

# Super Mario as a String: Platformer Level Generation Via LSTMs

**Adam J. Summerville, Michael Mateas**

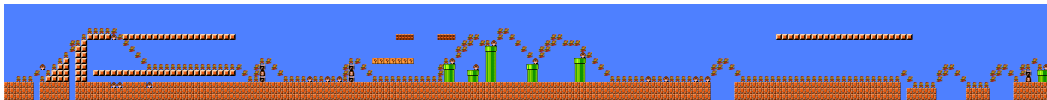
University of California, Santa Cruz

1156 High Street

Santa Cruz, CA 95066

(+1) 831-459-0111,

asummerv@ucsc.edu, michaelm@soe.ucsc.edu



## ABSTRACT

The procedural generation of video game levels has existed for at least 30 years, but only recently have machine learning approaches been used to generate levels without specifying the rules for generation. A number of these have looked at platformer levels as a sequence of characters and performed generation using Markov chains. In this paper we examine the use of Long Short-Term Memory recurrent neural networks (LSTMs) for the purpose of generating levels trained from a corpus of *Super Mario Bros.* levels. We analyze a number of different data representations and how the generated levels fit into the space of human authored *Super Mario Bros.* levels.

## Keywords

PCG, Video Games, Machine Learning, Recurrent Neural Networks, LSTM

## INTRODUCTION

Procedural Content Generation (PCG) of video game levels has existed for many decades, with the earliest known usage coming from *Beneath Apple Manor* in 1978. As an area of academic study, there have been numerous approaches taken ranging from human-authored rules of tile placement (Shaker et al. 2011), to constraint solving and planning (Smith et al. 2010). These approaches require the designer of the algorithm to specify either the rules of the generation or a method for evaluating the generated levels. More recently, machine learning approaches have been used so that a designer can train a generator from examples. These have ranged from matrix factorization (Shaker and Abou-Zleikha 2014, Summerville and Mateas 2015), to a number of different Markov chain approaches (Dahlskog et al. 2014, Snodgrass and Ontañón 2013, Summerville, Philip, et al. 2015), to different graph learning approaches (Guzdial and Riedl 2015, Londoño and Missura 2015), to Bayesian approaches (Summerville, Behrooz, et al. 2015). A number of these approaches treat the level as a sequence of characters, and use Markov chains to learn character-to-characters transition probabilities. These approaches can have good local coherence but struggle with global coherence. This can be remedied by considering longer and longer histories, but this leads to a rapidly exploding combinatoric space leading to a very sparse set of learned transitions.

Proceedings of 1<sup>st</sup> International Joint Conference of DiGRA and FDG

©2016 Authors. Personal and educational classroom use of this paper is allowed, commercial use requires specific permission from the author.

Long Short-Term Memory recurrent neural networks (LSTMs) represent the state of the art in sequence learning. They have been used for translation tasks (Sutskever et al. 2014), speech understanding (Graves et al. 2013), video captioning (Venugopalan et al. 2014), and more. In this paper we present a method for utilizing LSTMs for the purpose of learning and generating levels. We examine eight different data representations for the levels and compare the generated levels using a number of different evaluation statistics.

## RELATED WORK

Machine learned platformer level generation has mostly been performed via learning Markov chain transition probabilities. The transitions in the Markov chains have seen two major approaches, either tile-to-tile transitions or vertical slices of the level. The vertical slice approach was first used by Dahlskog et al. (2014) and was later used by Summerville et al. (2015). In this approach the states of the Markov chain are taken as a vertical slice of tiles and the transitions are learned from slice-to-slice. This has the benefit of directly encoding vertical structure which is important due to the fact that certain tiles are much more likely to be at the top (like empty tiles) and others are much more likely to be at the bottom (such as the ground). However, this comes with the severe drawback of removing the ability to generate novel vertical slices. It also has the problem that the state space is much larger than just the number of tiles, making the learned transition space much sparser than just learning tile-to-tile transitions.

Snodgrass and Ontañón (2013,2014) have used tile-to-tile transitions for the placement of tiles. Because of this, they do not have the problems of the vertical slice approach, but this brings a host of other issues. Namely, it can be far too likely for the system to generate sequences of tiles that violate implicit semantic constraints, such as generating ground tiles at the top of the level. Multiple attempts were made to reduce such problems via learning different chains for varying heights, or learning a hierarchical Markov chain (). We build on the tile-to-tile approach for the work we report here as it is better able to generalize over a larger collection of levels. With the column based approach it is very easy to imagine that a new level might have a column that has never been seen before and so the system will be unable to generalize, but it is much harder to imagine a single-tile-to-tile transition that has never been seen (unless it is one that should never be seen like an enemy embedded inside a pipe).

The above approaches both have the drawback of no guarantees about playability. In Snodgrass's work, over half of the levels produced were unable to be completed by a simulated agent (Snodgrass and Ontañón 2013). Summerville, et al. (2015) used Monte Carlo Tree Search as a way of guiding Markov chain construction of levels to help guarantee a playable level. This does allow for some authorial input on the part of a designer by letting them change the reward function to create levels with different features (e.g. more/less gaps), at the cost of biasing the generation away from the learned transition probabilities in ways that might not be easily understood or resolved. Despite the different applications of Markov chains, they all suffer from the same main drawback, that there is no good way to have a long memory in the chain without needing massive amounts of data to fill the transition table. This leads to good local coherence but poor global coherence, in that they can reliably pick the next tile in a way that makes sense, but have difficulty handling larger structures or patterns.

The work of Guzdial and Riedl (2015) used clustering to find latent groupings and then learned relative placements of tile blocks in Mario levels. This has the benefit of learning structure over much longer ranges than is possible in the Markov chain examples. While the work of Dahlskog was able to use 3 tiles worth of history Guzdial’s work considered tile distances up to 18 tiles in distance. A similar graph approach was used by Londoño and Missura (2015), but learned a graph grammar instead. Their work is unique in platformer level generation in that they used semantic connections such as tile X is reachable from tile Y instead of just physical connections.

