

# **Analysis and Implementation of Narrative Dialogue Engines Used in Procedurally Generated Games**

Ahad Khan  
KHA20776276



**UNIVERSITY OF  
LINCOLN**

2511778@lincoln.ac.uk

School of Computer Science  
College of Science  
University of Lincoln

Submitted in partial fulfilment of the requirements for the  
Degree of BSc(Hons) Games Computing

*Supervisor:* Dr. Olivier Szymanczyk

April 2023

## Acknowledgements

I would like to begin by thanking my parents, Kishwar Sultana and Dr. Muhammad Zafar Iqbal Khan for allowing me to be in the privileged position to study and excel in a field that both of them believed I could achieve great things in. Thank you for always believing in me and supporting me during every struggle I have had during my academic journey and celebrating every victory no matter how small.

I would also like to thank my siblings Dr. Fatima Zafar and Dr. Maryam Zafar for helping me when I was struggling being away from home and for believing in me even in times where I did not believe in myself.

My friends and peers and new found family that I have had the honor of working alongside these three years, I could list them all, but the list would become excessive. I appreciate every moment that we pushed one another to become the best versions of ourselves we could possibly be.

I would like to thank Dr. Olivier Szymanczyk for assisting me in the earlier stages of this dissertation, and for supporting the completion of this paper.

Lastly I would like to dedicate this dissertation to my nephew Mustafa, who passed away at the end of December this year. All my successes and ambitions have been watched over and I am blessed to have been his uncle.

## **Abstract.**

Creating an algorithm that outputs believable dialogue and text descriptions of items and scenery to be implemented in an RPG rogue-like game. To allow for a more immersive and enjoyable game experience that implements procedurally generated content into a more sustainable, traditional role-playing game experience. Focusing on implementing intentional dialogue lines and descriptive writing using machine learning and artificial intelligence to automate the process.

<b>Chapter 1 Introduction</b>	<b>5</b>
<b>Chapter 2 Literature Review</b>	<b>8</b>
2.1 Aims and Objectives	8
<b>Chapter 3 Requirement Analysis</b>	<b>12</b>
Transformer, Deciding between RNN and LSTM, and Prompt Generation	12
<b>Chapter 4 Design &amp; Methodology</b>	<b>14</b>
4.1 Project Management and Risk Analysis	14
4.2 Toolsets and Machine Environments	19
4.3 Design	24
4.4 Research	27
<b>Chapter 5 Implementation</b>	<b>36</b>
5.1 text_generator.py	36
5.2 gpt.py	50
<b>Chapter 6 Results and Discussion</b>	<b>52</b>
<b>Chapter 7 Conclusion</b>	<b>53</b>

## Chapter 1 Introduction

The current landscape for narrative based video-games focuses heavily on the production of story and description through a manual process. Consumers don't have the luxury of all narratives being epic sagas or short sweet harmonies alongside the video games that people play. A game's success relies heavily on how enjoyable it is to play and interact with, and games that set aside narrative as an afterthought can leave the player wanting more. The idea of immersion in video games has become a hot topic in the industry, worldbuilding, creating non-playable characters that have believable dialogue, and descriptions of the history and background of the items and areas that are interacted with.

Designers can help influence the atmosphere of a game through the dialogue of characters within their world, popularising the role of a Narrative Designer, ambient behaviours and animation loops, cohesively working together. There are entire genres of games that dedicate themselves to a handmade experience focusing on text-based narratives e.g. Roadwarden (Moral Anxiety Studio, 2022), and Zork (infocom, 1994). Games that rely solely on descriptive language and dialogue from NPCs to entice the player into suspending their belief and choosing to be immersed into the world that the creators have created for them. In contrast, games in recent years have begun to use procedural generation to create the main gameplay loop of their games. In a recent interview with French developers at Eidos and Bethesda, creators of Deathloop (Arkane Lyon Studios, 2021), the procedural generation ideas of multiplayer allow for an aspect of the player experience to be repeated much more than normal. That being whenever the player turns a corner or enters a new area, they are in for a different experience every time. Taking inspiration from the original PCG game Rogue in the 1980s, the content being produced is enticing because of its unpredictability. However, it merges with the industry norms of narrative devices such as the very NPCs it creates, leaving a hollow and very binary emotional response where the characters themselves are unempathetic or alien to their surroundings.

A relatively new term coined by **Clint Hocking**, ludonarrative dissonance (Clint Hocking 2007), the idea that the gameplay refuses to coincide with the narrative storytelling, pulls the player out of the total immersion that they are given, the internal conflict that occurs when the player is given opposing experiences from what they are interacting within the game itself and the narrative devices portraying something completely different.

This is one of the most prominent issues game designers are having within the procedural content space. The highlight failure of one of the highest-grossing games in recent times is **Hello Games' No Man's Sky** (Hello Games, 2016). One of the sole critiques was that there was so much content promised of expansive worlds, infinite exploration possibilities, and countless randomised instances, resulting in a lacking narrative experience and overall world design that felt artificial in nature. It wasn't unique or alluring enough for players to care about what they ultimately labelled as a procedurally generated universe. Unlike something made entirely by hand like **Fullbright's Gone Home** (Fullbright, 2013), which gave the player experience something much more humane, and tangible with character dialogue that feels real. Perhaps this is evidence that procedural generation has a long way before it can ever overtake complete manual generation.

The appeal for automation is still alive however, with companies spending millions on attempting to create the next big AAA game, anything that reduces the time spent in the production pipeline can be beneficial to the industry as a whole. On a much smaller scale, and the one this project will be focusing mainly on is the independent games scene, one that is notorious for an oversaturation of excess. Games that are in interaction unique, but in narrative experience lacking.

This project aims to solve some of these issues by creating a software application that will output text descriptions and dialogue lines for NPCs and

backgrounds in a role-playing video game, to allow for the automation and reduction in production cost of creating a believable dialogue engine and description generator. Aiming to use machine learning algorithms and narrative engines from already constructed descriptions to teach an AI to create an output that could potentially help the world-building process. Abstracting what it means to the player when they experience an immersive interaction.

The following pages will ascertain the aims and objectives, requirements for development and analysis as well as design and methodologies, the actual software implementation, its results and an overall evaluation of the findings.

## **Chapter 2 Literature Review**

### **2.1 Aims and Objectives**

Procedural content generation focuses on creating an exclusively unique experience within a varied set of genres. Specifically, in games, we can choose to look at roguelikes as a passageway for intelligent machines to calculate an outcome from a set series of rules that it is given to create a rich player experience through emergent behaviours, allowing for scenarios that are not necessarily hard-coded into the game, giving a different output each time it is run (Fredriks and DeVries, 2021).

This project endeavours to create a dynamic dialogue generator that runs off a trained AI using open-source narrative and online resources such as Twitter to create a foundation for a grammar-based system such as Tracery or Twine (Compton, K. et al, 2015). The dialogue algorithms possible to use will vary throughout testing, but the AI itself can be trained using TensorFlow libraries within Python and using third-party libraries that can integrate GPT inputs and their word databases. Abstracting the problem, there is an AI with several nodes indicating new inputs based on user inputs and the output that the program should give and a continuous exchange to create this spoken word dialogue. With two agents interacting with each other, what makes it more effective than a hardcoded dialogue system like that within more traditional RPGs (like Elder Scrolls, Final Fantasy), is that the agents can dynamically interact and change their view each time they interact with a new output, and the game will output a new response each time, unlike a set dialogue tree which while still effective, enforces a limited playing experience (Siegel and Szafron, 2009). A further point found within Siegel and Szafron's work is that the density and overall length of a conversation and description output are much more nuanced when it comes to actual dialogue. One model may be better at outputting shorter dialogue interactions than another but begins to falter when longer conversations are being attempted. This helps to



reinforce that the system is still not perfected and that even at a more basic level of single-line descriptions an attempt can be made to replicate and reproduce actual content that is normally hard coded.

In the conference paper from Gary Kacmarcik (Kacmarcik, 2021), he describes the normal NL (natural language) model that usually appears within AAA games to allow for a reinforcement of the dynamic nature of the interaction between player and agent. The effort used to hold up the illusion of expressiveness granted to the player is immense, having to account for a plethora of scenarios and variables that would eventually break the immersion, changing what should be a rich interaction to that of pure frustration. The option he proposes and the one that the project will be adapting from is dynamically creating these dialogue menus that take into account context and coded attributes as inputs to make a more intuitive response system. Kacmarcik achieves this by using interaction candidates, essentially separating the dialogue response into parts of information that the agent will be responding with. The scenario below is an example of pseudocode to recreate if the player is attempting to pick up a block of water which is also used as an environmental obstacle within the level. This is while also taking into account the character's knowledge base which is what the sub-conditions imply.

```
1 BE AGENT1
2   OBJ = TRUE
3   SUB_CONDITION = PLAYER_NAME + 's'
4   CONDITION = GET ITEM
5   [ITEM]
6   SUB_CONDITION = YOU
7   OBJ = WATER
8   ATTRIBUTE = OBSTACLE
9
10 BE AGENT2
11  OBJ = TRUE
12  WHILE OBJ IS TRUE
13    SUB_CONDITION = OBJ*
14    CONDITION = AGENT1*
```

This is aimed to be achieved by feeding in these prerequisite conditions through a database table of values of randomly generated characters. Although not fully random, it is a much more achievable prospect given my capabilities.

