Replicating Super Mario as a String

Joey Parker
University of North Carolina at Charlotte
ITCS 5156-051
Minwoo Lee
April 22, 2024

Original Paper
Title - Super Mario as a String: Platformer Level Generation Via LSTMs
Authors - Adam Summerville and Micheal Mateas
Year - 2016
Published at - https://doi.org/10.48550/arXiv.1603.00930

1 Introduction

In many video games, the world is divided into separate sections called ‘levels.’ These levels are designed and created by a level designer, and each can take a significant amount of time to create. In 2016, two researchers in the Center for Games and Playable Media at the University of California, Santa Cruz, proposed a method to generate Super Mario Bros. levels using Long Short-Term memory recurrent neural networks (LSTMs) (Summerville & Mateas, 2016).

1.1 Problems with Traditional Level Creation

The traditional design of video game levels poses significant problems related to time, required design skills, and creativity. Crafting levels manually demands a considerable time investment, often requiring designers to carefully place each level element individually to ensure gameplay flow. Additionally, the process demands a high level of design skills and creativity to create engaging and immersive environments. As a result, traditional level design can be resource-intensive and reliant on the expertise of skilled designers, limiting the quality and quantity of levels that can be created within a certain timeframe. This project aims to reduce the time, design, and creativity required to make platformer levels through procedural content generation with an LSTM.

1.2 Challenges of / Open Questions in Procedural Content Generation

Procedurally generating levels in video games faces several challenges. Firstly, there’s the issue of limited data availability. Most games contain only a maximum of a few hundred levels, which may not be sufficient data for training some generative models. How can generative models effectively leverage limited data to produce unique and engaging levels?

Secondly, platformer games like Mario often feature long levels containing thousands of tiles, posing a challenge for maintaining global coherence in generated content. Many machine-learning algorithms struggle to capture patterns over large sequences. How can machine-learning algorithms be adapted to maintain coherence over lengthy sequences?

Lastly, level design relies heavily on combining various patterns, a concept that machine learning algorithms struggle to understand due to their reliance on sequential data rather than overarching design principles. How can machine-learning models effectively learn and replicate the intricate patterns and design principles of current video game level design?

1.3 Motivation to Replicate

Game development is very complicated, time-consuming, and skill-intensive work. A large portion of developing a game is in level design. By automating level design, game developers’ focus can be spent on the other creation tasks. I wanted to investigate the problem due to the current entry of machine learning and AI in the game creation process. I want to spend my career

creating games, and while this project may not help me with anything I'm currently making, what I learned can easily be applied to other models that can aid in future game development projects.

1.4 Concise summary of this paper's method and approach

In this paper, I use a Long Short-Term recurrent neural network (LSTM) to generate Super Mario Brothers levels from a dataset of original levels. The process begins with pre-processing the original levels, breaking them down into sequences of characters. These sequences are then used to train the LSTM, evaluating its performance based on loss and accuracy. Training involves several steps: clearing gradients, forward passing, calculating loss, backward passing/computing gradients, and updating weights. After training, we can generate levels by continuously feeding a starting seed sequence into the model, with the model adding a character to it at each step. This updated seed is then fed back into the model to generate another iteration, and this process repeats until we generate an entire level.

2 Related Works

2.1 Online Level Generation in Super Mario Bros via Learning Constructive Primitives

In Online Level Generation in Super Mario Bros via Learning Constructive Primitives, they attempt to generate Mario levels through learning-based procedural content generation (Shi & Chen, 2016). Specifically, they sampled their levels using simple random sampling and then ran CURE, a clustering algorithm, on the sampled segment set. They then manually annotated several segments from each cluster to form a validation set. During active learning, they randomly chose 100 segments, annotated them, trained the weighted random forests model with them, and repeated until the accuracy stopped increasing. Once the model was trained, it could generate new levels endlessly.

This approach has two distinct pros. First, it offers a high degree of control, as designers can specify a number of different parameters when generating levels. Second, it has a high playability rate due to the mixture of rule-based and learning-based techniques over solely learning-based methodologies. However, it does have a significant distinct con. It relies heavily on manual labor and human evaluation during the learning process, taking a considerable amount of human time.

2.2 MarioGPT: Open-Ended Text2Level Generation through Large Language Models

In MarioGPT, they use a variant of transformers called Large Language Models. They use a pre-trained DistilGPT2 Model, which they fine-tuned to generate Super Mario Levels

(Sudhakaran et al., 2023). The fine-tuning involved taking slices of Mario levels and then randomly replacing one slice with another. If the new level is novel enough, it gets added to an archive of novel levels. Then, the process repeats 50,000 times until they get 200,000 training samples. These training samples are used to train the model. The model can then produce new levels by prompting it with wanted aspects of the level, such as pipe quantity, block quantity, enemy quantity, and elevation.