Procedural generation via neural networks has most been focused on image and text generation. Both have typically been byproducts of trying to train a network for some sort of classification process. Google’s Inceptionism (Szegedy et al. 2014) took the product of training an image classifier and manipulated images to better match the set classification class, such as dogs. Facebook’s Eyescream had image generation as the goal (Denton et al. 2015) and used adversarial networks, one network trying to generate images and the other trying to determine if what it was seeing was generated or real.

Text generation from neural networks has been done by numerous people for various goals such as translation (Sutskever et al. 2014), text classification (Graves 2013), and even trying to determine the output of code simply by reading the source code (Zaremba and Sutskever 2014) and then writing what it believes to be the output.

The work of Graves () went beyond text generation and used sequences of stylus locations and pressure to learn calligraphic character writing. Similarly, the DRAW system (Gregor et al. 2015) used LSTMs to generate address signs learned from Google street view by using an attentional system to determine where to focus and then using that as the sequencing of where to place a virtual paint brush and what color to paint, e.g. it learns a sequence of  $(x, y, r, g, b)$ .

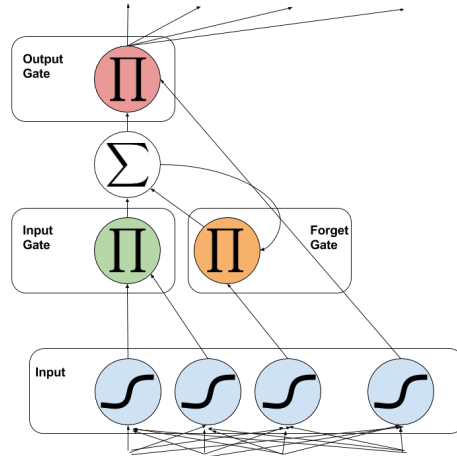
## **LSTM SEQUENCE GENERATION**

In this section we will first cover how LSTM Recurrent Neural Networks operate. We will then cover the basics of our data specification as well as the variants we used.

### **Recurrent Neural Network Architecture**

Recurrent Neural Networks (RNNs) have been used for a number of different sequence based tasks. They operate in a manner similar to standard neural networks, i.e. they are trained on data and errors are back-propagated to learn weight vectors. However, in an RNN the edge vectors are not just connected from input to hidden layers to output, they are also connected from a node to itself across time. This means that back-propagation occurs not just between different nodes, but also across time steps. However, a common problem with standard RNNs is known as “the vanishing gradient problem” (Hochreiter 1991), where the gradient of the errors very quickly dissipates across time. While different in why it occurs, the end result is similar to the problem found with Markov chains, i.e. good local coherence but poor global coherence.

LSTMs are a neural network topology first put forth by Hochreiter and Schmidhuber (Hochreiter and Schmidhuber 1997) for the purposes of eliminating the vanishing gradient problem.

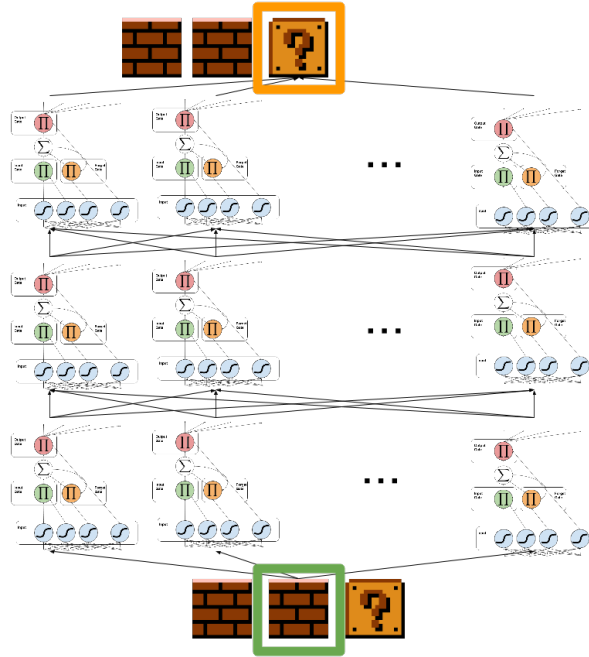


**Figure 1** Graphical depiction of an LSTM block.

LSTMs work to solve that problem by introducing additional nodes that act as a memory mechanism, telling the network when to remember and when to forget. A standard LSTM architecture can be seen in figure 1. At the bottom are nodes that operate as in a standard neural network, i.e. inputs come in, are multiplied by weights, summed, and that is passed through some sort of non-linear function (most commonly a sigmoid function such as the hyperbolic tangent) as signified by the S-shaped function. The nodes with  $\Sigma$  simply sum their inputs with no non-linear activation, and the nodes with  $\Pi$  simply take the product of their inputs. With that in mind, the left-most node on the bottom can be thought of as the *input* to an LSTM block (although for all purposes it is interchangeable with the node second from the left). The node second from the left is typically thought of as the *input gate*, since it is multiplied with the input, allowing the input through when it is close to 1, and not allowing the input in when it is close to 0. Similarly, the right-most node on the bottom acts as a corresponding *output gate*, determining when the value of the LSTM block should be output to higher layers. The node with the  $\Sigma$  acts as the *memory*, summing linearly and as such not decaying through time. It feeds back on itself by being multiplied with the second from the right node, which acts as the *forget gate*, telling the memory layer when it should drop whatever it was storing.

These LSTM blocks can be composed in multiple layers with multiple LSTM blocks per layer, and for this work we used 3 internal layers, each consisting of 512 LSTM blocks which can be seen in figure 2. The input layer to the network consists of a One-Hot encoding where each tile has a unique binary flag which is set to 1 if the tile is selected and all others are 0. The final LSTM layer goes to a SoftMax layer, which acts as a Categorical probability distribution for the One-Hot encoding.