In terms of a way to showcase just how the NPC Knowledge Base (KB), and PlayerCharacter KB are formatted and for ease of comprehension knowledge graphs can be used to display the pipeline process, in a similar way to Prithviraj Ammanabrolu (Ammanabrolu et al, 2020). They clearly outline the tools used within their work pipeline:

Holding the prerequisite information within plot details, separating various key characters and objects as unique entities, generating a prompt for the algorithm, asking the question about the entity, then running that through the AI and outputting the final solution.

Using these examples to introduce further aspects which will be beneficial to the project and development itself, Ammanabrolu discusses a pre-trained version of GPT-2 which is finetuned to a corpus of plot summaries collected from Wikipedia. This project will be using a similar model to make an output for its

descriptive and dialogue outputs. The basis of a Recurrent Neural Network (RNN) is to train itself off of a dataset of fantasy dialogue and/or game items, this can also be used to predict the next characters or pieces of text desired for the output.

An example of this being used in the industry can be seen within CTRL (Keskar et al, 2019). The system that Salesforce and OpenAI created uses controlled training of large-scale datasets being fed through their conditional language model, which is then tokenised to relay to its validation data. All while looking for an output which is contextualised or suitable for a given instruction.

They do so by incorporating a control code mechanism, which essentially acts as prompts for the users to declare the desired behaviours and tailors the output to match the specific prompts. Within their study, they chose news headlines as the large data set, and their control codes were that of the genre, review and rating. The output always signifies a learned response and is heralded as a breakthrough for natural language models and generating text and content which can be used in a plethora of applications.

Although the project's scope and complexity are scaled-down compared to the previous example, it still has the same foundations for the desired transformer process and output. Taking a dataset of fantasy dialogue from games or books, and/or items within a roleplaying game that can be used as the training data to be put through the natural language model such as a Tensorflow NL model, setting the validation data and generating an output of items and characters for the output to be stored as a prompt. The GPT functionality will allow the use of this prompt and set instructions based on the context that is desired e.g. "Create an item description for [prompt] within a fantasy game setting."

## **Chapter 3 Requirement Analysis**

### **Transformer, Deciding between RNN and LSTM, and Prompt Generation**

- The main requirement of having a development environment which is compatible with Python and several third-party packages and libraries which are core to development.
  - Pycharm
  - VS Code
  - Google Colab etc.
- From the research that has been carried out for the project, one of the first goals is to create a transformer model which takes data from a user-given dataset, primarily that of fantasy items or fantasy characters from an open-source website such as Kaggle or Google Scholar Databases.
  - Transformer - a deep learning architecture which is used to complete natural language processing tasks with its inputs in parallel. Encoding the input sequence to create a learned response to its context, then decoding it for its output. [e.g. Keras, Tensorflow, sklearn, etc.]
- A suitable dataset that can be read into the development environment through a text file or CSV file.
- There needs to be a suitable Artificial Neural Network (ANN) (Russel and Norvig, 1997) chosen for the task at hand, relevant options are mainly between Recurrent Neural Network (RNN) and a Long Short-Term Memory (LSTM) model.
  - RNN
    - Better for simpler tasks, and smaller datasets which don't require an imminent need for context and access to a plethora of example data. Has the benefit of being easily

analysed as the information layers are processed sequentially, step-by-step.

- LSTM
  - Better for more computationally complex tasks, which require context based on natural language and the input sequences that are being fed into it. This can often be seen within sentiment analysis - where text is analysed to see if the input is in a positive, neutral or negative tone (Kharde and Sonawane, 2016), or text generation.
- GPT-compatible libraries and access to pre-trained open-source generation for the output that the transformers and ANNs are creating, with a prompt input which is declared by the user e.g. “Generate a description for [X] in a fantasy setting” where X is the generated text generated output from the model
- Ensure that the final output is accessible to a text-based game engine e.g. Twine, Tracery etc.

## Chapter 4 Design & Methodology

This section explores the tools, project management techniques, methodologies, and implemented approaches that were researched and employed throughout the project's development to ensure its success.

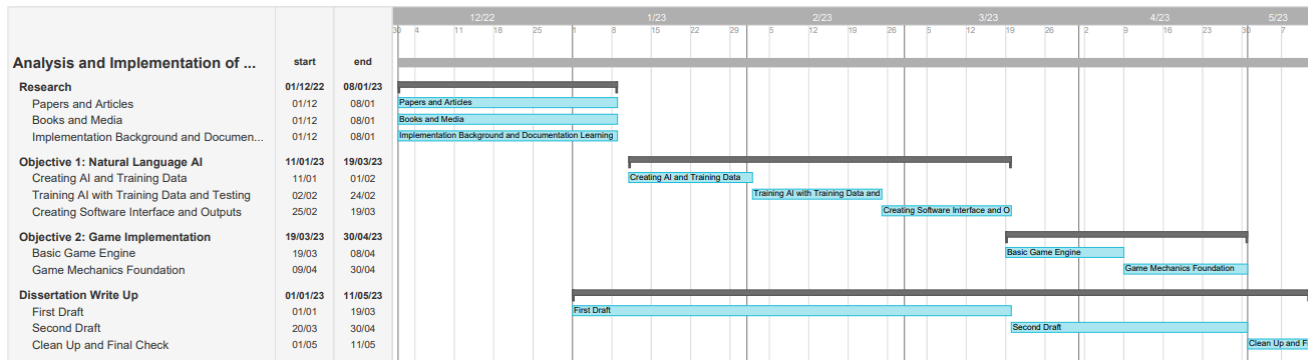
### 4.1 Project Management and Risk Analysis

The methodologies for project management should accommodate the aims and objectives within the scope and demands of the study that will lead to a success.

The study attempts to create a resource for game development which aids in the expansion of world-building and narrative engines within independent games that have limited resources and budgets to achieve such tasks. The way to measure this is completely qualitative, and the rate of which can feel very erratic when it comes to the evolution of the definition of world-building itself. From the words of Christian McCrea:

*“This is a productive confusion. What aspiring game designers and developers now see as world-building reflects this cloudy episteme. In the context of independent game development and an ever-diffusing information environment, worldbuilding’s definition is changing so quickly that the scholarly response has to be equally rapid.”* (McCrea, 2020)

From this objective, a discussion can be had about optimal risk assessments and mitigations and project management techniques, formats and scheduling.



**Fig.1** Original Gantt chart showcasing the timeline and schedule of the project and its implementation. (TeamGantt, 2009)

Figure 1. Showcasing the original Gantt chart created at the beginning of the project, it became clear that the chart did not accurately represent the time taken to complete the study, and the development methodologies that were employed.

Pressures from external sources, such as workloads outside of the general scope of the project, for example, assessments, revision for exams, lectures, and personal matters including family obligations caused a severe effect on the overall consistency of the schedule that had been set.

What was originally going to be a **Waterfall Methodology**:

It was planned from the start of the research to the end of development and write-up of the report, the logic and chronology of the system seemed like an adequate model to base the process of the study on.

The waterfall model aims to break down the entire project into tasks which are led through linear, sequential phases, one by one. *“Another reason for the waterfall approach's dominance is the logic of its structure. Projects proceed logically from feasibility through requirements analysis, a design phase, an implement-and-test (often involving programming) phase, through acceptance*

*and after delivery into the maintenance phase of the life cycle. This sequential model is simple, and the sequence of steps is easy to grasp.” (Light, 2009)*

The expectation is that tasks are being completed on time and that the chronology of the entire system of work allows for development to be focused and treat tasks as an ‘in-the-moment’ scope. It aims to accomplish a product for delivery into the maintenance phase of development as cleanly as possible. However, this becomes a very idealistic project model, not being able to revise or refine any pieces of work that may be an obstacle during development phases, creates delays in troubleshooting and ironing out the issues that eventually occur during a later stage.

