University
of Agder

Master

Transformer Reinforcement Learning for Procedural Level Generation

SEBASTIAN BEKKVIK AAS, LUKASZ FILIP MROZIK

SUPERVISOR
Per-Arne Andersen

University of Agder, 2023
Faculty of Engineering and Science
Department of Engineering and Sciences

Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.	Ja
2.	**Vi erklærer videre at denne besvarelsen:** • Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • Ikke refererer til andres arbeid uten at det er oppgitt. • Ikke refererer til eget tidligere arbeid uten at det er oppgitt. • Har alle referansene oppgitt i litteraturlisten. • Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	Ja
3.	Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31.	Ja
4.	Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.	Ja
5.	Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.	Ja
6.	Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.	Ja
7.	Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet.	Nei

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).
Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Acknowledgements

Thank you to Per-Arne Andersen for guidance throughout the project duration.

Abstract

This paper examines how recent advances in sequence modeling translate for machine learning assisted procedural level generation. We explore the use of Transformer based models like DistilGPT-2 to generate platformer levels, specifically for the game Super Mario Bros., and explore how we can use reinforcement learning to push the model towards a task like generating levels that are actually beatable. We found that large language models (LLMs) can be used without any major modifications from the original NLP focused models to instead generate levels for the aforementioned game.
However, the main focus of the research is connected to how advancement in the area of NLP by the use of reinforcement learning (RL) algorithms, specifically PPO, translates to the arena of procedural level generation in cases where the levels can be treated as token sequences.
We did however not find any combinations of hyperparameters that allowed the PPO to reach higher better results than our baseline model trained for next token prediction. Despite it's success in the area of NLP, we failed to find a combination of hyperparameters that improved upon the level generation by applying an reward for the whole level. However there are methods that we did not try yet, like finding specific parts of the level to reward and penalize.

Contents

Acknowledgements **ii**

Abstract **iii**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**
1.1 Large Language Models 2

2 Background **3**
2.1 Super Mario Bros. Level Generation 3
2.1.1 The Video Game Level Corpus 3
2.2 State of the art 4
2.2.1 Summerville and Mateas' LSTM 5
2.2.2 DagstuhlGAN 6
2.2.3 MarioGPT 7
2.3 Transformers 8
2.3.1 Multi-Headed Self Attention 8
2.3.2 GPT 9
2.3.3 PPO 9

3 Method **12**
3.1 Representation 12
3.2 Model 13
3.3 PPO 15

4 Experimental Results **16**
4.1 Reinforcement Learning 17

5 Discussions **19**
5.1 Autoregressive models 19
5.2 Negative Log-Likelihood as a evaluation score 20
5.3 Pre-training by other models 21
5.4 Divergence from training dataset 21

6 Conclusion **22**
6.1 Future Work 22
6.1.1 PPO avoiding invalid levels 22
6.1.2 Fine-tuning from human preference 22

A Examples of generated levels **24**

Bibliography **29**

List of Figures

2.1 Which character the Video Game Level Corpus uses for each Super Mario Bros. tile. 4
2.2 Snaking contrasted with regular column-wise parsing. 5
2.3 The architecture of the GAN developed by Volz et. al.[21] 7
2.4 The architecture of the MarioGPT model. 7
2.5 The architecture of Vaswani et. al.'s Transformer.[20] 9

3.1 The textual representation of a generated level, compared with with the game's textures. 13
3.2 A sketch of our model architecture. 14

4.1 DistilGPT-2 model fine-tuned for sequences of Super Mario Bros. tiles with no reinforcement learning. The first 2 columns is the prompt, telling the model that the level begins with ground at the lowest elevation 17
4.2 Training and evaluation negative log-likelihoods. 17
4.3 Level generated with high KL-divergence 17
4.4 Level generated with low KL-divergence 18

A.1 Possible to complete generated levels no PPO. 24
A.2 Impossible to complete generated levels no PPO. 25
A.3 Possible to complete generated levels KL target 0.1 26
A.4 Impossible to complete generated levels KL target 0.1 27
A.5 Impossible to complete generated levels KL target 6 28

List of Tables

3.1 Percentage distribution of tiles ignoring bottom tokens 15

4.1 Percentage distribution of symbols 18

Chapter 1

Introduction

Video games across various genres derive much of their gameplay appeal from the intricacies and challenges offered by their level designs. Among these genres, platformer games specifically rely heavily on well-crafted and engaging layouts to deliver a satisfying experience. To address the need for diverse and dynamic levels, game developers have increasingly turned to procedural level generation and other types of procedural content generation (PCG), which is the generation of characters, enemies, game rules, etc., all without human intervention. Notable examples of effective use of procedural level generation in the genre of platformers include *Spelunky*, *Dead Cells* and *Rogue Legacy*. The level generation greatly improves the replayability and adds an element of surprise to each run.

Super Mario Bros.[6], one of the most iconic and influential platformer games, has served as a captivating canvas for exploring the possibilities of procedural level generation. The game's rich history and enduring popularity have inspired researchers and enthusiasts to delve into the realm of procedural level generation, leading to the establishment of notable competitions such as the Mario AI Championship[12] and the Platformer AI Competition[13]. These events have served as platforms for showcasing innovative algorithms and techniques in generating Mario levels that can be traversed and conquered by both human players and intelligent agents.

In this thesis we explore the possibilities in the realm of procedural level generation by using techniques from another field of AI focused on generation, Natural Language Processing (NLP). The idea is not new [16], but because the advancements in the area of large language models (LLMs), it is worth looking at them and how they can be applied to procedural level generation.

What sets our approach apart from previous proposed solutions is the training objective. In many of the previous approaches mentioned in the Background section, the goal of the training is simply to learn the patterns in existing levels by the means of techniques such as sequence modelling in the form of next token prediction or GANs. However there has been previous research in using evolutionary algorithms to maximize desired properties of GANs based generators on arbitrary objectives by searching for a subset of the latent space. In this thesis we will explore the use reinforcement learning (RL) algorithm PPO in order to maximize desired properties in the case of sequence modelling.

1.1 Large Language Models

In recent years, there has been a notable advancement in the field of large language models (LLMs). Notably ChatGPT, derived from InstructGPT[5], has emerged as a prominent example. InstructGPT employs a reinforcement learning approach to fine-tune the GPT-3 model, which was initially trained on a large unlabeled dataset for the with for predicting the next token in a token sequence.

The remarkable progress observed in LLM research can be largely attributed to the effectiveness of scaling both the dataset and the model size in the case of transformer based model architecture. The GPT-3 model, for instance, incorporates a variant with an estimated 175 billion parameters. Meanwhile, the technical details of GPT-4 are undisclosed as of May 2023 due to the model's proprietary nature.

Inspired by the achievements of LLMs, this thesis aims to apply similar treatment to *Super Mario Bros.*[6] levels by considering them as sequences of tokens, similar to how natural language is processed. Each square in a Super Mario level can be regarded as a token, enabling us to generate a Super Mario level in an autoregressive manner. Specifically, this thesis will investigate the utilization of reinforcement learning methods to guide the model in generating playable levels. The objective is to encourage the model to generate levels that are feasible to complete.

This thesis will first examine the body of work making up the state of the art, in three parts. First, the game in question and its representation chosen for this project is discussed. Secondly, several of the primary papers in the line of research will be discussed, with particular regards to their methods used. Lastly, this project's chosen technology will be described.

Following that, our method will be elaborated upon. Not only will we go in depth on our chosen model and architecture, but also where we build upon the earlier work, as well as our reasoning for these design choices.

The thesis then goes on to list notable results of the methods, including graphs of numerical performance and examples of generated levels.

Following that, we discuss the results and what they mean for the project, ending on a conclusion and suggestions for future work.

Chapter 2

Background

As we begun this project, it was first necessary to understand the state of the art, as is it when understanding this project. This chapter will describe the preceding work for both the problem we are addressing, *Super Mario Bros.* level generation; and the architecture used to address it, a Generative Pre-trained Transformer (GPT) structurally inspired by InstructGPT and its derivatives.

2.1 Super Mario Bros. Level Generation

Super Mario Bros., released by Nintendo in 1985 on the Nintendo Entertainment System[6], is a 2D platformer game that served as the foundation for Nintendo's long-running Mario franchise. The game was chosen for this project in large part for its tile-based level design, easy to represent and work with both for computers of the time and our machine learning systems. Procedural level generation for *Super Mario Bros.* has gotten attention in recent years through various PCG competitions[21], particularly alongside adjacent development in AI agents playing the levels, Like the winner of the Mario AI competition ICE-GIC, A*[1], based on the pathfinding algorithm by the same name. This in particular has let projects like this one test the playability of their generated levels with minimal effort.

2.1.1 The Video Game Level Corpus

Our project uses level transcriptions from the Video Game Level Corpus[17], containing the levels from a variety of games for the NES/SNES consoles and other games of that era. These transcriptions, released in 2016[17], were made with machine-readability in mind. The project's abstract describes the need for accessible, usable training data in procedural level generation using machine learning. Notably, of the levels from *Super Mario Bros.* and *Super Mario Bros. 2*, the VGLC does not include levels set underwater or in castles. This is presumed done because of the difference in level design between those and the remaining levels.

[1]http://julian.togelius.com/mariocompetition2009/GIC2009Competition.pdf

Representation

In the *Video Game Level Corpus*, *Super Mario Brothers* levels are represented using text files. The levels are considered as a grid of tiles, of size equal to that of the level. The text file then codifies each tile as a character, the character used depending on the tile's type according to a JSON specification.

```
{
    "tiles" : {
        "X" : ["solid","ground"],
        "S" : ["solid","breakable"],
        "-" : ["passable","empty"],
        "?" : ["solid","question block", "full question block"],
        "Q" : ["solid","question block", "empty question block"],
        "E" : ["enemy","damaging","hazard","moving"],
        "<" : ["solid","top-left pipe","pipe"],
        ">" : ["solid","top-right pipe","pipe"],
        "[" : ["solid","left pipe","pipe"],
        "]" : ["solid","right pipe","pipe"],
        "o" : ["coin","collectable","passable"],
        "B" : ["Cannon top","cannon","solid","hazard"],
        "b" : ["Cannon bottom","cannon","solid"]
    }
}
```

Figure 2.1: Which character the Video Game Level Corpus uses for each Super Mario Bros. tile.