Our networks were trained using Torch7 (Collobert et al.) based on code from Andrej Karpathy (Karpathy 2015). In addition to choosing size and depth of the network, RNNs (and subsequently LSTMs) come with the additional hyperparameter of how many time steps to consider during the back-propagation through time, which we set to 200 data points. A common problem in machine learning is that of memorization, where the algorithm exactly



**Figure 2** Graphical depiction of our chosen architecture. The green box at the bottom represents the current tile in the sequence, the tile to the left the preceding tile, and the tile to the right 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.

memorizes the input data at the expense of being able to generalize to unseen data. There exist numerous methods to avoid this overfitting, but for this work we used dropout (Srivastava et al. 2014). Dropout is a technique where during training a pre-specified percentage of LSTM blocks are dropped from the network (along with their corresponding connections) at random for each training instance. This has the effect of effectively training an ensemble of networks during the training of one network and reduces the amount of co-adapting that occurs across nodes. Having decided on the network architecture, we will now get to the core of our data representation and what we mean by a data point.

### Data Specification

LSTMs are most commonly used to predict the next item in a sequence. They do this by producing a probability distribution over possible next items and predicting the most likely item. For this to work, our data must be a sequence. If we were to consider a format such as the one used by Dahlskog et al. (Dahlskog et al. 2014) we could simply consider a level as a sequence of slices progressing from left-to-right, but this comes with multiple drawbacks:

- It can only reproduce vertical slices encountered in the training set - meaning that it could be utterly unable to handle a previously unseen slice
- This drastically increases the size of the input space as we will see in a second

Instead we treat each individual tile of a level from *Super Mario Bros.* as a point in our sequence, that is, like a character in a string. The tile types we consider are:

- **Solid** - Any tile that is solid and has no other interactions, most commonly ground, giant mushroom, or inert block tiles of which no distinction is made
- **Enemy** - Any enemy, again no distinctions are made between enemies
- **Destructible Block** - A block that can be destroyed by Super Mario
- **Question Mark Block With Coin** - A ?-Block that only contains a coin
- **Question Mark Block With Power-up** - A ?-Block that contains a powerup, either a mushroom/flower, a star, or a 1-up
- **Coin** - A coin
- **Bullet Bill Shooter Top** - The top of a Bullet Bill shooting cannon
- **Bullet Bill Shooter Column** - The column that supports the top of the Bullet Bill shooter
- **Left Pipe** - The left side of a pipe
- **Right Pipe** - The right side of a pipe
- **Top Left Pipe** - The top left side of a pipe
- **Top Right Pipe** - The top right side of a pipe
- **Empty** - The lack of all other tile types

Distinctions are made between the variants of ?-Blocks due to the extreme difference in effect that they make in the game. The vast majority of ?-Blocks contain only coins and if we were to ignore the difference between ?-Blocks we would be losing crucial information about where the most important power-up bearing blocks are located. Pipes are one of the most important structures in the game. Since previous systems have shown difficulty accurately producing non-gibberish pipes (Snodgrass and Ontañón 2013), we want to consider the different parts of the pipe separately. It is important to know that the left half of a pipe is different from the right half in the sequence, as the latter can never be encountered before the former.

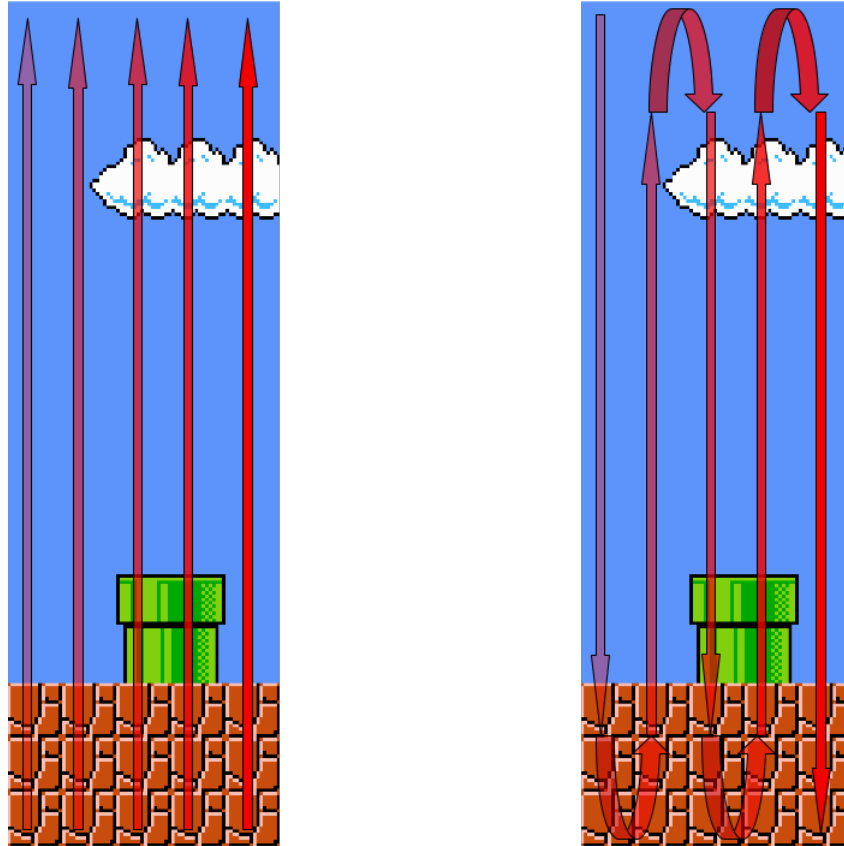
Each of these tile types can be considered as a character in a string, but a platformer level has 2 dimensions, not just 1, meaning that we have to induce an ordering across the 2 dimensions. The naïve ordering would be left-to-right and most likely bottom-to-top, as the levels progress from left-to-right and there is a higher density of important tiles at the bottom of a level due to the nature of simulated gravity. However, this sequencing would be highly problematic for most platformers. Levels in *Super Mario Bros.* are between 200 and 500 tiles in width, which would mean that the LSTM would need to remember for at least that distance what tile it had placed in the previous row. While LSTMs are good at remembering over longer distances, this would strain the algorithm for no particularly important reason.

