Åbo Akademi

Master's Thesis in Computer Engineering

Creating Unique Gameplay Scenarios Using Natural Language Generation

Alex Renkonen 38515
Supervisor: Marina Waldén
Åbo Akademi University
Faculty of Science and Engineering
2022

Abstract

Language and text generation is a complicated task for computers, there is a lot of nuances and context required to understand a sentence. For decades natural language processing has been a field of study, for computers to process, understand and generated texts. In the last few decades deep learning has been used to greatly increase the proficiency of computers for generating and understanding texts. Games such as Dungeons & Dragons are heavily based in language, the players talk during the game, and the dungeon master leads the game by describing what the players can do. However, these games also base themselves on rules, limiting the freedom the storyteller has. This thesis investigates creating a machine learning model, using GPT-2, which can create unique gameplay scenarios, called encounters, which fit within the rules of Dungeons & Dragons. The GPT-2 text generation models are based on the transformer architecture created by Google and can generate high-quality text without additional training. A fine-tuning process allows the model to train on specific types of text, teaching it how to create similar texts. Using gathered data this fine-tuning process can create a model which is able to generate Dungeons & Dragons encounters quickly.

Table of Contents

1. Introduction

Language is understood by every human alive. Whether it be English, Swedish, Sign language, or any other kind of language is vital to society. Effective communication is a major reason for humanity having come as far as it has [1].

A large part of language and communication is storytelling and using language to make things up. Research has even shown that storytelling and stories are, and used to be, a large part of why humans are able to cooperate so well [2]. Stories are uniquely human, as every story we have heard or seen was created by another human, at least until the last few years. Computers have a difficult time parsing language. Words have many different meanings and without a clear understanding of the context, it is difficult to assign the correct meaning to each word. Attempts have been made to use machine learning to create tools that will write a story automatically [3].

Natural Language Processing (NLP) is the term used when using a computer to analyse text. The definition provided by Elizabeth Liddy is "Natural Language Processing is a theoretically motivated range of computational techniques for analysing and representing naturally occurring texts at one or more levels of linguistic analysis for the purpose of achieving human-like language processing for a range of tasks or applications." [4]

This thesis investigates creating a tool that can generate text based on learned data and create similar text. The first goal is to be able to create new and varied encounters for Dungeons & Dragons fifth edition. Focusing on this allows us to focus the tool on learning a very specific set of data and use of language; even making certain that a few specific key terms are used correctly as often as possible.

Dungeons & Dragons was chosen as the target due to its popularity. Dungeons & Dragons has the largest player base by a large margin; on the popular online Virtual Tabletop website Roll20.net, more than 50% of games used the system according to their Q1 2020 report [5]. This popularity makes a large amount of content made by players available. This will help with gathering training data for the model.

2. Natural Language Processing

Natural Language Processing uses computers to process organic language. There are a multitude of tasks that use NLP, for example, speech recognition, changing each word in a text to its base dictionary form, segmenting text into sentences, and even more complicated tasks, such as generating new text with Natural Language Generation (NLG) [4].

2.1. History

The history of NLP stretches back to the 1940s and 50s when computer automation was born. In the late 1950s, the Transformations and Discourse Analysis Project was one of the first complete parsing systems to be developed [6]. Early on, Machine Translation was also researched with the expertise and technology based on cryptography from World War II. This early work based itself on a simple dictionary-lookup to translate words and then change the word order to be more appropriate to the new language. This resulted in poor translations. The research in machine translation continued, with the belief that results indistinguishable from human translations were only a few years away. This research continued into the 1960s, when it was concluded that machine translations of this calibre were not achievable, and funding was seized.

Due to the halted research in machine translation in the 1970s, research began to focus on semantics and human conceptual knowledge, such as goals, plans, human memory organisation and scripts. During the 1980s, natural language generation was also making progress; Kathleen R. McKeown created TEXT, and David D. McDonald and James D. Pustejovsky created MUMBLE [7] [8]. TEXT could respond to questions asked about things in its database, and MUMBLE could generate descriptions based on a script that defined the order which the information should be presented in.

Until the 1980s, most research into NLP used a symbolic approach, creating a set of rules that the computer followed to create text; in the late 1980s, a shift began to occur from the symbolic approach to a statistical approach. The growing power of computers and their increased availability allowed research to use machine learning. Probabilistic and data-driven models became the standard. This increase in power also allowed NLP to be used commercially with speech recognition, grammar, and spellchecking [9] [4].

Machine learning has continued to be used in NLP, and since the 2010s, neural networks and representation learning have become standard for NLP, as these

models have been shown to be effective and proved state-of-the-art results in many different NLP tasks [10].

2.2. Natural Language Generation

Natural language generation is an application of NLP, where data is taken and transformed into natural human-readable language. NLG has been part of NLP research since the beginning. In the 1950s, it was used as a minor aspect of machine translation; in the 1970s, the first dynamically generated answers to questions were done; and by the 1980s, language generation was a field of its own [11].

Even though NLG has been studied for decades, it is still a complicated subject. It is one thing to generate text, but to generate useable and readable text is much more difficult.

2.2.1. Types of Natural Language Generation

There are many different types of NLG systems, each with its positives and negatives, that have to be considered depending on the intended use case. Two of the more straightforward types are canned text and template filling, both of which use prewritten text to create new text.

Canned text uses text that has been prewritten and saved; this text is then copied and pasted. The different texts might be concatenated with some words in-between. This can be used to make simple text generators that follow a standard structure, and the messages do not have to vary too much. However, this type of NLG system cannot adapt to a new situation; if the generator is required to write something new, someone would have to go in and program a new set of rules for this new situation. Simple text generators such as a horoscope generator can use canned text [12].

Template filling is similar to canned text in that it uses prewritten text, in this case, a template. That template is then filled with data in certain slots to obtain a specific message. Junk mail is a typical application of template filling, where the receiver's name is placed in the right spots in a prewritten message [13].

2.2.2. Advanced NLG

So-called Advanced NLG systems require more in-depth knowledge and need to use stages of planning and merging of information to generate text that looks natural and is not repetitive. Ehud Reiter and Robert Dale propose the following typical stages in Building Natural Language Generation Systems [14]:

- Content determination

- Document structuring
- Aggregation
- Lexical choice
- Referring expression generation
- Realisation

Content determination decides what information should be included in the text from available data. Document structuring decides in which order the data should be expressed in the text and how the text should be laid out. Aggregation merges similar sentences to improve the readability of a text and make it more natural; having two sentences convey much of the same information is unnatural. Lexical choice decides which words to use for a concept; for example, whether the word *medium* or *moderate* would be more appropriate in a particular situation. Referring expression generation is used to decide which pronouns to use and identify objects and regions in a text. Realisation is finally responsible for putting the text together using correct syntax, morphology, and orthography, such as using *will be* as future tense or *to be*.

2.2.3. Deep Learning NLG

Apart from Advanced NLG, the above types of NLG do not make decisions; both canned text and template filling NLG only write using provided texts and data to make a new text. In modern times, machine learning solutions are used to create texts that are vastly more complicated without having to write texts ahead of time from which the machine can choose.

The simplest of these dynamic NLG models is a *Markov chain*. A Markov chain predicts which word should be next in a sentence based on which words usually follow the current word. It chooses randomly between all choices but with a higher chance to pick more common words. A Markov chain only focuses on the current word and does not consider the previous words. This strategy was used for early smartphone keyboards to suggest the following word in the sentence that was being written [15]. The Markov chain is simple but uses a lot of randomness to obtain its results. It also cannot use words that were not in the original text it was trained on, nor can it tell if it gets into an infinite loop. To create more complex texts, something that has a memory is required, something that can take more than the current word into account.

2.2.3.1. Neural Network

A *Recurrent Neural Network (RNN)* attempts to mimic a human brain. Nodes in a network send copies of outputs back as inputs to give the network a short-term memory. This allows the network to apply previous knowledge to current data,

which is then used by the algorithm to calculate the probability of the next word with higher accuracy. However, due to only having a short-term memory, longer sentences will begin to lose coherence towards the end due to the memory of earlier words disappearing [16].

Long short-term memory (LSTM) is the fix to the short-term memory problem of an RNN. LSTM adds memory to the algorithm beyond rewriting the output back into the input. LSTM can read, write, and delete information from this memory; it uses this to decide which parts are essential and which are less critical. When the memory is passed through the network to the node again, information might be added or removed based on whether it is considered important for the current word. The capacity of this memory is still limited, as the paths from previous knowledge increases the complexity and time to compute increases [17].

To handle the memory capacity problem that RNNs run into, whether they use LSTM or not, *attention* can be used. In the 2014 paper, Neural Machine Translation by Jointly Learning to Align and Translate, Attention was introduced as a way for machine translation to better align which words are relevant to each other in different languages [18]. Attention was later brought to a variety of different machine learning tasks. It allows the computing power of a neural network to focus on the essential parts of a text and spend less of the power on less critical parts. This helps allow the neural network to not waste computing power on parts that are not necessary [19]. There are multiple versions of attention, which are better suited for different tasks. When using it for machine translation, attention compares two sentences, one for each language and matching words to each other. Figure 1 shows an example of how attention aligns an English sentence with a generated French sentence. Each pixel is a correlation between the words in English and French. Attention uses soft alignment; this is compared to hard alignment, where each word is only related to one word in the other sentence. With soft alignment, the words can be related to multiple words and at different strengths. In Figure 1, towards the end of the sentence the soft alignment of attention allows the model to take, “the man said.” completely into consideration before translating, as in French, “the” can be translated into a variety of words depending on the context [18]. Self-attention is another version of attention where the same sentence is input twice; it is used in the Transformer and explained further below.

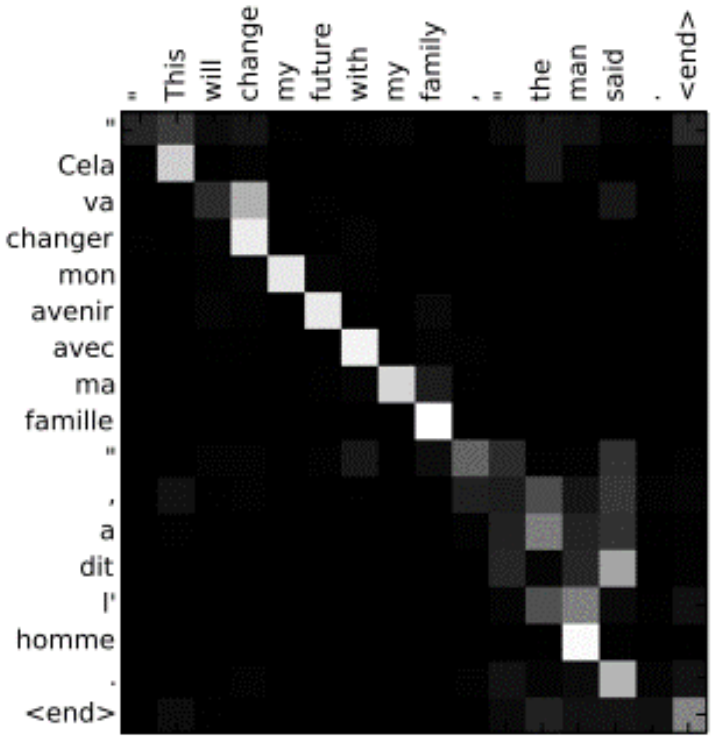

Figure 1. The matrix created by attention to align the two sentences in French and English. The whiter the pixel the higher the correlation between the words [18].

2.2.3.2. The Transformer

The Transformer is a neural network developed by Google in 2017; unlike an RNN, which processes the data sequentially, the Transformer is focused on parallel processing for quicker training times. Transformer is also an attention-based model. While attention has been used before Transformer as an addition to an RNN, Transformer is the first network with a sole focus on attention. Transformer uses a version of attention named self-attention, which allows the input of the Transformer to interact with other parts of the input. This allows it to calculate which parts are important to each other.

For example, "*The dog did not sit in on the tree branch because it could not climb."*

Using self-attention Transformer would be able to estimate that *it* in the example sentence refers to *the dog* and not *the tree branch* or anything else in the sentence. This helps in a variety of situations, for translating text knowing the context of words helps with translating those words, especially if the original text uses a word which could be translated into different words depending on the situation. The word, *bank,* could refer to the bank of a river, or the building bank. With the use of self-attention, the Transformer can use the sentence and figure out in what context

the word is used; is the word *river* or *swimming* in the sentence, then it most likely refers to a riverbank. However, is there mention of *building,* or *money*, then it probably refers to the building and not a riverbank.

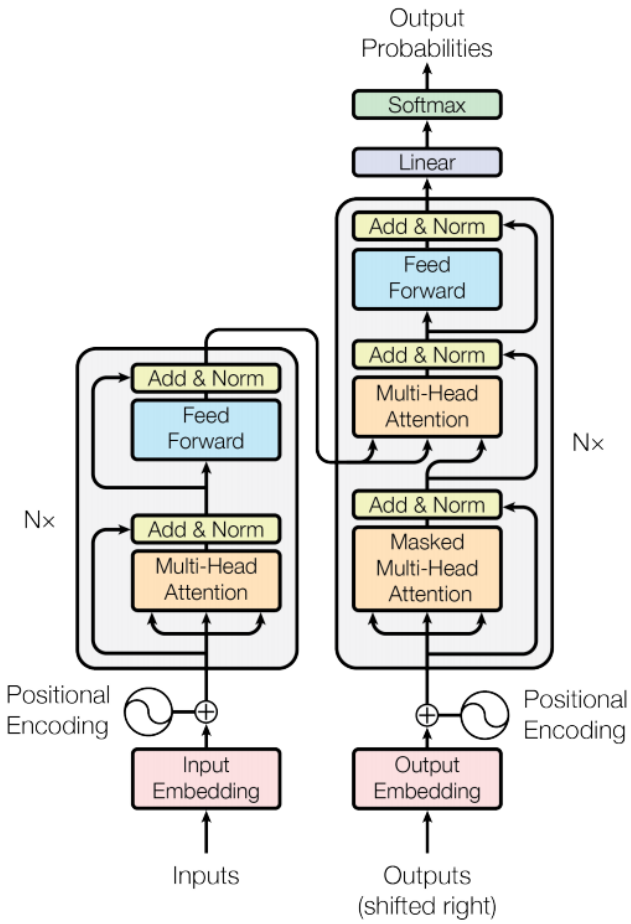

Figure 2. The Transformer model architecture [20].

The architecture of Transformer, seen in Figure 2, is built around an encoding component and a decoding component. The encoding and decoding components are both stacks of encoders and decoders, respectively. The encoders have a self-attention layer first and then a feed-forward neural network layer. The decoder also has a self-attention and feed-forward neural network layer, but in-between them is another attention layer to allow the decoder to focus on the relevant parts of the input sentence. The encoder's output is input into this second attention layer of the decoder [20]. Between the attention and feed-forward layers is layer normalisation in the Add & Norm layers, which will normalise the outputs to a set scale.