Studies have shown different ways of solving the issues that the Waterfall method encounter, one of the ideas that companies were attempting during the 2000s was altering the system slightly to incorporate sub-systems and a recursive step similar to Agile, that allowed for sectors of development to be repeated or extended in the case of obstacles that needed to be overcome during development, being coined the “sashimi model”. (McConnell, 1996)

The ‘final model’ from Winston Royce became very convoluted and became redundant to the extreme programming model. The key differences were that the tasks should be completed twice if possible and that all testing is planned beforehand while keeping in full contact with the customer throughout development. (Ji and Sedano, 2011)

The eventuality of the methodology and the context of the external issues that were occurring during the development phases of the study, another methodology had to be used.



### The transition to **Extreme programming**:

An agile software development methodology that focuses on iterative development, continuous changing of planning, and test-driven development. These practices aim to create a more flexible and responsive development process that can adapt to changing requirements of the project, and if obstacles arise, then changes can be made on the fly.

Organising tasks by reducing the costs of changes in requirements by having several shorter development cycles, allowing for more flexible planning, instead of just defining a set of requirements that must be fulfilled before moving on in development cycles.

This is much better suited to the current state of the project, as the aforementioned factors caused obstacles within development and conflicts of schedule. So a more reactive methodology ensures that at least a suitable output can be generated and built upon further if required. The ability to adapt and reflect on the go keeps this study within the basis of creating a pipeline which is more suited to the average university work structure and the constraints placed upon the study.

This rapid development environment allows for the scope of the project to continually advance and remain malleable, adding functionality when it is required instead of based on self-set deadlines. Achieving an iterative process of code reviewing and testing based on desired results, and improvement on the competency of the software used and comfortability with its documentation [Tensorflow and GPT-Neo].

**Possible risks and ways to mitigate their effects:**

- The artefact and product are incomplete and untestable
  - Showcase what working parts there are, and the pseudocode of how the engines and transformers work, and show examples instead of actual outputs
- The artefact does not work as intended and produces results which are not useful to the study
  - An adjustment to the success criteria must be made to reflect the progress of the project
- Access to third-party libraries is unobtainable or not sufficient for the project
  - Try to simplify the context and tasks to be more manageable and use open-source libraries to create a less complex but still relevant output
- Lack of time to complete the project
  - Reassess the tasks that have been set, and perhaps simplify the desired results to keep on track of the time constraints set
- The acquisition of a student license for GPT and OpenAI may be challenging as there is high demand for the software that is provided, alternate software and open-source data can be used instead if any problems arise e.g. NLTK (Natural Language Tool Kit for Python)

## 4.2 Toolsets and Machine Environments

### Python

Has access to a rich ecosystem of libraries and frameworks built for machine learning and ANNs, like scikit-learn, TensorFlow, Keras, PyTorch etc. Offering pre-built functions, algorithms and utilities to make it easier to develop complex models with minimal effort.

The familiar and simple syntax which eases development and simplicity in machine learning code, focusing on readability reduces the cognitive load when working with complex algorithms and ANN architectures.

Python integrates well with other technologies used in the workflow of the study, scientific computing libraries such as NumPy and SciPy, data manipulation libraries, visualization libraries etc.

Several online resources can be learned from based on the models and systems that will be used in the study, smoothing out the complexity gradient, and making for an easier understanding of the development process

C++ and Swift were considered for the project as it is the programming language that familiar to the distribution, and like Python has access to similar libraries and external software needed to complete the tasks required, however, the tools available are much less extensive and less well documented than its counterparts. Industry-wise, Python remains the primary language for TensorFlow development.

### **JetBrains PyCharm** (Jetbrains, 2010)

PyCharm is an integrated development environment (IDE) focused on Python programming. It incorporates code analysis tools, visual debuggers and unit/version control testing.

Formats code using its intelligent code engine, allowing for auto-completion, code highlighting during debugging, and refactoring tools, such as mass variable manipulation. This allows for a more accessible development process and one that encourages optimisation through iteration

Has built-in integration for version control systems such as Git and Mercurial, allowing for the management of code to become less of a hassle. With the context of reverting changes when applicable, in case of obstacles or further optimisation.

One of the main benefits for this study was the access to an IDE that has cross-platform support, allowing for development on Windows and macOS devices, because of the busy schedule of the recent few months, being able to work in multiple environments and from multiple machines allowed for a more flexible development schedule.

Another major benefit of using PyCharm is its virtual environment feature, which allows for the project to be isolated within its own capacity, already built-in so no external installs are needed. Being able to change between Python versions actually became very useful because of the way TensorFlow and NumPy libraries in Python had depreciated, this was easily overcome with testing within different versions of the Python interpreter being controlled within the venv

PyCharm also allows for ease of installation of external libraries and packages through its own project interpreter settings which use PIP directly through an easy-to-understand UI instead of having to directly install to program x files in directories

More lightweight text editors were considered such as SublimeText3 and VScode by Microsoft, however, they did not allow for the auto-formatting and runtime compatibility that a fully integrated Python IDE allowed for

### **Github / GIT (CLI)**

GitHub is an online version control platform which allows for the storage and management of code repositories during the development of this project. With in-built systems to reduce the redundancy when uploading new versions to their respective repos

The ability to have access to the repositories from anywhere, allowed for more mobile development within university computer labs and from home.

Compared to other online storage systems such as OneDrive and Google Drive, Github/GIT is although slightly less streamlined, much more versatile in its ability for version control and push and pull requests

### **TensorFlow**

TensorFlow is an open-source library for machine learning and artificial intelligence. Its main use within the project will be to run computational models and train a recurrent neural network to give the desired output.

With built-in support for deep learning and more importantly neural networks, through the use of optimization algorithms and the ease of constructing

and training complex neural network architectures for natural language processing, or sequence-based output

TensorFlow also offers flexibility and a portable platform across multiple devices, using CPU, GPU and embedded devices, deployment and efficiency configurations on hardware are intrinsically optimized

Being able to use high-level APIs such as Keras, a machine learning interface that abstracts the process of training and developing an ML model. This allows codebases to stay relatively smaller, more concise and maintainable. Encouraging iteration of the models produced and with access to data visualization tools, a clearer picture of the training pipeline can be made

Specifically for RNN development, TensorFlow has a high-level API that includes pre-built RNN cell implementations including LSTM functionality. With support for sequential data inputs i.e. text input, transformed into a numerical representation through a variety of vectorization formulae and tokenization.

Training and optimization is as simple as choosing the optimization algorithm within parameters, TensorFlow handles the entire backpropagation process for you.

PyTorch or Scikit-learn (sklearn) are considered alternatives for the project, they lack support for unstructured data i.e. text, meaning a more specialised library was needed, as well as wanting to learn an industry-standard tool rather than one for more basic implementation.

### **Microsoft Excel / Google Sheets**

Microsoft Excel is an industry-standard proprietary data processing software, which is offered for free to students under their respective University

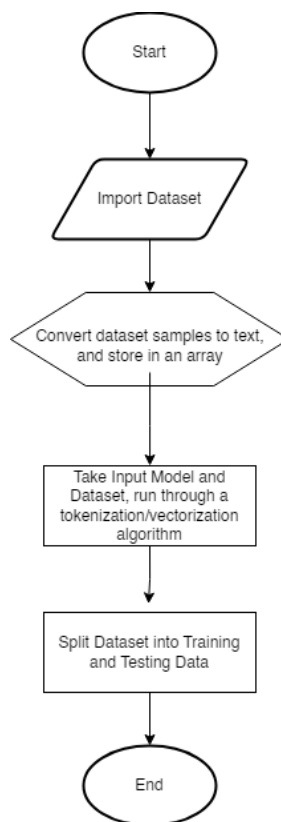
licenses. Enabling restructuring and reorganisation of data from the datasets that are being used within the study as well as helps with visualisation for our input and output.

Google Sheets is a similar proprietary software, with the inclusion of cloud-based functionality. Allowing for access to files from anywhere with access to the internet, allowed for the transfer of files to be seamless, it is however less feature-rich than Excel, so was used mainly at the beginning of development.

## 4.3 Design

The initial designs of the project were fairly simple and required an iterative process to refine and abstract tasks for the development pipeline.