Instead, we consider eight different induced tile orderings determined by answers to the following three questions:

- *Bottom-To-Top* or *Snaking*?
- *Includes Path Information*?
- *Includes Column Depth Information*?

We now discuss these possible orderings in depth.

### *Bottom-To-Top* vs. *Snaking*



**Figure 3** Illustration of the difference between going from bottom-to-top (LEFT) and *snaking* (RIGHT) for sequence creation

As mentioned, we are considering levels vertically and then horizontally. This comes first with a decision of whether to go bottom-to-top or top-to-bottom, of which we chose to go from bottom-to-top due to the higher density of important, non-empty tiles at the bottom of a level. This can be seen in the left of figure 3. To do this we include a special character in the sequences to indicate when we have reached the top of the level and are moving to the next column.

However, there is another, less intuitive manner of progressing through the level: *snaking*. By snaking we mean alternating bottom-to-top and top-to-bottom orderings as seen in the right of figure 3. In this snaking ordering we are following (Sutskever et al. 2014) which demonstrated that by reversing input streams one can induce better locality in the sequence. To see this, consider the pipe in figure 3. In the bottom-to-top ordering there are 17 tiles

between the left and right halves of the pipe, while there are only 7 in the snaking ordering. Snaking also comes with the hidden benefit of doubling the size of our dataset. When snaking we generate two sequences for a level, one that starts from bottom-to-top as well as one that starts from top-to-bottom.

### *Path Information*

In what we consider to be a novel contribution to the field of machine learned procedural level generation, we also considered the path(s) that a player would/could take through the level. A common problem with machine learned level generation has been in producing levels that are playable. Snodgrass and Ontañón found that only roughly half of all levels produced were able to be completed by a simulated agent. However, if we consider a player’s path through the level as a crucial piece of information in the specification of the level and include it in the training information, we are much more likely to generate playable levels since we are not just generating the level geometry but also a player’s path through the level.

We ran a simulated agent through existing levels using a tile-level A\* agent. Any empty space that the agent passed through was replaced with a special character denoting the player’s path. We considered all paths that were within 10 moves of optimal for this, but this could either be loosened to simulate worse paths or tightened to only consider optimal paths.

### *Column Depth*

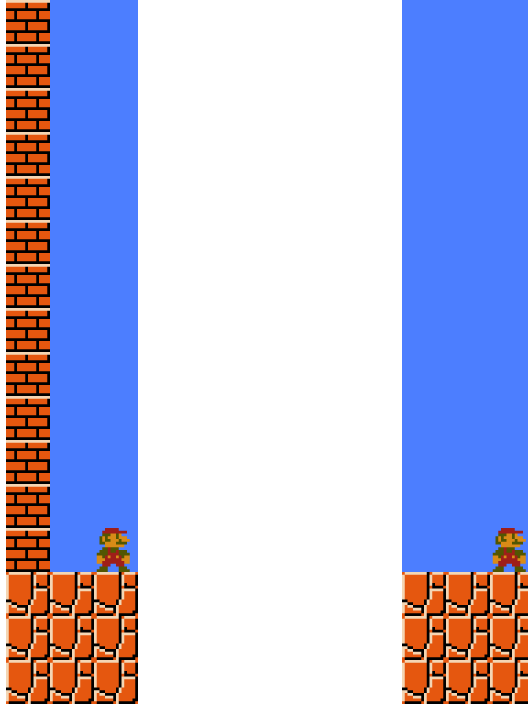
We also considered how far into the level we are. Obviously, the LSTM has some memory as to where it has been and therefore where it exists in the level, but our history of 200 tiles means that in essence it only remembers approximately 12 columns in the past. This captures most patterns that might be encountered, but is unlikely to capture larger considerations such as a ramping in intensity as the player progresses further and further into the level. To try to get the LSTM to understand level progression we introduced a special character that is incremented every 5 columns. In other words columns 0-4 have no special character, columns 5-9 have 1 special character, columns 10-14 have 2, etc.

## **RESULTS**

We trained eight networks, one for each of the induced sequences. We had a total of 15 levels from *Super Mario Bros.* and 24 levels from the *Japanese Super Mario Bros. 2* for a total of 39 levels. We used a 70%-30% training-evaluation split in the course of the training. After every 200 tiles in the training sequence we performed an evaluation on a held out evaluation set, determining how well the LSTM was able to predict unseen sequences. We save this version of the network if it has the best score on the evaluation set, with the belief that this will be the network that has the best ability to generalize from the training set. When the training reached a plateau for more than 2 epochs (runs through the training set) we stopped training. The error criterion is the negative log-likelihood, which is 0 if the correct value is predicted with 100% certainty and increases as the tile is less likely to be predicted.

The results of LSTMs over the evaluation task can be seen in table 1. The change that had the biggest effect on performance was the inclusion of path information, with the networks





**Figure 4** The below ground seed (LEFT) and the above ground seed (RIGHT).

with paths doing roughly twice as well as the networks without. Interestingly, the other changes considered without path information worsened network performance, but in tandem with path information improved overall performance. This demonstrates the difficulty of determining a good data specification; each non path change of the data specification away from the naïve base case decreased the performance of the LSTM, but when combined with the path information produced performance over twice as good as the next best network and  $20\times$  better than the naïve, despite the fact that the most complex specification had a vocabulary that was 36% larger from the naïve specification.

However, our goal is not to analyze how well a sequence of tiles might have believably come from an existing *Super Mario Bros.* game, but rather to generate new levels with properties hopefully similar to existing *Super Mario Bros.* levels. To that end we used each of the final trained networks to generate 4000 levels. Each generator was given a seed sequence and then asked to generate until it reached an end-of-level terminal symbol 2000 times for each of two different seeds, an underground and an above ground seed as seen in figure 4. Each seed consists of three columns of tiles, as well as an initial special character denoting the start of the level. We then analyzed these levels for a number of different properties. Some, such as linearity and leniency, have been used to evaluate platformer level generation since their inception (Smith and Whitehead 2010). Others are based off of more recent proposals from Cannosa and Smith (Cannosa and Smith 2015). In this evaluation we have also included the metrics for the levels from the dataset. It is hard to generate an intuition for whether a given metric is a good indicator that the generator is producing content that we should be happy with, but since our goal is to produce levels that have properties similar to existing levels we can compare how well the output space of our generator matches that of the existing levels.