MarioGPT offers three pros. First, it is very user-friendly, as it can be directed to generate levels through natural English commands. Second, it has a high degree of control. Users can control the number of pipes, enemies, blocks, and elevations generated. Lastly, it can create multiple paths. This is unique to MarioGPT, as no other level generator can do this intentionally. Multiple paths within a level give a player more options, making a more engaging level to play.

There are two notable cons. The first is the problem of temperature, like in all LLMs. A low temperature makes the model likely to reproduce provided levels. Raising the temperature will decrease this likelihood but will result in lower quality levels. A challenge for MarioGPT users is to find the optimal temperature to balance quality and originality. The second problem is with enemy generation. Due to the low number of enemies in the training data, MarioGPT has a lot of problems when provided with enemy-related prompts. This is mainly shown as the model ignoring users' enemy parameters, often resulting in unpredictable enemy placements or quantities in the generated levels.

2.2 In Relation

All three methods, including the related works and my approach, share the same objective: generating Super Mario Bros levels through applying machine learning algorithms. They all follow a similar workflow, beginning with a dataset of original Mario levels in a text format. Pre-processing is done to convert the original level data into model-trainable forms. The training phase is where the different methods diverge through the selected machine learning algorithm. After training, all of the models are used to generate Mario levels.

3 Quick Note on Contribution

For this project, I heavily modified Zhihan Yang's attempt to replicate the same paper (Yang, 2020). His project can be found at super_mario_as_a_string.

What is **Not Original** is the original Mario level data, *most* of the data pre-processing, creating the seed of level generation, and a tool that turns text levels into pngs.

Everything else is **Original**, including the model, training, generation, snaking, pathing, column depth, and metric calculation and evaluation.

4 Method

This paper's method has four core parts: data pre-processing, model creation, training, and level generation.

4.1 Input Format

Before discussing the different parts of the method, we must ask ourselves three questions about the model's input format in the original paper: Bottom-To-Top vs. Snaking? Include Path Information? Include Column Depth Information? To find the answers to these questions, we will test a mixture of them in the experiments section. But first, the options must be defined.

LSTMs take sequences as inputs. Mario levels are 2D and cannot be directly fed into an LSTM. We must figure out how to split the level to convert the 2D data into a 1D sequence. Bottom-To-Top involves initiating the sequence generation process at the bottom of each column within the input level, then progressing vertically upwards through each column, ensuring that the sequence is constructed column by column in an ascending manner. Snaking involves a similar process, except every other column progresses vertically downwards.

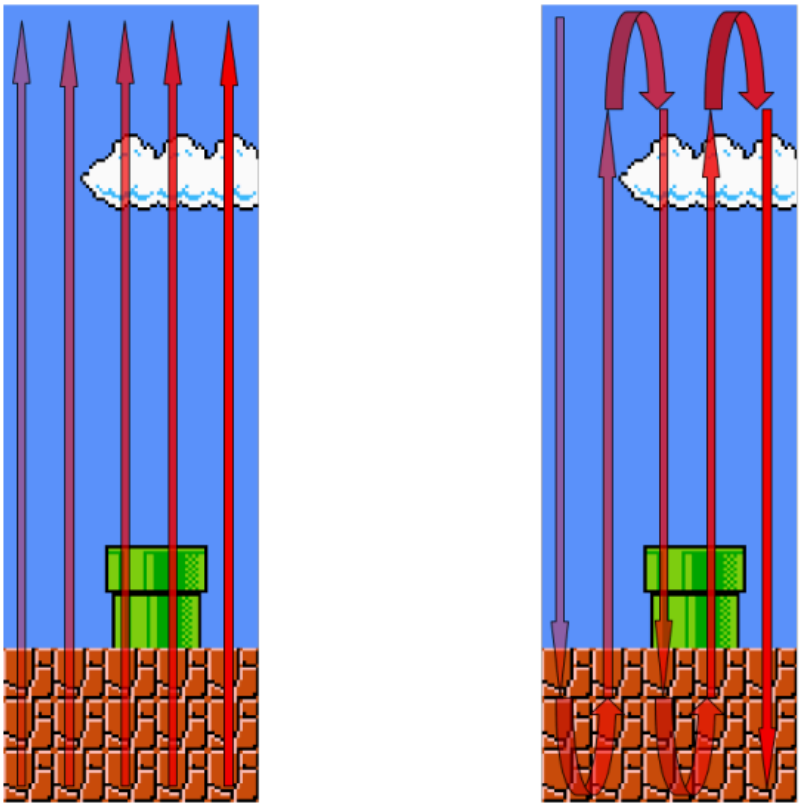

Figure 1: Illustration of the difference between going from bottom-to-top (LEFT) and snaking (RIGHT) for sequence creation. Adapted from Summerville and Mateas (2016)

Path information is included by default in every original Mario level. It marks the optimal path a player can take. The generator will try to create a path in the generated level by leaving these tiles in the input. We can remove the path information to prevent this.