2.2.4. Training and Fine-tuning an NLG Model

When an NLG model is created, it has to be trained; there are different steps to the training depending on the model. Some models are first trained on a set of generic

data to teach it language, grammar, and basic sentence structure; this creates the basic model that can create texts but does not have a specific focus. The training data is usually extensive and contains text from a variety of different sources. After the training step, the model can be fine-tuned to generate specific types of texts. This fine-tuning process uses fewer and more specific types of text. After the fine-tuning process, the model will generate texts in a specific style with language that fits the fine-tuning data.

There are various hyperparameters to adjust and data to consider when fine-tuning a model. Loss indicates how poor a model's predictions are; the learning rate dictates how much the model should change between each step. Optimisers use the hyperparameters to improve the efficiency of the model using regularisation.

Loss is a general number used to determine the quality of a model's predictions. The closer the model's prediction is too perfect, the closer the loss is to zero. When fine-tuning the model, the goal is to have the loss approach zero. However, as loss approaches zero, the risk of overfitting also increases. Overfitting occurs when the fine-tuning data is too small and the training time is too long. The length of the training must be proportional to the size of the data. When this happens, a model might start memorising part of the training data causing it to be more likely to directly copy part of the training data instead of generating new unique texts. Directly copying parts of the training data can produce a very low loss rating (the prediction is perfect if it generates the same sentence as the training data). However, it will not be helpful once the model is given something that it has not seen before [21].

The learning rate is one of the hyperparameters which can be adjusted to maximise speed while minimising loss. Finding the optimal learning rate can require some testing; a too-small learning rate will eventually reach an optimal loss value, but it could take a long time, while a too-large learning rate will quickly lower loss at first but might overshoot and begin to raise the loss value. A too-small learning rate also risks being stuck in a local minimum instead of reaching an optimal global minimum. An optimal learning rate will lower the loss with each training step rapidly without overshooting. The learning rate controls how much of the gradient the steps move along each update. A learning rate of 1 is equal to 100% of the gradient. The larger the step, the more the weights of the model change each iteration [22].

The time spent fine-tuning the model is, therefore, also important to consider. As with other parameters, it requires taking into account multiple factors such as the size of the fine-tuning data, the learning rate, and the model's original training data. If the model had extensive training before the fine-tuning on a large amount of data, it might require more training to have the desired impact on the result. Training for

a shorter amount of time might not allow the loss to lower to an optimal amount depending on the learning rate, while training for too long might overfit the model.

Optimisers use these hyperparameters to optimise the learning of the model. These optimisers can reduce overfitting while keeping the weight and complexity of the model as low as possible. They achieve this by using regularisation, which penalises large weight values in models by adding a term to the function. As the model's size and complexity increase, the regularisation function penalises various parameters to force the model to be more general and not overfit. The more complex a model becomes, the risk of overfitting increases, and more memory and processing power is required to run it. A version of regularisation is *L2 regularisation*, which penalises the model by adding to the model's loss function the sum of the squares of all the model's weights multiplied by a given hyperparameter. Each step also subtracts a small portion of the weight.

Weight decay is another technique that functions similarly to L2 regularisation; however, the most significant change occurs when weight decay occurs. The decoupled weight decay occurs later in the algorithm allowing it only to affect the gradient of the loss function, while L2 regularisation also would affect the gradient of the regulariser [23].

2.2.5. Generating Text

When generating text using an NLG model, various parameters control how the neural net creates this text. Each of these parameters affects the result in different ways.

Many NLG models allow for a *prompt* which the model uses as a starting prompt to generate text. The prompt can be anything from a single word to a longer sentence, it then depends on the model how it continues generating the text. This prompt can be immensely practical in allowing more guided generation.

Temperature controls how random the choice of the next token is generated at each step; if the temperature is higher, the more random the choice is. At a temperature of 0, the model will always choose the highest probability word, and the model becomes more deterministic. Most NLG models use a temperature between 0 and 1, but it can be higher than that too. The range depends on the model and its training. Commonly a temperature between 0.7 and 0.9 is considered optimal, but testing is generally required for the best results [24]. Figure 3 shows an example of how different the text can be between a low temperature and a high temperature. The low temperature is quite generic and not that interesting but does make sense, while the high temperature example is much more chaotic and nonsensical. More examples

of how temperature and the other parameters change the generated text are in chapter 5.3.

Low temperature: *"The adventurers have many adventures to their name, but it is important that they are well-equipped to handle the situation. Many of them are veterans who are on active duty."*

High temperature: *"The adventurers all ran into a nearby sewer grate (which had once held a miniature noble, but now rests years in ruin) infestation, but who wouldn't want a PC to help their problem?A monstrous mount is ridden by a gnome. Friend or foe?The party finds a wanted poster tacked to a tree swinging in the wind. Gnomes? Blasting?A small critter (CR 1D6+1) appears to be causing chaos, as everywhere it scampers people flee in horror."*

Figure 3. Examples of the higher and lower end of Temperatures.

Top-K sampling redistributes the probabilities amongst the *K* tokens with the highest probabilities and removes all tokens beyond those. The main benefit is that the model's chances of going off-topic are reduced, especially at higher temperatures when the model often can choose lower probability tokens. The hard limit can increase the likelihood of a loop occurring where the same words appear repeatedly, and no other options are available; this likelihood increases even more with a lower *K* value. A *K*-value of 1 means always taking the highest probability token, while a large *K*-value, often around $K=20$, is the same as having no *K*-value [25].

Top-p sampling (or Nucleus Sampling) was developed as a response to Top-K's hard limit. Top-p sampling collects the smallest set of tokens whose cumulative probability exceeds the set probability of *p*. This makes the number of words available at each step vary depending on how high the probability of the early words is. If $p=0.9$ and the highest probability word has a probability of 0.7 and the second 0.2, there will only be two words in that set, while Top-K would still have collected all *K* words [24].

2.3. Natural Language Understanding

Natural Language Understanding (NLU), or Natural Language Interpretation (NLI), is one of the most complicated applications of NLP. It requires the computer to understand the text it is processing to use it better. Having context and knowledge of what a text means is vital to translate texts accurately and to be able to answer complex questions.

The difficulty in understanding text lies in the fact that language is ambiguous. Daniel Jurafsky and James H. Martin give an example “I made her duck”, which they write could mean any of the following (directly cited [26]):

a. “I cooked waterfowl for her.”
b. “I cooked waterfowl belonging to her.”
c. “I created the (plaster?) duck she owns.”
d. “I caused her to quickly lower her head or body.”
e. “I waved my magic wand and turned her into undifferentiated waterfowl.”

For humans, it is usually easy to understand which of these are correct due to the context surrounding the sentence. For a machine, however, it is guesswork or using statistical probabilities in an attempt to have the correct understanding.

3. Existing NLP Tools

Natural Language Generation has been researched for decades, and many have made tools that involve generating text. This chapter discusses and analyses a few of these tools.

3.1. Textgenrnn

The tool textgenrnn was made by Max Woolf and is a Python 3 module which works on top of Keras and Tensorflow. Version 1.0 of the module was released in 2018 and has been updated multiple times since then [27].

Another Recurrent Neural Network implementation called char-rnn by Andrej Karpathy was the basis for the creation of textgenrnn [28] which adds to char-rnn with additional modern deep learning features, such as attention-weighting and skip-embedding. While char-rnn focuses on character-level generation, textgenrnn also allows for word-level generation. The neural network either attempts to fill in the next character based on the context and memory of preceding characters, or it does the same with the words. These modern features are used by textgenrnn to improve the training time of a model while keeping the results the same, if not better, in some cases.

Languages

Theoretically, textgenrnn should be able to handle any language; however, it might struggle with languages that do not use the Latin alphabet or peculiar word structures. It works on both the character-level, which entails training on which character is most likely to come after the current one and the word-level, where it trains to predict which word is most likely to come after the current one.

The pre-trained model in textgenrnn has created relationships and context for each character it has come upon during training. Additional fine-tuning done by the user can allow the model to learn new relationships between characters or words. When generating text, the model looks at the probabilities for the current character and uses what knowledge it has of the context stored in the LSTM to be able to generate the next word or character.

Architecture

The default model of textgenrnn's architecture contains seven layers, an input layer, an embedding layer, two LSTM layers, a concatenating layer, an attention layer, and an output layer. Each of these layers and how they connect are represented in Figure 4.

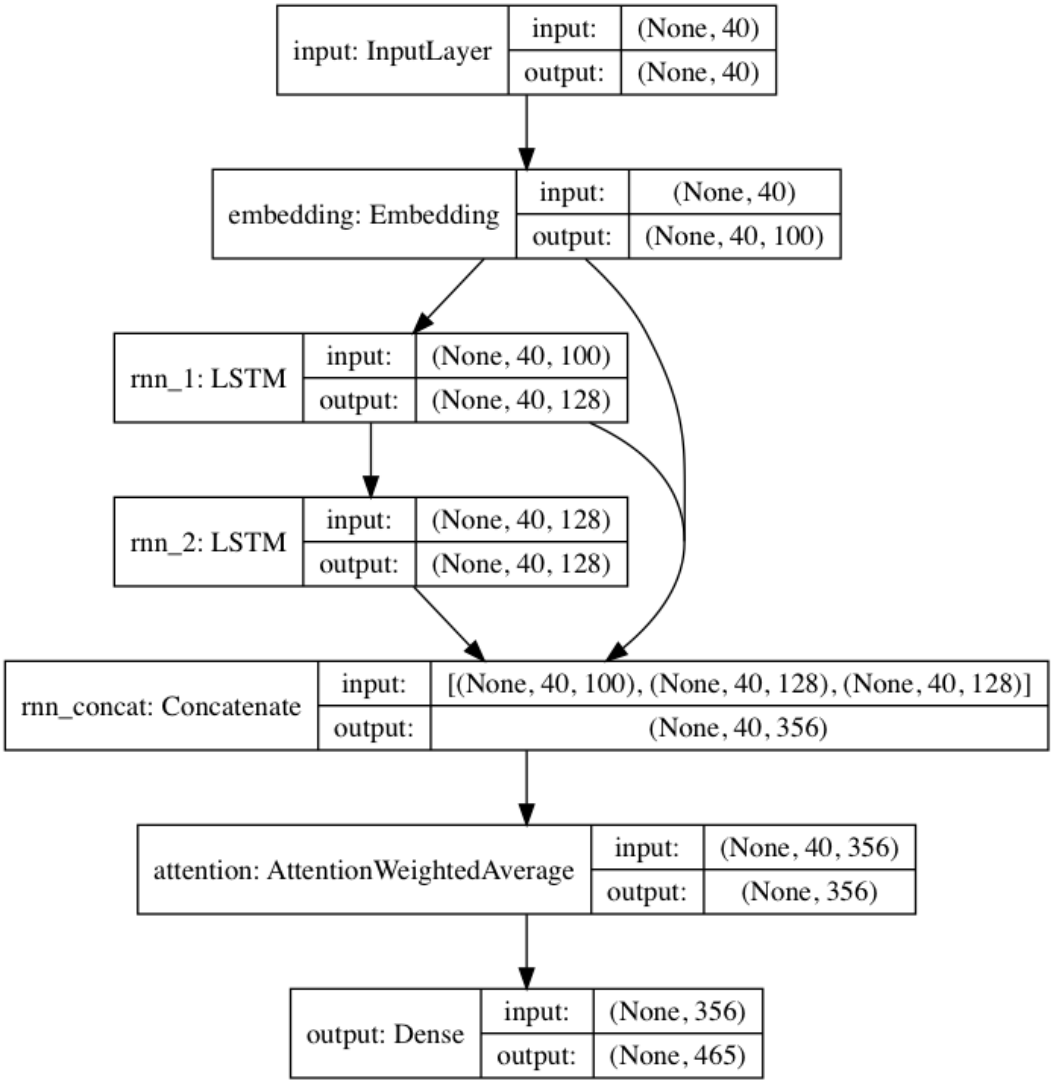

Figure 4. Default Architecture of textgenrnn [27].

The input layer takes an input of 40 characters which it sends to the embedding layer. The embedding layer converts each of these characters into a 100-D character embedding vector. This vector relates the characters to each other, so the LSTM layers will have information on how each character is related to others when it trains on them. The 100-D character embedding vector is then sent to the first LSTM layer and the concatenating layer. The 128-cell LSTM sends the output to the other LSTM layer and the concatenating layer. After having trained on both LSTM layers, the outputs are combined in the concatenating layer with the output of the embedding layer. After being concatenated, it is sent to the attention layer, which gives weight to the more essential parts. The last layer, the output layer, maps the output from the attention layer to the probability that they are the next character in the sequence relating to each character the model has access to.

Usage

Due to the ease of use, the tool has been used for many parody generators and smaller projects, usually with the intention of generating funny results. Examples of these are a tweet generator that can analyse the Twitter tweets of a single or multiple users and generate tweets in a similar style [29] and a Reddit community where all the posts are generated with textgenrnn [30].

3.2. GPT

GPT (Generative Pre-trained Transformer) are Transformer-based language models created by OpenAI. This chapter discusses GPT-2 and its successor GPT-3. These models have been trained on a large amount of internet text, 8 million web pages for GPT-2. GPT-2 boasts 1.5 billion parameters on its largest model, while GPT-3 has increased that many times with its largest model having 175 billion parameters. The parameter is a calculation in the neural network of the model, which will put a certain weight on aspects of the data [31] [32].

Languages

The pre-training which OpenAI used to train the GPT models before release was mainly focused on English texts; of the 40 GB of text which GPT-2 was trained on, only 10 MB were in French as they had tried to filter out any non-English texts. GPT-3 increased the amount of French and other languages in their pre-training data but still focused on English (93% of the texts were in English.) [32]

GPT uses pre-training to train the models on a vast amount of data, giving the model a general knowledge of sentence structure and order of words. It also gives the model a sense of the probability of which words come after each other. The models can be fine-tuned with more specific data to provide the context of what the user wants to be generated. For example, if movie scripts are used as input, the model will learn how movie scripts are written, which parts are common before other parts and so on [33]. For GPT-3, the researchers explored different types of fine-tuning. The standard approach of fine-tuning has been to provide numerous labelled examples for the model to learn from; with GPT-3, the Few-Shot, One-Shot and Zero-Shot versions of fine-tuning were attempted. Few-Shot gave the model a few examples, between 10 and 100. One-Shot gave one example and Zero-Shot zero examples. These all give a more accessible and quicker turnaround for using the model [32].

Architecture

GPT is based on Google's Transformer model; however, unlike the Transformer, which used both encoder and decoder blocks, GPT is built on only decoder blocks. The decoder blocks used in GPT also differ in that they do not contain the second attention layer, as there is no encoder input (see Figure 2 for the architecture of the Transformer.)

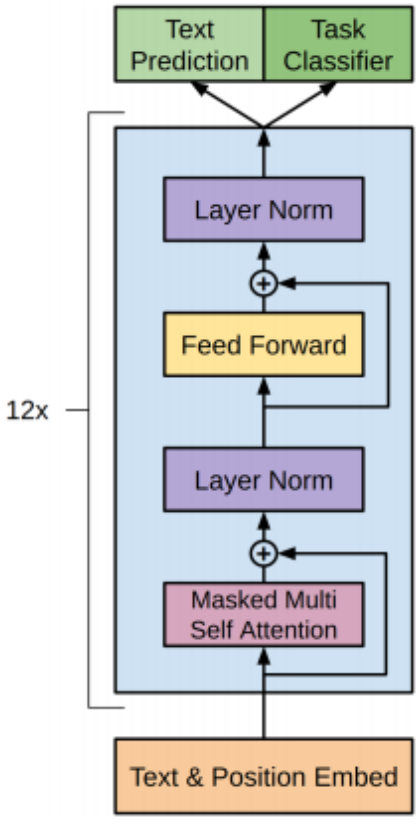

Figure 5. The architecture of GPT as presented in Improving Language Understanding by Generative Pre-Training [33].