Snaking?	Paths?	Line #?	Negative Log-Likelihood
N	N	N	0.0920
Y	N	N	0.1540
N	Y	N	0.0694
Y	Y	N	0.0573
N	N	Y	0.1376
Y	N	Y	0.1143
N	Y	Y	0.0404
<b>Y</b>	<b>Y</b>	<b>Y</b>	<b>0.0177</b>

**Table 1** The error of the best network for each data format. The **Snaking-Path-Depth** has the lowest error.

S?	P?	D?	$C$	$e$	$n$	$d$	$p$	$l$	$R^2$	$j$	$j_i$
Human Authored			100%	0.82	0.76	0.04	0.07	71.19	0.13	21.98	12.06
N	N	N	67%	<b>0.81</b>	0.55	<b>0.03</b>	0.09	<b>47.68</b>	<b>0.15</b>	<b>20.66</b>	<b>10.65</b>
Y	N	N	39%	<b>0.77</b>	0.52	<b>0.04</b>	0.1	<b>33.46</b>	<b>0.12</b>	<b>28.61</b>	<b>6.81</b>
N	Y	N	96%	<b>0.8</b>	<b>0.72</b>	<b>0.04</b>	0.09	<b>59.56</b>	<b>0.15</b>	<b>24.17</b>	<b>12.17</b>
Y	Y	N	93%	0.72	<b>0.67</b>	0.11	0.1	<b>48.06</b>	<b>0.19</b>	<b>19.2</b>	<b>9.34</b>
N	N	Y	51%	<b>0.82</b>	0.51	<b>0.02</b>	0.09	<b>59.53</b>	<b>0.17</b>	<b>19.69</b>	<b>9.96</b>
Y	N	Y	37%	<b>0.76</b>	0.44	0.09	0.1	<b>45.99</b>	<b>0.1</b>	<b>15.04</b>	<b>6.68</b>
N	Y	Y	94%	<b>0.81</b>	0.64	<b>0.03</b>	0.09	<b>46.84</b>	<b>0.17</b>	<b>18.31</b>	<b>8.21</b>
Y	Y	Y	97%	<b>0.81</b>	<b>0.68</b>	<b>0.03</b>	0.09	<b>42.14</b>	<b>0.19</b>	<b>19.02</b>	<b>8.04</b>

**Table 2** The mean values for each of the considered metrics. Values in bold are within 1 standard deviation of the original, human-authored levels.

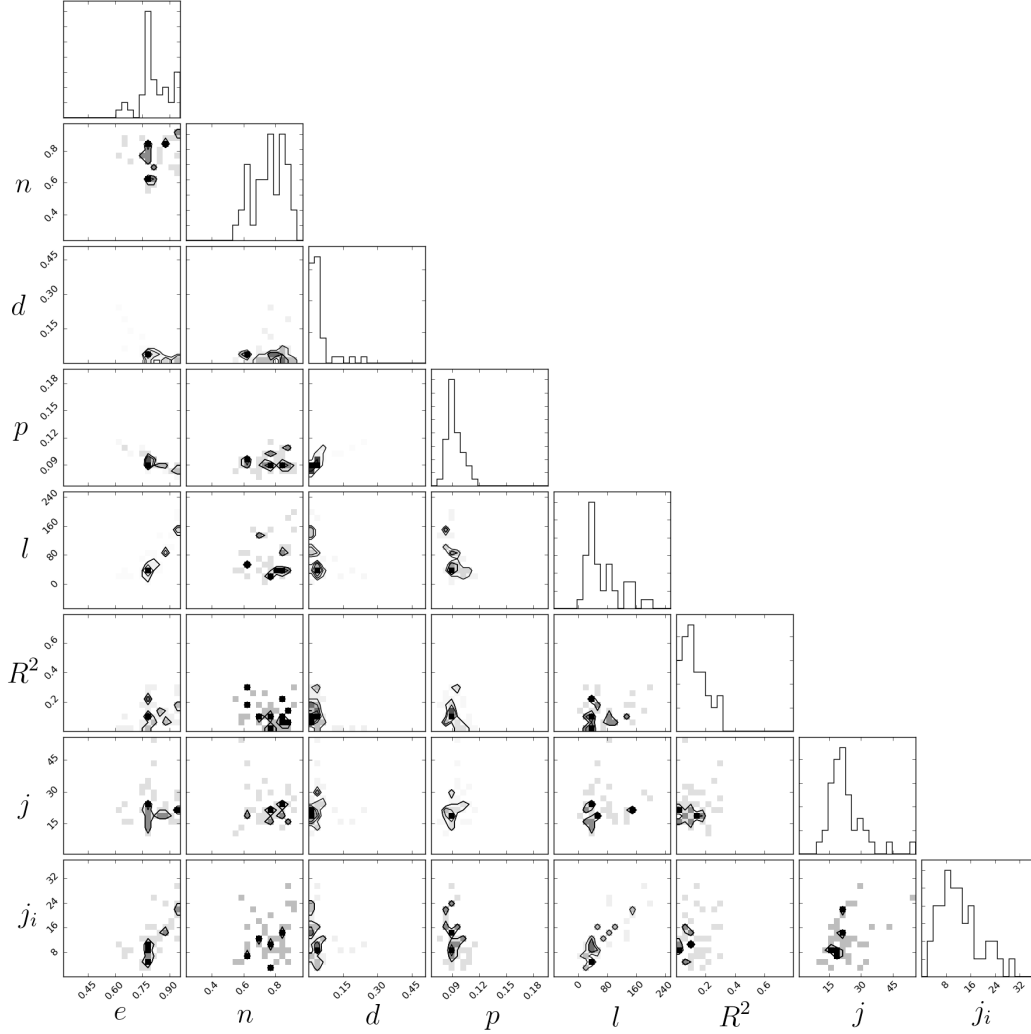
The variants of generators are:

- **S?** - Y if the generator was trained on *snaking* data, N if bottom-to-top
- **P?** - Y if the generator had path information, N if not
- **D?** - Y if the generator had depth information, N if not

The metrics we considered were:

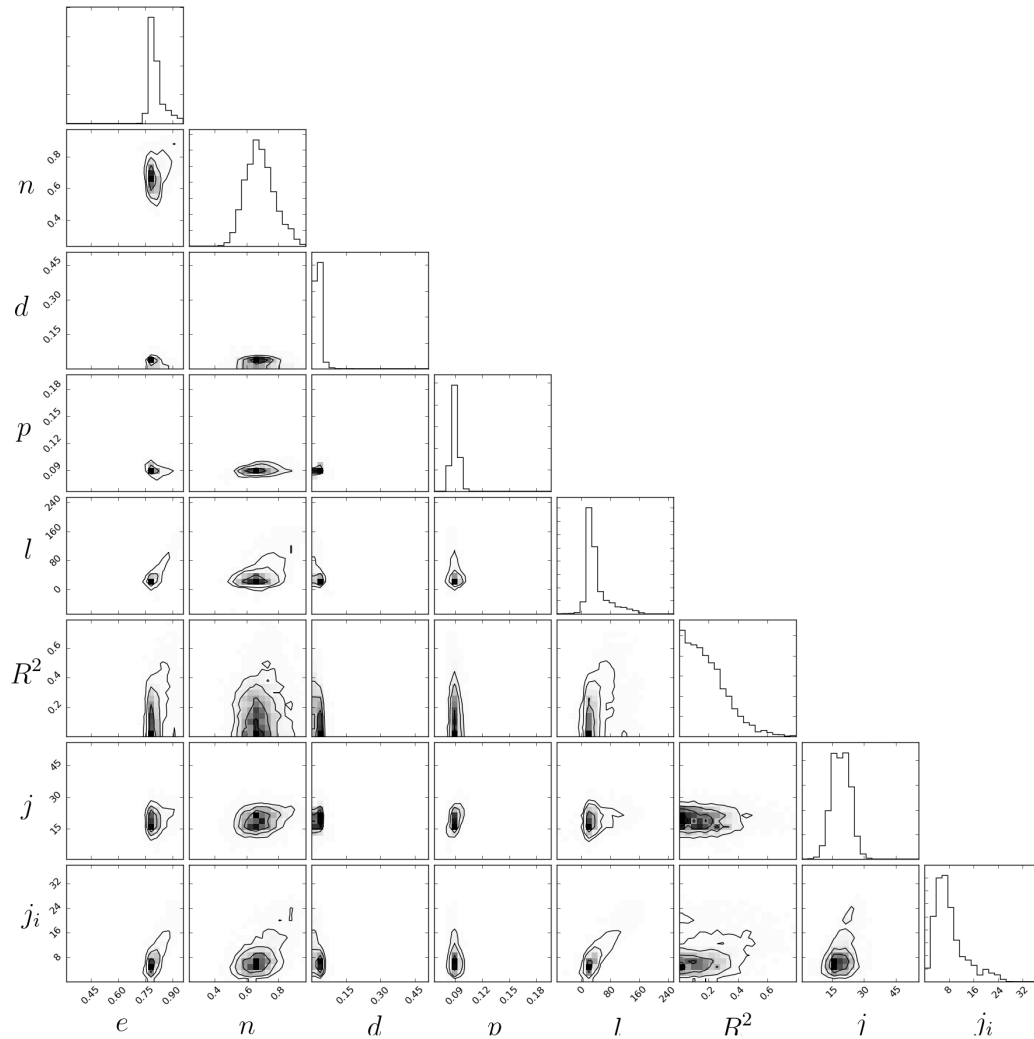
- $C$  - The percentage of the levels that are completable by the simulated agent
- $e$  - The percentage of the level taken up by empty space
- $n$  - The negative space of the level, i.e. the percentage of empty space that is actually reachable by the player
- $d$  - The percentage of the level taken up by “interesting” tiles, i.e. tiles that are not simply solid or empty
- $p$  - The percentage of the level taken up by the optimal path through the level
- $l$  - The leniency of the level, which is defined as the number of enemies plus the number of gaps minus the number of rewards

- $R^2$  - The linearity of the level, i.e. how close the level can be fit to a line
- $j$  - The number of jumps in the level, i.e. the number of times the optimal path jumped
- $j_i$  - The number of meaningful jumps in the level. A meaningful jump is a jump that was induced either via the presence of an enemy or the presence of a gap.



**Figure 5** A corner plot of the expressive range for the original levels. The histograms along the diagonal are for each of the metrics listed along the bottom. The others are 2D contour plots showing the density of levels with the bottom metric being the X axis and the left metric being the Y axis.