[17]

Notably, the VGLC collapses some categories of tiles into a single character in their representation. Examples include impassable and unbreakable blocks like ground, large mushroom caps and stairs all being represented with 'X', and every kind of enemy being represented with 'E'.

For solid blocks, this works well, as all the blocks behave the same way, only graphics differing. However, some information is lost in regards to enemies, as several kinds of enemies exist with unique behavior. Thus, when treating them as one for the purpose of annotating levels, an AI model generating levels would place them as though the same kind of enemy.

Often, this is not a big problem, as which enemy type something is supposed to be is visible from context. Pirhana Plants are always placed on pipes, Lakitus are the only enemies in the air, and so on.[6]

2.2 State of the art

This project is built upon a history of work on the topic, which this section will detail. Primarily, their approaches will be discussed, both in terms of model architecture and other related facets of their approaches that are relevant to the subject.

2.2.1 Summerville and Mateas' LSTM

The 2016 paper *Super Mario as a String: Platformer Level Generation Via LSTMs*[16] by Summerville and Mateas explores level generation using an LSTM, or Long-Short Term Memory, and discusses ways to represent the training data.

LSTMs are a subcategory of Recurrent Neural Networks (RNNs), which by sending the output of certain nodes to other ones earlier in the network, can "remember" past inputs and decisions.[14] LSTMs, specifically, were developed to solve the Vanishing Problem with RNNs, where after 5-10 iterations, backpropagated error signals fluctuate wildly or vanish.[14] In response, LSTMs use Constant Error Carousels, tightly regulating access to special cells, thus maintaining a strict error flow and retaining the backpropagated information.

In their paper, Summerville and Mateas explore ways to encode the levels to make the most sense to their LSTM, discussing snaking, A*'s pathing information, and an additional character every couple columns indicating the progress into the level.[16]

Snaking is a technique used in encoding the level data for input.[16] Instead of recording each column from top to bottom, a snaking parser alternates between top to bottom and bottom to top, even running over the same level twice by alternating which direction it starts on. In this paper, snaking is used to keep pipes closer together in the encoded string.

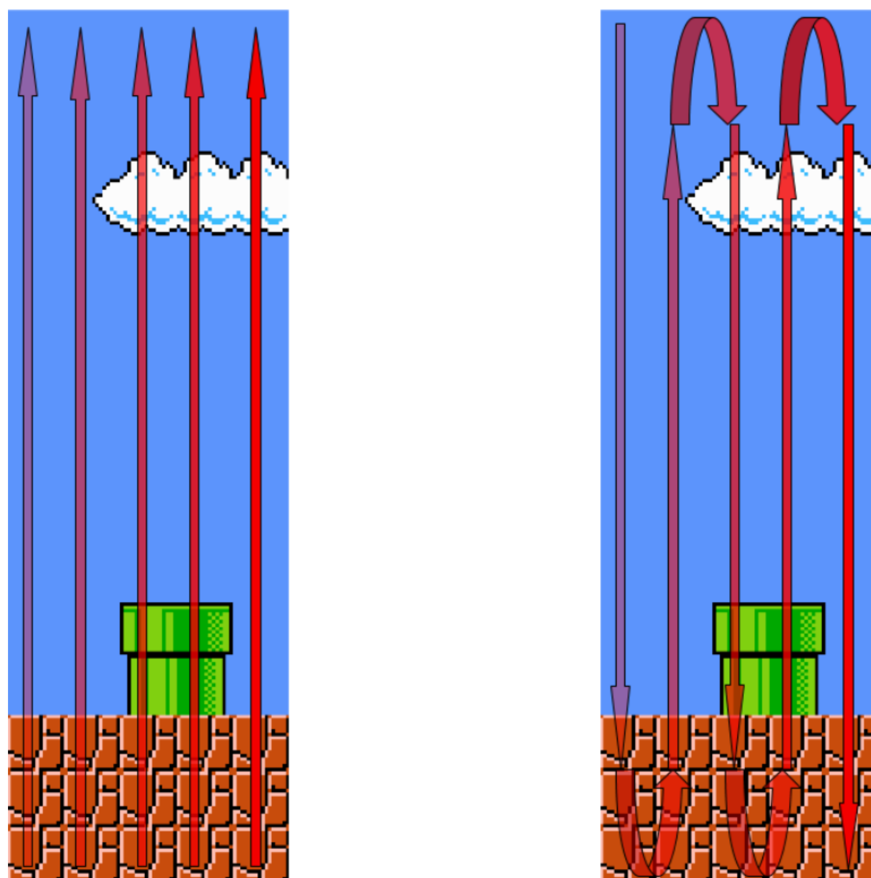


Figure 2.2: Snaking contrasted with regular column-wise parsing.

Pipes are the only features of *Super Mario Bros.* levels spanning more than one tile. In order to reliably fit the entire pipe in the LSTM's short-term memory, snaking both ways is used to keep both sections of the pipe close together in at least one dataset, separated only by the ground rather than by several tiles of air.

A* is as mentioned a pathfinding algorithm developed in 1968 by Hart, Nilsson and Raphael[3], though in this context the term refers to an agent by the same name by Baumgarten[12][19], trained for efficiently navigating *Super Mario Bros.* levels. It allows developers to not only verify whether generated levels are beatable, but the path it finds to be the most efficient through the level can inform the level design of future iterations.

Summerville and Mateas encode depth information by adding a unique token to every five columns. That is because they do not always begin training at the beginning of the level, rather when training the LSTM they used strings 200 tokens which encode around 12 columns. The reasoning behind adding the current position in the level as an additional encoding is how levels can sometimes change intensity based on how close the player is to the end of the level.

The study ultimately concludes that using all three techniques result in significantly better performance than any other combination of them, performance in this case being based on the negative log-likelihood the LSTM had on existing levels that were in the test set from a 70%-30% training-test split used for this project. The generated levels' similarity to the original ones was judged along several other metrics such as the ratio between blocks, enemies and empty space; negative space; linearity of optimal path through; and amount of jumps in that path.

2.2.2 DagstuhlGAN

Published in 2018, *Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network* by Volz et. al.[21] describe their attempt to use a PyTorch implementation of a Wasserstein GAN to generate levels for *Super Mario Bros.*

Their process is twofold. First, a generator with Gaussian noise and a discriminator compete to produce levels that appear like the original levels, typical for a GAN. Secondly, treating the generator/discriminator-pair as a mapping from an input vector to a tile-level description of a generated level, they then use CMA-ES to explore the latent vector space of input-to-level and categorizing them by several fitness functions.

CMA-ES, short for Covariance Matrix Adaptation Evolutionary Strategy[21], is a category of evolutionary algorithms in which the problem space, expressed as a multivariable distribution, seeks to optimize the pairwise dependencies of those variables, expressed as a covariance matrix.

When encoding the existing levels for their training set, they used two unique approaches. Firstly, their training set is made from a single level, the first level from *Super Mario Bros.* They write "By training on only a single level, we are able to show that even with a very limited dataset, we can apply the presented approach successfully."[21] Due to the low size of the dataset, even when using the additional levels from *Super Mario Bros. 2*, proving their concept worked on limited data was crucial.

Secondly, they encoded the level as a series of screen-wide snapshots, 28 by 14 tiles, sliding tile by tile across the level.

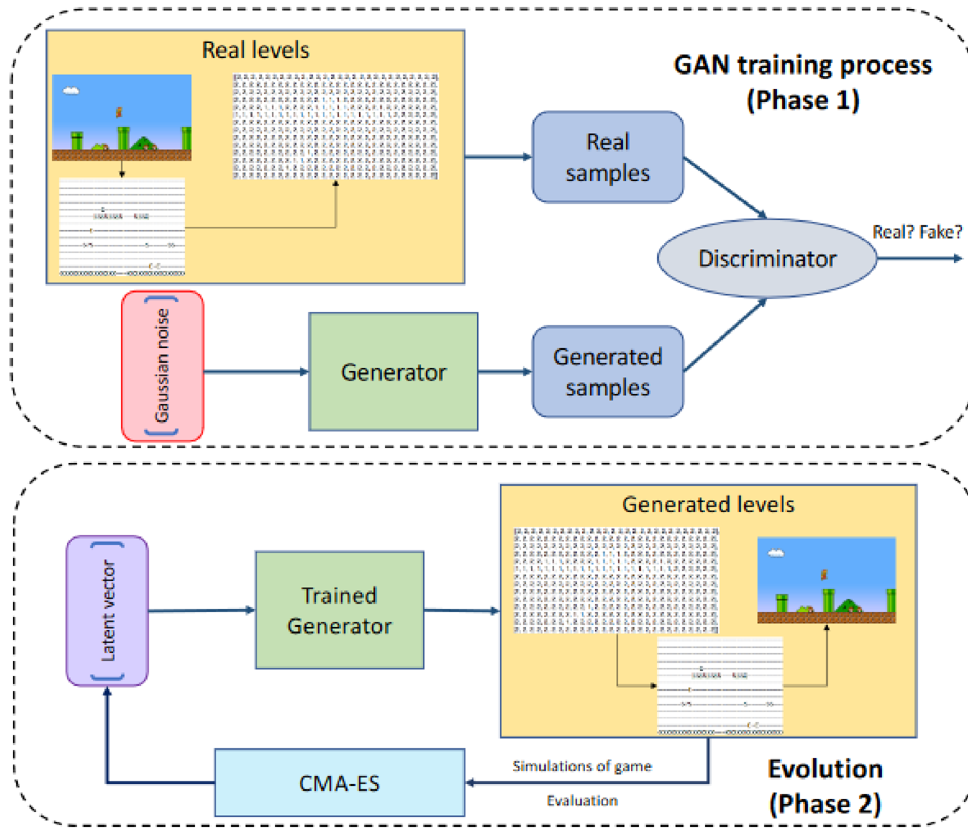


Figure 2.3: The architecture of the GAN developed by Volz et. al.[21]

