Aline Paes, and Esteban Clua. "Learning How to Play Bomberman with Deep Reinforcement and Imitation Learning". In: Nov. 2019, pp. 121–133. ISBN: 978-3-030-34643-0. DOI: 10.1007/978-3-030-34644-7_10.

[9] Alex Graves and Navdeep Jaitly. "Towards End-To-End Speech Recognition with Recurrent Neural Networks". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Bejing, China: PMLR, June 2014, pp. 1764–1772. URL: http://proceedings.mlr.press/v32/graves14.html.

[10] gwicks.net. *USA English Dictionary*. 2011. URL: http://www.gwicks.net/dictionaries.htm.

[11] Hugsy. *English Adjectives*. 2017. URL: https://gist.github.com/hugsy/8910dc78d208e40de42deb29e62df913.

[12] Navdeep Jaitly et al. *A Neural Transducer*. 2016. arXiv: 1511.04868 [cs.LG].

[13] Prasad Kawthekar, Raunaq Rewari, and Suvrat Bhooshan. "Evaluating Generative Models for Text Generation". 2017. URL: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1174/reports/2737434.pdf.

[14] Yuxi Li. *Deep Reinforcement Learning: An Overview*. 2018. arXiv: 1701.07274 [cs.LG].

[15] *Magic Items for Dungeons and Dragons Fifth Edition (5e)*. https://www.dndbeyond.com/magic-item. Accessed: 2021-02-08.

[16] Sanidhya Mangal, Poorva Joshi, and Rahul Modak. *LSTM vs. GRU vs. Bidirectional RNN for script generation*. 2019. arXiv: 1908.04332 [cs.CL].

[17] Ian Osband et al. *Behaviour Suite for Reinforcement Learning*. 2020. arXiv: 1908.03568 [cs.LG].

[18] M. Phi. "Illustrated Guide to LSTM's and GRU's: A step by step explanation". In: *Towards Data Science, Medium* (Sept. 2018). URL: https://towardsdatascience.com/illustrated-guide-to-lstms-and-grus-a-step-by-step-explanation-44e9eb85bf21.

[19] D. Sage and M. Unser. "Teaching Image-Processing Programming in Java". In: *IEEE Signal Processing Magazine* 20.6 (Nov. 2003), pp. 43–52. URL: http://bigwww.epfl.ch/publications/sage0303.html.

[20] H. Sak et al. "Learning acoustic frame labeling for speech recognition with recurrent neural networks". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015, pp. 4280–4284. DOI: 10.1109/ICASSP.2015.7178778.

[21] Elizabeth Salesky et al. *The Multilingual TEDx Corpus for Speech Recognition and Translation*. 2021. arXiv: 2102.01757 [cs.CL].

[22] Lifeng Shang, Zhengdong Lu, and Hang Li. *Neural Responding Machine for Short-Text Conversation*. 2015. arXiv: 1503.02364 [cs.CL].

[23] Rosaria Silipo and Kathrin Melcher. "Product Naming with Deep Learning". In: *Datanami* (Apr. 2019).

[24] *String Comparison*. https://jellyfish.readthedocs.io/en/latest/comparison.html. Accessed: 2021-04-20.

[25] Wikipedia contributors. *English phonology — Wikipedia, The Free Encyclopedia*. [Online; accessed 12-February-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=English_phonology&oldid=1006107464.

[26] Ho Chung Wu et al. "Interpreting TF-IDF Term Weights as Making Relevance Decisions". In: *ACM Trans. Inf. Syst.* 26.3 (June 2008). ISSN: 1046-8188. DOI: 10.1145/1361684.1361686. URL: https://doi.org/10.1145/1361684.1361686.

[27] Tom Young et al. *Recent Trends in Deep Learning Based Natural Language Processing*. 2018. arXiv: 1708.02709v8 [cs.CL].