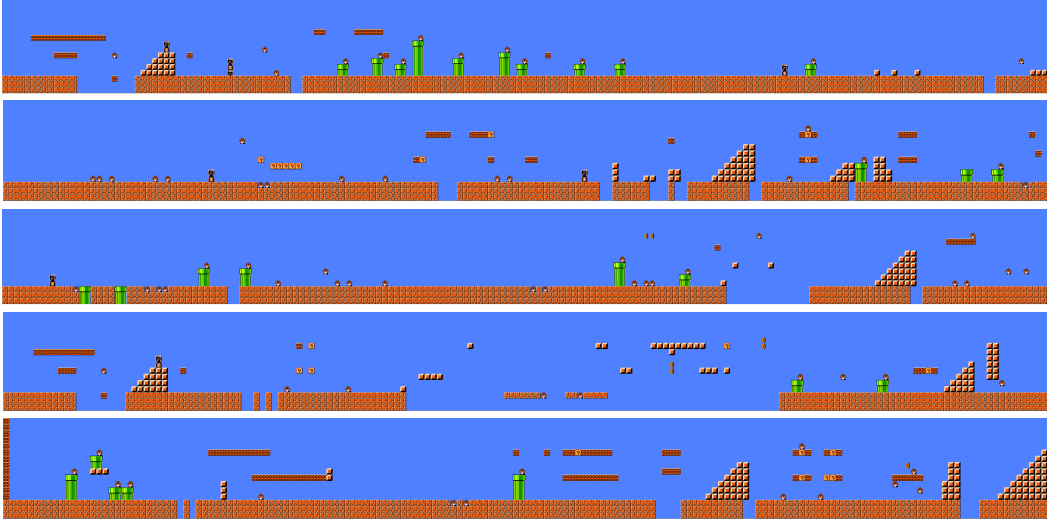
In table 2 we can see the mean values of the level metrics for the different generators. A metric value is **bold** if the value was within one standard deviation of the value of the original levels. All of the generators generally performed well at matching the existing levels, at least in these metrics. Certain metrics, like percentage of empty space, leniency, linearity, percentage of decoration, # of jumps, and # of induced jumps were fairly consistently good across all generators. Interestingly, the lengths of paths through the generated levels were all consistent with each other and all were consistently higher than in the existing levels.



**Figure 6** A corner plot of the expressive range for the levels generated by the **Snaking-Path-Depth** generator.

Not surprisingly, the generators with path information did much better at producing completable levels. To our knowledge, the playability of our generated levels is the best of any generator employing machine learning, beating the best reported of 66% (Snodgrass and Ontañón 2015). The 97% beats even the best reported human-authored system from the 2011 Mario AI Competition by Mawhorter of 94% as reported by Snodgrass and Ontañón (Snodgrass and Ontañón 2015).

Following are two corner plots which show the density of generated levels across a single dimension (the outer diagonal corresponds to a histogram for the metric along the bottom) or across two dimensions (the left and the bottom corresponding to x and y dimensions). In figure 5 we can see the original human-authored levels and in figure 6 we can see the results from the **Snaking-Path-Depth** generator. Generally, the generator does a very good job of matching the expressive range of the original levels.



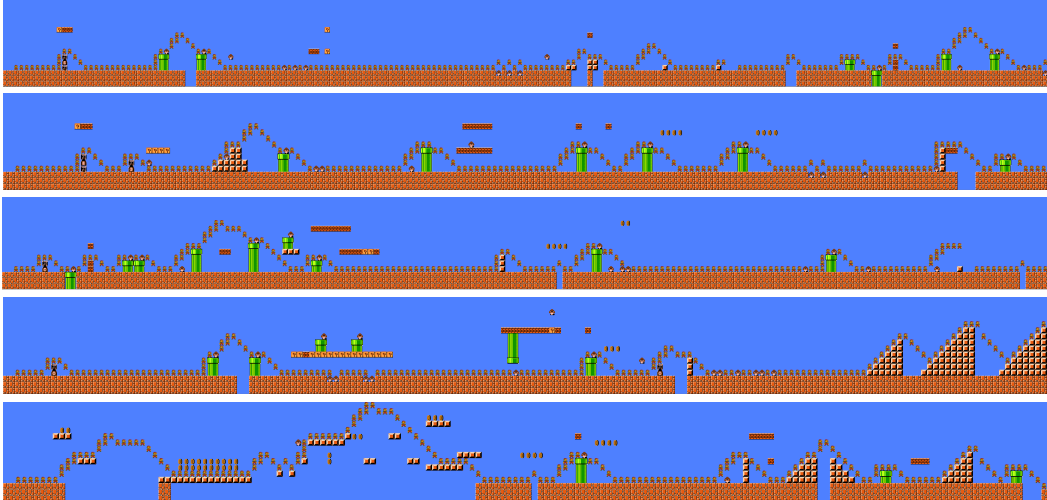
**Figure 7 5** levels from the **Snaking-Path-Depth** generator without path information displayed.

However, until we are able to train a system to analyze a level for how good it is or how fun it is (which seems a lofty AI-complete goal), such metric-based evaluation must be supplemented with an informal analysis of example generated levels to verify that the metrics actually correspond to human perceptions of high-quality levels. Following are a selection of levels chosen at random from the **Snaking-Path-Depth** generator. The first 5 are levels as they would be shown, the latter 5 with the generator’s included path information. We believe that these random samples demonstrate the quality of the generated levels and showcase the breadth of patterns it has learned, such as a series of pipes of differing heights, pyramidal structures, and even sequences of unconnected platforms that are still playable. But it is just a random sampling and all of the generated levels can be found online<sup>1</sup>.

Ignoring the fact that the LSTMs are able to generate playable levels, it is interesting that the LSTMs are even able to produce coherent column-to-column tile placements. Even the fact that the LSTMs never incorrectly produce a column delimiting character is testament to their sequence learning capabilities. Without fail the LSTMs only produce columns of 16 tiles, exactly the size of the input columns. This does not come from telling the generator to stop or reset after 16 tiles, but from learned sequences.

## CONCLUSION AND FUTURE WORK

In this paper we have demonstrated a novel use of LSTM Recurrent Neural Networks for the generation of platformer levels, specifically Super Mario Bros. levels. Towards this, we have created multiple novel ways of interpreting a Super Mario Bros. level as a linear sequence. Most importantly, we have demonstrated that the introduction of player path information can dramatically improve the quality of generated levels, most specifically in the playability of generated levels, but also in terms of exhibiting level metrics closer to the human-authored levels. We have also used a wider range of metrics for characterizing our levels than have previously been used, allowing us to perform a more nuanced comparison between human-authored and generated levels.



**Figure 8 5** levels from the **Snaking-Path-Depth** generator with generated path information displayed.

We believe that the introduction of path information to the generation process also opens new avenues for automated analysis of levels. When a designer first designs a level, they often have a hypothesis of what the player will do before any playtesting has occurred. This system works in a similar way by assigning likelihoods to where a player will go based on observation. We would like to extend this system so that it not only generates levels, but also analyzes presented levels. While it is possible to send simulated agents through the level to generate analysis, something we did for this work, future work may enable the system to make suggestions based solely on observations without explicitly modeling or simulating a player. We would like to utilize human playthroughs gathered from gameplay video, to be able to suggest paths that actual human players would take instead of just simple A\* agents.