To start with find a way to take textual input and create an algorithm that is capable of generating a learned response similar to that of the input dataset. If a focus is first put on the pre-process setup, making sure our inputs are valid and ready to be used within the RNNs later:

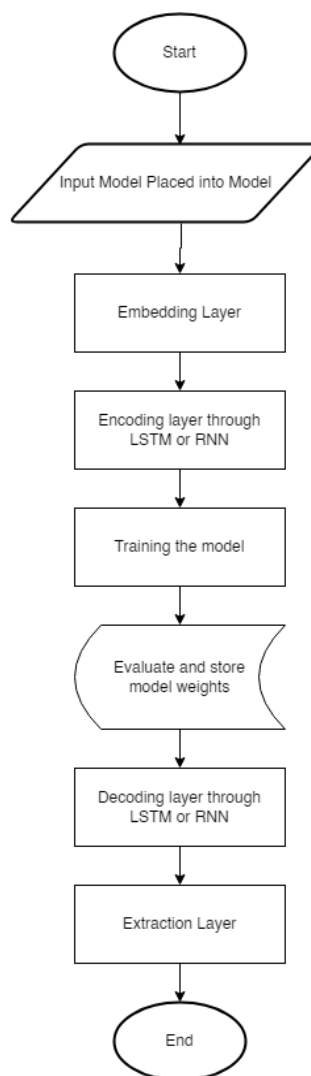


**Fig 2.** Flowchart for dataset preprocessing for the ANN model



This process will take our text inputs, convert them into numerical storage, and split the data to be fed into the training stage. This step is known as the normalization process, essentially, encoding the characters and strings into categorical variables, and splitting them to later evaluate the model's performance after being trained as a generative model.

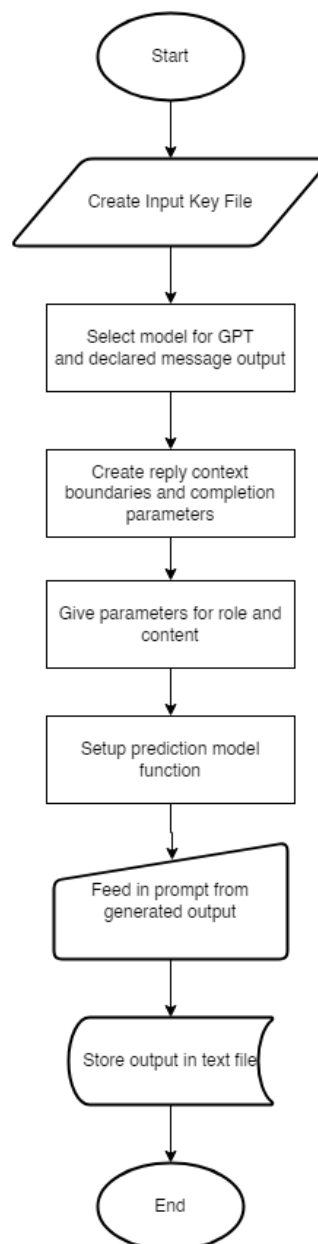
The next step is training the generative model using the dataset:



**Fig 3.** Flowchart for training and evaluating the model

The data after being evaluated is used to generate a new set of data, then goes through a post-process phase where constraints and minor adjustments are made to refine the model. Then going through the process of iteration. Improving the quality and diversity of the generated output.

After getting the generated output, it will be fed into the GPT API that is being given a prompt to declare an output for a character or description:



**Fig 4.** Flowchart for creating a GPT prompt feed and output to text

This is the final output design for the study, relatively simple functions using an API to create a user-generated prompt with parameters for role and context added into the equation. The output is finally saved in a text file for use within a text-based engine.

## 4.4 Research

The overall study aims to convincingly create and analyse descriptive text and dialogue about fantasy items, because of the non-quantitative nature of the project, the research mainly attains to the methodology and theory for the machine learning models and natural language being used within the video game industry.

### Dataset

The datasets that are being used within the study are sourced from Kaggle (Kaggle, 2010). The datasets are also all under CC0 1.0 Universal Copyright Law and CC BY-SA 4.0 (Attribution-ShareAlike 4.0 International), essentially no copyright infringements are possible as the work that is being used under creative commons has been dedicated to the public domain, and/or the data is being used in a way that is adapted and built upon for open-source educational purposes.

The first dataset used is from a fantasy massively multiplayer online role-playing game (MMORPG) developed and published by Jagex, a British games developer called RuneScape 3. With over 200 million accounts created for a free multiplayer game, the player base had a consistent flux of interactions and an active in-game economy.

The virtual economy, '*The Grand Exchange*' (abbreviated to GE), was a trading system for players to trade nearly every item within the games ecosystem for gold which could be earned from quests, looting etc. Nearly every item in the game came with a numerical value, and supply and demand came into effect with rarer items or resources and materials that were used more often by hardcore players during the 'endgame' stages of the player experience.

The player base for RuneScape categorises the gameplay loop as repeating monotonous tasks to increase experience and gain level-ups in their in-game skills to earn rewards and show off to other players. At the core of the activities players are required to complete, earning optimal experience points (XP) devolves into time cost efficiency always dominating resulting in purchasing resources that are required for the activity through the GE. For example, if a player wants to level up their fire-making skill they can opt to cut down trees themselves, or purchase logs off of the GE which the majority of the time is the most time-efficient way to gain XP.

Because this gameplay loop is the main driving force, players that come to these games with seemingly active player bases and it misses out on the narrative and worldbuilding aspects by only having brief descriptions of items. So it seemed like a perfect item dataset to include in this study for training and generating outputs.

The RuneScape dataset itself includes three CSV files:

- Runescape\_Item\_Names
  - Name\_ID: a unique identifier given to an item
  - Name: a text formatted name of the item
- Runescape\_Item\_Prices
  - Name\_ID: a unique identifier given to an item
  - Price: an integer value of the daily average price of an item

- Date: the date on which the average price of an item was collected, directly from the games live economy
- Runescape\_Item\_URLS
  - Name\_ID: a unique identifier given to an item
  - Name\_URL: official RuneScape grand exchange website URL for the associated items

The second dataset being used from another game series known as Final Fantasy, a turn-based fantasy role-playing game originally developed in 1987 in Japan. Unlike the previous example, the game focuses much more on its narrative and world-building through character dialogue and item descriptions.

The games are heralded as bringing vibrant, historic, and alive visions of their fantasy worlds to the screen. Each title in the series brings new characters and influences from the media, and reviews and sales signify that players will continue to come back to experience the worlds that are built in each iteration.

Although older entries of Final Fantasy take liberties when it comes to item and enemy descriptions within the games, the series had an evolutionary stage of expert writing within their narratives. In the 9th entry, the game revolved around mimicking the era of Shakespeare's theatrical worlds and the fantasy lore that has been built from previous entries.

This diverse nature of world design allowed the writers to create a narrative that not only focuses on dialogue from characters and the relationships between them but also uses in-game item descriptions to tell the player about certain areas and enemies.

One of the main things to point out about the dialogue systems in several older RPGs from the 2000s, the dialogue presentation was very similar to that of a play, and character-centric dialogue was a highly dominant form of portraying a

narrative. It wasn't until action games began to grow in popularity that more graphical forms of narrative storytelling were being used as a more exclusive form. This suits the study well as it is mainly focusing on character agents when it comes to descriptive speech. There is the option to amplify how a character would respond to an area based on an item in theory, however, in the scope of the project, there will be only a focus on if the generated output can be accurate, and useful to the development and narrative design process as a whole.

The dataset itself contains scripts from specific sections of the game series where multiple characters are speaking within a cutscene:

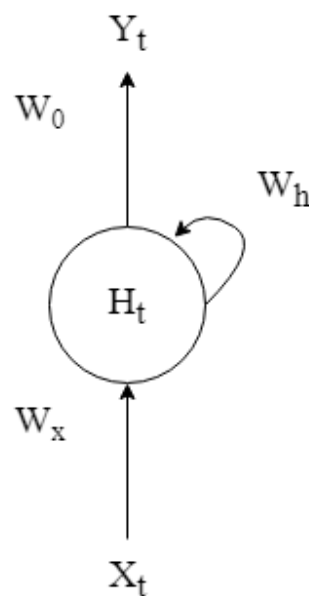
There are a total of 10 CSV files:

- (Dissidia) dff-operaomnia-intersecting-wills-script.csv
- dff-operaomnia-lost-script.csv
- dff-operaomnia-lost-script.csv
- ff5-vba-script.csv
- ff6-script.csv
- ff7-crisiscore-script.csv
- ff7-remake-script.csv
- ff7-script.csv
- kingsglaive-script.csv
- World-of-ff-script.csv
  - Original: base cutscene/gameplay dialogue string, in format with the character -> dialogue
  - Character: the character that is speaking, whether they are named or unnamed
  - Dialogue: the dialogue in an isolated string format
  - Wordcount: integer value of total wordcount of specific dialogue strings

### Algorithm/model design and selection

The architecture of a traditional recurrent neural network compared to a regular feedforward network is a folded inwards passage of X to Y, with a recursive layer for the weights. Essentially becoming multiple copies of the feedforward network looping in on itself.

