Learning Based Video Game Level Generation

ERIC ALZHEIMER, Rowan University

Many games rely on procedural generation, the use of algorithms to generate content, as a way of constructing game levels. In "Super Mario as a String: Platformer Level Generation Via LSTMs", Summerville and Mateas [14] use LSTM neural networks to generate levels for Super Mario Brothers, with promising results. In this project, I applied a similar methodology to the game Wolfenstein 3D, a game with larger, more complex levels. In addition to the LSTM, I used a Markov chain, multi-layer perceptron neural network, and convolutional neural network to implement game level generators and create sets of 50 levels for each experimental model. These levels were then compared to the game's original levels using various criteria and then ranked with a simple scoring mechanism. The rankings show that, based on the criteria, Markov chains had the best performance.

Additional Key Words and Phrases: Procedural Generation, Recurrent Neural Networks, Convolutional Neural Networks, Vdeo Games, Level Generation, Markov Chains, Wolfenstein 3D

ACM Reference format:
Eric Alzheimer. 2017. Learning Based Video Game Level Generation. 1, 1, Article 1 (April 2017), 10 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many games rely on procedural generation, the use of algorithms to generate content, as a way of constructing game levels. Some examples include Minecraft, The Binding of Isaac, Nethack, and many more. These procedural generation techniques usually require a game designer to pick the algorithms used and hand tune them to produce quality levels. For example, a game may generate terrain by using Perlin noise and a filter to produce a height map. The parameters of the noise generator and filter will have to be hand tuned until the terrain generated meets the game designer's requirements. This is often a tedious or confusing process.

Through machine learning, an alternative method of procedural generation can be used: train a machine learning algorithm on a collection of sample data to have it learn what the designer wants. This way, a designer need only create a collection of levels that contain the rules and designs they want and then an algorithm can be learned to produce more levels.

In "Super Mario as a String: Platformer Level Generation Via LSTMs", Summerville and Mateas[14] introduce a machine learning method for generating levels for the game Super Mario Brothers using LSTMs. Each level is a 2D array of "tiles" (effectively, integers). Some of these tiles represent solid bricks that the player can walk on, while others represent game objects like pipes, coins, and enemies. The player's goal is to travel from the start of the level (on the far left), to the end (the far right) without getting hit by any enemies or falling into pits.

An LSTM is trained to predict the next tile given the current one, using the levels from the original game as data. By feeding its output back into itself and filling a level with the output tiles, it will produce a sequence of tiles constituting an entire level.

In this work, I applied a number of machine learning techniques, including LSTM, to a different game: Wolfenstein 3D. Wolfenstein 3D is a first person perspective game, but uses a 2 dimensional top-down map comprised of tiles. Like Super Mario Brothers, the player's goal is to travel from the start location to the end location. Like the blocks in Super Mario Brothers, the player navigates a maze of rooms made from walls and doors, with enemies placed in the rooms. Unlike Super Mario Brothers, where levels are traversed by the player in a left-to-right manner, the player's path through a Wolfenstein 3D level can be arbitrary and maze-like.

In addition to constructing a level generator using an LSTM architecture, I constructed generators using Markov chains, multi-layer perceptron neural networks, and convolutional networks. The ouput of all four methods was compared using various criteria to determine their effectiveness.

2 RELATED WORK

Methods of procedural content generation fall into roughly three categories: search-based, learning-based, and tiling[13].

Tiling based approaches, which are among the oldest content generation methods, use algorithms that construct a full level or map from smaller parts called tiles (not the same as the level tiles described above), and focus on methods and rules for how to arrange and assemble these tiles. For example, Compton et. al.[3] create levels using an initial collection of "patterns" and construct new levels by representing the rules for combining the patterns as a context free grammar.

In search based approaches, a state space of potential maps is defined and searched, using some fitness function to evaluate each map. For example, a genetic algorithm may be used to mutate a map by changing some tiles; then, a fitness function grades the maps to produce a new generation. A survey of search based methods can be found in Togelius[15].