2.2.3 MarioGPT

MarioGPT is a similar project that released during this project. It uses a combination of DistilGPT-2 and a BERT transformer to generate levels from a prompt. The BERT model allows the user to type a prompt describing the level and the amount of pipes, enemies, blocks and height variation it contains. Next, the latent encoding from the BERT model is used for generating the level similar to previous LSTM based implementations.[15]

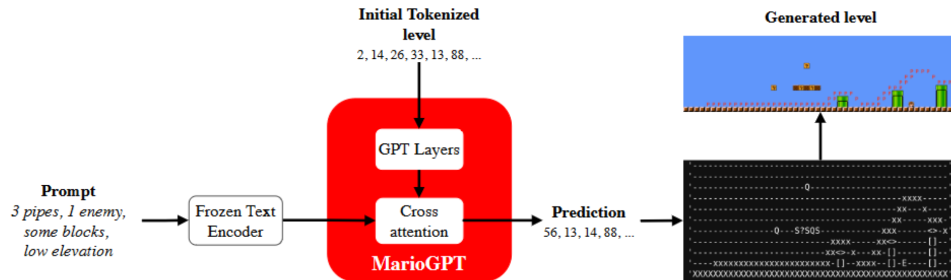


Figure 2.4: The architecture of the MarioGPT model.
[15]

Sudhakaran et. al. compare their work to previous projects, mentioning previous work using A* to check for playability and for finding the optimal path through the level compared to them instead using levels with the optimal path through annotated in advance to train. They also handle the generation of levels in response to a prompt, contrasted with the work of Volz et. al. preemptively generating the levels and then exploring their vector space for suitable levels afterwards.

Unlike *Super Mario as a String: Platformer Level Generation Via LSTMs* by Summerville and Mateas[16], they do not evaluate or mention the negative log-likelihood as a way to tell which model is best. Instead they evaluate models based on how much different the actual path taken by A* agents is from the path the neural network predicted and the percentage of tiles the model guesses correctly. Despite the model performing worse than the LSTM implementation when trained from scratch, the MarioGPT model from fine-tuning a pretrained DistilGPT-2 model performs better in both tile accuracy and path accuracy.[15]

2.3 Transformers

Introduced in 2017 in the paper *Attention Is All You Need* by Ashish Vaswani et al.[20], Transformers are a type of attention-based neural network showing superior performance in the field of NLP. This architecture, unlike previous architectures like CNN or LSTM, has a constant number of operations to relate signals from two arbitrary input or output positions, making it easier for the model to relate two signals that are far away from each other. Transformers do this by avoiding layers like LSTM and convolutions, instead opting for the use of only attention. This is the reason for the papers title: *Attention is all You Need*, the models for sequence modeling and transduction do not need convolutions and LSTM on top of attention. Transformer based architectures have shown superior performance in various NLP tasks such as machine translation [20], language understanding and question answering.[1][4][8]

2.3.1 Multi-Headed Self Attention

Attention mechanisms, the paper describes, is a method for sequence description and transduction, that used to be normally paired with convolution or an RNN. A variant, self-attention, uses solely connections between positions in a single sequence. More specifically, attention used in *Attention Is All You Need* by Ashish Vaswani et al.[20] is Multi-Headed Self Attention.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_K}})V \tag{2.1}$$

An head in the attention mechanism refers an an instance to scaled dot-product attention defined by Equation 2.1[20]. Here Q stands for query, K stands for key and V stands for value. Both keys and values represent tokens in the sequence and together they represent the attention between them. The value matrix stores information that can be retrieved based on thee attention weights. The query represents the current position word Multiple instances of scaled dot-product attention is used in parallel and the results concatenated to be used by a following linear layer as shown in Equation 2.2. Running multiple attention instances is called multi-head self attention. Figure 2.5 shows how the attention layers are used in the original transformer architecture.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) \cdot W^O \tag{2.2}$$

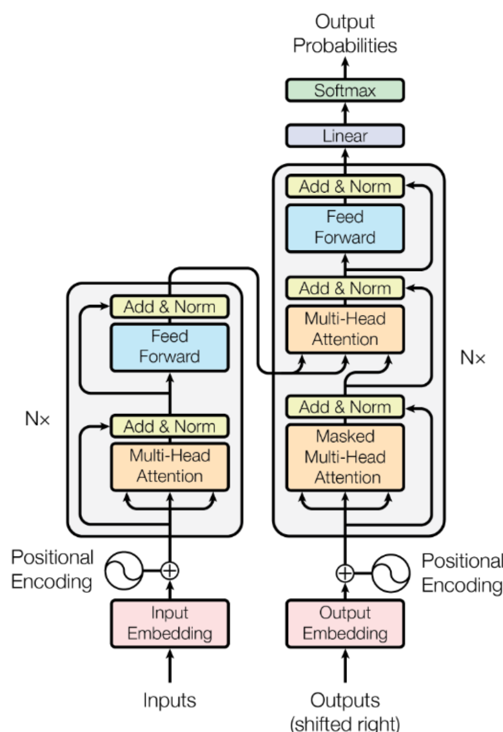


Figure 2.5: The architecture of Vaswani et. al.'s Transformer.[20]

2.3.2 GPT

The original GPT model, introduced by Radford et al. in *Improving language understanding by generative pre-training*[7] marked a major milestone in NLP by demonstrating impressive preformance on various language tasks. GPT builds upon the Transformer architecture (see section 2.3) and by pretraining on a massive corpus of unannotated text data, GPT is able to learn a rich representation of natural language.
GPT-2 is a subsequent iteration of the GPT series released in 2019[8]. The model has a larger scale with up to 1.5 billion parameters and a bigger training dataset (went from approximately 5GB to 40GB of text). It also introduces a "top-k" sampling to avoid sampling very unlikely logits resulting in an nonsensical response. There exists an distilled version of GPT-2 model based on the distillation method from DistilBERT[9].
Generative, Pre-trained Transformers (GPT) are a series of transformer models released by OpenAI[7], based around generatively pre-training a transformer using a large corpus of text, then later discriminatively fine-tuning it with text specific to the target domain. In its introductory paper, GPT was tested on tasks like question answering and natural language inference. Today, a variety of GPT-derived models exist, like BERT[1].

2.3.3 PPO

The vanilla Policy Gradient (PG) introduced in *Policy gradient methods for reinforcement learning with function approximation* by Sutton et al.[18] In policy gradient methods learn a policy function that maps states to actions. To do this we first collect trajectories, that is the sequences of states, actions and rewards experienced by the agent. Next we compute policy gradients and use them to update the parameters of our policy. The policy gradients are the logarithm of the policy's probability distribution multiplied by an estimated advantage

which is typically the returns we got for the actions we took with an estimated values function subtracted from it. Some limitations of the Vanilla Policy Gradient methods are that they only use samples from the environment one time leading to bad sample efficiency, and also suffer from instability caused by the policy performing a big step that changes it significantly.

To address limitations of vanilla Policy Gradients John Schulman et al. instroduced in the paper *Trust Region Policy Optimization*[11] Trust Region Policy Optimization constraint the update of the policy within a "trust region" based on KL divergence from the policy before the step in order to avoid making the step too big.

PPO[10] was invented to be simpler than TRPO, but the authors found the additional benefit of the sampling efficiency being empirically higher than that of TRPO. Like TRPO and PG, PPO updates the policy based on the policy gradient, but instead of using a hard KL-divergence constraint like in TRPO, it just keeps the policy somewhat in the proximity of the previous policy. Two commonly used methods for keeping the policy in the proximity of our old policy are clipping and a penalty (rather than a hard constraint) based on KL-divergence from the old policy.

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[L(s,a,\theta_k,\theta)\right] \tag{2.3}$$

$$L^{PPO}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A_t - \beta \mathrm{KL}\left(\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right) \right] \tag{2.4}$$

With Equation 2.3 in combination with Equation 2.4 we see how the policy gets updated in KL-penalty based PPO implementation. θ represents the current policy parameters, and θ_{old} represents the previous policy parameters. s_t denotes the state at time t, and a_t denotes the corresponding action. $\pi_\theta(a_t|s_t)$ represents the probability of taking action a_t in state s_t under the current policy parameterized by θ. Similarly, $\pi_{\theta_{\text{old}}}(a_t|s_t)$ represents the probability of taking action a_t under the previous policy parameterized by θ_{old}.

The term A_t represents the advantage estimate.

The last term in the objective function is the KL-divergence between the old and current policies. The coefficient β controls the strength of the penalty. The penalty is often but not always dynamically adjusted to reach a target KL-divergence, which helps stabilize the training process.

By optimizing the PPO objective function, the policy parameters θ are updated iteratively to improve the policy's performance in the reinforcement learning task.

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \ \ \mathrm{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right) \tag{2.5}$$

If we replace Equation 2.4 with Equation 2.5 we instead have the clipping implementation of PPO. In this implementation instead of of the KL-penalty we keep the ratio between the probabilities from the policies are within ϵ of each other.

We can see the training loop for both clipping and KL-penalty PPO in Algorithm 1.