Depending on which model of GPT is used, there is a different number of decoder blocks stacked on each other. In Figure 5, the architecture of GPT is presented as in the paper that introduced the first version of GPT, Improving Language Understanding by Generative Pre-Training [33]. That first model of GPT (and the smallest version of GPT-2) has twelve layers of the decoder Transformers. The self-attention layer also has twelve attention heads in the masked multi self-attention layer seen in Figure 5. The number of layers has vastly increased in the larger GPT-2 and GPT-3 models; the largest GPT-2 model has 48 layers, while GPT-3 has 96 layers and 96 attention heads. After the attention layer, a normalisation layer helps keep the data in a set range.

Usage

Since its release, GPT has been used in a multitude of applications. As with textgenrnn, parody generators have been created with GPT-2. Similarly, to textgenrnn a Reddit subreddit where bots based on other subreddits create posts and comments was created [34]. Another implementation that uses GPT is AI Dungeon, a text-based Role-Playing Game developed by Latitude, inspired by text-based adventure games popular in the 1970s and 1980s, where players interacted with the world and story by entering keywords to tell the game what they wanted to do (e.g., "Enter Door", "Look South".) Unlike the games that inspired AI Dungeon, it is not limited to the actions the developers thought of and implemented. The user is encouraged to write longer sentences of actions instead of single words, and the AI of AI Dungeon responds and continues the story; anything goes. AI Dungeon has both options to use GPT-2 and GPT-3; however, the latter requires a subscription to access [35].

Researchers at the University of Toronto used GPT-3 to detect hate speech. The model was able to detect hate speech with a 78% accuracy using few-shot learning; however, intentional misspelling of profane words made the model less accurate to detect them [36]. The Trevor Project, a non-profit suicide prevention hotline, began using a chatbot powered by GPT-2 in early 2021 to train its volunteers to handle calls [37].

Misuse

GPT-2 was initially released in February of 2019. However, due to the worry of misuse, the models were released in stages, starting with their smaller 117-million-parameters model, eventually releasing the final 1.5B model in November of the same year. In one study, GPT-2 was used to create news articles that were almost as convincing as the actual articles in *New York Times*. The generated article was considered credible by 72% of the respondents. In comparison, the actual *New York Times* article was considered credible by 83% [38].

With how realistic GPT-2 could create these articles, the researchers at OpenAI decided not to release the GPT-3 models publicly. GPT-3 has increased accuracy, and the ability to use the models maliciously would also be increased with this new power. Instead of releasing the models publicly, OpenAI has created an API that is accessible after joining their waitlist. This API allows the team to keep track of how the models are being used and terminate the access of anyone who misuses it [39]. For GPT-2, a detector model was also created to be able to spot the likelihood of a text having been generated.

3.3. Other Tools

Outside of the two tools above, there are other tools that do not focus on neural networks for their text generation. These tools might be focused on canned text, template filling, or some other type of text generation for different purposes.

3.3.1. RosaeNLG

RosaeNLG is an open-source NLG library for node.js [40]. RosaeNLG was created as an open-source alternative to a commercial NLG system that could be used for production-grade NLG applications. RosaeNLG focuses on template filling and has full support for English, French, German, Italian and Spanish. However, other languages are supported with a limited feature set. Suppose one of the supported languages is used. In that case, the generated text will have correctly written grammar, such as using *a* or *an* correctly depending on the word in English or using the correct gender for words in languages like Spanish and German.

3.3.2. RiTa

"RiTa is a toolkit for generative writing and natural language. It is implemented in Java and JavaScript." [41] RiTa's text generation uses Markov chains; there are also a handful of NLP features such as tokenization and feature-analysis. RiTa focuses on English.

RiTa has been used to create a variety of *Parody Generators*. Mark Sample created a Godard Film Generator, which creates plots for movies in the style of Jean-Luc Godard [42]. As one of the features of RiTa is finding rhymes, many generators focus on different kinds of poems.

The RiTa toolkit was initially created to provide students with an easy-to-use toolkit that can aid in various writing tasks. It was intended to be used by both those familiar with programming and novices. The main focus of RiTa was not text generation but to have one library which can support multiple different NLP and NLG tasks [43].

3.4. Comparison Between textgenrnn and GPT-2

Both textgenrnn and GPT set out to do similar things, generate text with pre-trained models, which can then be fine-tuned by the user using a smaller sample size. However, textgenrnn was created by a single person, while GPT has a large team of researchers and a company with funding to back them. This comparison will focus on the 124M model of GPT-2.

The generated texts from both have a clear difference in quality. GPT-2 has more sensible and legible texts on average compared to textgenrnn; GPT-2 also has a more consistent tone. The most significant upside of textgenrnn is its speed and size; the training times for textgenrnn is usually quicker than that of GPT-2, and the size of the model on the disk is much smaller. The size of a model in textgenrnn is usually between 1 to 10 MB, while the smallest model of GPT-2 (124M) is 500 MB and the largest (1558M) 5.8 GB. GPT-2 and textgenrnn use different parameters when fine-tuning their models, making a direct comparison complicated. When fine-tuning textgenrnn, the following parameters will decide training time the most

- Word level or character level.
- Number of LSTM cells in each LSTM layer
- Number of LSTM layers
- Number of epochs

The number of epochs will be the biggest decider; each epoch is a complete pass-through of the training data, which means each epoch can add significant time to the training. The time it takes for an epoch to train will depend on the number of cells in the LSTM layer and how many of those layers are used; if trained on word level, there will be less data for the model to consider, but the results may be less varied. GPT-2 uses steps; each step uses a batch of tokens from the data (the default is a batch size of 1 or 1 024 tokens).

3.4.1. Results

For this comparison, I fine-tuned both textgenrnn, with five 256-cell LSTM layers, and the 124M model for GPT-2 on the same training data. I trained textgenrnn for 20 epochs and GPT-2 for 2 500 steps (2 500 steps will run the model through the data about 20 times in the same way as the 20 epochs for textgenrnn will.)

Using Google Colaboratory with an Nvidia Tesla P100 GPU [44] the fine-tuning took 45 minutes for textgenrnn and 53 minutes for GPT-2. The training data was the same data described below in chapter 4 and 5.

A few examples from the training data:

“You see a human trapped in a bear trap. Upon release she shifts into bear form and runs away.”

“A pack of (2d4 +2) rabid dogs bursts from an alley, directly in the path of the party.”

"A group of goblins standing on each other's shoulders in trenchcoats trick travelers into coming to help fix their broken down wagon, and rob them for every copper they're worth."

After the fine-tuning was complete, the model generated a few of the following examples. When generating the text, a temperature setting can be selected, usually between 0 and 1; the temperature sets how far the model will stray from the original training data. Closer to zero means that it is more likely to generate text directly from the original data; closer to one means more unpredictable results. Once complete, the textgenrnn provides examples for three different temperature settings 0.2, 0.5 and 1.0. Texts in *italics* below are generated by textgenrnn and GPT-2. Any spelling errors are left as created by the model.

3.4.2. textgenrnn

Temperature: 0.2 example: *"man in rags walks listlessly towards you are to his blue from the shakes, but it doesnt have to strip off and looks like a prominent, protruding a few healon ssget cries for help."*

Temperature: 0.5 example*: "A farmer with black liberal splits the party."*

Temperature 1.0 example: *"A tinker or tanner near a clothed sharpener. If the PCs attempt to see in it, but where it made of helpling winds. With giant, violently, it leads to a nearby tow nShrine experienced and attacks."*

The text created by textgenrnn is often nonsensical, often using words that do not fit into the current context. When using character level generation, often non-existent words will be generated, such as *"healon ssget"*

3.4.3. GPT-2

GPT-2 generally performs better than textgenrnn and other models at higher temperatures (0.7 - 1.0) [45], giving readable, varied, and coherent text even at high-temperature settings. For comparison, here are three examples using the same temperatures using GPT-2.

Temperature: 0.2 example*: "A man in a white robe runs up to the group, eyes pleading with the party to help him. The robe is a Arsenean one, the draugr are from another plane. The PCs know that the man in the man in robe is a great wizard, and wants to make sure he is the son of a powerful wizarding family. The wizard is in his th elision at the top of the stairs, and is frantically scrabbling to get out of the way. The man"*

Temperature: 0.5 example: *"The man in the red hoodie and carrying a box of crackers begs the party for food or water. They will flee upon finding it."*

Temperature 1.0 example*: "A glimpse into the dreams of a wealthy race of adventurers."*

As is pretty evident, the GPT-2 results are more legible and closer to natural language than that generated by textgenrnn. The context of the text is also kept, often referring to characters or things set up earlier (such as the man in the white robe in the first example.) GPT-2 does not have a consistent way to know when to end the generation and will sometimes leave a sentence halfway complete, such as the first example, which ends in "*The man"* due to reaching the character limit. Non-existent words are also sometimes generated by GPT-2; however, compared to textgenrnn, these words are more often based on something or follows more structure compared to a jumble of letters. The quality may go up even further if using a larger model of GPT-2 but depending on the size of the training data when fine-tuning the model, the output might also worsen. Too little data for fine-tuning on too large of a model means that the final model will not have enough data to base what it generates text on. While it will generate coherent text, it might not be related to the training data.

3.4.4. Comparison

The benefit of GPT-2 is the quality; however, depending on the use case, textgenrnn will provide a quicker result. If there is a specific style you are looking for, textgenrnn offers word-level training, which is quicker than character training as there is less data to consider. GPT-2 will also require the use of a GPU for a reasonable training time; while textgenrnn benefits from one, it is usually feasible to fine-tune the model using only a CPU.

4. The Text Generation Tool

4.1. Goal

The goal of the tool is to be able to generate texts that have a basis in a rule-based system. Dungeons & Dragons fifth edition was chosen as a case study for the first test due to the amount of content available for it. Using the rules of fifth edition Dungeons & Dragons, the goal is to generate new and unique social, combat, or other hazard encounters that would be allowed within the rules of Dungeons & Dragons and that the players of the game could encounter during play. GPT-2 was chosen due to its power in generating coherent text.

Using the training data gathered, I fine-tuned GPT-2 to learn useful terms for generating encounters for Dungeons & Dragons. The fine-tuned model is then used to generate encounters.

4.2. Dungeons & Dragons

Dungeons & Dragons (D&D) is a tabletop role-playing game created by Gary Gygax and Dave Arneson, with the first edition published in 1974 [46]. Since then, multiple incarnations of the system have been released; in 2014, the latest version, Dungeons & Dragons Fifth Edition, was released [47].

4.2.1. Players, Characters and Rules

Playing Dungeons & Dragons is usually done with one *Dungeon Master* (DM) and one or more players. Playing Dungeons & Dragons has been compared to improvisational theatre with rules; the DM is the keeper of the rules and the leader of the storytelling, the players are characters in the DM's story. The Dungeon Master controls all enemies and friendly characters the players meet (*Non-Player Characters* or NPCs) while each player plays a single character (*Player Character* or PC). The characters and enemies all have attributes, skills, and powers. When playing Dungeons & Dragons, the DM describes a situation and location that the players encounter, acting as any characters they meet or controlling the enemies. The players react to that situation by acting (or role-playing) as their characters and rolling dice to overcome obstacles. The obstacles can be anything from fighting enemies, trying to get a deal on an item in a shop or jumping over a rushing river.

The rules in Dungeons & Dragons are there to guide the play. These rules come with many terms that are specific to it and other similar game systems. Every PC and NPC has six ability scores that each has its specific usages, Strength, Dexterity,

Constitution, Intelligence, Wisdom and Charisma. These ability scores have skills and saving throws based on them. Saving throws are used when something is done to a character that they try to resist or save themselves from, a Constitution saving throw to resist the effects of a poison, or a Dexterity saving throw to jump out of the way of a boulder or explosion. Dungeons & Dragons Fifth Edition has one type of saving throw for each ability score. A skill check, in contrast, is used when a character wants to take a specific action, a Charisma (*Persuasion*) skill check is used to convince a group of bandits that you are not worth attacking or an Intelligence (*History*) to see if they remember information about a person or event in history. Each skill is based on a specific ability score. However, there are multiple skills for most of the ability scores.

4.2.2. Dice and Randomness

Randomness is essential to Dungeons & Dragons, and rolling dice is used for this purpose. Instead of writing "two six-sided dice" in-text, it is shortened to 2d6. The number before the *d* indicates how many dice to roll and the number after the *d* which kind of die to use. Figure 6 shows examples of the dice commonly used in Dungeons & Dragons.

Figure 6. The dice of Dungeons & Dragons generally include a d4, d6, d8, d10, d12 and d20 [63].

There are six types of dice used in Dungeons & Dragons, d4, d6, d8, d10, d12 and d20. Sometimes a d100 is called for; this is done by rolling 2d10 and choosing one for the 10s and one for the 1s; a d3 is rarely asked for, but when it is a d6 is commonly used. Then the one rolling decides what the 4, 5 and 6 faces will be counted as. There are many speciality dice produced and other game systems might also have their own dice or use the dice in different ways. One of the most common things that occur while playing Dungeons and Dragons is rolling a skill check. A skill check occurs when a character attempts to do something based on one of the 18 skills, at which the character might not succeed. When attempting to take any action which could reasonably fail, the player rolls 1d20, one twenty-sided die, and adds any bonuses their character has to the result. The DM then explains if he succeeded or failed based on how high the result was, generally the higher, the better. To decide if the check succeeded, the DM sets a *Difficulty Clas*s or DC for the challenge, which is the number the player has to reach to succeed. A DC of 10 is relatively easy, and a character without any bonuses has about a 50% chance of success. At the same time, a DC of 25 is very difficult, and the character must have significant bonuses (which represents the training said character has in a skill) to even have a chance of succeeding. Combat follows the same general rules with the *Armor Class* or AC taking the place of the DC when attempting to hit an enemy.

Most other dice are used to generate a random number with specific ranges. The d4, d6, d8, d10 and d12 are all used to obtain a damage number for different weapons or spells during combat. Sometimes they are also used to generate a random number, e.g., how many creatures the PCs encounter. In combat, when someone takes damage, different dice are used to generate damage based on what caused it. A small dagger does 1d4 damage, and a large great axe does 1d12, while a powerful Fireball spell does 8d6 damage. These damage numbers are added up with any relevant modifiers and removed from the target's hit points, indicating how healthy and able to continue fighting they are. The d100, also known as d%, is used when there is a percentile chance of something happening; unlike other dice, the aim when rolling a d100 is often to get below a specific number; if there is a 10% chance of something happening, then you hope to roll a 10 or less on the dice.