My method falls into the category of learning based content generation. A learning-based algorithm uses existing data to learn patterns that are then used in content generation[13]. Learning based approaches often use character representations of game levels, allowing methods of character or word prediction to be used to generate the levels.

The earliest attempts at machine learning based game level generation were primarily based on Markov chains, with two major approaches. In the first approach, the Markov chain models the transition probability from any one tile type to

2017. XXXX-XXXX/2017/4-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

another. Snodgrass and Ontañon[13] used a Markov chain to predict single tiles given a configurable choice of preceding tiles - both horzontal and vertical. An issue with single tile Markov chains is that they often fail to enforce important design constraints - for example, in Super Mario Brother's it may place ground blocks up in the sky. Later work by Snodgrass and Ontañon (in [12]) attempts to fix these issues by using clustering to learn tile groups first and then predicting groups of tiles with multi-dimensional Markov chains.

To work around this issue, another approach predicts not single tiles but instead entire columns of tiles. The Markov chain in this case provides the probability of a certain tile of columns given the previous columns. This is analagous to an n-gram model in a word prediction problem, where each column of tiles is effectively a word (and the tiles characters). This method was first used by Dahlskog et. al.[4], This method produces much better results than single tile Markov chains, but no longer allows the generation of new vertical arrangements of tiles.

There are two prior examples of neural network based approaches. Hoover et. al.[7], represented a Super Mario Brothers game level as a series of vectors, called voices. There is one voice for each tile type which indicates, for each column, what height a tile of that type is placed at. Neural networks were trained to predict the value of a single voice at column t, given the values of all other voices at that column and the previous columns.

In a paper by Summerville and Mataes[14], upon which this paper's LSTM architecure is based, single tiles are predicted using an LSTM. An LSTM is a form of recurrent neural network - a neural network with internal feedback, effectively providing memory. This allows the prediction performed in one step to be based on knowledge from any previous prediction. LSTM, put forth by [6] is a variant of recurrent neural networks that eliminates the vanishing gradient problem through the addition of extra nodes.

In addition to training the LSTM, Summerville and Mateas also include a modification to the training data to improve the LSTM's ability to generate valid levels. Prior to training, special tiles are added to each input level to mark the traversable path the player can take to get from the start to the finish. By adding this path and having the LSTM learn and generate this, they manage to significantly increase the number of completable levels over previous methods. (A level is completable if a player can actually traverse the level from the start to the finish).

3 PROBLEM (LEVEL GENERATION)

A Wolfenstein 3D level consists of a 2D array of tile numbers, size 64×64. The original game contains a very large number of tile types, most of which are used to specify graphics and decorations; they do not define the level geometry. To simplify the data and focus soley on the generation of good level geometry (that is, the layout of the rooms in the level), I reduced the original tile set down to three tile types: floor, wall, and door.

Depending on the learning algorithm used, this data is represented in one of three ways:

- A matrix $M \in \mathbb{T}^{64\times 64}$, where $\mathbb{T}$ is the set of tile types.
- A tensor $M \in \mathbb{B}^{3\times 64\times 64}$ ($\mathbb{B}$ is the set $\{0,1\}$), where $\sum_{i=1}^{3} M_{i,j,k} = 1$, for $j = 1 \dots 64$ and $k = 1 \dots 64$. This is analogous to one-hot encoding each of the matrix entries into a size 3 vector.
- A string $M \in \Sigma^*$.

To represent the level as a string, use an alphabet $\Sigma = \mathbb{T} \cup \{\alpha, \beta\}$, where α is a character representing the end of a row, and β is a character representing the end of a game map. You then convert the matrix into a string like so:

$$m_{str} = str(M_1^T)\ \alpha\ str(M_2^T)\ \alpha \dots str(M_m^T)\ \alpha\ \beta$$

where $str(x) = x_1 x_2 \dots x_n$ - that is, the string made by concatenating each vector element. Now, each game map can be represented as a string in Σ^*, and existing methods of character prediction can be used.