Algorithm 1 Proximal Policy Optimization (PPO)

1: Initialize policy parameters θ
2: Initialize old policy parameters $\theta_{\text{old}} \leftarrow \theta$
3: **for** iteration = 1, 2, ... **do**
4: **for** actor = 1, 2, ..., N **do**
5: Run policy $\pi_{\theta_{\text{old}}}$ in the environment for T timesteps
6: Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
7: **end for**
8: Optimize surrogate L wrt θ using K epochs and minibatch size M $\leq$ NT
9: $\theta_{\text{old}} \leftarrow \theta$
10: **end for**

Fine-Tuning Language Models from Human Preferences

The paper *Fine-Tuning Language Models from Human Preferences*[23] by Daniel M. Ziegler et al. explores the usage of reinforcement learning to improve on auto-regressive models previously trained only for text prediction to improve based on human feedback. This way the model is trained for the actual objective instead of what text is most likely. By training the GPT-2 model pre-trained for text prediction using PPO, the model would output answers that were evaluated as better by humans, but also copy whole sentences from input. One of the drawbacks was that to train the model this way the authors needed to collect large amounts of labelled data, in the form of questions, answers, and how the answer was evaluated by humans. Together, this is significantly harder to collect than unlabeled data.

InstructGPT

InstructGPT, as well as its more popular sister model ChatGPT, are based on the previous research on fine-tuning language models from human preferences using PPO, but are doing this on a much larger scale with the much bigger GPT-3 model.[5] By fine-tuning GPT-3 models from human feedback the models authors, OpenAI was able to align the GPT-3 model better with what a chatbot is supposed to do, that is increasing the amount of thruthful information the model responds with, making it less toxic, better at following instructions and more. The reason for this is that InstructGPT was trained for these things specifically in contrast to base GPT-3 which just tries to follow the likelihood present in the dataset.[5]

Chapter 3

Method

3.1 Representation

The method used in this project consists of multiple parts. First we want the levels to be represented as sequences of tokens. We decided that a good representation to try is to use token being the same as a tile/block in a Super Mario Level. In NLP, byte pair encoding is used by GPT. However, we avoided combining any tiles together into one token, instead opting into using an tokenizer which outputs the same number of tokens as the number of input tiles.

There are several reasons for our choice of not combining multiple tiles into single tokens. The first one is making the transformer understand its current position. At the end of each column we append a special token we call bottom token to signify that the transformer reached the bottom of a column and now is generating the next column. When each tile has its own token and each column is 16 tokens high including the bottom token at the end, the model can learn to tell the vertical position of the tile it is getting the logits for without learning how wide the tokens are. It can also look at the token with the position embedding being 16 indices from the end to tell what tile is to the left independently of what tiles are in-between, the tile will always have the same position embedding. We could say that the position of the tokens is a kind of grammar for the levels.

Another reason is the size of the dataset being fairly small compared to the datasets used to train transformers for NLP tasks. We only have a few levels from *Super Mario Bros.* and *Super Mario Bros.: The Lost Levels* that we are using to fine-tune the model that is pre-trained for NLP, with the tokens having completely different meaning and the grammar being completely different. By combining multiple tiles into single tokens we have fewer examples of how each of the tokens are used, and additionally, the tokenwise length of the dataset becomes shorter.

A drawback to not using byte pair encoding is that if we train the model on samples of up to 700 tokens, the model will only see the previous 700 tiles and forget everything that happened before in the level. If the byte pair encoding based tokenizer produced token sequences that are on average two times longer than the tile sequences, the tokenizer would be able to remember around two times as many tiles.

To convert the two-dimensional level into a one dimensional sequence that we can treat as a natural language, we read the columns from top to bottom with the previously mentioned

bottom token at the end of the column, and then begin at the next column. The length of the encoding used in this project is 700 tokens.

Figure 3.1: The textual representation of a generated level, compared with with the game's textures.

3.2 Model

The model used for this task was DistilGPT-2, a variant of the generative pre-trained transformers popular for generative NLP tasks. Despite the levels not being natural languages, we theorized that as long as the levels could be represented as a sequence of tokens similarly to natural language with a distinct grammar, the model could learn that as well. As seen in the background section, similar approach to how GPT generates natural language was tried before with a LSTM-based model.

The DistilGPT-2 implementation used in this project was the one present in the Hugging Face library. Although the implementation used more memory than NanoGPT, at least in our case, there is a bigger ecosystem around Hugging Face's implementation with existing implementations for reinforcement learning we can use to further fine-tune the model for a specific task. The training process consisted of using the trainer class from Hugging Face's transformers library which is usually used for NLP, however our representation of the levels as a sequence of tokens is similar to how NLP is represented as a sequence of tokens allowing us to use training loop implementation.

The batch size for the model used is 32, with 5000 batches of training total. Initial learning rate is 0.00003, with most of the other arguments being the default ones for the TrainingArguments class from Hugging Face's transformers library. We did not add weight decay to the optimizer and we did not change the default probability of the dropout layers activating from the default for the Hugging Face library that is 0.1. After 10 training iterations the model was evaluated on 100 batches to test how if the negative log likelihood begins to increase for levels that the model has never seen or not.

However, we do not know whether the assumption that having low negative log likelihood on the evaluation dataset is a good thing is a valid one. One of the reasons that having the same negative log probability on the training and evalutation dataset can be a bad thing is because of how the probabilities of the blocks of levels are different of the different levels. For example, if all levels depicting Bowser's castle got placed inside of the evaluation dataset, we would expect the the negative log-probability for the predictions of the model on this

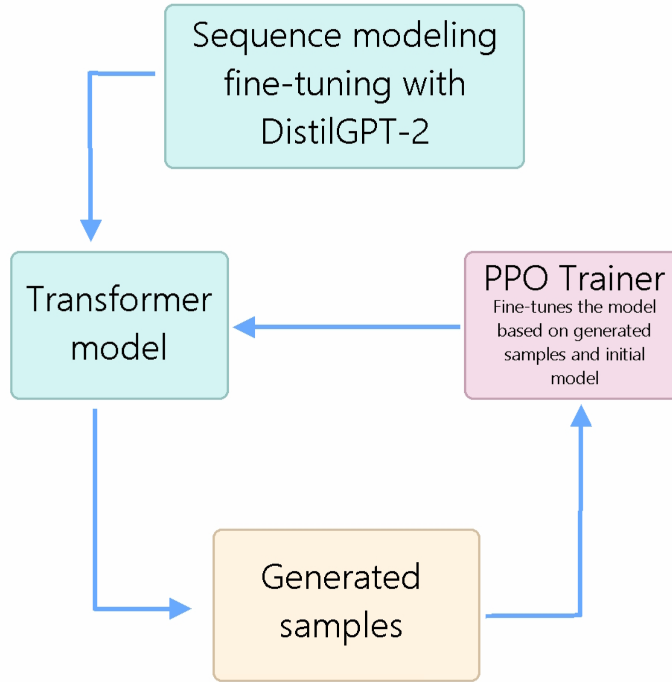


Figure 3.2: A sketch of our model architecture.

level to be worse. Additionally, if we deem the levels too unoriginal we could try to make the predictions more erratic by increasing the sampling temperature.

The sampling process for both this model fine-tuned only on existing levels, as well as models resulting from reinforcement learning, is mostly the normal process of sampling GPT models for NLP. We get the logits for the next token from the model, optionally apply a temperature then softmax and sample from the predicted probabilities. In our work we did not apply a temperature or k-top. A modification we made for the purpose of generating Super Mario Bros. levels is related to the bottom token. After the last token in the column we append bottom token to the list of generated tokens rather than sample the token from the model since that's the only valid token in that situation and sampling any other token in this situation would result in the level having invalid syntax with the height of the columns being inconsistent. Similarly when sampling any other tile, we ignore logits for the bottom token to avoid making a column that is too short.

When it comes to reinforcement learning we label levels as being possible to finish or not by using an A* agent. Despite the A* agent not being perfect and the possibility of a level that is possible to finish being marked as being impossible, we assume that the reinforcement learning algorithm has some tolerance to label noise. We are using Hugging Face's TRL library to reward the model for generating levels that are possible to complete according to the A* agent as well as penalize impossible levels using PPO. This library is based on the paper *Fine-Tuning Language Models from Human Preferences* by D. Ziegler et al. [23]. We are however not generating natural language or training the model based on human feedback. When processing the level as 15 tiles and a bottom token we add air tiles on top of or clip tiles from levels that are too high or short. After parsing the levels we get the percentage of tiles as shown in Table 3.1 in the training dataset.

Tile	Percentage	Amount
−	86.824%	96922
<	00.177%	198
>	00.176%	197
?	00.090%	100
B	00.035%	39
E	00.550%	614
Q	00.214%	239
S	02.095%	2339
X	08.547%	9542
[	00.427%	447
]	00.429%	449
b	00.022%	25
o	00.411%	459

Table 3.1: Percentage distribution of tiles ignoring bottom tokens

3.3 PPO

When fine-tuning the model using PPO, we have cycles of collecting advantage estimates and improving our agent. We opted to use the TRL[22] library which provides an implementation of the method described in *Fine-Tuning Language Models from Human Preferences* by Daniel M. Ziegler et al.[23]. This implementation of the PPO algorithm is the one based on KL-divergence with and adaptive KL-penalty. Before an iteration we collect 1000 using the current model and the A* agent. We used simple SDG optimizer with learning rate of 1.0×10^{-5}. The default target KL-divergence for the trl library is 6, however in the paper the repository is based on a much lower target KL-divergence of 0.1 is mentioned. The results of using them is mentioned in the results chapter. We use batch size of 10 and iterate over the samples we collected. The way KL-divergence is implemented in trl library when a PPOTrainer is instantiated it takes a reference model to which to calculate the KL-diveregence so we are not updating the reference model every batch and keep training the model on samples which the reference model generated. We also tried to use the latest version of the model to generate all the batches, however it was yielding similar results despite being much slower as we now need to run A* on separate levels rather than allowing us to run PPO on the levels we previously generated for evaluation.