One tool which Dungeon Masters commonly use to introduce conflict, add interesting moments into the game or have something new and unexpected occur is using encounters. Encounters can introduce new characters or put the PCs into danger or other situations that do not necessarily have to tie into the main story being told. Encounters are often used when travelling and can be based on the area that it occurs in, such as meeting a yeti while travelling in the arctic north, seeing a hunting pack

of wolves in the forest, or meeting a travelling caravan of traders while on the road. The goal of this thesis is for the model to apply some of these rules to the encounters it generates. A DM could use these encounters in play, if they do not have time to prepare any encounters or the players do something unexpected.

4.2.3. The Tools of Play

Playing Dungeons & Dragons can be done in a few different ways depending on the preference of the Dungeon Master and the players. The original and still popular way to play is in person using miniatures (or any other small object) to represent the PCs and NPCs, as seen in Figure 7; a grid sheet or terrain tiles can be used to represent the world which the characters move around in. In the figure, the grid sheet has been drawn on with markers to show the walls of the room and different architectural features.

Figure 7. Two players (to the left and right) and a Dungeon Master (at the top behind the screen) fighting monsters in a dungeon using miniatures and a grid sheet [64].

If one does not wish to use miniatures, grid sheets, or terrain tiles, one can use *Theatre of the Mind,* where the DM describes the locations even in combat, and everyone imagines the locations. This vastly reduces cost and preparation time for playing but can make fights more confusing and make it more challenging to keep track of combatants. In-person games often rely on Theatre of the Mind for

impromptu fights, and the Dungeon Master might have prepared maps for combats he was expecting. Online games have become more common in the last decade, allowing players from different locations to play together. Using voice and video chat tools like Discord, Skype and Zoom allow people to communicate with each other, and Virtual Tabletops (such as FoundryVTT and Roll20) as stand-ins for the grid sheet. Virtual Tabletops also track the characters' information (such as their gear and statistics). They can have features that are not possible for in-person games, such as not letting a player see locations on a map that the character has not been to and helping with the maths of adding bonuses to rolls for smoother game-play.

4.3. Libraries

Aitextgen [48], a python package created to enable simpler model management, fine-tuning, and text generation, allows for training on both CPUs and GPUs. Max Woolf created Aitextgen as a successor of textgenrnn and gpt-2 simple to take the best of both packages and add more features. It is the main package used for this tool's fine-tuning and generation process.

The package uses PyTorch [49], pytorch-lightning [50] and Hugging Face Transformers [51] to optimise text generation. PyTorch is used to improve GPU usage during training, and pytorch-lightning allows multiple GPUs to be used for quicker training. Support for TPUs (Tensor Processing Units) is eventually planned as this is a feature that PyTorch supports.

Hugging Face Transformers, also stylised as 🤗 Transformers, has a repository of pre-trained models for different NLP tasks and provides APIs to use these. It allows users to quickly obtain a base model and fine-tune it for more specific tasks; the models can then be shared with the Hugging Face community. Hugging Face Transformers allows aitextgen to have increased compatibility and functionality with GPT-2. The models on the Hugging Face model repository can be used with aitextgen, and models trained with aitextgen can be uploaded to the Hugging Face repository.

Unlike GPT-2 simple aitextgen allows for the usage of architectures beyond OpenAI's GPT-2 as EleutherAI's GPT Neo/GPT-3 [52] architectures are also available. EleutherAI's GPT Neo is based on the GPT-3 architecture created by OpenAI with additional functionality and broader availability. GPT Neo was trained on the Pile, an open-source language modelling data set, which combines 22 smaller data sets [53]. The Pile is 825GB large compared to the 40GB training set on which GPT-2 was trained.

4.4. Training Data

The goal of GPT and its derivatives was to have a sizeable pre-trained model capable of high-quality text generation without spending as much time fine-tuning it. The basic models do generate competent text, which is legible for the most part without fine-tuning. However, the text will not be based on a specific writing style or form. To teach a model to create a specific style of content when generating, fine-tuning data can be provided; this data should be able to convey the style and language of a particular type of text. If the goal is to generate movie scripts, feeding the model movie scripts will help it understand how they are generally written, the kinds of language used and what kind of content they include. If the goal of the generation is less specific and does not require a particular type of language, or is based on something already in the large amount of content the model was initially trained on, a minimal amount of fine-tuning can be used, if any is required at all. Few-shot and one-shot learning presented in the GPT-3 chapter above are examples of these quicker, more generalised fine-tuning options. Sometimes no fine-tuning is required, and as the models become more advanced, this might become more common. This tool aims to use prewritten Dungeons & Dragons encounters as fine-tuning data to let the model learn the language the encounters typically would use.

The input for the fine-tuning or the training data must be prepared in a few specific ways for GPT-2 to use it optimally. The training data must have a tag at the beginning and end of examples to tell the model where a new example has begun. These are called beginning of sentence (BoS) and end of sentence tags (EoS). The standard practice for this is <|*startoftext*|> for the beginning of sentence tag and <|*endoftext*|> for the end of sentence tag. Aitextgen will automatically add these tags to each line of a single column .csv-file.

4.4.1. Examples of the Fine-tuning Data

The training data used for the model for the tool has been gathered from various online sources. Websites such as Dndspeak and Reddit communities (or "subreddits") such as r/d100 have lists of encounters created by the users which have been gathered and edited into a format usable for fine-tuning the model. Currently, there are 3250 encounters in the file. These encounters vary from simple ones, which indicate a certain number or a random number of enemies.

For example:

"2d6+3 orcs"

"1 mammoth"

The DM can roll the indicated dice and obtain the number of creatures the party encounters and possibly fight. These encounters are quick and straightforward but do not offer much in terms of storytelling. There are more complex encounters in the data which set up characters that the party can interact with in other ways than fighting them. Some set up a mystery that the party can investigate; others are interesting things the characters see to make the world they play in seem more inhabited.

"While walking by the stocks, one of the prisoners recognizes someone from the party, and asks a favor. Could be something as simple as scratching an itch they can't reach themselves, or perhaps something more involved."

"You find a corpse with its eyes and mouth filled with dirt, still grasping the axe that's embedded in the nearest tree."

Some encounters are merely set dressing with a description of a landmark the characters pass by.

"A pond in a clearing, clean and still. No animal drinks from it, no insects buzz above it. No monsters will approach here beyond the tree line. Is it safe for you to stay here? The forest path is narrow and overgrown. Looking down the tunnel-like trail gives you vertigo and puts you off balance. Shadows appear to move, and you feel like you're just going in circles."

A few encounters involve making a die check, either a saving throw or a skill check.

"The party finds a sword in a stone. A DC15 Strength check pulls it out. It is a +1 Longsword."

"A DC15 Perception check spots a small chest hidden in a tree. It contains 44gp."

None of these encounters considers the difficulty of the encounter relative to the party. As the party grows stronger, the enemies in an encounter that was difficult earlier might become trivial, and an encounter might include enemies that are near impossible for weaker characters to defeat. Items and money that might seem like a lot to characters early on, might be irrelevant to experienced characters. The DC of skill check also often increases as the characters increase in power and their abilities are greater. It will be up to the dungeon master to consider if an encounter is appropriate for the characters and adjust it or choose another one.

4.4.2. Optimisers

Optimising the models and their learning is essential to reduce overfitting while keeping its weights and complexity low. To this end aitextgen uses the AdamW optimiser [23]. AdamW uses weight decay a change from the original Adam which used L2 regularisation. Both weight decay and L2 regularisation were explained further in 2.2.4 Training and Fine-tuning an NLG Model. Loshchilov and Hutter explain that they are often seen as equivalent because stochastic gradient descent (SGD). However, AdamW decouples the weight decay from the gradient-based update causing it not to be equivalent to L2 regularisation. In aitextgen, the AdamW optimiser is default and has the hyperparameters of learning rate, weight decay and warmup steps. The authors of AdamW showed experimentally that the L2 regularisation approach yields better training loss and that models trained with AdamW generalize better than those trained with the original Adam [23].

5. Implementation

This chapter discusses the implementation of the tool, how the choice of model impacts the result, and the testing of the tool.

5.1. The Choice of Model

When creating the tool, the first consideration was to decide which model to use; there is a wide variety of models available with various amounts of previous training and competence. However, choosing the model with the highest proficiency in language and most training might not be beneficial. A larger model often means that the time spent generating text will be longer and require more powerful hardware. Different models will also have different requirements for the fine-tuning data. For use with aitextgen, the model options come down to GPT-2 and GPT Neo. The models also come in various sizes, the larger sizes of the model generally mean that it has more parameters and is more capable of generating coherent texts. It does also mean that they require more memory and storage space to handle, and generating text generally takes longer and requires more input data to fine-tune. The time it takes to fine-tune is also longer.

The original GPT-2 model was trained on 40 GB of training data and GPT Neo on 825 GB, which is much more than the around 500 KB, which makes up the gathered input data for fine-tuning. Using a larger model would make it less likely to be influenced by the fine-tuning data and more likely to stick to what it learned in the initial training. Learning which model would be best takes some trial and error.

The third option, textgenrnn is much simpler and smaller in scale than GPT; it makes up for that with the speed of training and generation. However, the general quality of the outputs is still less than satisfactory. Using a considerable amount of fine-tuning data may resolve this. However, the larger models' basic generation quality would still most likely be superior to textgenrnn. GPT-2 and GPT Neo are being considered and compared in this chapter.

5.1.1. Size of the Model

Another consideration is the size of the model. Both GPT-2 and GPT Neo are available in a variety of sizes. The large ones generally provide higher quality generation at a basic level but require more fine-tuning data to generate specific styles of text. Larger models also require more memory to fine-tune. When fine-tuning a GPT-2 model with aitextgen, the options are between the models with 124M, 355M or 774M (million) parameters; for GPT Neo, the models with 125M and 355M

parameters are available. It is also possible to train a completely new GPT-2 or GPT Neo model. The 124M model of GPT-2 and 125M model of GPT Neo are the ones that are most likely to be useful for the size of the fine-tuning data. The larger the model, the more fine-tuning data or the longer time the training data would have to train, and even then, it might not be enough to obtain a good result without overfitting the model.

5.1.2. LAMBADA Evaluation of the Models

Both EleutherAI and OpenAI have published data on the results and skills of their models based on different evaluations available for testing the proficiency of language models [54] [32] [31]. The LAMBADA (Language Modeling Broadened to Account for Discourse Aspects) data set tasks the model to predict the final word of a sentence based on the context of previous sentences [55]. The data set consists of passages of text, including context and a target sentence. The goal is for the model to use the context to find the last word of the target sentence. The contexts are on average 4.6 sentences. LAMBADA scores the models on their accuracy (higher is better) and perplexity (lower is better). English speaking humans have on average a 95% accuracy and a ~1-2 perplexity score.

Table 1. Comparison between GPT-2, GPT-3 and GPT Neo on the LAMBADA data set. The numbers in parenthesis are EleutherAI's result of GPT-2 and 3 on the data set.

Models' names and size	**LAMBADA Accuracy (%)**	**LAMBADA Perplexity (Score)**
GPT-2 117M	45.99%	35.13
GPT-2 1.5B	63.24% (51.21%)	8.6 (10.634)
GPT Neo 125M	37.36%	30.266
GPT Neo 2.7B	62.22%	5.626
GPT-3 2.7B (Zero-Shot)	76.2% (67.1%)	3.00 (4.60)
GPT-3 2.7B (One-Shot)	72.5%	3.35
GPT-3 2.7B (Few-Shot)	86.4%	1.92
Human	95%	~1-2

Table 1 shows OpenAI's and EleutherAI's test results on the LAMBADA data set. GPT-2's largest model (1.5B) has an accuracy rating of 63.24% and an 8.6 perplexity rating. The smallest model (117M) had a 45.99% accuracy and a 35.13 perplexity rating [31].

The largest model of EleutherAI's GPT Neo (2.7B) had a 62.22% accuracy and a 5.626 perplexity rating. The smallest model (125M) had a 37.36% accuracy and a 30.266 perplexity rating [52]. EleutherAI also evaluated GPT-2 and GPT-3 on the same tests and received inconsistent results with OpenAI's report (shown in the parenthesis in Table 1.) The 1.5B GPT-2 model had a 51.21% accuracy and 10.634 perplexity rating in their tests.

In comparison, GPT-3 had a 76.2% accuracy score and a 3.00 perplexity rating in their zero-shot [32] and in EleutherAI's test, 67.1% accuracy and a 4.60 perplexity rating. OpenAI also evaluated GPT-3 with one-shot and few-shot training, receiving 72.5% and 86.4% accuracy and 3.35 and 1.92 perplexity ratings, respectively.

Both GPT-2 and GPT Neo fare similarly; the difference in accuracy of the larger models is negligible. In general, GPT-2 has higher accuracy, while GPT Neo has a better perplexity rating. However, in EleutherAI's testing, GPT-2 had overall worse results than GPT Neo; this could be due to the OpenAI team having more proficiency with setting up their model and EleutherAI's team having more proficiency with theirs. The EleutherAI team noted this discrepancy and wrote, "Some results for GPT-2 and GPT-3 are inconsistent with the values reported in the respective papers. We are currently looking into why, and would greatly appreciate feedback and further testing of our eval harness." [54]

Since the scores difference can be quite severe (according to EleutherAI's result, the GPT-2 1.5B model has worse accuracy than their own GPT Neo 2.7B model, while the GPT-2 model has better accuracy in OpenAI's evaluation), the question if the scores are comparable comes up. Is GPT-2 and 3 as powerful as OpenAI's result, and why do other evaluations not reach identical scores. Is the difference between the models as big as the original scores indicate? In the next section, more examples between GPT-2, Neo and 3 show that the trend continues with other evaluation tests.

5.1.3. Other Evaluations

Table 2 shows other similar evaluation tests done by both the OpenAI and EleutherAI teams on the WikiText-103 data set [56], where the perplexity rating for the

largest GPT-2 model is 17.48 and 37.50 for the smallest. At the same time, the rating for GPT Neo is 11.39 for the largest and 32.285 for the smallest.

Table 2. Comparison between GPT-2, GPT-3 and GPT Neo on the WikiText-103, WinoGrande and HellaSwag data sets. GPT-2 was only evaluated on WikiText-103 by OpenAI. GPT-3 was only evaluated on HellaSwag by OpenAI. The results in parenthesis on the GPT-2 and GPT-3 model are evaluations by EleutherAI.

Models' name and size	**WikiText-103 Perplexity (Score)**	**WinoGrande Accuracy (%)**	**HellaSwag Accuracy (%)**
GPT-2 117M	37.50	—	—
GPT-2 1.5B	17.48	(59.40%)	(40.03%)
GPT Neo 125M	32.285	50.43%	28.67%
GPT Neo 2.7B	11.39	56.50%	42.73%
GPT-3 2.7B (Zero-Shot)	—	(62.3%)	78.9% (62.8%)
GPT-3 2.7B (One-Shot)	—	—	78.1%
GPT-3 2.7B (Few-Shot)	—	—	79.3%
Human	—	94%	95.6%

GPT-2, GPT-3 and GPT Neo were also evaluated with WinoGrande [57] and HellaSwag [58], also shown in Table 2. EleutherAI evaluated GPT-2, GPT-3 (their results are in parenthesis in Table 2), as well as their own GPT Neo for WinoGrande and HellaSwag.