Column depth is a proposed idea by the original paper to try to get the LSTM to understand level progression. This is done by adding column depth markers, a special character that is incremented every five columns. In other words, the first column has no marker, the sixth has 1, the 11th has 2, etc.

4.2 Data Preprocessing

Data pre-processing is done in five steps. First, the original Mario levels are combined into a single text file. Second, we modify each level in the master text file to include the optional snaking, pathing, and column depth parameters. If we choose to snake the input, every other column is flipped. If pathing information is undesired, every character denoting a path is replaced with the character denoting air. If column depth markers are wanted, we add the necessary ascending number of column depth markers every five columns of each level. Third, we find the number of unique characters in the input to be used for one-hot encoding later, which is called the vocab size.

The fourth step is more involved. Each level string is turned into a series of overlapping sequences of a specified sequence length. For each input sequence, we have a target sequence, which is the input, just with every index shifted up by one. The inputs are then one-hot encoded by the vocab size. The fifth and final step involves splitting the inputs and targets 70-30 into training and evaluation datasets.

4.3 LSTM Model

Long Short-Term recurrent neural networks, or LSTMs, are a type of artificial neural network designed for sequential data processing. LSTMs address the vanishing gradient problem in traditional recurrent neural networks (RNNs) by introducing specialized memory cells with gating mechanisms. There are three kinds of gates: forget, input, and output. These gates enable LSTMs to selectively store, update, and retrieve information, allowing them to capture long-range dependencies. Effectively, LSTMs create sequences that have good global and local coherence.

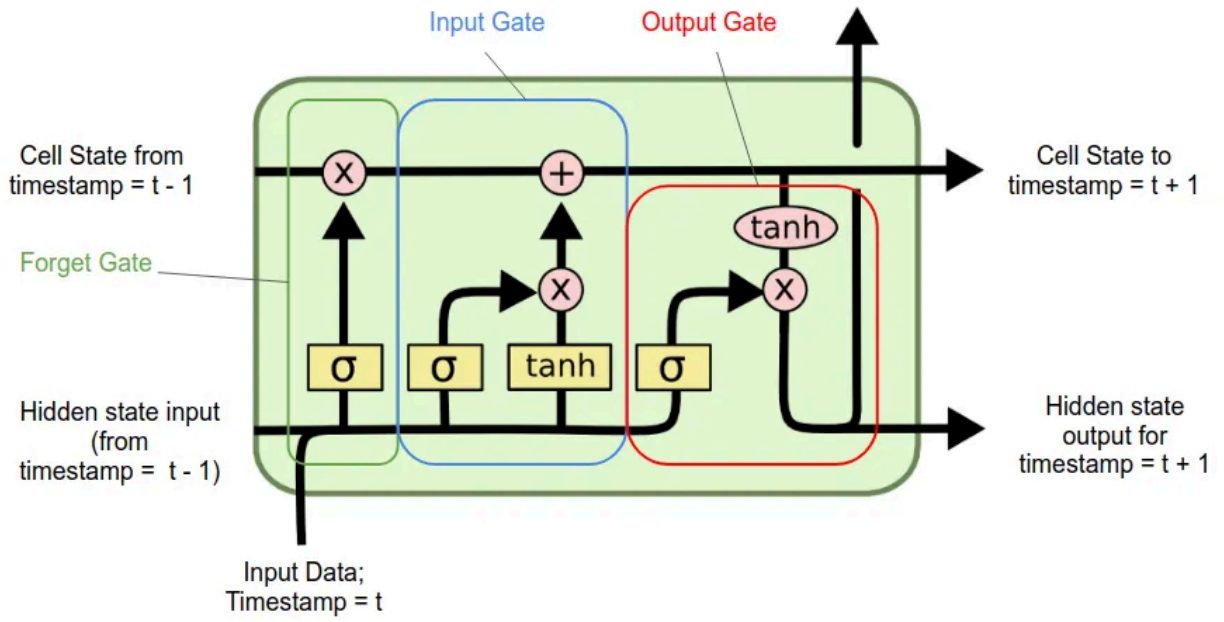

Figure 2: Model of a single LSTM cell. Adapted from T. J. J. (2024)