In order to generate game levels, the algorithm must learn a probability distribution:

- For matrix/tensor: $P(M_{x,y} \mid \{M_{x-i,y-j} \mid i = 0 \dots N, j = 0 \dots N, i \neq 0 \vee j \neq 0\})$
- For strings: $P(M_x | M_{x-i}$ for $i = 1 \dots N)$

In other words, we want to know the probability that a location will have each tile type, given the previous tiles in the level. The parameter N determines how many previous tiles are used in the condition.

When training a learning algorithm, the input data samples consist of the previous tiles that make up the probability condition. For the matrix form, each sample will be an $(N+1)\times(N+1)$ matrix with one dummy entry where the predicted tile would be. For the tensor, you will have a $3\times(N+1)\times(N+1)$ tensor with a zero vector where the predicted tile would be. For the string, you only use the one previous character, since the LSTM architecture keeps memory of past inputs internally. Each input sample is classified as the type of the tile that follows it. For the matrix or tensor, this is the tile in location (x, y)) and for the string representation, this is simply the next character. Each algorithm is trained as a probabilistic classifier on this data.

Once you have learned a function to produce this distribution, level generation can be performed by sampling from it. Start at $(1, 1)$ and sample the distribution, and assign the result to that location. Then, do $(1, 2), (1, 3)$ and so on, filling the level in row-major order, until you reach the end. Naturally, the conditions used when sampling will use data that was generated from the distribution earlier.

When using a recurrent architecture (LSTMs), the string form of the level is used. Since the LSTM has internal memory, it does not get fed more than one previous character. To generate with it, you simply feed its output back into it as input, causing it to create a character sequence.

The goal of a good level generator is to output levels that have the same characteristics of the original levels in the training data without being simple copies of the them.

4 BACKGROUND OF LEARNING ALGORITHM'S USED

4.1 Markov Chain

A Markov chain is defined as a set of states $S = \{s_1, s_2, \dots\}$ and a probability distribution on those states $P(S_t|S_{t-1})$, representing the probability of a transition from one state to another[13]. This distribution is often represented as either a transition matrix or a graph with nodes representing states and edges defining transition probabilities between them. Basic Markov chains depend only on the previous state, but a higher order Markov chain conditions the probability on multiple prior states: $P(S_t|S_{t-1} \dots S_{t-N})$[13]. In this problem, each state is a location $M_{x,y}$ and the and probability distribution is defined as:

$$P(M_{x,y} \mid \{M_{x-i,y-j} \mid i = 0 \dots N, j = 0 \dots N, i \neq 0 \vee j \neq 0\})$$

which, as explained in the previous section, is the distribution we want to learn. N is the order of the Markov chain, and it is a user selected parameter.

Markov chains can be used as a learning algorithm by constructing the distribution from training data. The distribution is constructed via the rule:

Let $C^{(x,y)}$ be the 2D array of tiles defined by $\{M_{x-i,y-j} \mid i = 0 \dots N, j = 0 \dots N, i \neq 0 \vee j \neq 0\}$

$$P(M_{x,y} = t \mid \{M_{x-i,y-j} \mid i = 0 \dots N, j = 0 \dots N, i \neq 0 \vee j \neq 0\}) = \frac{\text{Number of times that } C^{(x,y)} \text{ is paired with } M_{x,y} = t \text{ in the data.}}{\text{Number of times that } C^{(x,y)} \text{ appears in the data.}}$$

Thus, a model can be constructed by iterating through the data and keeping a count of the following tiles found for each unique conditional array $C^{(x,y)}$ encountered. This can be done using either a hash table, or, to reduce memory usage, a trie.

4.2 Multi-layer Perceptron

In a multi-layer percepton neural network, each perceptron layer consists of a weight matrix and an activation function[2]:

$$h(x) = \theta(W^T x)$$

The input vector is x and θ is the activation function, which transforms the output vector in some way, while preserving its dimensions. Each layer receives the output of the previous layer as input, with the first layer receiving the input vector and the last layer producing outputs. By training the network using backpropagation and stochastic gradient descent, it can be configured to perform classification.