WinoGrande tests for the accuracy of the model, similar to LAMBADA. It was only tested by EleutherAI and not OpenAI. For their own model, EleutherAI received a 50.43% accuracy rating for its smallest 125M model and a 56.50% for their largest 2.7B model. GPT-3 received a 62.3% accuracy and GPT-2 59.40%. Overall, GPT-2 and 3 both received better accuracy than GPT Neo, continuing the trend that OpenAI's GPT models have higher accuracy. However, the difference between the smallest GPT Neo and the largest GPT-3 evaluated was quite minor, with only an 11.87 percentile point difference. Comparing this to the estimated accuracy of a human at 94%, it is a relatively small difference [57].

HellaSwag also tests for accuracies like WinoGrande and LAMBADA. OpenAI evaluated GPT-3 on HellaSwag, as a zero-shot, one-shot and few-shot, using their largest model for each. EleutherAI evaluated GPT-2's largest model, their own GPT Neo models and GPT-3 as a zero-shot on HellaSwag. GPT-2's largest model received an accuracy score of 40.03%. GPT Neo's smallest model only received a 28.67% accuracy and the largest a 42.73% accuracy. OpenAI's result on GPT-3 was 78.9% accuracy for the zero-shot, 78.1% for the one-shot and 79.3% for the few-shot. Surprisingly the one-shot performed the worst of the evaluations compared to the zero-shot, which would have received even less information. The few-shot model only did marginally better [32]. Meanwhile, EleutherAI's result on GPT-3's 2.7B model was a 62.8% accuracy rating. Humans have an estimated score of 95.6% accuracy on HellaSwag, quite a significant step up from even GPT-3's highest accuracy. It does, however, put GPT-3 quite a lot ahead of the other models; there is only a 16.3 percentile point difference between the few-shot model's accuracy and the human accuracy, while the GPT Neo accuracy is 36.57 percentile points behind GPT-3. Here the difference between GPT Neo and GPT-3 is quite significant. The largest GPT Neo model barely has a higher accuracy rating than the largest GPT-2 model, with a 2.7 percentile point difference. Compared to WinoGrande, the accuracy of all models except GPT-3 were significantly lower. GPT-3 is the only model which did better; even EleutherAI's evaluation rated it at 62.8% compared to 62.3% accuracy on WinoGrande [52]. GPT-3 is consistently the most powerful, which may be primarily due to OpenAI's access to the most resources for their research. OpenAI's founders collectively pledged US$1 billion to their research in 2015 [59]. EleutherAI, in comparison, is a decentralized collective of volunteers working on open-source AI research founded in 2015 [60].

As with the LAMBADA evaluations, the EleutherAI results for the HellaSwag evaluation on GPT-3 are much lower than OpenAI's. The overlap between evaluations by EleutherAI and OpenAI are not as considerable as the LAMBADA, which makes it more difficult to estimate how accurate the evaluations are. In general, the EleutherAI results have tended to be lower than the OpenAI results, even when using the same models. This could, in large parts, be due to inexperience with the model. It does take into question how well the GPT-2 WikiText-103 evaluations line up with the rest as EleutherAI did not evaluate it.

5.1.4. Choosing the Model

The three different model architecture evaluated are all capable of generating high-quality text. GPT Neo might be more consistent than GPT-2 but requires more power to run; GPT-2 goes awry now and then and produces erroneous results more

often than GPT Neo. Either way, both models do not have perfect generation, and without oversight the output will often be nonsense. GPT-3 creates impressive high-quality generation, but due to fear of misuse, OpenAI has decided to provide limited access to it.

GPT Neo was made by EleutherAI as a response to the low accessibility of GPT-3 and boasts many of the same features as GPT-3. It boasts similar generation skills as GPT-3 [52].

Overall, there is a trend of GPT-2 having higher accuracy while GPT Neo has a better perplexity in these evaluations. GPT-3 is still the most proficient of the models; even with EleutherAI's evaluations, where it performed worse than during OpenAI's tests, it still outperformed even the largest GPT Neo model. The inexperience with the model can explain the discrepancy between the results of the EleutherAI team and the OpenAI team, and in general, the difference is not that big. Due to the limited access, OpenAI provides to GPT-3 it is not chosen as the model for this tool.

Due to the similarities in proficiency between both models and GPT-2 having smaller models and requiring less memory and computing power, it is chosen as the model.

5.2. Fine-Tuning Settings

The model is fine-tuned on the gathered data to train it on how encounter texts are often written. The following chapter discusses some of the hyperparameters and data used to fine-tune the model to achieve the desired results. The hyperparameters change how long and how efficiently the model learns during fine-tuning. Usually, it takes some testing to obtain ideal results.

The learning rate is an essential hyperparameter for fine-tuning and is discussed more in 2.2.4 Training and Fine-tuning an NLG Model. The learning rate significantly impacts how quickly the model is overfitted; avoiding overfitting is vital to creating an NLG model which creates unique texts. *Loss*, also discussed in Training and Fine-tuning an NLG Model, is one of the data points used to discover overfitting, but it is not perfect. Another way to discover overfitting is to see if the model consistently generates texts that already exist in the fine-tuning data. The model should generate unique texts, but if it is overfitted, it will more or less copy text from the fine-tuning data, sometimes with minor changes. This is the clearest evidence of overfitting, as the model has learnt that to create the most accurate text to the training data, is to create the exact text as that in the data.

This text was generated without any additional input to the generator:

*"In the middle of day**, a monstrous mount lumbers into town**. Will the adventurers notice?"*

is partially identical to parts from the fine-tuning data:

*"**A monstrous mount lumbers into town**. It is ridden by a gnome."*

However, the other parts of the generated encounter are original text, indicating that the model is overfitted or close to it. In an ideal scenario, all the text will be unique, but this could also be evidence of the model not having enough training data to learn from. If the model is overfitted, the fine-tuning must be done again, using fewer training steps and a lower learning rate in the next attempt. Gathering more data for fine-tuning will also help avoid overfitting. At this point, it can help to be conservative, as it is always possible to continue training the same model if it turns out it could use more fine-tuning after quitting the process early.

The learning rate explained in Training and Fine-tuning an NLG Model is by default set to 0.001 when using aitextgen; however, with the small size of the fine-tuning data, the learning rate will most likely have to be much lower to receive a useable result. Unfortunately, there are no hard rules on what loss is ideal as it depends on the use case, the model, and the training data.

5.3. Generation Settings

When generating text, various parameters allow for control over how the neural net creates the text. The most basic and perhaps most important, depending on the goal, is the prompt that tells the model how it should begin.

***"The adventurers** now have the charred skeletons of 1-4 adventurers, along with a note saying they died of natural causes"*

***"The adventurers** find a large cage which is actually a cage made of wood and made of glass. Inside is a large pool of water that attracts a swarm of locusts. The pool is home to a collection of locusts that are attracted by the swarm of locusts.*

Figure 8. Generated text without hyperparameters set.

The prompt **"The adventurers"** tells the model to generate a text which begins with the words the adventurers. The examples in Figure 8 use **"The adventurers"** as a prompt, but the outputs are each distinct.

This allows for a high level of control, but some prompts can limit the model, especially if it is overfitted or does not have enough data to choose from; in which case it might often generate the same text. The prompt is not necessary, and the model can still generate appropriate texts based on its fine-tuning, but a prompt helps steer it in the right direction. There are many options of how this could be used to generate encounters that would fit in a Dungeons & Dragons game. Prompting the model to generate texts which begin with "The adventurers", "The party", "You come upon", or similar text, which is common in many of the example encounters in the fine-tuning data, helps set the neural network on the right path of the kinds of text we want to generate. The prompt only sets up the text we want; it does not control how the model generates it. The most important parameters to consider are temperature (how random the choice of the next token is), Top-K sampling (redistributes the probabilities over *K* tokens) and Top-p sampling (Like top-K but using the *p* most likely tokens); these terms were further explained in 2.2.5.

5.3.1. Temperature

Even though a temperature between 0.7 to 0.9 is usually recommended, the chosen model, GPT-2, can generally do well at temperatures even above 1. Continuing with the example above in Figure 8, Figure 9 shows what the model generates at different temperature levels using the same prompt.

temp = 0.01: *"The adventurers have many adventures to their name, but it is important that they are well-equipped to handle the situation. Many of them are veterans who are on active duty."*

temp = 0.2: *"The adventurers are hired to transport live cargo to the house of a noblewoman. One of the PCs is actually a shape-changed harpy."*

temp = 0.5: *"The adventurers all run into the same bugbear, who is desperately trying to find his pet dog."*

temp = 0.8: *"The adventurers all ran into the dungeon at night, yet they had not come face to face with a single enemy. The party had a simple task they had to accomplish; they were transporting a magical item to market in a small city, and the merchant has no problem with the item's failure. Trading and security are available, but does the merchant have a problem with magic items?"*

temp = 1.0*: "The adventurers all ran into the dungeon at night (or more importantly, because they were traveling in a rather large city) and were exhausted on their feet. Half of their party is exhausted on their feet. The other half is covered in blood.*

temp = 1.5*: "The adventurers all ran into a nearby sewer grate (which had once held a miniature noble, but now rests years in ruin) infestation, but who wouldn't want a PC to help their problem?A monstrous mount is ridden by a gnome. Friend or foe?The party finds a wanted poster tacked to a tree swinging in the wind. Gnomes? Blasting?A small critter (CR 1D6+1) appears to be causing chaos, as everywhere it scampers people flee in horror."*

Figure 9. Generated text using various temperature settings.

The appropriate temperature depends on the original model and the specific fine-tuning. Figure 9 shows quite a large difference between the first generated text with a temperature of 0.01, which ends up being quite generic. The last one, which has a temperature of 1.5, ends up being much more chaotic and contains a great deal of nonsense such as a sewer grate infestation, a monstrous mount, a wanted poster and so on with no sense of direction. The texts generated with a more moderate temperature tend to have more coherent text, but it is not perfect.

A temperature of 0.8 at this stage is the most coherent and useable, but as Top-K and Top-p are added below a higher temperature can be used without the text becoming nonsense.

5.3.2. Top-K

Top-K sampling redistributes the probabilities amongst the *K* tokens with the highest probabilities and removes all tokens beyond those. The difference between a large K and no set K value is often non-existent since it is already more likely that a high probability choice is taken. When using the same example as above, not using a K-value generates the same results as above. There is no limiting and redistributing of the token's probabilities.

A K-value of 1 means always taking the highest probability option; this ends up being the same as having a very low temperature and results in the same text as a temperature of 0.01 above.

K=2*: "The adventurers are hired to transport live cargo to the house of a noblewoman. One of them takes issue with the PCs, mistaking them for a reviled enemy".*

K=3: *"The adventurers are searching for a certain wizard. If they pursue him, they find the wizard is a ruthless wizard who is afraid of the outside, and seeks to use the PCs as an example."*

K=4: *"The adventurers are recruited as substitute racers for a down-on-their luck faction after a mysterious 'accident' disabled the previous team."*

Figure 10. Generated text using various top-K values, and a temperature of 0.8

When set to a temperature of 0.8; if $1 < k < 10$ as seen in Figure 10, the model ends up generating different text from the example in Figure 9.

The difference in text quality is not as noticeable as with temperature, especially once the temperature is at a good level. There are many repeats of texts at higher K-values as all the words the model uses already are available. In the examples, K=2 has a relatively straightforward text, but as it only has the top two choices, the variation of generated text will be limited. Similarly, with K=3, the limitation of choices has resulted in the model repeating wizard multiple times. When K=4, the generated text ends up mostly copying one of the existing texts from the training data, then once K > 10, the generated text ends up being the same as with no set K-value. The cut-off for the model creating the same text as infinite K can vary from

generation to generation. While K=4 seems good here, the usage of Top-p sampling, which is discussed next, also requires some change to Top-K for optimal generation results.

5.3.3. Adding Top-p

When Top-K and Top-p are used in tandem, it is recommended to make K larger than when it would be used by itself. When testing different combinations of K and p, it is noticeable that once the p-value gets large enough, the model generates the same text from that point on (when the model uses a seed for generation). Generally, this occurred at p=0.85, and as pointed out earlier, a lower K-value also causes the model to begin repeating text. The model's temperature can also be adjusted with all these hyperparameters in place. Figure 11 has a few examples of text generated with various values for each hyperparameter.

K=20 & p=0.1*: "The adventurers have many adventures to their name, but it is important that they are well-equipped to handle the situation. Many of them are veterans who are on active duty."*

K=20 & p=0.95: *"The adventurers all ran into a nearby den. The inn is a little farther away from the cave than the party, and neither party has been able to come inside."*

K=4, p=0.85 & temp = 1.5: *"The adventurers were recruited to search the forest for extinct creatures. But when they approach, they are ambushed by a mammoth shark-man, but the whale-man is gone."*

K=10 & p=0.85: *"The adventurers all ran into a nearby cave hours before. Closer investigation reveals that the fire had turned a small creek into a massive raging river."*

Figure 11. Generated text with various values for top-K, top-p and temperature.

Having a low p-value generates the same result as a low temperature and low K-value. When the p-value increases, the generated text begins to be more varied. Having a high K-value and a high p-value creates varied but unpredictable texts. Having a lower K-value and a higher temperature value can give varied texts.

Together each setting gives the model more limitations but helps guide it to generate more appropriate texts. A p-value of 0.85 got good results most of the time, and as seen earlier, a higher value often did not change the resulting text. Setting the K-

value between 10 and 20 tended to give the most exciting results, much higher than the K=4 used when not using top-p. When using top-K and top-p, the temperature can also be increased, and the generation will still be less sporadic than usual. A temperature of 1.5 will not be as nonsensical as shown in the temperature section.

A few other parameters can affect the generation of the model, repetition and length penalty, which penalise the model for repeating text and making long texts, respectively.

Another parameter penalises n-grams (a sequence of n words.) The parameter stops the model from repeating an n-gram twice by setting the probability of the sequence of words which would cause a repetition to 0. This penalty reduces the repetition often found in the generated texts and can lead to more natural and unique texts. The last few parameters do not change the resulting text much but bringing all the settings together, the rate of useable text increases. There is still much nonsense, but it is reasonable to remove the worst using some testing.

Generally, a temperature of 1.5, a top-K value of 20, a top-p value of 0.85, a length penalty of 2 and an n-gram limit of 3 caused the model to generate good encounters a good amount of the time.

5.4. Testing

After testing multiple permutations of the hyperparameters, does the model create good texts and does the results fit within the goals of the tool? The most basic goal is to create readable text. After that, the text should fit within Dungeons & Dragons as an encounter, and thirdly it should include the rules of Dungeons & Dragons correctly. GPT-2 is already capable of creating good texts due to its extensive training by OpenAI, and the fine-tuning done should not change the outcome a great deal. Good text can be difficult to test, as what is considered good can be subjective. The first test is to test how well the model does in general.

5.4.1. Evaluating the Fine-Tuned Model

Using the evaluation tests from 4.5.1. The Choice of Model can show if the quality of the model has changed, and if so, how much. To that end, EleutherAI provides a framework Language Model Evaluation Harness (LM Evaluation Harness), which allows for evaluation tasks on GPT-2, GPT-3 and GPT Neo on over 200 different evaluation tasks [54]. These evaluations test how good the model is at accurately predicting the next token. However, the evaluations test the general skill, and the more specific knowledge of Dungeons & Dragons encounters might reduce the accuracy as the model will expect different words than would generally be used.