Displaying an unfolded RNN, the loop of the conversions of independent activations into dependent activations can be showcased, providing the same weights and biases in all layers.



**Fig 5. Simple Recurrent Unit with Weights**

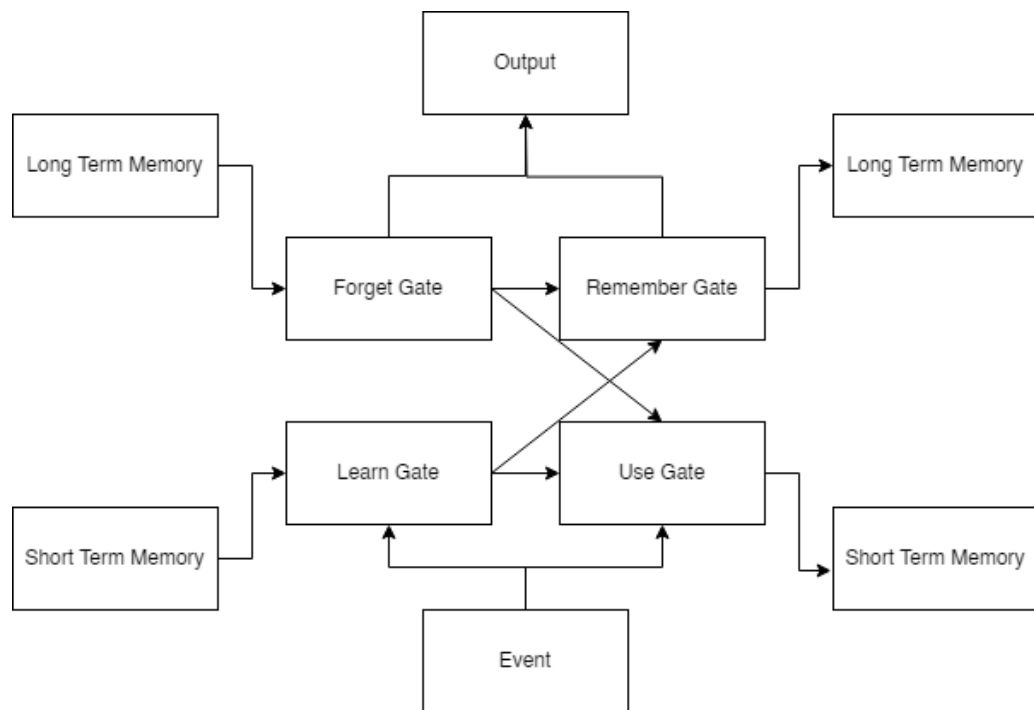
- $H_t$  - current state at  $t$
- $X_t$  - input at  $t$
- $W_h$  - weight at recurrent neuron

- $W_x$  - weight at input neuron
- $Y_t$  - output at  $t$
- $t$  - time step

The architecture for LSTM includes Long Term Memory (LTM) and Short Term Memory (STM), and abstracting the process, to improve effectiveness it uses gates

- Forget Gate
  - The LTM enters the forget gate to erase currently stored data that is deemed not useful
- Learn Gate
  - The current input and STM are merged to allow for the necessary information to be stored and used elsewhere
- Remember Gate
  - LTM data that needs to be stored, STM that is currently stored and the final event data that is stored working to update the LTM through phases
- Use Gate
  - Uses the data from LTM, STM and the Event to predict the output of current event then stores as an updated STM





**Fig 6. Abstracted representation of LSTM model architecture**

LSTM models are a type of RNN that are aimed at retaining long-term dependencies in data. They are architecturally more complex than regular RNNs and are often used for tasks such as language translation or modelling.

The key differences are just in how the two architectures are designed to handle memory and dependencies between time steps, the study will mainly be using a simple RNN model, however, for further development in the future LSTM may be a better-suited model for adding more functionality later on.

### **Parameter Tuning**

This section will be explaining the fine-tuning of hyperparameters within the ANN during each iteration of the model going through each dataset. In this scenario, there are several ways of measuring and evaluating how accurate a set

of results are as well as if the gradient of accuracy is increasing per epoch/iteration, or if it hit a standstill.

One of the main ways is using Keras to run an evaluate or prediction function to log the results of each iteration, an assignment of callbacks to set checkpoints is used to enable a summary view of the results graphically using numpy and pandas, having robustness and precision be our two main factors of tuning.

As explained earlier in the literature review and within the requirements analysis, the changes being made will be unique to a dataset depending on what goals are aiming to be achieved, for a dataset of items, there is less computational complexity, therefore adjustments can be made to the learning rates and details of embedding layers with more or fewer epochs than previous iterations. Changing the batch size is also an acceptable change to make the model learn at a faster rate, reducing the size can make the process faster, but causes higher variance in the validation dataset accuracy

The other section of tuning will be in the prompt that the GPT API is given to produce the in-game description and or assist in the way the generated output from the RNN can be used in a more retroactive way, i.e. the prompt can be formatted into a list instead of a series of strings, or input one by one instead of a mass compile.

### **Performance metrics**

The evaluation of the ANN models will be mainly from the results of its predictive text output and generative text output, comparing these metrics during separate stages of iterative development i.e. by taking into account the hyperparameters that are being changed in context with the datasets being used.

Perplexity is one of the metrics that could be used for this study, commonly used within language modelling tasks such as natural language translation. Since the algorithm and ANN model being used is sequential, the measurement of how well a predictive result matches the validation data is used.

$$P(W) = P(w_1, w_2, \dots, w_N)$$

Above is a generic probability of the chance of outputting a full sentence or in context case a full item name. For the study an extrinsic evaluation can be offered to look at the final loss of accuracy within the model, to tangibly see how different models and datasets interact with one another, depending on the parameters that are being changed within testing as mentioned prior.

As the information stored is not binary, such as a classification problem, choosing F1-score as a metric would be benign, and the same would go for any other categorical evaluation metric, the study should have a main focus on a loss of accuracy through its perplexity value, the lower the value the better the model is at predicting and outputting the next item, therefore is more accurate when generating the fantasy item names.

The generative text from the GPT API will have separate more qualitative metrics such as clarity, coherence, vividness etc. As these metrics are much more subjective and vary depending on the consumer, the accuracy will focus more on if the output is sufficient to be used inside a game for item descriptions.

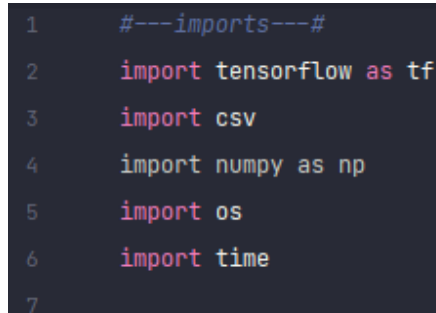
- Are the ideas presented logically to the context?
- Is the text engaging, more so than a simple description?
- Is the language used to describe the item improving on the world-building?

## Chapter 5 Implementation

In this section there will be a demonstration of the final solution from a technical perspective, outlining the different components of the software development. This will include the text generation and training of a neural network model, as well as generating a text description of an item through the use of GPT API.

### 5.1 text\_generator.py

Within text\_generator.py, the script aims to create, model, train, evaluate and output a generated text from an assortment of datasets that are mentioned as prior in 4.4. To prepare the interpreter and the project scope environment, libraries that the project uses had to first be imported:



```
1      #---imports---#
2      import tensorflow as tf
3      import csv
4      import numpy as np
5      import os
6      import time
7
```

**Fig 7.** Import statements of libraries and packages within text\_generation.py

TensorFlow is the main library used during the construction and training of the neural network model, as well as NumPy, being used for mathematical and string formatting processes. OS and time are used in the coordination of runtime processes, to give users an indicator for completion time of specific tasks. CSV is allowing Python to read and write from CSV files, which is essential to get relevant data from the datasets used in the study.