Chapter 4

Experimental Results

When excluding a level from training, the evaluation loss for predicting the next token seem to plateau after 1 000 steps, however even after training for 5 000 steps we do not observe results of overfitting and the evaluation loss rising. This was against our expectation given the size of the model compared to the dataset, however when looking at MarioGPT it looks like that model was trained for 10 times at many batches. Suspected culprits are that the complexity and the size of the dataset actually poses a challenge for a model of the size of DistilGPT2. Another thing that could prevent overfitting is the regularization in the form of 10% dropout chance which is the default value for the Hugging Face library. When we increased the amount of training steps to 50 000 we began observing overfitting. Negative log-likelihood for training with both 5 000 and 50 000 steps is shown in Figure 4.2.
Negative Log-Likelihood however does not tell us directly how good level the model is going to generate, rather it tells us how well the model predicts the level and is dependent on how much information the model has about the level. Further discussion in section 5.2.
When generating levels we generate 100 columns which equals 1600 tiles including the bottom tokens for positions. We found that the initial success rate for the A* agent was only 65% win rate. It is worth mentioning MarioGPT[15], a similar approach to ours in terms of also using DistilGPT-2 reached 88.33% valid 100 column level chunks after 50 000 training steps. MarioGPT research paper does not mention negative log-likelihood the model got and instead measures the model performance in the amount of tiles the model guesses (tile accuracy) and how similiar the generated paths are to paths taken by A* agent (path accuracy).[15]
We did not measure the amount of valid level for our 50 000 training step models since we assumed the model be overfitting based on evaluation negative log-likelihood. Novelty score implemented in MarioGPT[15] could be considered in future work in to check whether the model actually begins to generate unoriginal levels or just loses it's ability to predict levels it has never seen before.
While generating 65% is not great in and of itself, it provides a good balance to check whenther we can improve our model using a reinforcement learning approach like PPO, rather than sequence modelling approach. The next step of our training is to use RL in the form of PPO in order to try to align the model with the goal of generating actual valid levels similarly to how we can align an NLP oriented model to generate text that align with what humans want and not only with what the probability distribution of the training dataset is.

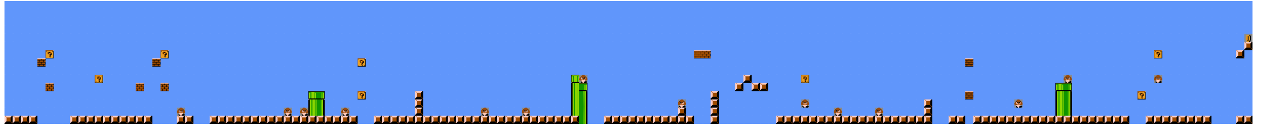
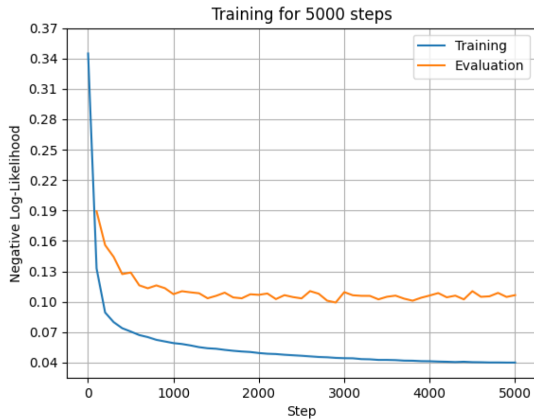
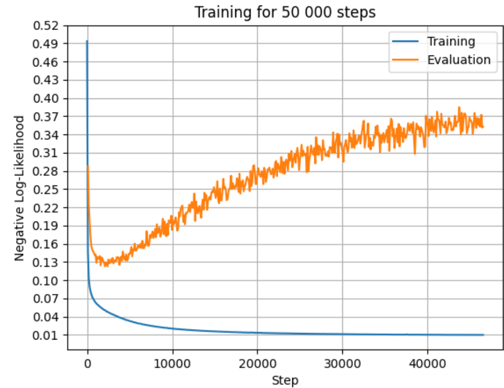


Figure 4.1: DistilGPT-2 model fine-tuned for sequences of Super Mario Bros. tiles with no reinforcement learning. The first 2 columns is the prompt, telling the model that the level begins with ground at the lowest elevation



(a) Training and evaluation scores when fine-tuning DistilGPT-2 model for Super Mario Bros. sequence modelling



(b) Training and evaluation scores when fine-tuning DistilGPT-2 model for Super Mario Bros. sequence modelling with ten times as much steps causing overfitting.

Figure 4.2: Training and evaluation negative log-likelihoods.

4.1 Reinforcement Learning

We found that training with the KL-divergence target of 6 shifts the probability distribution way too much off the original causing the generated level to be unbeatable. By taking a look at the generated levels to see what actually is wrong, the generated levels generally seem to contain a lot of empty space with gaps impossible to jump over. An actually beatable level can be seen in Figure 4.3, showing how a lot of empty space is generated after one step with too high KL-divergence target. An explanation for this behaviour we had was that in the levels we generated before PPO fine-tuning many of the unbeatable levels had obstacles that the agent could not get around, however after running the algorithm with different hyperparameters we found that there must be another explanation.

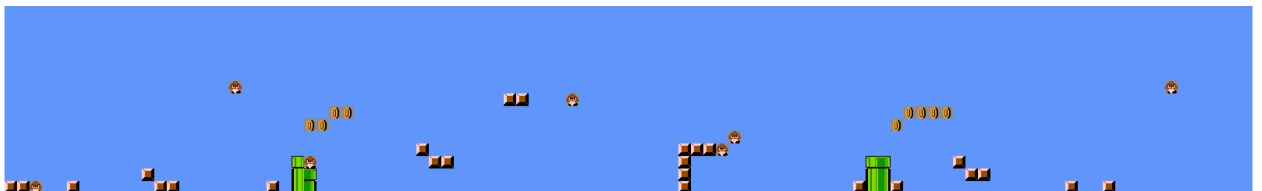


Figure 4.3: Level generated with high KL-divergence

Reducing the target KL-divergence to 0.1 line the one used for fine-tuning NLP models to human preference in the paper that proposed PPO as a method of fine-tuning transformer this way resulted in better results. After an PPO step we still observed the chance of the level being beatable to fall down, but this time to only 52%. Surprisingly this time the

fine-tuned model seemed to generate more times than the original levels. The amount of air tiles in with 0.1 as KL-divergence target in PPO fine-tuning was 81.445%. The amount of air tiles with 6 as KL-divergence target in fine-tuning was 96.097%. The training data was equal for the first PPO iteration for both KL-divergence and contained 88.055% air tiles.

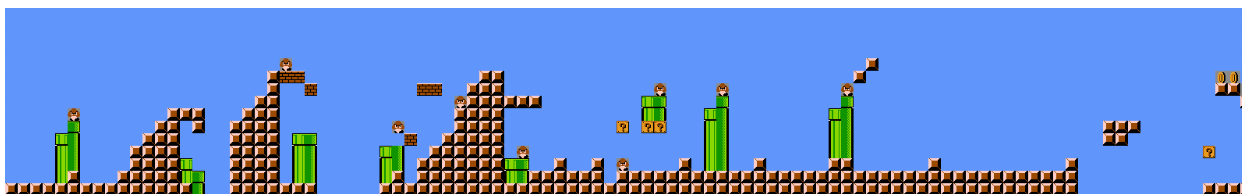


Figure 4.4: Level generated with low KL-divergence

Symbol	**Percentage Real Levels**	**Percentage No RL**	**Percentage KL target 6**	**Percentage KL target 0.1**
$-$	86.824%	88.055%	96.097%	81.445%
$<$	00.177%	00.253%	00.103%	00.243%
$>$	00.176%	00.250%	00.099%	00.241%
?	00.090%	00.064%	00.037%	00.069%
B	00.035%	00.000%	00.000%	00.000%
E	00.550%	00.558%	00.205%	00.483%
Q	00.214%	00.172%	00.015%	00.169%
S	02.095%	00.932%	00.283%	01.044%
X	08.547%	08.227%	02.172%	14.746%
[	00.427%	00.555%	00.245%	00.569%
]	00.429%	00.574%	00.223%	00.571%
b	00.022%	00.000%	00.000%	00.000%
o	00.411%	00.359%	00.519%	00.417%

Table 4.1: Percentage distribution of symbols

Table 4.1 shows the percentage distribution of symbols.

Chapter 5

Discussions

In our experiments we failed to find a combination of hyperparameters that would actually allow us to improve the model based on the A* agent. We theorize this is caused by the model finding random coincidences when it comes to what is common in the levels are are impossible to complete and possible to complete and just deviate from the original probability distribution of the levels. Instead of identifying the reason or reasons a generated level got a low score and improving on that, the model instead randomly makes subsequent levels less like the training data. It is possible that we made the task of PPO figuring out that impossible to overcome obstacles are the thing that makes the scores low. The current implementation penalizes the model for the whole level after it detect that the level is impossible, however often only a part of the level is impossible.

5.1 Autoregressive models

It's worth discussing how various machine learning based level generation strategies differ from each other, and one of the things that is not always the same is if the model is autoregressive. Autoregressive models such as GPT and LSTM generate the level in a specific direction and can see the previously sampled parts of the level. Our solution being based on DistilGPT-2, which is an example of an autoregressive model, sampling the level tile by tile and each sampled tile being visible to the model when sampling the next tile. In contrast, the GAN based approach MarioGAN[21] did not use autoregression. It would, however, be possible to combine a GAN based approach with a generator that takes the previous part of the level as input to generate the next part of the level by employing an architecture more similar to an autoencoder that takes the previous part of the level as input and outputs the next part of the level.