Table 3 shows the comparison between, the original GPT-2 117M model, the GPT Neo 125M model and the fine-tuned GPT-2 117M model used by the tool using both the HellaSwag and WinoGrande evaluation methods. With HellaSwag the original models had an accuracy of 28.67% to 79.3% depending on the model used and its size (compared in 5.1.3 on Table 2.) HellaSwag was not evaluated on GPT-2 117M, which is the one used in this tool, the closest to it was the GPT Neo 125M, which had an accuracy of 28.67%. Using LM Evaluation Harness to evaluate the base GPT-2 117M, the result was an accuracy of 28.94%. When evaluating the fine-tuned model, the result was a 27.72% accuracy—a decrease by 1.22 percentage points from the original model. WinoGrande's accuracy was 51.62% on the original model, which decreased to 49.57% on the fine-tuned model, a decrease by 2.05 percentage points. The model has decreased a bit in general accuracy, but as the accuracy is already so low, their choices are near random and as such the small change in accuracy can not be considered a reduction in quality.

*Table 3. Comparison between the results of GPT-2 117M, GPT Neo 125M and the fine-tuned GPT-2 117M models' evaluation results. * Results by EleutherAI*

	GPT-2 117M	**GPT Neo 125M ***	**Fine-Tuned GPT-2 117M**	**Difference**
HellaSwag (Accuracy %)	28.94%	28.67%	27.72%	1.22%-points and 0.94%-points
WinoGrande (Accuracy %)	51.62%	50.43%	49.57%	2.05%-points and 0.86%-points

Both WikiText and LAMBADA, which evaluate the model in its ability to guess the last word in a target sentence and calculate a perplexity score resulted in a score of several hundreds of thousands. However, according to the paper published about LAMBADA [55], choosing a random vocabulary word should result in a perplexity score of 60 000. The accuracy results from HellaSwag and WinoGrande also were close to random choices.

Even if the accuracy has decreased, the general knowledge of the model is not a significant concern, as long as it can generate text which is useable as encounters. To see if it can generate good encounters is more problematic as it can be subjective, and there is no programmatic way to test it. A time-consuming but straightforward way, which would introduce a lot of subjectivity, is to generate a large sample of encounters and see which percentage of them are useable in any form. Comparing the generated texts to the fine-tuning data to see if it is copying instead of creating new text is also an important test.

As will be discussed further in Discussion the generated text sometimes includes rules from Dungeons & Dragons, such as dice rolls and DC's. This mainly occurs when the model copies texts, but sometimes it does use them correctly without prompt. It does better at suggesting a random number of enemies (e.g., *"1d3+1 kobolds"*), as a part of the fine-tuning data was a list of enemies from Dungeons & Dragons and numbers of dice.

HellaSwag

HellaSwag is an evaluation model created by Rowan Zellers et al. Which evaluates NLP models by using a similar system as LAMBADA (explained further in 5.1.3. LAMBADA Evaluation of the Models). It presents the model a sentence and asks it to choose the right option to continue it. HellaSwag was designed to have problems that are easy for humans to complete (humans have an average of 95.6% accuracy), but challenging for machines, with most models having <50% accuracy. Even on models trained explicitly on the same data as the test data. HellaSwag generates the wrong answers with a technique called Adversarial Filtering which is able to generate texts which are easy for humans to see as wrong but difficult for machine learning models.

LM Evaluation Harness implements HellaSwag which allows a model to be tested on the HellaSwag data set. This allows the fine-tuned model of the encounter generator to be tested on the HellaSwag data set easily. The evaluation asks the model 40 000 questions and calculates an average accuracy based on the responses.

WinoGrande

The WinoGrande was created by Keisuke Sakaguchi et al. and was made as a modern improvement on The Winograd Schema Challenge (WSC) which modern NLP models were able to complete with an accuracy of around 90%. The questions in WSC were constructed to have question pairs which are almost identical except a trigger word which changes the correct answer. The example below would ask the model what ***it*** represents, the suitcase or the trophy.

*"The trophy doesn't fit into the brown suitcase because **it**'s too **large**."*

*"The trophy doesn't fit into the brown suitcase because **it**'s too **small**."*

WinoGrande expanded the number of questions with the help of crowdsourcing. The questions were also removed which had some sort of bias, such as the trigger word being strongly indicating one answer over another. The resulting evaluation model went from 90% accuracy on the original WSC to 79% on the new set of questions. Humans were still at around 94%. On the set of questions without bias,

most models only have around 50% accuracy, which would be the same as picking a random choice [57].

LM Evaluation Harness implements WinoGrande which allows a model to be tested on the WinoGrande data set. By default, the XL-data set is used, which includes all 40 000 questions, even the biased ones.

WikiText

The WikiText data set was created by Stephen Merity et al. It is made up of Wikipedia articles, with over 28 000 articles as part of the data set overall. The WikiText data set is available in two size, WikiText-2 and WikiText-103. The perplexity is calculated as the model attempts to predict the next word in a text, in this case the texts are from the Wikipedia articles gathered for the data set [56].

LM Evaluation Harness implement WikiText allowing a model to be tested on the WikiText-2 data set by default. The WikiText-2 data set is smaller than the 103 version which contains all the articles.

5.4.2. Analysing Generated Texts

Text that has been generated by the model can generally be placed in one of five different categories.

1. *Copying and Replacing Words* for the texts which merely copies fine-tuning data and replaces one or two words.
2. *Unique but Nonsensical* for the texts which are unique but do not mean anything.
3. *Rules Implementation is Wrong or Missing* for the texts which use the rules of Dungeons & Dragons wrong or partially.
4. *Unique and Useable with Work* for the texts which can be used with a little bit of work.
5. *Ideal Encounters* for text which achieve the goal of this thesis, creating an interesting and unique encounter.

All the texts below have been generated with the same parameters, as the model has some randomness when choosing which token to pick next the texts created can be unpredictable. This allows the model to have a near infinite number of texts it can create, but not all of them will be good.

Similarity to Fine-tuning Data

Using Natural language toolkit (NLTK) [61] and Gensim [62] to compare a set of 300 generated encounters to the fine-tuning data receives a result of 1.75% average

similarity between the encounters on the two documents. The encounters which are found to be most similar sometimes have just one word switched for a synonym and other times is more of an amalgamation of two or more similar encounters from the fine-tuning data. These encounters often fit in the first category, *Copying and Replacing Words*.

Parameters

The parameters used for the generation of these encounters are the ones discussed in 5.3. Generation Settings. The *temperature* which sets how random the choice of the next token is set at 1.5. A rather high number but the other parameters remove less likely tokens making the options available better.

Top-K limits the number of available choices to the *k* most likely choices, a *K*-value of 20 works well here. *Top-p* further limits the selection of tokens by only keeping the tokens which reach a combined probability of *p*, a *p*-value of 0.85 in combination with Top-K = 20, makes for an appropriate set of tokens for the model to choose from at each step.

The *length penalty* penalises the model if the generated text gets too long. Using a value of 2 allows the model to create texts which tends to not be too long, but do not cut off suddenly.

Category 1: Copying and Replacing Words

The text in category 1 has copied a text from the fine-tuning data and merely replaced a word with another. The goal is to create unique encounters, which this completely fails at.

In this generated example: *"The road passes through a dense forest."* The model has copied the text "*The road leads through a dense forest."* From the fine-tuning data and replaced *leads* with *passes*. As the NLG model largely works on combining texts which it has learned from comparing texts like this is not completely fair, the combined texts which are found similar are not always similar to a single encounter but using a lot of different parts of various encounters to make something original.

Category 2: Unique but Nonsensical

Even if the text is not copying, it can still be unusable, such as if the text is nonsensical, or goes off on irrelevant tangents. These texts could possibly be used; however, it would require a lot of work and assumptions by the DM.

This is an example of a text which is nonsensical and goes off on a weird tangent: *"A large and colourful herd of deer cross the road this day and night, they are known for their frenzy, hunt one and the flock is very merry. However, one night one of their mounts has been murdered by a crazed gnome who appears to have a petrification spell on them."*

The text starts off as an encounter which could occur to break up the monotony of travel or increase immersion at the table. Something to help the players feel like the world they are pretending to inhabit actually is alive, but not something they have to interact with. However, it starts to lose coherency after halfway through the first sentence. The flock of deer turns from frenzied to merry, and the second sentence has nothing to do with the first. Different parts of the encounter could maybe be used by a DM, but it would take some liberal interpretation of the text. The text is also beginning to be quite long, which can be a reason why the text loses coherency. Ideally an encounter should setup the information needed and then let the DM and players react and interact with the situation. One or two sentences should be enough. The second sentence could be a good encounter in itself too; with the players being ambushed while resting for the night.

Category 3: Rules Implementation is Wrong or Missing

The texts in this category are not as nonsensical as the ones in the previous category. However, rules elements are either left out or misused. The rules of Dungeons & Dragons are important, but they are difficult for the model to be able to implement, especially with the small sample size it learned from.

Example 1: *"Sickly green gas rises from a tar pit in a cave ahead, it is very difficult to detect."*

Would have been a good place for the generator to have asked for a Wisdom (Perception) check to see if the PCs detect the gas as they approach. Asking for a Constitution saving throw could also be appropriate if the characters enter the gas, to see if they are affected by it or not. However, there are few examples of these in the fine-tuning data so it is not something the model would have a high likelihood of generating. The texts which include saving throws or skill checks usually are quite nonsensical. The following two examples show how the text are generally generated when they include saving throws and skill checks.

Example 2: *"The door creases into a stout gatehouse, guarded by an amiable but naive young woman, who asks her to escort them into some forbidden room of the dungeon's dungeons, where they will have to endure torment for the rest of their*

days. They will also have to pass a perception check, like a WIS save to halve their short rest. The save DCs are Charisma-based."

As a whole example 2 is nonsense. The last sentence uses a lot of words which are based on Dungeons & Dragons, however without context they are meaningless. The text *"They will also have to pass a perception check, like a WIS save to halve their short rest."* Asks the players to make a perception check like a Wisdom saving throw to halve their short rest. This ends up meaning nothing, how a perception check is used as a Wisdom saving throw is unclear, and why perceiving something would let someone rest up quicker is also confusing. The text *"The save DCs are Charisma-based."* Also ends up being meaningless, as there are no other mentions of charisma nor are there any pointers to whose charisma to base the DC off.

Example 3: *"If any humanoid within 5 feet of you is poisoned or otherwise severely poisoned, halve the damage until cured by taking a DC 12 Wisdom saving throw, taking 1d10 psychic damage on a failure. On a later performance of the druid the poisoned or critically poisoned humanoid takes 1d6 psychic damage with a WIS roll (perception) using Greater Restoration, taking 2d6 bludgeoning damage on an additional roll (not combat related)."*

This text could almost be useable, but it requires some generous interpretation of the content. To ask for a DC 12 Wisdom saving throw is an accurate application for resisting psychic damage. If you are generous, it also says that you take half the damage if you succeed, which is common with saving throws throughout Dungeons & Dragons. The second half of the text is not as coherent, and as with the example above, it only uses words from Dungeons & Dragons in random ways.

Category 4: Unique and Useable with Work

Some text generated are close to being useful, but it would require some work. They are often using a word wrong, or using something as if it were a creature, even if no such creature exists in Dungeons & Dragons.

An example of such a text would be the following: *"2d6 of mist from the canopy, followed by 1d6 mist from deep within the ground."*

The text is almost completely unique, and it does use the rules of generating a random number of creatures correctly. However, mist is not a creature. Mist occurs a few times throughout the fine-tuning data, but never in a similar context. It seems that the model has inserted mist instead of a monster into the text. There are monsters which are mist like in Dungeons & Dragons, however, they do not appear in the fine-tuning data so the model would be unfamiliar with them. The encounter

could be a good combat focused encounter if the Dungeon Master replaced the mist with a monster which fits better. Something that can climb trees and dig holes, or a humanoid monster which can set an ambush. In these cases, the generated encounter builds a good foundation, but it would still take some work for the DM to be able to use the encounter effectively. The encounter text also does not explain what the "mist" does, only that it exists.

Category 5: Ideal Encounter Generation

The ideal encounter is unique, while following the rules of Dungeons & Dragons and being legible. Such an encounter does not copy large portions of text from the fine-tuning data, while staying coherent and avoiding nonsensical tangents. The encounters should also not be too long, one or two sentences are usually ideal.

The following example is almost completely unique: *"3d6+2 oozes are circling around and attacking a beholder."*

The specific combination of 3d6+2 does not occur in the fine-tuning data. Oozes are a type of monster in Dungeons & Dragons which often are depicted as amorphous sludge which moves and can manifest fists and similar blunt instruments out of their bodies. Oozes are mentioned a few times in the fine-tuning data; however, the context is often very different. A beholder is another monster from Dungeons & Dragons, a floating creature with a sphere body, the beholder is known for its massive central eye and tentacles with smaller eyes at the end of each. Once again, the monster is mentioned a few times throughout the fine-tuning data but is never mentioned in a similar context. The premise of the encounter where a group of enemies are circling another enemy is also unique and does not occur in a similar fashion in the fine-tuning data.

This encounter is the ideal of what the generator can create. An encounter which is unique, adds something interesting to the situation, and lets the players ask questions. "Why is the beholder there?" "How did it get surrounded by the oozes and where did they come from?" It also uses the rules for how to indicate dice appropriately. 3d6+2, or three six-sided dice plus 2, has quite a wide span, the lowest result is 5 and the highest 20, and the average result is 12 and 13. It is not out of the question that this range would be used, however, a result of 12 or 13 would be considered a lot of enemies if the party were to fight them. Combat in Dungeons & Dragons tends to have about the same number of enemies as the number of PCs, unless it is a fight against a single extremely powerful enemy or against a group of very weak enemies. However, there are examples of 3d6 being used in the fine-tuning data.

Concluding Remarks

When the model generates unique text which is not nonsensical, the results are good. It can make encounters which would be useable while playing, but it fails some of the time. A lot of the encounters generated has some part of a flaw mentioned above; some nonsense, misplaced rules or copied parts of the fine-tuning data. Sometimes these flaws are minor and could as discussed with the examples above still be used. To use them would take more work, and assumptions must be made by the Dungeon Master, but it would be possible. Some encounters still are generated as mostly nonsense, which makes them useless.

The best encounters are generated when the model takes words from the fine-tuning data and apply them in a unique way. Such as placing a creature common to Dungeons & Dragons and placing it in an unusual situation. These encounters work best when they can be used without any context and let the player interact or experience something interesting which does not necessarily relate to the plot.

6. Discussion

This chapter discusses the goals and the failures of the finished tool, contains a reflection on the development of the tool.

6.1. Goals

The goal was to create a tool which could generate unique gameplay encounters for Dungeons & Dragons Fifth Edition on the fly, which with little preparation could be used when playing. The first goal was to have generic encounters which could be used anywhere. The second goal was to have it to asks for appropriate rolls such as saving throws and skill checks at appropriate times.