In this experiment, two activation functions are used:

- The rectified linear unit, or ReLU: $\theta_i(x) = max(0, x_i)$. The rectified linear unit zeroes all negative values in the output vector. This activation function is used in all but the final layer.
- The softmax function: $\theta_i(x) = \frac{e^{x_i}}{\sum_{n=1}^{N} e^{x_n}}$. This function produces an output vector that sums to 1 and is used to create an output that can be interpreted as a probability distribution. It is used in the final layer to perform the classification.

4.3 Convolutional Neural Networks

Convolutional networks are a form of neural network that are commonly used with image data. Convolutional neural networks include, in addition to normal fully connected layers as described above, a number of convolutional layers that are used to perform feature extraction. A convolutional layer learns a small set of weights called a kernel, which is applied as a convolution to the input data to produce a smaller dimension output[10]. For example, if you convolve a 3×3 kernel with a 20×20 image, the result is an 18×18 dimension image.

Also used are max pooling layers. A max pooling layer takes a region of an image (for example, every 2×2 region), and reduces that region to a single sample by taking the maximum value in the region. For example, if you take a 24×24 image and pass it through a 2×2 max pooling layer, the output will be 12×12.

4.4 LSTM

An LSTM network is a form of recurrent neural network. A recurrent neural network (RNN) is a form of multilayer perceptron neural network where the output of a hidden layer is fed back as part of its input to itself. That is, instead of the update function:

$$h(x) = \theta(W^T x)$$

you have the recurrence relation:

$$h_t = \theta(W^T x_t + U^T h_{t-1})$$

where U is a weight matrix like W and h_t is the output of iteration number t.

From this recursive definition, it is clear that this architecture allows the output at time t to be comprised of information from all previous inputs $x_t, x_{t-1}, x_{t-2} \dots$. However, this presents a new issue - the vanishing gradient problem. Because of the repeated multiplication of each h_i, each of which are in the range $(0, 1)$, the gradient approaches zero. See for example figure 1 from [1], which shows the effect of repeatedly nested sigmoids. This vanishing gradient problem prevents gradient descent methods from being used to train the network effectively.

The LSTM architecture provides a solution to the vanishing gradient problem. In an LSTM, each perceptron is replaced with a collection of nodes to form a memory cell, as in figure 2 (from [14]). The bottom-most nodes all receive the normal and recurrent input (x_t and h_{t-1}). The bottom-left most node is considered to be the normal input into the cell. Its output is multiplied by the node to its right, which is typically called the input gate, since it decides whether to allow the input through or not. The sum in the center is the memory, since its input includes both the input of the whole cell and itself. The third-leftmost node at the bottom is the forget gate,

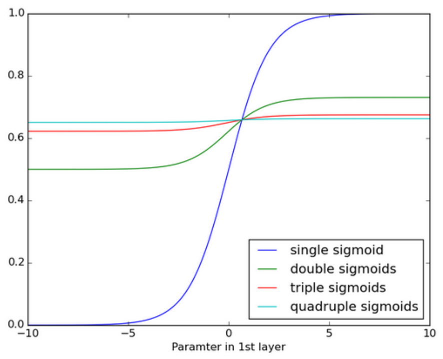


Fig. 1. The effect of continually nesting sigmoids. Notice the slope approaching zero.