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

Figure 3: Forget gate sigmoid function.
Adapted from PyTorch documentation (*LSTM, n.d.)*

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$

Figure 4: Input gate sigmoid function
Adapted from PyTorch documentation (*LSTM, n.d.)*

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg})$$

Figure 5: tanh function
Adapted from PyTorch documentation (*LSTM, n.d.)*

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$

Figure 6: Output gate sigmoid function
Adapted from PyTorch documentation (*LSTM, n.d.)*

The forget gate in an LSTM determines how much of the previous cell state information to discard. It does this through a sigmoid activation function (Figure 3), which outputs normalized values between 0 and 1 for each component of the cell state. A value close to 0 indicates that a lot of the cell state should be forgotten, and a value close to 1 indicates that a lot of the cell state should be retained.

The input gate in an LSTM determines how much new information should be fed into the cell state. Similarly to the forget gate, it uses a sigmoid activation function (Figure 4) to indicate what parts of the input should be stored. The same data fed into the sigmoid function is also fed into a tanh function (Figure 5). This function normalizes values between -1 and 1 instead of 0 and 1. This function is meant to scale the input based on how important the data is to remember.

The last gate, the output gate, determines the new hidden state. Again, a sigmoid function (Figure 6) determines what parts of the cell state should be outputted. Another tanh function (Figure 5) controls the magnitude of the outputted values.

While the functions of the gate are all very similar, it is important to note that each gate has separate weights attached. We train the model by updating these weights to be as accurate as possible.

The LSTM used in this paper has three fully connected internal layers, each consisting of 512 LSTM blocks. The vocab size varies depending on whether we have enabled snaking, pathing, or column depth. Dropout is applied to each layer, a technique that drops a number of LSTM blocks randomly to prevent memorization and overfitting. After the LSTM layers, we fully connect the output to a linear transformation layer, which is then returned out of the forward pass.

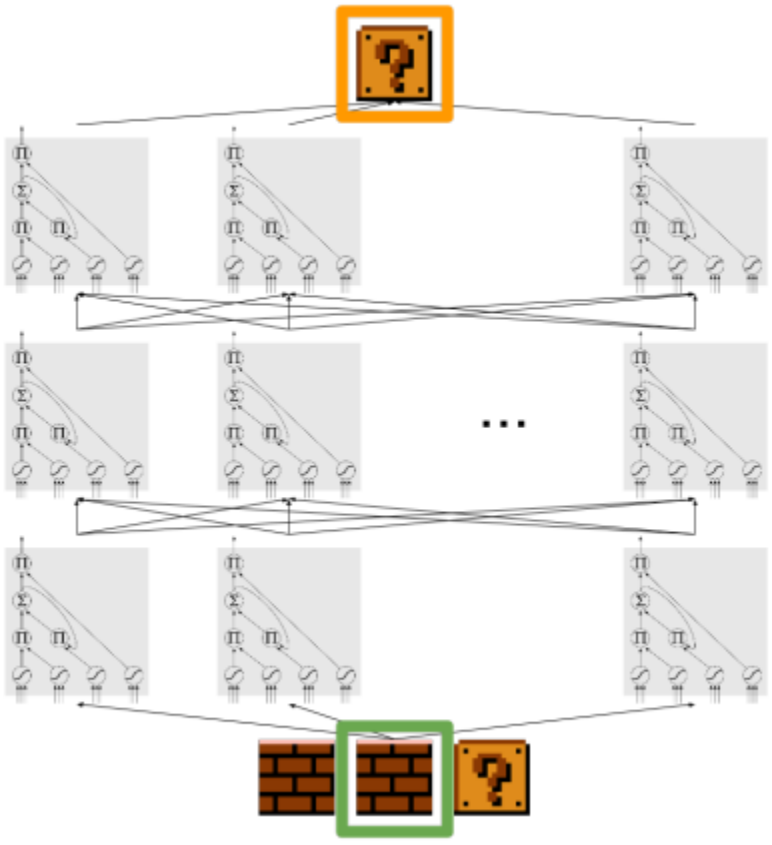

Figure 7: Graphical depiction of our chosen architecture. The green box at the bottom represents the current tile in the sequence, the tile to the left of the preceding tile, and the tile to the right of the next tile. The tile in the top orange box represents the maximum likelihood prediction of the system. Each of the three layers consists of 512 LSTM blocks. All layers are fully connected to the next layer. Figure and notes are taken from Summerville and Mateas (2016)

I do not include a LogSoftmax layer like in the original paper. This is because I use CrossEntropyLoss as the loss function when training, which effectively applies a LogSoftmax itself and then a Negative log-likelihood loss.