By employing an autoregressive implementation for level generation we gain the capability to enforce specific rules on the model's generation process based on the preceding input. This is done by manipulating the logits to make sampling values that would break these rules impossible. As an example, we will take a look at the pipe element from the game *Super Mario Bros.*

The pipe in *Super Mario Bros.* is a multi-tile structure, comprising two tiles in width, with a distinct tile type for the upper block. When generating a level from top to bottom, column by column, we know that normally the only possible part of the pipe structure to

generate is the top-left portion of the pipe. Once the top-left part of the pipe is obtained, the model gains knowledge about the expected layout for the subsequent rows - specifically, the bottom-left parts of the pipe follow until we reach a ground tile. on the next column we know what positions the right side of the pipe structure needs to be in. By restricting the sampling process to correct tile types we can ensure generation of the pipes and other structures consistent with the desired level design.

Another example is the restriction of the bottom token element always and only generated at the ends of columns as a guide where currently the model is in. Since we turn the 2 dimensional level into a sequence, the bottom token element is supposed to help the model remember what the current vertical position is.

5.2 Negative Log-Likelihood as a evaluation score

Similarly to previous implementations treating the levels as sequences of tokens and generating the levels by outputting an predicted probability distribution, we used negative log likelihood during training and on the validation dataset to tell how well the model predicts and generalizes the predictions to levels it has not seen before. However drawbacks of this method should be considered before using it, especially to compare different training methods and models to each other.

One example we should consider is the possibility of adding the path of an A* agent to the level's representation or not. A problem with relying on the predicted probabilities when comparing models trained on dataset with A* path and ones with normal level representation is that the path of the A* agents gives the model more information about what it can expect in the future than it can conclude from the level itself. If the model sees that the A* agent began an jump in the prompt it can conclude that the agent could be trying to jump over something increasing the probabilities of obstacles that could stand in the way requiring the A* agent to jump like pipes. Likewise if the model observes the agent not jumping it can conclude that there should be no obstacle in the path. When comparing the models for the levels with paths and without paths a problem is that the negative log likelihood will get lower pointing towards the model being better in the case of the model with paths, however we can not know if the levels are actually going to be better when generating them with the use of autoregression because in that case the paths are going to be imagined by the model in contrast to the training and evaluation dataset when the paths are generated by something like an A* agent or an actual player. The model with pathing information in prompt has unfair advantage over models without in the case of the training and evaluation datasets.

While paths are one obvious example, other additional information like the expected amount of pipes in the level, expected amounts of enemies in the levels and so forth are also going to improve the negative log-likelihood scores, but not necessarily make the levels more realistic.

The biggest thing the negative log-likelihood can tell us is if the models still works on levels it has never seen before or if it is overfitting. This method is however not necessarily a good method of comparing different level representation and types of input since giving more information could improve how well the model predicts existing levels but not how realistic levels the model is actually going to generate itself.

To actually compare different approaches of level generation with each other one good way should be collecting enough human feedback, since that is our actual goal when generating levels and not prediction. This would however require a lot of human effort and time. Another statistic that gives us more information about how good the levels are things like how often the model generates levels that are possible to complete.

5.3 Pre-training by other models

When training the model, the dataset we used consisted of levels from just two *Mario* games, which is relatively small compared to datasets like OpenWebText [2] that were used to train GPT models. What we relied on to get somewhat good results despite training an relatively large model compared to the size of the dataset, was to use transfer learning in order to fine-tune a NLP model for the task of level generation. However, NLP and *Mario* level generation are two different tasks that contain a large number of differences with the *Mario* levels not being an natural language at all. An possible solution for this problem could be to use other level generators to generate levels to fine-tune the NLP model to before training it on real levels or maybe not even using NLP at all and training an GPT model from scratch of AI generated levels. The possibility of using other models like a step between the task of NLP and fine-tuning the GPT model on real *Super Mario Bros.* levels could be the basis for future work.

5.4 Divergence from training dataset

As we fine-tune our model with PPO, we do not have any reward for the model to keep the levels looking like the original human made levels. Therefore it is possible the model would stop generating some types of levels in response to being fine-tuned by PPO only. For example, the model gets penalized for making big gaps, potentially causing the model to stop generating gaps for the player to jump over at all. In our experiments we kept the same reference model throughout the training, which allows us to search for levels with a given KL-divergence from the original distribution at the beginning of the training. Nevertheless, when we fine-tune the model using PPO we are moving away from the original probability distribution we had, making the levels less realistic.

Chapter 6

Conclusion

In our experiments we found that we can indeed use GPT models for the purpose of procedural level generation, even with the limited dataset that is Super Mario Bros. and Super Mario Bros.: The Lost Levels when making use of pre-trained GPT models. However we failed to make use of PPO algorithm to fine-tune the level generator to produce better levels. As we used examples sampled from our model and checked them our initial success rate for possible levels was 65%, but applying PPO to try to improve the score only worsened the score.

6.1 Future Work

6.1.1 PPO avoiding invalid levels

While we did not find success in fine-tuning the model to reduce the amount of invalid levels with our method, that doesn't invalidate PPO as a method of fine-tuning level generators. It's possible that by improving on the fine-tuning process by methods like finding the exact part of the level that is impossible to pass and penalize the model for that part rather than penalizing the whole level could result in better scores. Possible future work is trying to create a better PPO implementation possible being more precise with what we actually penalize and reward in order to make it easier for the model to align with chosen goals.
Also while PPO is popular for use in the field of NLP, it is possible that another way for rewarding and penalizing the model based on our goal could fit better in this scenario like using some other RL algorithm.

6.1.2 Fine-tuning from human preference

Possible future work when it comes to the reward function in an RL based approach for fine-tuning a level generator would be to test how fine-tuning the model using human feedback would improve results. With enough data this would allow to train the generator to generate levels that are perceived by humans as better. In *Fine-Tuning Language Models from Human Preferences* by Daniel M. Ziegler et. al.[23] it is shown that "offline" data collection, that is collecting labels only for data generated by the original model, is less effective than labeling data from the newest version of the model at the same time as the model is fine-tuned using RL. Doing this was omitted from this research as it would require a human participant to

label potentially tens of thousands of levels similarly to how *Fine-Tuning Language Models from Human Preferences* by Daniel M. Ziegler et. al.[23] has up to 60 000 examples, which all show improvement in the perceived quality, in addition to evaluation by humans.

Appendix A

Examples of generated levels

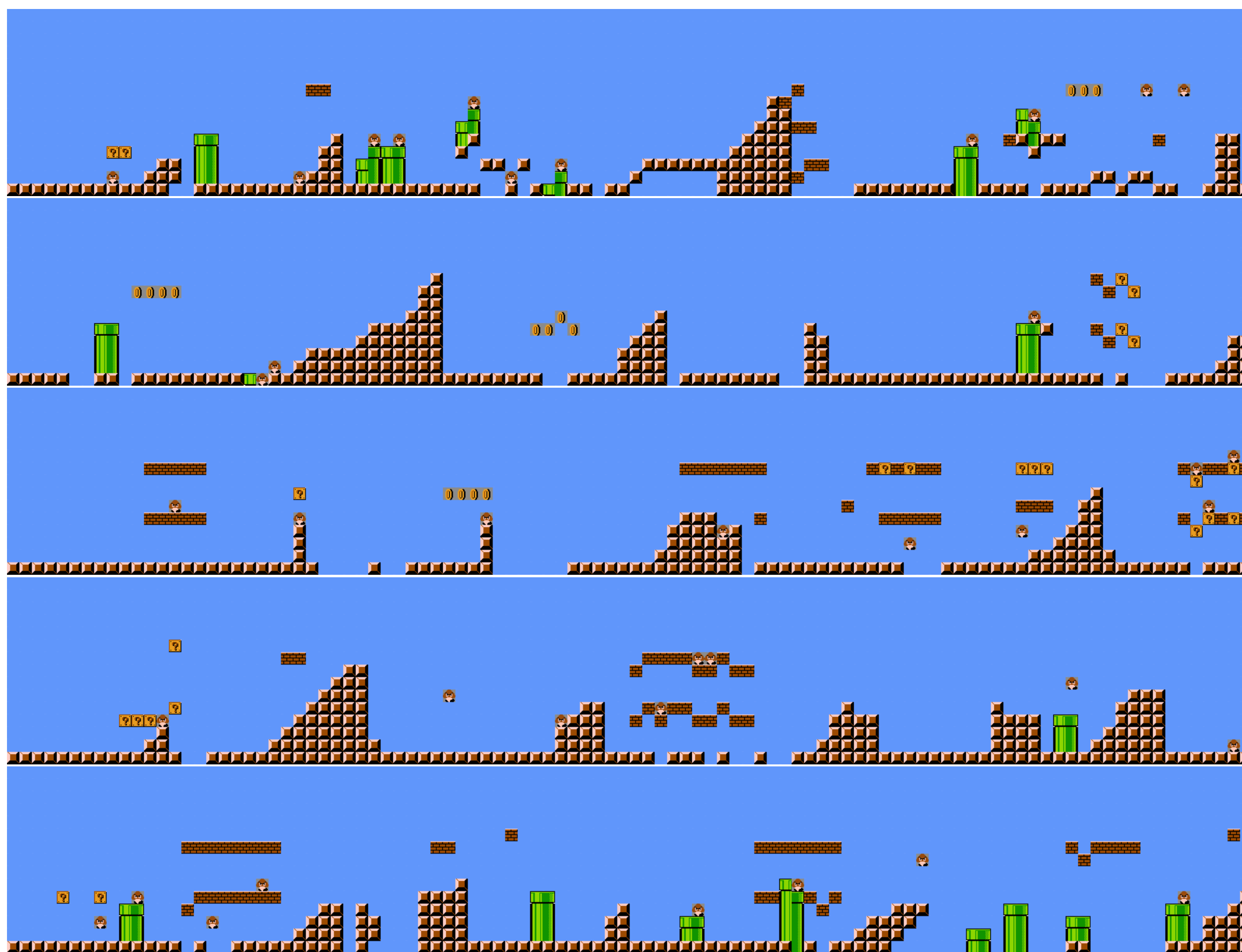


Figure A.1: Possible to complete generated levels no PPO.