The code itself is structured inside a main procedure which is split into a series of chronological functions. Starting with preprocessing the data required for the model to function.

```
155     def main():
156         #---PREPROCESSING---#
157         csv_data = convert_to_list()
158         text, path = write_to_file(csv_data)
159         decoded, vocab = decode_py2(path)
160         id_from_chars, chars_from_id, chars = tokenization(vocab)
161         dataset = formatting_data(decoded, id_from_chars, chars_from_id)
162         training_sets = create_training_batches(dataset)
```

**Fig 8.** The main procedure within text\_generation.py showcasing 6 preprocessing functions

```
65     #---preprocessing---#
66
67     # reads text from csv file taking only the item names
68     1 usage
69     def read_item_data(file_path, column_index):
70         data = []
71         with open(file_path, 'r') as file:
72             reader = csv.reader(file)
73             for row in reader:
74                 if len(row) > column_index:
75                     data.append(row[column_index])
76             return data
77
78     # outputs text that has been put into a string list
79     1 usage
80     def convert_to_list():
81         file_path = "Runescape_Item_Names.csv"
82         column_index = 1
83         csv_data = read_item_data(file_path, column_index)
84         del csv_data[0] # to delete heading entry
85         return csv_data
```

**Fig 9.** read\_item\_data() and convert\_to\_list() within text\_generation.py

The first two functions `convert_to_list()` and `read_item_data()` are used to extract and reformat the data to be usable within a vectorized array. First begin by taking the csv file path “Runescape\_Item\_Names.csv”, and opening it within the environment using a read only command. Then lead on to outlining which columns of the dataset should be used within the variable `csv_data` which is list data type, holding each entry for the item names. Specifically in the `convert_to_list()` making sure to delete the heading entry, so to not cause irrational logic within the testing data used for our neural networks later on.

```
85     # writes list to text file for proper formatting
      1 usage
86     def write_to_file(list_of_items):
87         file_name = 'items.txt'
88         file = open(file_name, 'w')
89         for each in list_of_items:
90             file.write(each+"\n")
91         file.close()
92         return file, file_name
```

**Fig 10.** `write_to_file()` within `text_generation.py`

After putting the data into a list format, a new file is created titled ‘items.txt’, which will be used to hold each entry of the dataset of item names, in a string, text format. This is achieved once again by opening the file, except now in write-only mode, and using the in built python function of `.write()` within a for loop to append each item, returning the file and `file_name` to the procedure.

```

1 usage
94 def decode_py2(name):
95     py2 = open(name, 'rb').read().decode(encoding='utf-8')
96     print(f'Length of text: {len(py2)} characters')
97     vocab = sorted(set(py2))
98     print(f'{len(vocab)} unique characters')
99     print("ok")
100     return py2, vocab
101
102 1 usage
102 def tokenization(file):
103     example_text = ['abcdefg', 'qrstuvwxyz']
104     chars = tf.strings.unicode_split(example_text, input_encoding='UTF-8')
105     print(chars)
106     id_chars = tf.keras.layers.StringLookup(vocabulary=file, mask_token=None)
107     id_token = id_chars(chars)
108     print(id_token)
109     chars_id = tf.keras.layers.StringLookup(vocabulary=id_chars.get_vocabulary(), invert=True, mask_token=None)
110     chars_token = chars_id(id_token)
111     print(chars_token)
112
113     return id_chars, chars_id, chars
114

```

**Fig 11.** Decode and tokenization functions in text\_generator.py

Up next after is `decode_py2()` and the process of `tokenization()`. Inside of `decode_py2()`, there is an output to the user how many unique characters are inside the dataset, as well as the length. These values are used later on inside the training data to check for accuracy within the model, and for loss of data triggers during exponent calculations. The function uses an automatic sort of and the `decode()` function to interpret the bytes that are stored within the text file and sort them numerically.

The tokenization process is simplified to converting the string input that is apparent in our current dataset, into numerical integer values to be used during training. Give a set of example text that is run through a UTF-8 converter to translate it into unicode encoding. The encoded characters are then reverse checked against the original data using a `StringLookup()` function and additional masking to ignore erroneous data within conversion.

Once the conversion is successful, the three values returned to the main procedure are the id's for the character encoding and decoding. Also known as IDs and tokens.

```

114
    4 usages
115     def text_from_ids(chars_id, ids):
116         return tf.strings.reduce_join(chars_id(ids), axis=-1)
117
    1 usage
118     def split_data(sequence):
119         input_data = sequence[:-1]
120         target_data = sequence[1:]
121         return input_data, target_data
122
    1 usage
123     def formatting_data(text, data_id, data_chars):
124         all_ids = data_id(tf.strings.unicode_split(text, 'UTF-8'))
125         print(all_ids)
126         ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)
127         for ids in ids_dataset.take(10):
128             print(data_chars(ids).numpy().decode('utf-8'))
129
130         seq_len = 100
131         sequences = ids_dataset.batch(seq_len+1, drop_remainder=True)
132         for i in sequences.take(1):
133             print(data_chars(i))
134
135         dataset = sequences.map(split_data)
136
137         for input_example, target_example in dataset.take(1):
138             print("Input :", text_from_ids(data_chars, input_example).numpy())
139             print("Target:", text_from_ids(data_chars, target_example).numpy())
140
141         return dataset

```

**Fig 13.** split\_data() and formatting\_data() function with text formatting in text\_generator.py

text\_from\_ids() is a short function which uses the TensorFlow library to rejoin characters together to form cohesive sentences within the tensor object parameters. I.e. converting “‘T’ ‘e’ ‘n’ ‘s’ ‘o’ ‘r’ to Tensor”.



The `split_data()` function takes the batch of data that is being extrapolated from and separates it into input data and target data, which is used later on for validating and the predictive model. The formatting uses a sequential prediction model to get the model to work out the next character from a baseline.

The main section for preprocessing is the `formatting_data()` function, it creates training examples and targets by assigning ids to the encoded strings and using a predefined sequence length which dictates the size of the tensor object shape. After doing so, the dataset is then mapped to the target data, and then both the target and the input data is returned to the main procedure

```
Usage
143 def create_training_batches(dataset):
144     batch_size = 32
145     buffer_size = 1000
146
147     dataset = (dataset
148                 .shuffle(buffer_size)
149                 .batch(batch_size, drop_remainder=True)
150                 .prefetch(tf.data.experimental.AUTOTUNE))
151     print(dataset)
152     return dataset
153
154
```

**Fig 14.** `create_training_batches()` function

Now that tensorflow has been used to create splices of data from our data set, to sort into manageable sequences, it must all be shuffled before feeding it back into the model and packed into batches. Luckily `tf.data` is designed to work with an infinite amount of sequences, the buffer is used to stop the entire sequence shuffling within memory, and instead just the elements within the sequence.

```

164     #---CREATING THE MODEL---#
165
166     vocab_size = len(id_from_chars.get_vocabulary())
167     embedding_dim = 256
168     rnn_units = 1024
169
170     model = MyModel(
171         vocab_size=vocab_size,
172         embedding_dim=embedding_dim,
173         rnn_units=rnn_units
174     )
175
176     for input_example_batch, target_example_batch in training_sets.take(1):
177         example_batch_predictions = model(input_example_batch)
178         print(example_batch_predictions.shape, "# (batch_size, sequence_length, vocab_size)")
179
180     model.summary()
181
182     sampled_indices = tf.random.categorical(example_batch_predictions[0], num_samples=1)
183     sampled_indices = tf.squeeze(sampled_indices, axis=-1).numpy()
184     print(sampled_indices)
185
186     print("Input:\n", text_from_ids(chars_from_id, input_example_batch[0]).numpy())
187     print()
188     print("Next Char Predictions:\n", text_from_ids(chars_from_id, sampled_indices).numpy())
189

```

Fig 15. The creation of the model within main() in text\_generation.py

```

8     class MyModel(tf.keras.Model):
9         def __init__(self, vocab_size, embedding_dim, rnn_units):
10             super().__init__(self)
11             self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
12             self.gru = tf.keras.layers.GRU(rnn_units,
13                                             return_sequences=True,
14                                             return_state=True)
15             self.dense = tf.keras.layers.Dense(vocab_size)
16
17     def call(self, inputs, states=None, return_state=False, training=False):
18         x = inputs
19         x = self.embedding(x, training=training)
20         if states is None:
21             states = self.gru.get_initial_state(x)
22         x, states = self.gru(x, initial_state=states, training=training)
23         x = self.dense(x, training=training)
24
25         if return_state:
26             return x, states
27         else:
28             return x
29

```

Fig 16. MyModel class using tf.keras and the call method

These two code snippets showcase the keras.Model subclass and its layers include embedding, GRU and dense. The embedding layer is equivalent to the input layer, which acts as a trainable lookup table that maps the char IDs to vectors with dimensions equal to that of embedding\_dim. GRU is a type of recurrent neural network that uses a size indicator, and finally dense is the output layer, which outputs one logit for each character in the vocabulary i.e. the log-likelihood of each character appearing in the model.