4.4 Method Sequence

The whole process of the method involves several steps. First, the model is initialized with the proper hyperparameters, loss function, and optimizer. Then, the preprocessed inputs and targets are split 70-30 into training and evaluation datasets. The model is then trained using the training dataset, where each training iteration involves clearing the gradients, performing a forward pass to compute predictions, calculating the loss, executing a backward pass to compute gradients, and updating the model weights accordingly. The model is evaluated after each epoch using the evaluation dataset, and if the loss has reached a plateau, the training process is halted. Finally, with the trained model in place, the level-generation phase begins. We start with a pre-made starting "seed" sequence for each level we want to generate. This 'seed' sequence is repeatedly fed into the model, adding a new character to the end of the sequence each iteration. This continues until we have a sequence with our desired level's length. At this point, we have generated a whole Super Mario Level in text form.

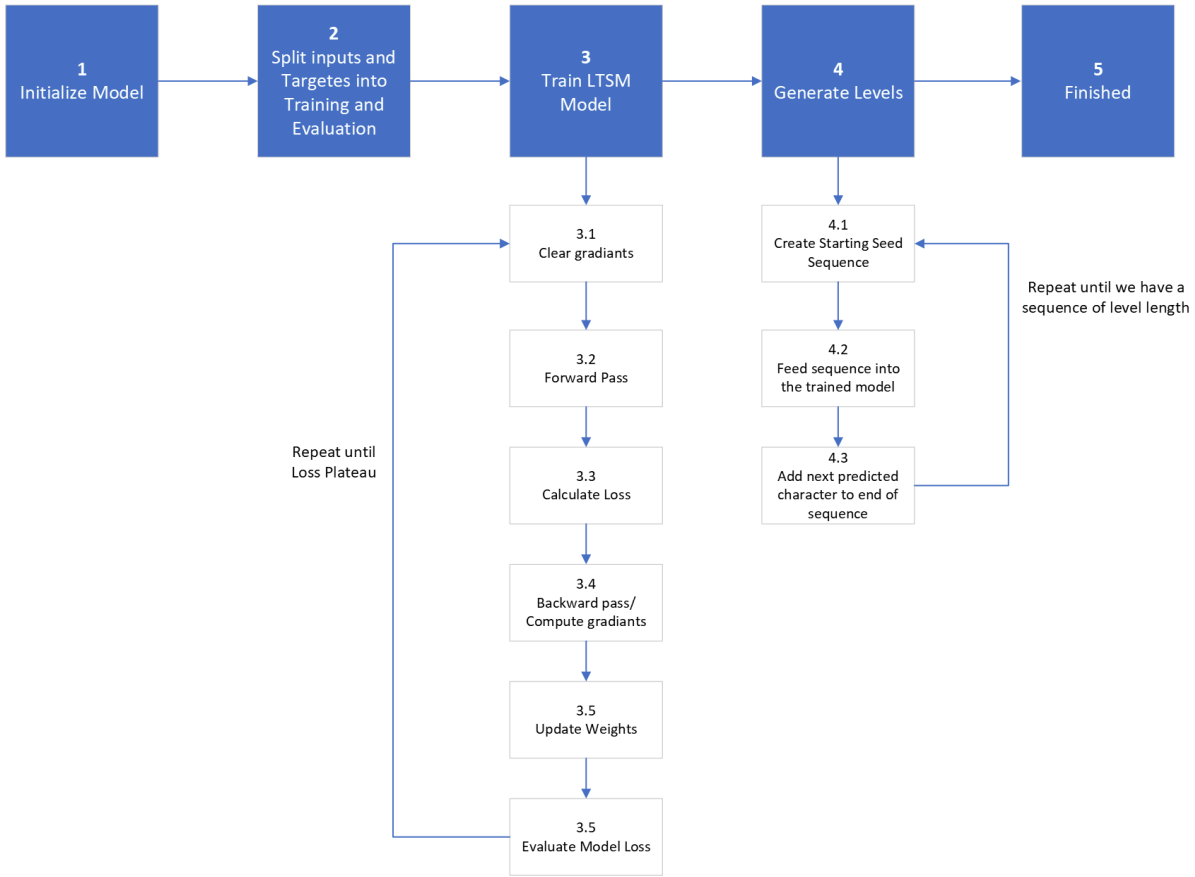

Figure 8: Step-by-step diagram of the method sequence

5 Experiments

5.1 Experimental Setup

In replicating the experimental setup from the original paper, I tested various combinations of input formatting, exploring options such as enabling snaking (S), including path information (P), and adjusting column depth (D). However, due to constraints in time and resources, I generated 100 levels per trained network instead of the original 4000 levels per network. To evaluate the performance of the trained networks, I used several metrics from the original paper. These include the percentage of empty space (e), the percentage of "interesting" tiles that are not simply solid or empty (d), the leniency of the level (l), and the linearity of the level (R2). It is important to note that I do not have access to how the original paper calculated the metrics, so they should not be used as 1-to-1 comparisons, though they will be compared.