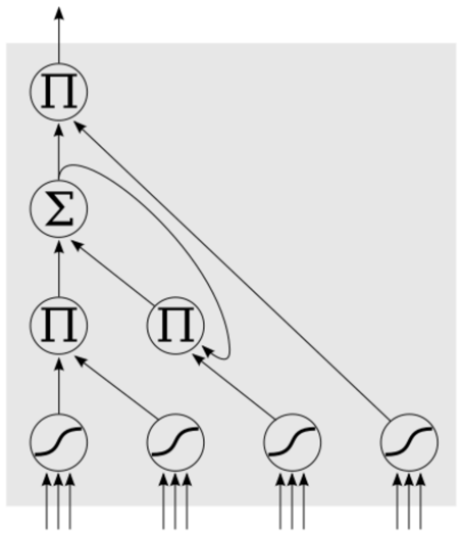


Fig. 2. Depiction of a single LSTM cell.

since its output is multiplied by the memory feedback. Lastly, the bottom rightmost node is the output gate, whose output is multiplied by the output of the memory to produce the output of the cell.

The key to the LSTMs effectiveness is in the summation node. Unlike normal RNNs, the subsequent cell state is determined by a sum of the input and previous state, instead of the product, avoiding the vanishing gradient problem.

5 EXPERIMENTAL RESULTS

5.1 Data Set

As mentioned previously, I generated levels for the game Wolfenstein 3D. So, as training data, the original game levels were used. The original game's data is in a cumbersome, poorly documented binary format (as was necessary for space and performance reasons at the time of the game's release). To obtain the level data, I loaded the game data into the Havoc Wolf3D level editor software[5] and exported it in the well documented WAD file format. The WAD data was then converted by a custom program into a text representation of the game levels.

The original game has a very large number of tile types. There are about 96 meaningfully unique types of tiles. Many of these tile types are for specifying graphical decorations, like added a flag to a wall, or placing a barrel, etc. To simplify the problem and focus solely on the generation of level geometry, the original tile types were reduced down to three types: wall, floor, and door.

Wall tiles represent a location that is solid; the player cannot travel into these tiles. Floors represent empty space where the player can move. Doors are used to interconnect rooms (spaces of floor). A door must have walls on two of its sides. The player can travel through doors, just like floors.

Each level has 64×64 tiles. From the original game, 47 levels were extracted and converted into the 3 tile text format, producing a data set with a total of 192512 training samples.

5.2 Experimental Design

5.2.1 Model Architectures. Each of the four learning algorithms were trained with a number of different parameter settings. In each case, a random 90% to 10% training/testing split was applied to the data. The architectures for each algorithm were as follows:

- **Markov chain:** The Markov chain was implemented using a hash-table. The distribution is conditioned on both the previous tiles ($C^{(x,y)}$) and the row number (y) at which the data was found. It has a single parameter, N, the order of the Markov chain, specifying the size of the conditional array $C^{(x,y)}$. If the Markov chain encounters a unique $C^{(x,y)}$ that cannot be found in the data for the distribution of order N, it then attempts to find it (using a submatrix of $C^{(x,y)}$) in the distribution of order $N-1$, and so on. If order 1 is reached without finding the condition, the uniform random distribution is used.
- **Multi-layer perceptron:** The multi-layer perceptron consists of four layers. The first three layers are each fully connected ReLU layers of size S. The final layer is a softmax layer of size 3, producing a probability distribution output. The input to the network is a tuple $(y/64, C^{(x,y)})$, where $C^{(x,y)}$ is the $3 \times (N+1) \times (N+1)$ tensor of previous tiles. Note that $\sum_{n=1}^{N} C_{n,1,1}^{(x,y)} = 0$, since that is the location of the tile to be predicted. The network was trained using stochastic gradient descent (SGD) using a negative log loss (NLL) criterion for 250 epochs. There are two configurable parameters, the layer size S, and the input order N.
- **Convolutional NN:** The convolutional network consists of 6 layers:
  - A convolution layer of 6, 3×3 kernels.
  - A max pooling layer with a region size of 2×2.

  - A convolution layer of 16, 3×3 kernels.
  - A fully connected ReLU layer of size S
  - A fully connected ReLU layer of size S
  - A softmax layer of size 3.