The first goal is a mild success. While the tool does not have a high success rate with creating interesting, unique, and useable encounters it does occasionally succeed in creating fun, engaging, and interesting encounters which would fit perfectly in many Dungeons & Dragons games (at least those that are of a less serious tone.) There is still a lot of repetition, nonsense, and incoherent texts generated and some filtering is required to find something appropriate.

The second goal is not as successful. The model does sometimes include these rules, however, it does so without having the context for why they are used nor an understanding on how to use them properly. A dexterity saving throw is as likely to be asked for as a constitution, as they are both often used, however, in vastly different contexts. If the tool gets it right, it is usually by luck. It is better at providing a random number of foes for the players to face as these encounters are often similarly structured to each other. The most basic ones only give a set of dice to roll for the number of enemies and the name of the enemies. If the enemy that is named exist these encounters are completely useable. However, the model has no understanding of which kinds of dice exist and sometimes suggests a d7 or similar.

The generated text uses appropriate language most of the time. Anachronistic and irrelevant concept and items such as phones, and modern-day countries and locations do occasionally occur, but this is rare. Avoiding these words as much as possible helps with fitting the generated encounters in a generic fantasy world, which is the most commons setting in which to play Dungeons & Dragons. As most of the original training data that the model was trained on was modern internet text avoiding such idiosyncrasies completely can be difficult without filtering out these words manually afterwards.

6.2. Reflections on the Tool

The first part of creating the tool was to figure out what Machine Learning model to use. At first, I tested textgenrnn, a simple and quick way to train and generate text. However, due to its simplicity and speed the text it generates is less than ideal, a lot of it is incoherent and less than satisfying. The original training data was small and not very extensive, and the gathered data for fine-tuning was also not large enough to make up for this. It is possible that a larger set of fine-tuning data could make textgenrnn viable for a tool like this.

GPT and its derivatives were much more competent from the outset at creating readable texts, they also required a low amount of fine-tuning data to be able to copy a style. There are a variety of models based on the GPT family of models; OpenAI has made three versions of GPT, and other groups have created their own versions based on OpenAI's research. EleutherAI made GPT Neo and its derivatives. After comparing different sizes of GPT-2 and GPT Neo, a smaller model of GPT-2 was chosen. The speed of training and generating was slightly slower than textgenrnn and the size of the final model larger. However, the resulting generated text was much more coherent and legible.

The fine-tuning of the model used different encounters gathered from a variety of internet sources of mostly fan made encounters. These encounters were used to aid the model in learning how encounters are usually structured and what language is used. The model also learned the vocabulary of Dungeons & Dragons, such as creature and the typical character which occur in the text, but also the rule elements such as how different dice are indicated in text. For the fine-tuning process there are multiple hyperparameters to modify, to get the model to learn at an appropriate rate. By using a smaller model of GPT-2 it allowed the model to be influenced by a smaller amount of fine-tuning data. However, it also risked the model being overfitted quicker, which would make the model merely repeat text from the fine-tuning data instead of generating new. Finding a balance between training the model for as long as possible, while not overfitting took some trial and error.

Once the model's fine-tuning was complete, I had to find the appropriate settings for the parameters used for generating the text. There are many parameters that can be changed and tweaked to get the desired results from the model when generating texts. Finding the ones which would influence the model the most towards the right direction also took some trial and error.

The testing was done in a few steps, first a comparison between the final model and the original model before fine-tuning. Here the model was minorly worse in general

accuracy, but the difference was minimal, and the goal was not to generate general texts which is what the evaluations tested.

7. Conclusion

This thesis examines the creation of a machine learning model which can create playable encounter scenarios for Dungeons & Dragons Fifth Edition using natural language generation. With the goal to be able to create encounters which could be used spontaneously and without preparation or adaptation from the Dungeon Master. The goal was also for the model to be able to use Dungeons & Dragons rules appropriately in these generated texts.

The model was created using the transformer based GPT-2 by OpenAI and fine-tuned using aitextgen. The result was a model which could create encounters but still flawed. The texts often had flaws in them, but were often useable, nonetheless. Missing words or using words in the wrong context are common mistakes. The usage of Dungeons & Dragons rules is often wrong and does not work as intended. Saving throws and skill checks are called for at random and often without any logical reasoning.

The general accuracy of the model was found to be slightly lower than the original GPT-2 model when evaluated using some evaluations frameworks. However, the difference was in general negligible and mostly compared to be comprehensive in the analysis of the model. The model's quality of the generated texts was more difficult to objectively test, especially as there is a lot of subjectivity in what is good when it comes to specific texts such as Dungeons & Dragons encounters. The analysis of the texts was done by the author, generally the texts fit into a few different categories ranging from copying texts from the fine-tuning data the model was trained on or being complete nonsense, to being an interesting and unique encounter.

The right parameters and prompts can achieve unique and interesting encounters for the DM to use, but there is frequently something in the texts which makes them difficult to immediately bring into play. A skilled DM could still use the encounters quickly, but assumptions are often necessary for them to work.

7.1. Future Work

A larger number of encounters in the fine-tuning data, especially where saving throws and skills are used could provide the model with a more accurate knowledge of how these rules are applied in encounters. A larger knowledge of creatures and concepts intrinsic to Dungeons & Dragons would also let the generated encounters be more varied.

The created model was limited to a small GPT-2 model due to the small size of the fine-tuning data and how much memory and size a larger model would require to be used efficiently. GPT-3 was not used due to its limited accessibility. Either a larger GPT-2 model with more fine-tuning data or GPT-3 could realistically create for a much more accurate and varied model. GPT-3 was created with the goal to be able to create specific types of texts with fewer fine-tuning examples so the relatively small fine-tuning data used in this thesis might be enough for it.

Another possibility that was not investigated in this thesis is schema-based generation. Schemas allow the user to input keywords which the generation is based around. Unlike a prompt these keywords do not have to appear at the beginning of the text. Using a schema to ask for DC's or skill checks could allow for more varied and natural text which includes these concepts. Compared to a prompt that would require the DC or the skill check to be generated at the beginning of the text, which forces the model into creating texts in the same way, often leading to repetition.

8. Svensk sammanfattning

8.1. Introduktion

Språk används världen över, och det har till stor del varit varför människan har haft den framgång hon har haft [1]. Datorer har använt språk i ca 80 år. Men att förstå språk är komplicerat, och datorer har ännu idag svårt att förstå de små språkliga detaljerna som används naturligt av människor. Samma ord kan ha olika betydelser beroende på kontext; kontext som det är svårt datorer att förstå [3].

Datorlingvistik är forskningsområdet som fokuserar på att använda datorer för att analysera text. Målet för datorlingvistik är att få en dator att förstå och manipulera naturlig text lika bra som en människa [4]. Målet med denna avhandling är att skapa ett verktyg som med datorlingvistik kan skapa text som kan användas i Dungeons & Dragons Fifth Edition. Målet är att skapa text som kan används för att lägga spelarna i olika scenarier som är varierande och unika. Texten bör också använda termer och regler som används i Dungeons & Dragons på korrekt sätt.

Dungeons & Dragons används som bas för att det är det mest populära bordsrollspelet [5]. Det betyder att det finns en stor mängd text skapad av spelare tillgängligt för den maskinlärda modellen att träna på.

8.2. Dungeons & Dragons

Den första versionen av Dungeons & Dragons gavs ut 1974, och skapades av Gary Gygax och Dave Arneson som ett av de första bordsrollspelen [46]. Sedan dess har flera versioner getts ut, och 2014 kom den senaste versionen ut, kallad Dungeons & Dragons Fifth Edition [47].

Dungeons & Dragons spelas vanligtvis med en spelledare (Dungeon master) och en eller flera spelare. Varje spelare har en karaktär som de spelar och kontrollerar, medan spelledaren spelar alla andra karaktärer och monster. Spelledaren beskriver också platserna där spelarna befinner sig.

Varje karaktär har flera attribut, färdigheter och krafter som de kan använda för att interagera med världen som karaktärerna befinner sig i. Då en spelare använder någon av dessa attribut, färdigheter eller krafter behöver hen ofta kasta en tärning. I Dungeons & Dragons används oftast av sex olika tärningar, och den mest använda är en 20-sidig tärning kallad d20. Då någon försöker göra något som kan misslyckas, kastas en d20 och relevanta bonusar läggs till resultatet, summan jämförs mot en svårighetsgrad (difficulty class) som bestäms av spelledaren eller någon annans attribut. Om summan är den samma eller högre så lyckas handlingen. Andra

tärningar används då ett slumpmässigt nummer i ett visst omfång krävs, som då en attack eller magi gör skada.

För att skapa variation, introducera mera konflikt eller göra något oväntat kan spelledaren använda sig av scener (encounters). Dessa scener används oftast för att bryta upp monotona delar av spelet, såsom att resa långa vägar, och de är ofta relaterade till området spelarna reser igenom. Scenarierna kan vara något som att möta fiender på vägen, eller bara en intressant situation. Ofta kan spelarna interagera med karaktärer eller fiender i dessa scener, antingen genom att slåss mot fienderna eller tala med en annan vänlig resenär.

Målet med avhandlingen är att skapa en modell som kan generera dylika scenarier. Detta kan vara något som är unikt och intressant för spelarna att interagera med, men ska också följa reglerna som används i Dungeons & Dragons. Målet är att spelledaren ska utan att förbereda sig kunna skapa en intressant situation ifall de inte haft tid att planera på förhand.

8.3. Datorlingvistik

Forskning inom datorlingvistik har pågått sedan 1940-talet då datorer blev automatiserade. Maskinöversättning var ett av de första områdena inom datorlingvistik som forskningen fokuserade på, baserat på kryptografiteknologin från andra världskriget [6].

Då detta visade sig vara svårare än forskarna hade förväntat sig ändrade ett fokus i huvudsak till informationshantering och textgenerering, och på 1980-talet skapades de första textgenereringsverktygen. Text skapad av Kathleen R. McKeown kunde svara på frågor om information i verktygets databas. Mumble skapad av David D. McDonald och James D. Pustejovsky kunde skapa text baserat på ett script som definierade ordningen informationen i texten skulle presenteras [7] [8].

Forskningen inom olika områden inom datorlingvistik fortsatte och fokusen ändrades till maskininlärning, och idag används neuronnät vid många tillämpningar av datorlingvistik [10].

8.3.1. Textgenerering

Textgenerering är ett område inom datorlingvistik som fokuserar på att lära datorer generera ny, läsbar text automatiskt. Det finns många olika typer av textgenerering, *mallgenerering* (template filling) baserades på att endast fylla ut texter som förprogrammeras [12] [13]. Djupgenerering är mer avancerad och tillåter generatorn att välja vilka ord som skulle passa bättre i en mening. En Markovkedja är den enklaste

modellen av djupgenerering. Markovkedjan förutspår vilket ord som skulle passa bäst som följande i en mening genom att slumpmässigt välja mellan de ord som oftast följer det nuvarande ordet, med en större chans att välja det vanligaste [15].

Denna strategi används ännu i de mer avancerade modellerna byggda på neuronnät. Dessa modeller har längre minne, vilket betyder att orden som fanns att välja mellan har mer relevans för meningen. Denna metod av textgenerering blev speciellt populär runt 2010-talet, och olika metoder att öka modellernas minne och kapacitet att använda text som genererats tidigare för att skapa mer sammanhängande text har utvecklats [16].

Långt korttidsminne (Long short-term memory, LSTM) var ett tidigt försök att öka minnet på neuronnät. LSTM försöker ta bort information som är onödig, så att endast de viktigaste stannar kvar i minnet. Minnet är ändå begränsat men räcker mycket längre då endart det viktigaste sparas [17].

År 2014 introducerades *attention* (uppmärksamhet), en algoritm som hjälper ett neuronnät att fokusera på de viktigare delarna av en text och inte slösa beräkningskraft på delar som inte är viktiga. LSTM och attention kan användas tillsammans [18].

8.3.2. The Transformer

The Transformer (Transformatorn) är ett neuronnät skapat av Google. Transformatorn fungerar till skillnad från tidigare neuronnät genom parallell bearbetning av data. Det är också det första neuronnätet som baserast helt på attention. Transformatorn använder *self-attention* som jämför indata vid olika steg för att hitta vilka delar som är viktiga jämfört med varandra. Det hjälper att hitta kontext för ord, speciellt ord som kan ha flera betydelser.

Ordet *bank* kan ha olika betydelser beroende på situationen. Det kan vara byggnaden bank eller en sandbank. Genom att jämföra texten kan Transformatorn hitta ifall vatten, sand, stränder, simmande eller dylika ord finns i texten, och förstå att det då är större chans att texten handlar om en sandbank. Finns det ord som byggnad, stad, eller pengar så är det större chans att det handlar om byggnaden bank [20].

8.3.3. GPT

GPT (Generative Pre-trained Transformer) skapades av OpenAI som en transformatorbaserad språkmodell. Hittills har OpenAI skapat tre versioner av GPT (GPT,

GPT-2 och GPT-3.) Dessa modeller tränades på en stor mängd text från internetet. GPT-2 tränades på 8 miljoner webbsidor. Denna avhandling fokuseras på GPT-2.

GPT har till största del tränats på engelska texter, (hela 93 % av GPT-3 träningsdata var på engelska.) Dessa generella träningsdata har tillåtit GPT att lära sig meningsstruktur och ordföljd. Det har också tillåtit modellen att bygga upp ett allmänt sinne för vilka ord som ofta kommer efter varandra. Denna baskunskap kan sedan finjusteras med en mindre mängd specifika texter för att styra modellen att generera en specifik typ av texter. Till exempel, ifall filmmanus används som finjusteringsdata tränas modellen på hur manus skrivs, vilka ord som ofta används, hur de arrangeras och dylikt.

GPT skapades baserat på Googles transformator men har delvis ändrad arkitektur. Det finns inga kodarblock, endast avkodarblock, och dessa avkodarblock ser något annorlunda ut än i den ursprungliga arkitekturen.

GPT har använts till en stor mängd olika tillämpningar. Latitude skapade AI Dungeon inspirerat av textbaserade äventyrsspel som var populära på 1970- och 1980-talen. I de spelen kunde spelaren agera genom att skriva in nyckelord som spelet hade blivit förprogrammerat att förstå (“Gå Norr”, “Öppna Dörren”.) AI Dungeon till skillnad från de äldre äventyrsspelen använder GPT och är inte begränsat till de förprogrammerade nyckelorden. Användaren uppmuntras att skriva längre meningar som förklarar hur de vill agera i olika scenarier och den artificiella intelligensen reagerar och fortsätter berättelsen [31] [32] [33].

8.4. Implementering av ett textgenereringsverktyg

Det första steget vid implementeringen av ett textgenereringsverktyg var att hitta vilken textgenereringsmodell som skulle passa bäst för ändamålet. Tre olika modeller testades. GPT-2 skapat av OpenAI [31], GPT Neo skapat av EleutherAI baserat på GPT-3, men med mer tillgänglighet [52], och textgenrnn, skapat av Max Woolf som är en lättillgänglig och snabb textgenerator [27].

Att textgenrnn är så primitiv gör också att texten den genererar innehåller mycket nonsens och onödig text. Speciellt när texten blir längre blir den nästan oläsbar. Det gör att textgenrnn inte är ett bra val för detta verktyg.