5.2 Test Results

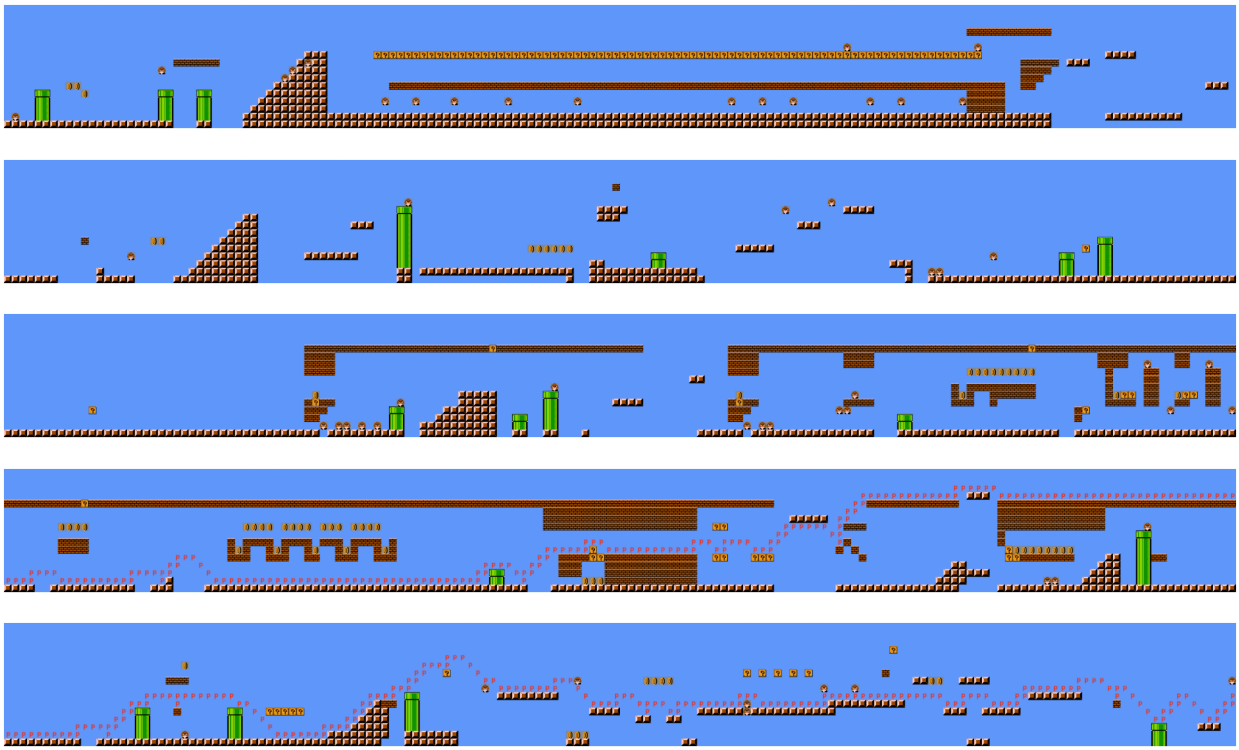

Figure 9: Sampled generated levels with snaking disabled

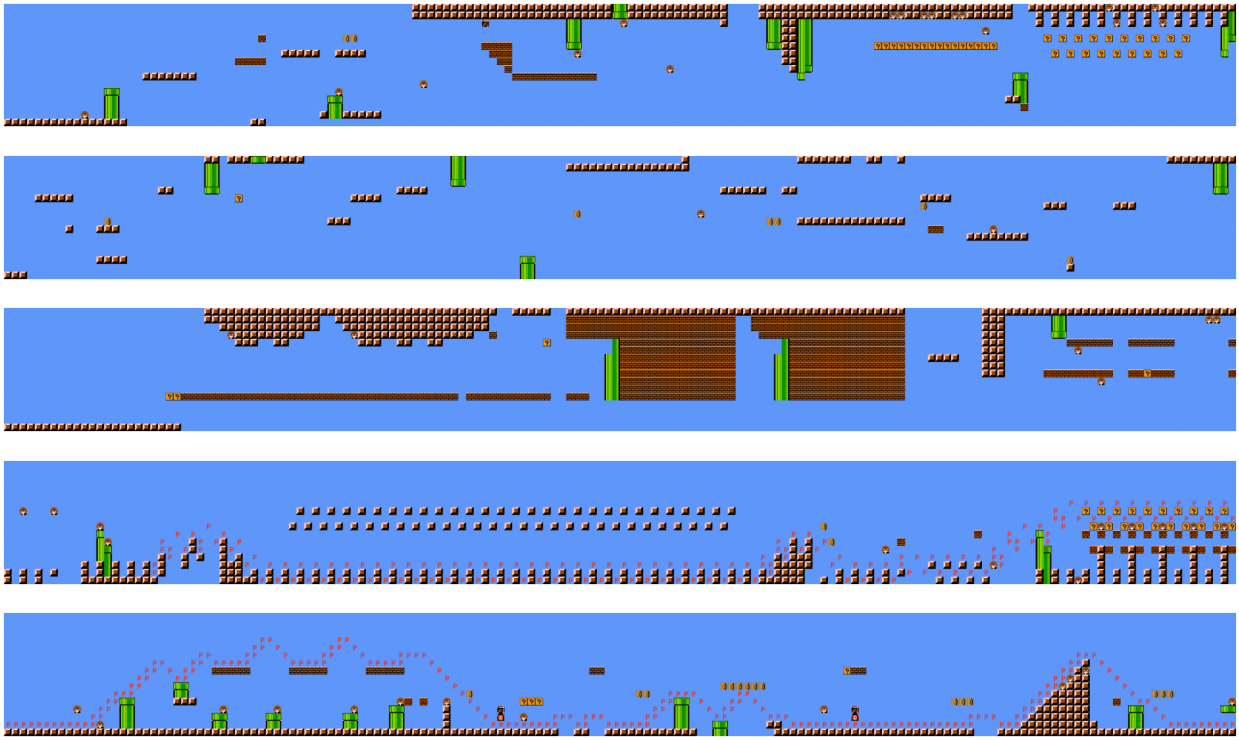

Figure 10: Sampled generated levels with snaking enabled

S?	P?	D?	e	d	l	R2
Human Authored			0.82	0.04	71.19	0.13
N	N	N	0.81	0.03	47.68	0.15
Y	N	N	0.77	0.04	33.46	0.12
N	Y	N	0.8	0.04	59.56	0.15
Y	Y	N	0.72	0.11	48.06	0.19
N	N	Y	0.82	0.02	59.53	0.17
Y	N	Y	0.76	0.09	45.99	0.1
N	Y	Y	0.81	0.03	46.84	0.17
Y	Y	Y	0.81	0.03	42.14	0.19

Table 1: The mean values for each considered metric from the original paper's levels. Adapted from Summerville and Mateas (2016)

S?	P?	D?	e	d	l	R2
N	N	N	0.87	0.04	-7.21	0.08
Y	N	N	0.87	0.05	-2.22	0.27
N	Y	N	0.86	0.07	-13.57	0.09
Y	Y	N	0.89	0.03	-3.75	0.21
N	N	Y	0.88	0.04	-4.45	0.08
Y	N	Y	0.87	0.05	-0.6	0.24
N	Y	Y	0.88	0.04	-3.8	0.09
Y	Y	Y	0.88	0.04	2.71	0.18

Table 2: The mean values for each considered metric from my original metrics

5.3 Test Results Analysis

As shown in Figure 9, we have successfully replicated the paper's method and generated playable Super Mario Bros. levels! Pipe generation was perfect, a feat done in the original paper but not in the referenced GitHub replication project. I do not have a simulated agent like the original paper, so I cannot say exactly how many levels are playable. Through an informal assessment of their playability through sight, most of the non-snaking levels are playable.

Notice I specifically mentioned non-snaking levels. Figure 9 only sampled levels with no snaking, and from Figure 10, it is clear that our LSTM has problems with a snaked input. First, some snaking levels did not achieve perfect pipe formation. This is surprising as the original paper hypothesized that snaking the levels would help perfect pipe generation. Second, some snaking levels have platforms that change their height every column, such as the fourth sample level in Figure 10. This is caused by the level generator placing either an end-of-column or column depth marker in the middle of the sequence. Third, some snaking levels flip the map mid-way through the level, as shown in the first and second levels in Figure 10. This is undoubtedly caused by the LSTM forgetting if it was generating a column that was going up or down. While the original paper succeeded with snaking, I recommend not snaking the input.