Throughout the code there are multiple test print statements that are meant to indicate during runtime the variable outputs during setup and the execution of training the model. On line 180 the summary() function which will output the layers, output shapes and parameter count.

```
###SETUP FOR TRAINING THE MODEL###

loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
example_batch_mean_loss = loss(target_example_batch, example_batch_predictions)
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_size, sequence_length, vocab_size)")
print("Mean loss:      ", example_batch_mean_loss)
print(tf.exp(example_batch_mean_loss).numpy())

model.compile(optimizer='adam', loss=loss)
```

**Fig 17.** Training data setup into model and compile

At this point in the code training essentially begins, an optimizer and loss function are attached and applied across the last dimension of all the predictions and a flag must be set to allow for logits to interact with the crossentropy loss function. The model is ultimately compiled with the 'adam' optimizer, which is an extended version of the stochastic gradient descent.

```
200      #---CONFIGURE CHECKPOINTS---#
201      directory = './training_checkpoints'
202      checkpoints_prefix = os.path.join(directory, "ckpt_{epoch}")
203      checkpoints_callback = tf.keras.callbacks.ModelCheckpoint(
204          filepath=checkpoints_prefix,
205          save_weights_only=True)
```

**Fig 18.** Checkpoint system for weights for the model calling at each epoch

This checkpoint system is used to store weights after the model compiles and for after training, stored in a local directory titled 'training\_checkpoints'. Again tf.keras is being utilised for simplifying this entire process.

```
206      #---EXECUTE TRAINING---#
207      epochs = 30
208      history = model.fit(training_sets, epochs=epochs, callbacks=[checkpoints_callback])
209
```

**Fig 19.** The declaration of the number of epochs and fitting the data to the model

A short use of the fit() function to start the execution of training, it will run through the training multiple times for the number of epochs that has been designated, on average of 60s for the Runescape\_Item\_Names.csv dataset.

```
210     #---GENERATING TEXT---#
211     one_step = OneStep(model, chars_from_id, id_from_chars)
212
213     start_execute = time.time()
214     states = None
215     next_char = tf.constant([' '])
216     result = [next_char]
217
218     for n in range(1000):
219         next_char, states = one_step.generate_single_step(next_char, states=states)
220         result.append(next_char)
221
222     result = tf.strings.join(result)
223     end_execute = time.time()
224     with open("output.txt", "a") as f:
225         print(result[0].numpy().decode('utf-8'), '\n\n' + '_' * 80, file=f)
226     print('\nRun time:', end_execute - start_execute, file=f)
227
228     tf.saved_model.save(one_step, 'one_step')
229     one_step_reloaded = tf.saved_model.load('one_step')
230
231     states = None
232     next_char = tf.constant([' '])
233     result = [next_char]
234
235     for n in range(100):
236         next_char, states = one_step_reloaded.generate_single_step(next_char, states=states)
237         result.append(next_char)
238
239     print(tf.strings.join(result)[0].numpy().decode("utf-8"))
```

**Fig 20.** Generating texts using the one\_step class methods and then saving the outputs to output.txt

```

1 usage
30 class OneStep(tf.keras.Model):
31     def __init__(self, model, chars_from_ids, ids_from_chars, temperature=1.0):
32         super().__init__()
33         self.temperature = temperature
34         self.model = model
35         self.chars_from_ids = chars_from_ids
36         self.ids_from_chars = ids_from_chars
37
38         # creates mask to stop unknown outputs
39         skip_ids = self.ids_from_chars(['[UNK]'])[:, None]
40         sparse_mask = tf.SparseTensor(values=[-float('inf')]*len(skip_ids),
41                                         indices=skip_ids,
42                                         dense_shape=[len(ids_from_chars.get_vocabulary())])
43         self.prediction_mask = tf.sparse.to_dense(sparse_mask)
44
45     2 usages (1 dynamic)
46     @tf.function
47     def generate_single_step(self, inputs, states=None):
48         # Convert strings to token IDs
49         input_chars = tf.strings.unicode_split(inputs, 'UTF-8')
50         input_ids = self.ids_from_chars(input_chars).to_tensor()
51
52         # Run the model
53         predicted_logits, states = self.model(inputs=input_ids, states=states,
54                                             return_state=True)
55         predicted_logits = predicted_logits[:, -1, :]
56         predicted_logits = predicted_logits / self.temperature
57         predicted_logits = predicted_logits + self.prediction_mask
58
59         predicted_ids = tf.random.categorical(predicted_logits, num_samples=1)
60         predicted_ids = tf.squeeze(predicted_ids, axis=-1)
61
62         predicted_chars = self.chars_from_ids(predicted_ids)
63
64         return predicted_chars, states

```

**Fig 21.** OneStep class which is a subset of tf.keras used to do one single iteration of generation

The generation for one item iteration is all carried out within the class method of OneStep. It contains the tokens and ids for the characters and encoding/decoding data. As a way to ignore unknown inputs a mask is created that goes through each input and skips the decoding and output of text for those ids. The shape of the output vocab is then matched to the output\_ids and placed inside a prediction mask.

The method `generate_single_step` converts strings to token IDs, then runs the model for predicting the following character. Once the model had finished running, the `predicted_logits` used are only the last prediction as they will be the most accurate after going through the most steps of training. The prediction mask is then applied and samples are taken from the logit outputs to generate token IDs. These ids are converted to characters and then returned to main.

`One_step_model` is the variable which calls the method to run the method a singular time, by placing it in a for loop a larger set of text is output, printing the runtime and the generated text in a combined string format, that is output to a text file called `output.txt`.

Example of `output.txt` for one test run:

```
skirt

Medium blunt rune salvage

Small blunt iron salvage

Medium plated bronze full helm

Orikalkum pickaxe + 2

Adamant longsword

Rune whirl (Run)

Blue dragonhide boots

Blue dragonhide boots

Imphide sot (unchecked)

Dragonfire shield

Leather shield
```

Backly- seed

Meckladius ere geg helm

Spider on shield token

Stein weaste

Rune crossbow

Black boots

Oryfield robe tops

Team-32 cape

Leadhers necklace

Yellow feather

Glatial spikes

Necronium kiteshield + 3

Orikalkum armoured boots + 2

Orikalkum armour set + 2

Necronium armoured boots

Orikalkum pickaxe + 2

Orikalkum warhammer + 2

Mithril off hand warhammer + 2

Mithril off hand warhammer + 1

Dragon scimitar

Iron off hand scimitar



Rune longsword

Rune were shield

Rune kiteshield

Rune armoured boots + 2

Orikalkum armoured boots + 3

Necronium armoured boots

Sandall apron token

Shorts (blue)

Fregene Beller

Gold seal

Stone seal

Mystic gloves

Gold necklace

Jade amulet

Emerald Cane

Rubyd attack (r)

## 5.2 gpt.py

This python script is essentially taking the generated text that was created from text\_generator.py and running it through the OpenAI api using the gpt-3.5-turbo model for a user, recipient exchange.

```
1  import openai
2
3  items = []
4  items_file = open('output.txt', 'r').read()
5  data = items_file.split("\n")
6  data = list(filter(None, data))
7  print(data)
8
9  openai.api_key = open('key.txt', 'r').read()
10 message_history = []
11
```

**Fig 22.** Opening section of gpt.py

The script first begins by formatting the generated text into a list format that can be used to output into elements for the prompt being asked to the AI. Importing openai and using the free version of the Open AI API key which is read in from an external text file to allow the rest of the functionality of the API to work.

An empty list called message\_history is created that will be used to store logs of user and assistant call and responses

```
Usage
def call(user_input, role="user"):
    print("User input >> ", user_input)
    message_history.append({"role": role, "content": user_input})

    completion = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=message_history,
    )

    reply_content = completion.choices[0].message.content
    with open("item_description.txt", "a") as f:
        print(reply_content, '\n\n' + '_' * 80, file=f)
    message_history.append({"role": "assistant", "content": reply_content})
    return reply_content
```

**Fig 23.** Call function that takes user input and requests a response from the GPT model

The call() function takes our user input and logs the request into message history using a {role, content} format. openai.ChatCompletion is run to request a response, the model is chosen as gpt-3.5-turbo as it is the most recently updated model for Python development and has the most concurrent support for independent projects.

The reply content is printed to the user and then stored in a text file called “item\_description.txt” and then returned.

```
28 for i in range(1):
29     user_input = "Write a brief description of {0} as if it were an item in a fantasy game.".format(data[38])
30     print(user_input)
31     print()
32     call(user_input)
33     print()
34
35     print(message_history)
36
```

**Fig 24.** For loop that calls and holds the user input and generates the description

## Chapter 6 Results and Discussion

Here are some examples of the final generated description that the API output:

Medium blunt rune salvage is a rare and precious item that can be found in the most dangerous and mysterious places of the realm. This item is a fragment of an ancient and powerful rune, imbued with magical energy that can be harnessed by skilled craftsmen and enchanters. The medium blunt rune salvage is recognized for its particular shape, which resembles the head of a warhammer or a mace. It is said that this rune was created by a legendary blacksmith who imbued it with special properties that allow it to increase the effectiveness of blunt weapons and grant them additional strength and durability. Warriors and adventurers seek this item to upgrade their weaponry and increase their chances of success in battle. However, obtaining the medium blunt rune salvage is not an easy task, as it is guarded by powerful creatures and hidden in treacherous locations. Those brave enough to seek it out are sure to be rewarded with a boon to their strength and power.