Recent work in neural network learning have used attention based systems to learn sequences from data that is not typically thought of as a sequence, such as an image. While the sequencing in this work is able to produce good results, we feel it could be made more robust by incorporating the sequencing into the learning task. We also believe that this would enable the system to operate in regimes where a simple left-to-right progression is unsuited, such as games like *The Legend of Zelda* or *Metroid*. These games tend to have highly non-linear progressions with numerous backtrackings and dead-ends required during play. An attentional system could learn to navigate these non-linear maps.

A similar concern for future work is deconvolving time and space. In standard platformer levels, the player progresses from left-to-right, which has led to many generators, this one included, to operate as if time has a linear mapping to space, e.g. x-coordinate. If we were to treat the screen as a tensor with dimensions of width, height, and tile type, we could perform a Tensor auto-regression across time. For a game like *Super Mario Bros.*, this would work simply as progressing from left-to-right, but for the above mentioned non-linear games, this would work to eliminate the spurious correlation of time and x-coordinate.

## ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. IIS-1409992.

## ENDNOTES

1. <http://tinyurl.com/SMBRNN>

## BIBLIOGRAPHY

- Canossa, Alessandro, and Gillian Smith. 2015. "Towards a Procedural Evaluation Technique: Metrics for Level Design." In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.
- Collobert, Ronan, Koray Kavukcuoglu, and Clément Farabet. *Torch7: A Matlab-like Environment for Machine Learning*.
- Dahlskog, Steve, Julian Togelius, and Mark J. Nelson. 2014. "Linear levels through n-grams." In *Proceedings of the 18th International Academic MindTrek Conference*.
- Denton, Emily L., Soumith Chintala, Arthur Szlam, and Robert Fergus. 2015. "Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks." In *CoRR*.
- Graves, Alex. 2013. "Generating Sequences With Recurrent Neural Networks." In *CoRR*.
- Graves, Alex, Abdel-rahman Mohamed, and Geoffrey E. Hinton. 2013. "Speech Recognition with Deep Recurrent Neural Networks." *CoRR* abs/1303.5778.
- Gregor, Karol, Ivo Danihelka, Alex Graves, and Daan Wierstra. 2015. "DRAW: A Recurrent Neural Network For Image Generation." In *CoRR*.
- Guzdial, Matthew, and Mark O. Riedl. 2015. "Toward Game Level Generation from Gameplay Videos." In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.
- Hochreiter, Sepp. 1991. "Learning Causal Models of Relational Domains." In *Master's thesis, Institut fur Informatik, Technische Universitat, Munchen*.
- Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long Short-Term Memory." *Neural Computing*.
- Karpathy, Andrej. 2015. *The Unreasonable Effectiveness of Recurrent Neural Networks*. <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>.
- Londoño, Santiago, and Olana Missura. 2015. "Graph Grammars for Super Mario Bros Levels." In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.
- Nintendo RD1. 1986. *Metroid*. [Famicom] Nintendo, Kyoto Japan.
- Nintendo RD4. 1983. *Super Mario Brothers*. [Famicom] Nintendo, Kyoto Japan.
- Nintendo RD4. 1986. *Super Mario Brothers 2*. [Famicom] Nintendo, Kyoto Japan.
- Nintendo RD4. 1986. *The Legend of Zelda*. [Famicom] Nintendo, Kyoto Japan.
- Shaker, Noor, and Moahamed Abou-Zleikha. 2014. "Alone We Can Do So Little, Together We Can Do So Much: A Combinatorial Approach for Generating Game Content." In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

- Shaker, Noor, Julian Togelius, Georgios N. Yannakakis, Ben George Weber, Tomoyuki Shimizu, Tomonori Hashiyama, Nathan Sorenson, et al. 2011. “The 2010 Mario AI Championship: Level Generation Track.” *IEEE Trans. Comput. Intellig. and AI in Games* 3:332–347.
- Smith, Gillian, and Jim Whitehead. 2010. “Analyzing the Expressive Range of a Level Generator.” In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.
- Smith, Gillian, Jim Whitehead, and Michael Mateas. 2010. “Tanagra: a mixed-initiative level design tool.” In *FDG*, 209–216. ACM.
- Snodgrass, Sam, and Santiago Ontañón. 2013. “Generating Maps Using Markov Chains.” In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.
- . 2014. “Experiments in Map Generation using Markov Chains.” In *Proceedings of the 2014 Conference on the Foundations of Digital Games (FDG 2014)*.
- . 2015. “A Hierarchical MdMC Approach to 2D Video Game Map Generation.” In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.
- Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting.” *Journal of Machine Learning Research* 15:1929–1958.
- Summerville, Adam, Morteza Behrooz, Michael Mateas, and Arnav Jhala. 2015. “The Learning of Zelda: Data-Driven Learning of Level Topology.” In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.
- Summerville, Adam, and Michael Mateas. 2015. “Sampling Hyrule: Multi-Technique Probabilistic Level Generation for Action Role Playing Games.” In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.
- Summerville, Adam, Shweta Philip, and Michael Mateas. 2015. “MCMCTS PCG 4 SMB: Monte Carlo Tree Search to Guide Platformer Level Generation.” In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.
- Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. 2014. “Sequence to Sequence Learning with Neural Networks.” In *CoRR*.
- Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. “Going Deeper with Convolutions.” In *CoRR*.
- Venugopalan, Subhashini, Huijuan Xu, Jeff Donahue, Marcus Rohrbach, Raymond J. Mooney, and Kate Saenko. 2014. “Translating Videos to Natural Language Using Deep Recurrent Neural Networks.” *CoRR* abs/1412.4729.
- Zaremba, Wojciech, and Ilya Sutskever. 2014. “Learning to Execute.” In *CoRR*.