Unlike snaking, including path information helped generate playable levels. From a glance of all the levels, levels created with path information are completable 75% of the time. I recommend including path information in the input.

The original paper's authors believed adding column depth markers would help the LSTM understand level progression. This idea does not hold, as when generating levels with column depth characters, the characters were inconsistent in number throughout the generated level. Additionally, the LSTM only takes sequences of length 200 into account. When including column depth characters, a large part of each sequence is filled by these characters instead of the level tiles themselves. This results in levels that are less likely to be playable. I recommend not including column depth characters within the input.

When comparing our calculated metrics in Table 2 against the original paper's calculated metrics in Table 1, we see several differences. First, the percentage of empty tiles (e) in our generated levels is consistent regardless of the input format. The empty percentage is slightly higher than either the original Mario levels or the papers' generated levels. The percentage of "interesting" tiles that are not simply solid or empty (d) is consistent with both the original Mario levels and the paper's levels.

A significant difference is found in the leniency metric. Leniency is defined as the number of enemies plus the number of gaps minus the number of rewards. A high leniency means a more challenging level, and a low leniency means an easier level. Our generated leniency is extremely low compared to both the original Mario levels and the paper's levels. This hints that the leniency is being calculated differently, and a quick comparison between the generated levels shows a similar number of challenges and rewards in the levels. This is likely caused by how gaps are counted, as I only count gaps along the bottom column. It is unknown how the paper is defining gaps. This metric should be ignored when comparing.

The last calculated metric is the linearity of the levels, which is how close the level can fit into a line. Notice that the linearity jumps up when snaking the input. This is due to the level flipping caused by snaking. When not snaking the input, we have a lower linearity than the original or the paper's levels.

6 Conclusion

Overall, I believe I was highly successful in replicating the original paper's method. I'm pretty amazed that I managed to generate realistic and playable levels through a neural network I made. This project took a significant amount of time to complete. Part of my original plan was to convert Zhihan Yang's Keras model to one using fast.ai. Figure 11 shows what I was stuck generating for multiple weeks.

Figure 11: Best generated level from the fast.ai model attempt

This was a massive obstacle to my progress, so instead of converting the model to fasi.ai, I ended up creating my own original model using PyTorch directly. I enjoyed the experience despite the struggles in my attempt to fix the fast.ai model. I learned so much about LSTMs, loss functions, optimizers, and ways to train and evaluate a model. These challenges were overcome with continuous research, repeated deep dives into the sequences lab, and an overwhelming desire to finish.

For future work on the same topic, I suggest three suggestions. First, test different hyperparameter combinations. For this project, I used the same hyperparameters as the original paper to compare them directly. Testing different hyperparameters is crucial in finding the optimal ones. Second, a large language model (LLM) should be used instead of an LSTM. The second related work, MarioGPT, utilizes an LLM and generates absolutely fantastic levels. Third, expand the dataset to include more Mario levels. I used only levels from Super Mario Bros 1 and 2, yet there are millions of player-created Super Mario Bros levels in Super Mario Maker 1 and 2. With access to such a large dataset of levels, better and more complex generators can be trained to generate even better and more complicated levels.

I wanted to thank you for your class. At the beginning of the semester, I knew absolutely nothing about machine learning, but by the end, I had learned a perfect introduction to the subject. The weekly videos and labs were extremely helpful and well-planned. Thank you for the semester.

References

PyTorch. (2023). LSTM. LSTM - PyTorch 2.3 documentation. https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

Shi, P., & Chen, K. (2016). Online level generation in Super Mario Bros via learning constructive primitives. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. https://doi.org/10.1109/cig.2016.7860397

Sudhakaran, S., González-Duque, M., Glanois, C., Freiberger, M., Najarro, E., & Risi, S. (2023). *MarioGPT: Open-Ended Text2Level Generation through Large Language Models.* Arxiv. https://doi.org/10.48550/arXiv.2302.05981

Summerville, A. & Mateas, M. (2016). *Super Mario as a String: Platformer Level Generation Via LSTMs.* ArXiv. https://doi.org/10.48550/arXiv.1603.00930

T. J. J., R. (2024, Sept 2). *LSTMs explained: A complete, technically accurate, conceptual guide with keras.* Medium. https://medium.com/analytics-vidhya/lstms-explained-a-complete-technically-accurate-conceptual-guide-with-keras-2a650327e8f2

Yang, Z. (2020). *super_mario_as_a_string.* GitHub. https://github.com/zhihanyang2022/super_mario_as_a_string

Sharing Agreement

You can share all of my work from this semester, including the GitHub repository of my project. You don’t need to hide my name.