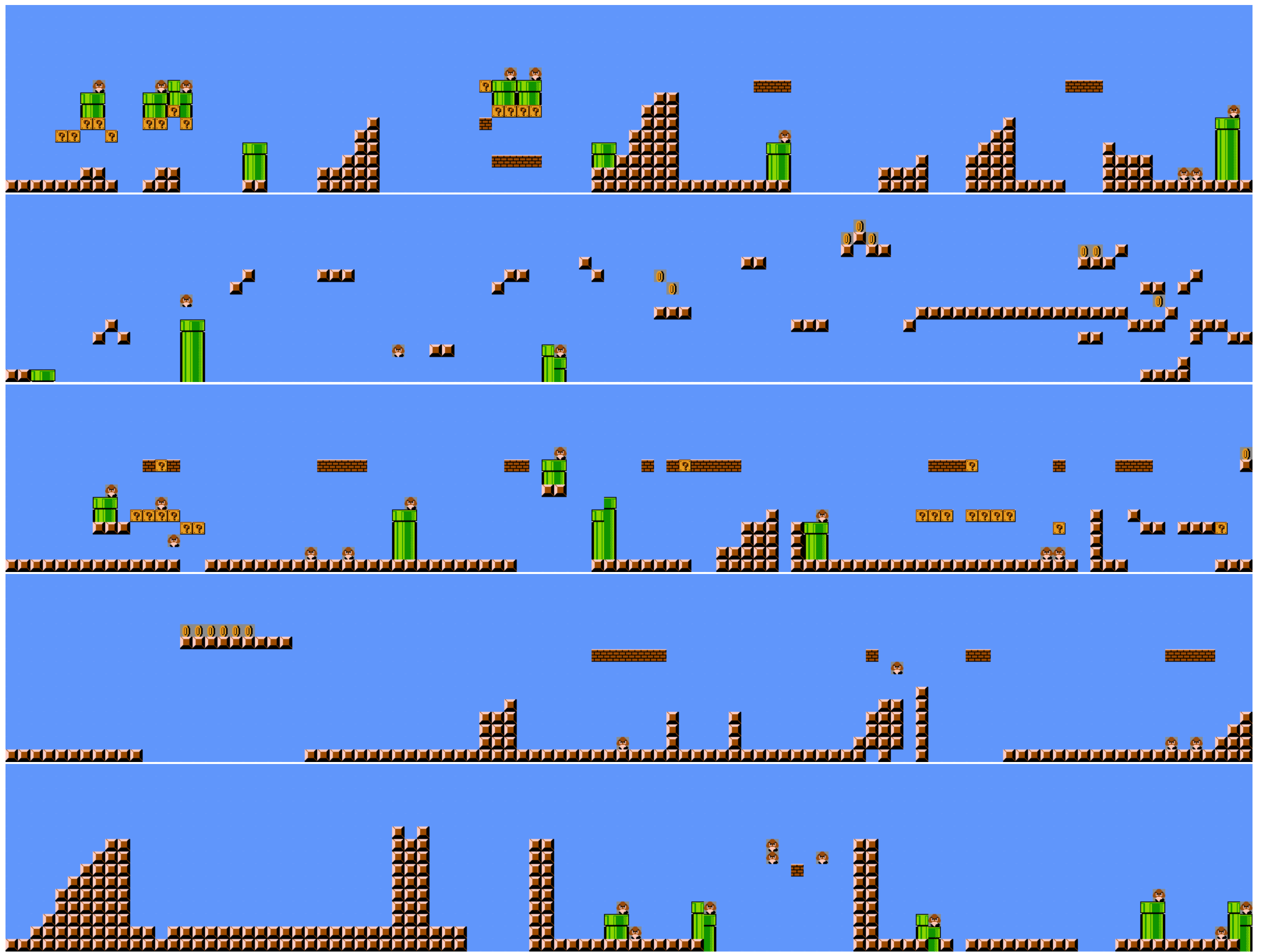


Figure A.2: Impossible to complete generated levels no PPO.