GPT-2 och GPT Neo skapar båda läsbar och användbar text, och båda skulle fungera som val för detta verktyg. Baserat på evalueringar gjorda av både OpenAI och EleutherAI hade GPT-2 något högre exakthet i textförståelse, speciellt på mindre modeller. Därför valdes den minsta versionen av GPT-2 124M. Storleken på modellen ändrar kvalitén av texten som genereras, men också hur mycket data som

krävs för att finjustera modellen och hur mycket minne och utrymme som krävs för att använda den. Eftersom det finns relativt lite finjusteringsdata används en liten modell för detta verktyg.

8.4.1. Finjustering och generering

Efter att modellen har valts behöver den finjusteras. Finjustering är det steg som tränar modellen i hur den ska skapa specifika texter. Texter samlade från olika internetsidor används för att träna modellen på reglerna i Dungeons & Dragons och hur scenerna oftast skrivs. Texterna är till stor del skrivna av spelare som spelar Dungeons & Dragons.

Den viktigaste parametern för finjusteringen är inlärningshastigheten (learning rate). Inlärningshastigheten justerar hur mycket modellen ändrar vid varje steg av finjusteringen, ju mindre den är desto längre kommer det räcka innan modellen når en ideal träningsnivå. Men är parametern för stor kommer modellen bli överanpassad (overfitted) och endast kopiera text från finjusteringstexten i stället för att generera egen. Baserat på inlärningshastigheten krävs olika mängd tid för finjustering [22].

Efter att modellen har blivit finjusterad på finjusteringsdata så kan den användas för att generera text. Det finns flera parametrar för att justera hur texten genereras efter finjusteringen. En uppmaning (prompt) som modellen använder att börja med och fortsätter generera från, är viktig och tillåter hög kontroll över hur modellen genererar text.

Temperatur kontrollerar hur slumpmässigt val av nästa tecken som blir genererad kommer vara. Högre temperaturer betyder mer slumpmässigt val. En låg temperatur betyder att ordet eller tecknet med högst sannolikhet att följa det nuvarande tecknet alltid kommer bli valt. Detta gör modellen deterministisk, men för höga temperaturer kan göra att genererade texter blir oläsbara [24].

Parametrarna Top-K och Top-p ämnar minimera mängden val som modellen har vid varje steg. Top-K tar bort alla andra val än de med k högst sannolikhet att följa det nuvarande tecknet, och Top-p samlar den minsta satsen vars sammanlagda sannolikhet når p. Tillsammans skapar Top-K och Top-p en mindre uppsättning av ord som modellen har att välja mellan, men varje ord bör passa bättre i meningen [25] [24]. Optimal Top-K, Top-p och temperatur kan styra generatorn att generera intressanta och unika kombinationer.

8.4.2. Test och analys

Den färdiga modellen jämförs med de ursprungliga modellerna, med samma verktyg. Det visar sig att den finjusterade modellen har något sämre noggrannhet än de ursprungliga modellerna. Dessa evalueringar testar modellens allmänna noggrannhet, och eftersom denna modell skapades för att göra specifika texter blir noggrannheten lägre. Kvalitén på de genererade texterna har inte påverkats.

Själva texterna som genereras av modellen kan läggas in i några olika kategorier:

- *Kopierande och utbyte av ord*
- *Unik men nonsens*
- *Fel eller ingen användning av regler*
- *Unik och användbar efter bearbetning*
- *Ideala scenarier*

De flesta scenarier som genereras är inte ideala och något misstag finns i texten. Några av dessa kan ändå användas ifall spelledaren bearbetar scenariot. Många har något ord fel använt, speciellt ifall det är ett ord som hör specifikt till Dungeons & Dragons.

8.5. Resultat

Målet med denna avhandling var att skapa ett verktyg som kan skapa unika scenarier till Dungeons & Dragons med textgenerering baserad på neuronnät. Dessa scenarier skulle helst följa spelets regler och vara användbara utan att kräva extra förberedelsetid för spelledaren.

Verktyget som skapades för avhandlingen kan generera unika texter, och texterna är ofta lämpliga för Dungeons & Dragons. På grund av modellens oförutsägbarhet skapas dock inte alltid texten så att den blir användbar. Ibland kan dessa texter omarbetas och användas, men några texter förblir helt oanvändbara. Att använda regler vid genereringen fungerar inte heller alltid och då reglerna tillämpas korrekt beror det oftast slumpmässigt. En annan strategi för att implementera reglerna i modellen skulle kunna undersökas i framtiden för att förbättra resultaten.

9. References

[1] E. A. Smith, “Communication and collective action: language and the evolution of human cooperation,” *Evolution and Human Behavior,* vol. 31, no. 4, pp. 231-245, 2010.

[2] D. Smith, P. Schlaepfer, K. Major, M. Dyble, A. E. Page, J. Thompson, N. Chaudhary, G. D. Salali, R. Mace, L. Astete, M. Ngales, L. Vinicius and A. Bamberg Migliano, “Cooperation and the evolution of hunter-gatherer storytelling,” *Nature Communications,* no. 8, 2017.

[3] L. Yao, N. Peng, R. Weischedel, K. Knight, D. Zhao and R. Yan, “Plan-and-Write: Towards Better Automatic Storytelling,” *AAAI,* vol. 33, no. 1, pp. 7378-7385, 2019.

[4] E. D. Liddy, “Natural Language Processing,” in *Encyclopedia of Library and Information Science, 2nd Ed.*, New York, Marcel Decker, Inc., 2001.

[5] Roll20, “Roll20 Blog: The Orr Group Industry Report: Q1 2020 - Reimagining the Classics,” 5 May 2020. [Online]. Available: https://blog.roll20.net/post/617299166657445888/the-orr-group-industry-report-q1-2020. [Accessed 18 January 2021].

[6] Department of Linguistics, University of Pennsylvania, “Transformations and Discourse Analysis Project,” University of Pennsylvania, [Online]. Available: https://cs.nyu.edu/cs/projects/lsp/pubs/tdap.html. [Accessed 24 Januray 2021].

[7] K. R. McKeown, “The TEXT system for natural language generation: an overview,” *Proceedings of the 20th Annual Meeting on Association for Computational Linguistics,* pp. 113-120, 1982.

[8] D. D. McDonald and J. D. Pustejovsky, “A computational theory of prose style for natural language generation,” *Proceedings of the Second Conference on European Chapter of the Association for Computational Linguistics,* pp. 187-193, 1985.

[9] E. Kumar, Natural Language Processing, New Delhi: I.K. International Publishing House Pvt. Ltd., 2011.

[10] Y. Goldberg, "A Primer on Neural Network Models for Natural Language Processing," *Journal of Artificial Intelligence Research,* no. 57, pp. 345-420, 2015.

[11] R. Dale, H. Moisl and H. Somers, Handbook of Natural Language Processing, New York: Marcel Dekker, Ink, 2000.

[12] C. Manning and H. Schutze, Foundations of Statistical Natural Language Processing, Massachusetts: MIT Press, 1999.

[13] P. Semaan, "Natural Language Generation: An Overview," *Journal of Computer Science & Research,* vol. 1, no. 3, pp. 50-57, 2012.

[14] E. Reiter and R. Dale, Building Natural Language Generation Systems, Cambridge: Cambridge University Press, 2000.

[15] P. A. Gagniuc, Markov Chains: From Theory to Implementation and Experimentation, Hoboken: John Wiley & Sons, Inc, 2017.

[16] L. Medsker and L. C. Jain, Recurrent Neural Networks: Design and Applications, CRC Press, 2001.

[17] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," *Neural Computation,* vol. 9, no. 8, pp. 1735-1780, 1997.

[18] D. Bahdanau, K. Cho and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," arXiv:1409.0473, 2014.

[19] A. Galassi, M. Lippi and P. Torroni, "Attention in Natural Language Processing," *IEEE Transactions on Neural Networks and Learning Systems,* 2020.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoereit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention Is All You Need," in *31st Conference on Neural Information Processing Systems*, Long Beach, 2017.

[21] Z. Xie, "Neural Text Generation: A Practical Guide," arXiv:1711.09534, 2018.

[22] S. Ruder, “An overview of gradient descent optimization algorithms,” arXiv:1609.04747, 2017.

[23] I. Loshchilov and F. Hutter, “Decoupled Weight Decay Regularization,” arXiv:1711.05101, 2019.

[24] A. Holtzman, J. Buy, L. Du, M. Forbes and Y. Choi, “The Curious Case of Neural Text Degeneration,” arXiv:1904.09751, 2019.

[25] A. Fan, M. Lewis and Y. Dauphin, “Hierarchical Neural Story Generation,” arXiv:1805.04833, 2018.

[26] D. Jurafsky and J. H. Martin, Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, Upper Saddle River: Prentice Hall, 2008.

[27] M. Woolf, “textgenrnn,” 24 April 2018. [Online]. Available: https://github.com/minimaxir/textgenrnn/. [Accessed 16 November 2020].

[28] A. Karpathy, “char-rnn,” 21 May 2015. [Online]. Available: https://github.com/karpathy/char-rnn. [Accessed 21 February 2021].

[29] M. Woolf, “Tweet Generator,” 13 April 2018. [Online]. Available: https://github.com/minimaxir/tweet-generator. [Accessed 2021 February 8].

[30] M. Woolf, “SubredditNN,” 4 May 2018. [Online]. Available: https://www.reddit.com/r/subredditnn. [Accessed 2021 February 8].

[31] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, “Language Models are Unsupervised Multitask Learners,” OpenAI, San Francisco, 2019.

[32] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agrawal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clarke, B. Chrispher, S. McCandlish, A. Radford, I. Sutskever and D. Amodei, “Language Models are Few-Shot Learner,” OpenAI, 2020.

[33] A. Radford, K. Narasimhan, T. Salimans and I. Sutskever, “Improving Language Understanding by Generative Pre-Training,” OpenAI, San Francisco, 2018.

[34] u/disumbrationist, “SubSimulatorGPT2,” 27 May 2019. [Online]. Available: https://www.reddit.com/r/SubSimulatorGPT2/. [Accessed 24 March 2021].

[35] “AI Dungeon,” Latitude, 5 December 2019. [Online]. Available: https://play.aidungeon.io/main/home. [Accessed 24 March 2021].

[36] K.-L. Chiu and R. Alexander, “Detecting Hate Speech with GPT-3,” arXiv preprint arXiv:2103.12407, Toronto, 2021.

[37] A. Ohlheiser and K. Hao, “An AI is training counselors to deal with teens in crisis,” 26 February 2021. [Online]. Available: https://www.technologyreview.com/2021/02/26/1020010/trevor-project-ai-suicide-hotline-training/. [Accessed 6 April 2021].

[38] S. Kreps and M. McCain, “Not Your Father's Bots,” Foreign Affairs, 2 August 2019. [Online]. Available: https://www.foreignaffairs.com/articles/2019-08-02/not-your-fathers-bots. [Accessed 23 February 2021].

[39] G. Brockman, M. Murati, P. Welinder and OpenAI, “OpenAI API,” OpenAI, 11 June 2020. [Online]. Available: https://openai.com/blog/openai-api/. [Accessed 23 February 2021].

[40] RosaeNLG, “RosaeNLG Documentation,” [Online]. Available: https://rosaenlg.org/. [Accessed 27 January 2021].

[41] D. Howe and J. Cheung, “RiTa: a toolkit for generative writing and natural language,” [Online]. Available: https://github.com/dhowe/rita/. [Accessed 27 January 2021].

[42] M. Sample, “The Godard Film Generator,” [Online]. Available: https://rednoise.org/rita/gallery/TheGodardFilmGenerator/. [Accessed 16 February 2021].

[43] D. C. Howe, “RiTa: creativity support for computational literature,” in *C&C '09: Proceedings of the seventh ACM conference on Creativity and cognition*, Berkeley, 2009.

[44] Google Research, “Welcome to Colaboratory,” Google LLC, 2017. [Online]. Available: https://colab.research.google.com/notebooks/intro.ipynb. [Accessed 18 April 2021].

[45] M. Woolf, “gpt-2-simple,” 14 April 2019. [Online]. Available: https://github.com/minimaxir/gpt-2-simple. [Accessed 19 April 2021].

[46] G. Gygax and D. Arneson, Dungeons & Dragons, Lake Geneva: Tactical Studies Rules, 1974.

[47] J. Crawford, M. Mearls, B. R. Cordell, J. Wyatt and R. J. Schwalb, D&D Player's Handbook, Renton: Wizards of the Coast LLC, 2014.

[48] M. Woolf, “aitextgen,” 29 December 2019. [Online]. Available: https://github.com/minimaxir/aitextgen. [Accessed 7 June 2021].

[49] PyTorch, “PyTorch,” 2021. [Online]. Available: https://pytorch.org. [Accessed 13 June 2021].

[50] W. Falcon and et al., “PyTorch Lightning,” *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning,* vol. 3, 2019.

[51] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest and A. M. Rush, “Transformers: State-of-the-Art Natural Language Processing,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Association for Computational Linguistics, 2020, pp. 38-45.

[52] S. Black, L. Gao, P. Wang, C. Leahy and S. Biderman, “GPT Neo,” EleutherAI, 2021. [Online]. Available: https://github.com/EleutherAI/gpt-neo. [Accessed 2021 June 13].

[53] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser and C. Leahy, “The Pile: An 800GB Dataset of Diverse Text for Language Modeling,” arXiv:2101.00027, 2020.

[54] L. Gao, J. Tow, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, K. McDonell, N. Muennighoff, J. Phang, L. Reynolds, E. Tang, A. Thite, B. Wang, K. Wang and A. Zou, “A framework for few-shot language model evaluation,” Zenodo, September 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5371628. [Accessed 16 February 2022].

[55] D. Paperno, G. Kruszewski, A. Lazaridou, Q. N. Pham, R. Bernardi, S. Pezzelle, M. Baroni, G. Boleda and R. Fernández, “The LAMBADA dataset:

Word prediction requiring a broad discourse context," arXiv:1606.06031, 2016.

[56] S. Merity, C. Xiong, J. Bradbury and R. Socher, "Pointer Sentinel Mixture Model," 2016.

[57] K. Sakaguchi, R. Le Bras, C. Bhagavatula and Y. Choi, "WinoGrande: An Adversarial Winograd Schema Challenge at Scale," arXiv:1907.10641, 2019.

[58] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi and Y. Choi, "HellaSwag: Can a Machine Really Finish Your Sentence?," arXiv:1905.07830, 2019.

[59] G. Brockman, I. Sutskever and OpenAI, "Introducing OpenAI," OpenAI, 11 December 2015. [Online]. Available: https://openai.com/blog/introducing-openai/. [Accessed 18 November 2021].

[60] EleutherAI, "About Us | EleutherAI," EleutherAi, 2021. [Online]. Available: https://www.eleuther.ai/about/. [Accessed 18 November 2021].

[61] S. Bird, E. Loper and E. Klein, Natural Language Processing with Python, O'Reilly Media Inc., 2009.

[62] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, 2010.

[63] Diacritic, Artist, *Dice (typical role playing game dice).* [Art]. Wikimedia Commons, 2010.

[64] P. L. Rocco, Artist, *Dungeons and Dragons game.* [Art]. Wikimedia Commons, 2005.