Small blunt iron salvage is a common material found amidst the ruins and battlefields of the fantasy world. Composed of broken weapons and scraps of armor, it holds little value to most adventurers. However, those with a keen eye and a skilled hand can craft it into useful items such as arrowheads, small tools, and even some types of armor. Though unremarkable on its own, small blunt iron salvage can be a vital resource in the hands of a skilled blacksmith or craftsman.

The Iron Off-Hand Scimitar is a deadly weapon known for its uniquely crafted design and unmatched precision. Its massive blade, forged from the strongest iron, is sharp enough to slice through even the thickest of armor and deal devastating blows. This scimitar is designed for those skilled enough in the art of dual-wielding, as it is the perfect complement to any warrior's arsenal. With an ergonomic grip and a sturdy build, it is sure to provide the edge needed to win any battle. Its dark and sharp appearance is sure to strike fear into the hearts of any opponent, making the Iron Off-Hand Scimitar an invaluable tool for any adventurer.

The overall accuracy and level of detail for the descriptions that are generated are more than what was expected for the project, the predictive generative text model also worked very well, with fine tuning the training sets and loss values could be decreased, however the outputs that were generated from `text_generator.py` were accurate enough to be legible and understandable to the API.

For further improvements, the formatting of the descriptive text could be improved, as well as creating a UI and UX for the system instead of running it all through an interpreter. Perhaps using tkinter or converting the code base into an application would be better purposed for distribution. If there was sufficient time and better time management, being able to showcase the code run and then be played through an engine mentioned prior in Chapter 1 would have been ideal.

## **Chapter 7 Conclusion**

In conclusion, the study has been an overall success, being able to develop an RNN model that successfully uses predictive text generation to output different variations of a desired context, which syncs well with the use of third-party software in that of OpenAI and GPT-3.

In the future improvements made to the training model, increasing the complexity of the dataset, changing the RNN model to a LSTM model, scaling upwards from just items and dialogue to grand-scale environments and design. Creating a more userfriendly UX to use for potential open-source distribution, and automate the process in a less invasive manner.

The experience has been overwhelmingly rewarding, and I have refound a passion for games development and more specifically narrative design and the level of discipline AAA studios are having to achieve as technology improves, focusing on smaller scale development has been a good insight on how developers are attempting to keep up with the increase in automation and workflow pipeline process. The plan is to continue to work on the project after submission.

## Chapter 8

## References

### 1.1 Introduction

Moral Anxiety Studio (2022) *Roadwarden* [game], Wiesbaden: Assemble Entertainment. Available from <https://store.steampowered.com/app/1155970/Roadwarden/> [Accessed 10/11/2022]

Infocom (1994) *The Zork Anthology* [game], Santa Monica: Activision. Available from [https://store.steampowered.com/app/570580/Zork\\_Anthology/](https://store.steampowered.com/app/570580/Zork_Anthology/) [Accessed 10/11/2022]

Arkane Lyon (2021) *Deathloop* [game], Maryland: Bethesda Softworks LLC. Available from <https://store.steampowered.com/app/1252330/DEATHLOOP/> [Accessed 10/11/2022]

Epyx, Inc (1985) *ROGUE* [game], UK: Pixel Games UK. Available from <https://store.steampowered.com/app/1443430/Rogue/> [Accessed 10/11/2022]

Fullbright (2013) *Gone Home* [game], Portland, Oregon: Fullbright. Available from [https://store.steampowered.com/app/232430/Gone\\_Home/](https://store.steampowered.com/app/232430/Gone_Home/) [Accessed 10/11/2022]

Hello Games (2016) *No Man's Sky* [game], Guildford UK: Hello Games Ltd. Available from [https://store.steampowered.com/app/275850/No\\_Mans\\_Sky/](https://store.steampowered.com/app/275850/No_Mans_Sky/) [Accessed 10/11/2022]

Clinton Hocking (2007) *Ludonarrative Dissonance in Bioshock* [blog], 7 October. Available from [https://clicknothing.typepad.com/click\\_nothing/2007/10/ludonarrative-d.html](https://clicknothing.typepad.com/click_nothing/2007/10/ludonarrative-d.html) [Accessed 10/11/2022]

## 2.1 Literature Review

Fredriks M. E., DeVries B. (2021) *(Genetically) Improving Novelty in Procedural Story Generation*, Allendale, USA: Grand Valley State University. Available from <https://doi.org/10.48550/arXiv.2103.06935> [Accessed 2 January 2023]

Compton, K. and Kybartas, B. and Mateas, M., (2015) *Tracery: An Author-Focused Generative Text Tool*, Santa Cruz, USA, Springer International Publishing, Switzerland, 154-161. Available from [https://www.researchgate.net/publication/300137911\\_Tracery\\_An\\_Author-Focused\\_Generative\\_Text\\_Tool](https://www.researchgate.net/publication/300137911_Tracery_An_Author-Focused_Generative_Text_Tool) [Accessed 2 January 2023]

Siegel, J. and Szafron, D. (2009) Dialogue patterns - A visual language for dynamic dialogue, *Journal of Visual Languages and Computing*, 20(3) 196-220. Available from <https://www.sciencedirect.com/science/article/pii/S1045926X09000147> [Accessed 2 January 2023]

Kacmarcik, G. (2021) Using Natural Language to Manage NPC Dialog. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2(1) 115-117. Available from <https://ojs.aaai.org/index.php/AIIDE/article/view/18757> [Accessed 2 January 2023]

Ammanabrolu, P., Cheung, W., Tu, D., Broniec, W., & Riedl, M., (2020) Bringing Stories Alive: Generating Interactive Fiction Worlds, *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1) 3-9. Available from <https://ojs.aaai.org/index.php/AIIDE/article/view/7400> [Accessed 2 January 2023]

Shirish Keskar, N., McCann, B., Varshney, Lav R., Xiong, C., Socher, R., (2019) *CTRL: A Conditional Transformer Language Model For Controllable Generation*, San Fransisco, USA, Salesforce, Available from <https://doi.org/10.48550/arXiv.1909.05858> [Accessed 6 May 2023]

### 3.1 Requirement Analysis

Mitchell T. and McGraw H. (1997) *Machine Learning*, 3rd edition, India: McGraw Hill

Kharde, Vishal A. and Sonawane, S.S., (2016) Sentiment Analysis of Twitter Data: A Survey of Techniques, *International Journal of Computer Applications* 11(139) 5-15, Pune, India: Pune Institute of Computer Technology, University of Pune. Available from <https://doi.org/10.5120/ijca2016908625> [Accessed 6 May 2023]

## 4 Design & Methodology

### 4.1 Project Management and Risk Analysis

McCrea, C., (2020) *Videogame World-Building as Ideation, Praxis and Design Model*, Melbourne, Australia: Royal Melbourne Institute of Technology, World-Building Lab. Available from [https://digraa.org/wp-content/uploads/2020/01/DiGRAA\\_2020\\_paper\\_17.pdf](https://digraa.org/wp-content/uploads/2020/01/DiGRAA_2020_paper_17.pdf) [Accessed 6 May 2023]

TeamGantt (2009) *teamgantt* [online software], Baltimore, Maryland: TeamGantt. Available from <https://www.teamgantt.com/> [Accessed 8 May 2023]

Light, M., (2009) *How the waterfall methodology adapted and whistled past the graveyard*, Gartner Research.

McConnel, S., (1996) *Rapid Development: Taming Wild Software Schedules*, 2nd edition, Microsoft Press. Available from <https://archive.org/details/rapiddevelopment00mcco> [Accessed 8 May 2023]

Ji, F. and Sedano, T., (2011) *Comparing Extreme Programming and Waterfall Project Results*, Silicon Calley Campus Mountain View, USA: Carnegie Mellon University. Available from <https://doi.org/10.1109/CSEET.2011.5876129> [Accessed 8 May 2023]



## **4.2 Toolsets and Machine Environments**

JetBrains (2010) *PyCharm* [development software], Prague, Czech Republic: JetBrains. Available from <https://www.jetbrains.com/pycharm/> [Accessed 8 May 2023]

## **4.4 Research**

Kaggle (2010) *kaggle* [online software], San Francisco, USA: Google. Available from <https://www.kaggle.com/> [Accessed 9 May 2023]