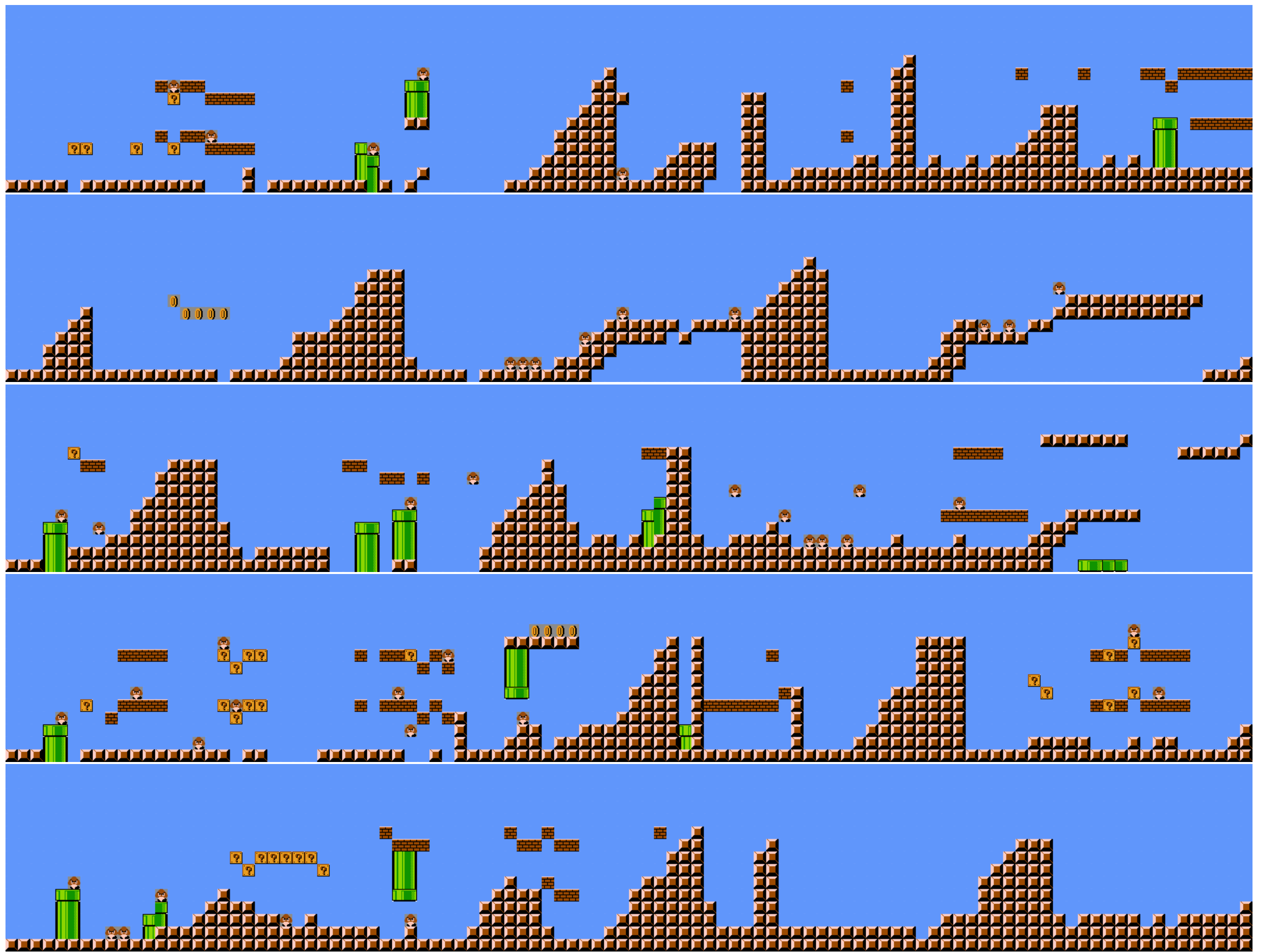


Figure A.3: Possible to complete generated levels KL target 0.1

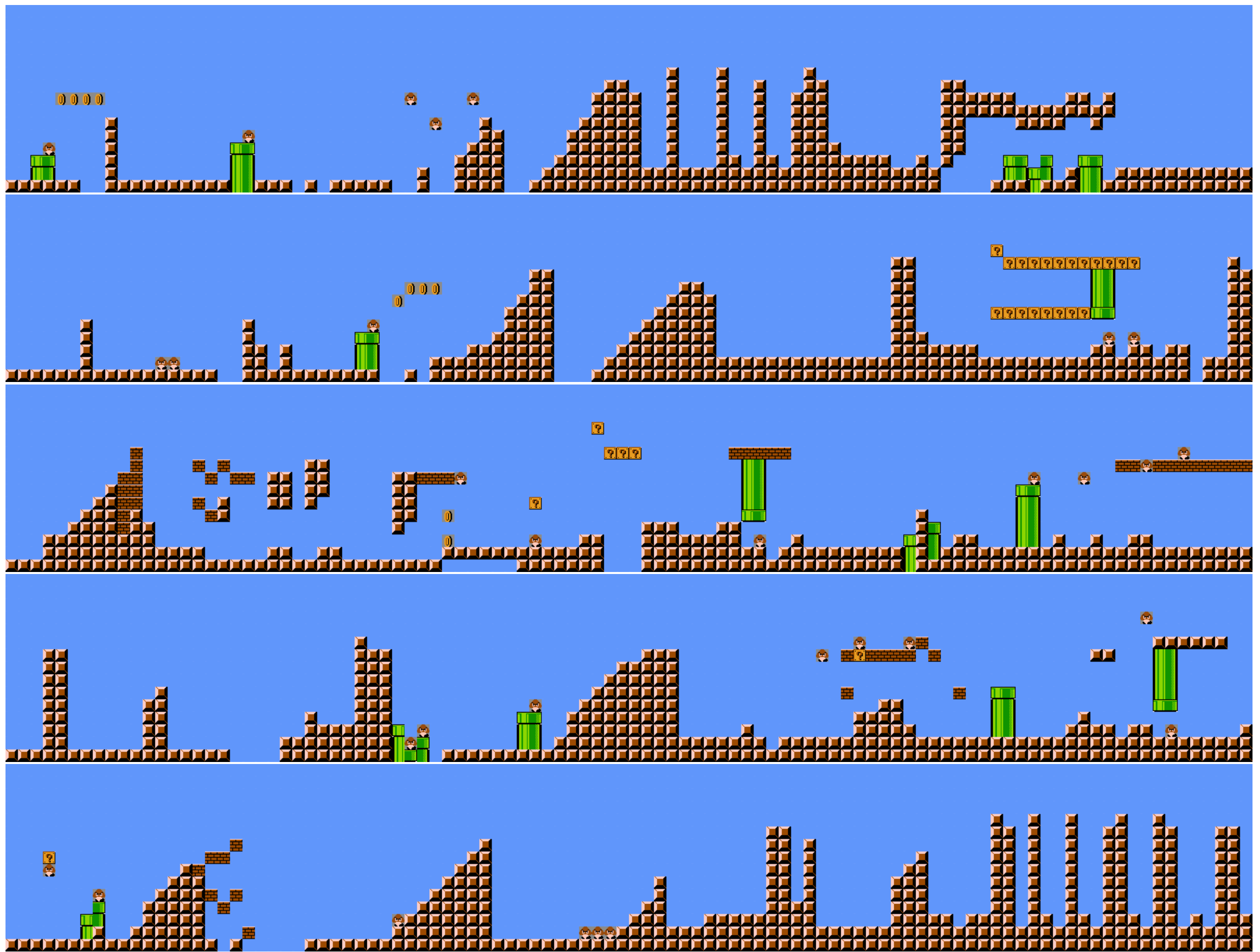


Figure A.4: Impossible to complete generated levels KL target 0.1

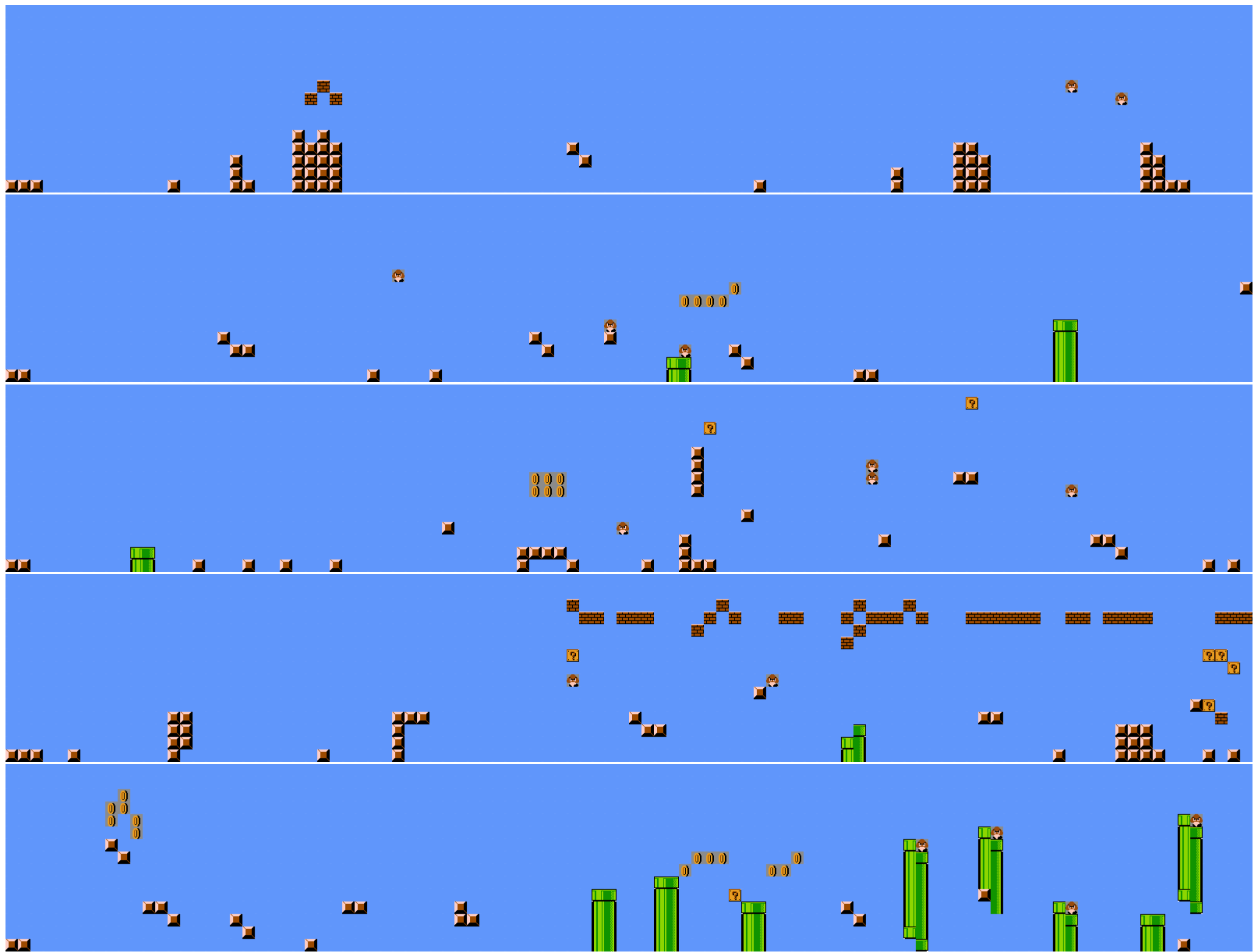


Figure A.5: Impossible to complete generated levels KL target 6

Bibliography

[1] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[2] Aaron Gokaslan et al. *OpenWebText Corpus*. http://Skylion007.github.io/OpenWebTextCorpus. 2019.

[3] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[4] Yinhan Liu et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach." In: *CoRR* abs/1907.11692 (2019). arXiv: 1907.11692. URL: http://arxiv.org/abs/1907.11692.

[5] Long Ouyang et al. "Training language models to follow instructions with human feedback." In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 27730–27744.

[6] Nintendo R&D4. *Super Mario Bros.* 1985.

[7] Alec Radford et al. "Improving language understanding by generative pre-training." In: (2018).

[8] Alec Radford et al. "Language Models are Unsupervised Multitask Learners." In: 2019.

[9] Victor Sanh et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter." In: *NeurIPS EMC^{2}Workshop*. 2019.

[10] John Schulman et al. "Proximal Policy Optimization Algorithms." In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[11] John Schulman et al. "Trust Region Policy Optimization." In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[12] Noor Shaker, Julian Togelius, and Georgios Yannakakis. "Mario AI Championship 2012." In: 2012. URL: https://web.archive.org/web/20160312112028/http://www.marioai.org/home.

[13] Noor Shaker, Julian Togelius, and Georgios Yannakakis. "Platformer AI Competition." In: 2013. URL: https://web.archive.org/web/20150907121258/http://www.platformersai.com/.

[14] Ralf C. Staudemeyer and Eric Rothstein Morris. "Understanding LSTM - a tutorial into Long Short-Term Memory Recurrent Neural Networks." In: *CoRR* abs/1909.09586 (2019). arXiv: 1909.09586. URL: http://arxiv.org/abs/1909.09586.

[15] Shyam Sudhakaran et al. *MarioGPT: Open-Ended Text2Level Generation through Large Language Models*. 2023. arXiv: 2302.05981 [cs.AI].

[16] Adam Summerville and Michael Mateas. "Super Mario as a String: Platformer Level Generation Via LSTMs." In: *CoRR* abs/1603.00930 (2016). arXiv: 1603.00930. URL: http://arxiv.org/abs/1603.00930.

[17] Adam James Summerville et al. "The VGLC: The Video Game Level Corpus." In: *Proceedings of the 7th Workshop on Procedural Content Generation* (2016).

[18] R. S. Sutton et al. "Policy gradient methods for reinforcement learning with function approximation." In: *Advances in Neural Information Processing Systems 12*. Vol. 12. MIT Press, 2000, pp. 1057–1063.

[19] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. "The 2009 Mario AI Competition." In: *IEEE Congress on Evolutionary Computation*. 2010, pp. 1–8. DOI: 10.1109/CEC.2010.5586133.

[20] Ashish Vaswani et al. "Attention is all you need." In: *Advances in neural information processing systems* 30 (2017).

[21] Vanessa Volz et al. "Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network." In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2018)*. Kyoto, Japan: ACM, July 2018. DOI: 10.1145/3205455.3205517. URL: http://doi.acm.org/10.1145/3205455.3205517.

[22] Leandro von Werra et al. *TRL: Transformer Reinforcement Learning*. https://github.com/lvwerra/trl. 2020.

[23] Daniel M. Ziegler et al. "Fine-Tuning Language Models from Human Preferences." In: *CoRR* abs/1909.08593 (2019). arXiv: 1909.08593. URL: http://arxiv.org/abs/1909.08593.