  In one experiment, I used a variation where the number of kernels used in the convolutional layers was doubled. The convolutional network receives the same input as the multi-layer perceptron, and likewise, it was trained with SGD and a NLL criterion for 250 epochs. There are two configurable parameters: the layer size S, and the input order N.
- **LSTM:** The LSTM architecture consists of 3 layers of size S, followed by a softmax layer. An existing software package named torch−rnn[8], based off of Andrej Karpathy's char-rnn package[9], was used to implement and train the network. As input data, it was fed the entire text file of level data. It trained using the ADAM algorithm and a NLL criterion for 2500 epochs. A drop-out rate of 0.5 was used between each layer during training. There is only one parameter, the layer size, S.

5.2.2 Experiments and Comparison Criteria. Using these models, I ran the following experiments:

- markov−3, markov−12, markov−20, Markov chains of various order sizes, (3, 12, or 20).
- lstm−256, lstm−512, lstm−1024, LSTMs of various layer sizes, (256, 512, or 1024).
- mlp−256−12, mlp−256−20, mlp−1024−12, mlp−1024−20, multi-layer perceptrons of various layer sizes (256 or 1024) and order sizes (12 or 20).
- conv−256−12, conv−256−20, conv−1024−12, conv−1024−20 , convolutional networks of various layer sizes (256 or 1024) and order sizes (12 or 20).
- mlp−longtrain−1024−20, conv−longtrain−1024−20, a multi-layer perceptron and convolutional network with layer size 1024 and input order 20, which were trained for 500 epochs instead of 250.
- bigconv−longtrain−1024−20, a convolution network with layer size 1024, input order 20, with double the number of filter kernels, trained for 500 epochs.

In each experiment, I trained the model and used it to generate 50 levels. I compared the levels to the original game levels using the following criteria:

- **E** - Percentage of the level that is walkable (that is, consists of floor or doors).
- **N** - Percentage of walkable space that belongs to the largest connected component of walkable tiles. This region will be referred to as the LWCC (Largest Walkable Connected Component) and is considered the portion of the level that the player character will navigate.
- **D** - Percentage of the LWCC that is door tiles.
- **C** - Percentage of doors in the LWCC that are correct (that is, they have walls on two opposite sides).
- **R** - The number of rooms in the LWCC.
- **S** - Rooms per tile in the LWCC.
- **V** - Tiles per room in the LWCC.
- **M** - Percentage of level that is equal to the most similar original game map.
- **L** - The level number of the most similar game map.

A level generator that performs well generates levels with criteria values similar to that of the original game levels, with the exception of criterion M. M represents how much of the level is just a copy of one of the originals. If M is 1, then the level is an exact copy of an original game level. So, M should be minimized.

To rank each model, I computed the criteria for each experiments' 50 levels, and then saved the mean and standard deviation for each experiment. Both values were rescaled into a score by comparing them to the corresponding mean and standard deviation for the original levels. The rescaling was calculated as follows:

$$\begin{aligned} P_{diff} &= |P - p_{game}| \\ p_{min} &= min(P_{diff}) \\ p_{max} &= max(P_{diff}) \\ P_{rescaled} &= \mathbf{1} - \frac{P_{diff} - p_{min}}{p_{max} - p_{min}} \end{aligned}$$

where P is a column vector of criteria results, one for each experiment. p_{game} is the criteria value for the original game, or, in the case of the mean of criterion M, 0. This rescaling makes it so that the experiment with the best results gets a score of 1, and the one with the worst results gets a score of 0. The rescaling is done for the column vector of means and for the column vector of standard deviations, for each criteria. The scores for the mean and standard deviation of each criteria were summed to produce a single final score for each experiment. Because the criteria S and V are redundant, only V was used. So, the highest score an experiment can earn is 16 (8 criteria, mean and standard deviation).

5.3 Results

5.3.1 Scoring Results. The full results for each experiment are found in table 1 (means) and table 2 (standard deviations). More interesting, however, are the bar charts of the total scores for each experiment. Figure 3 shows the scores for each experiment with regards to the mean of each criteria. Figure 4 shows the scores for each experiment with regards to the standard deviation of each criteria. Figure 5 shows the final score (combined mean and standard deviation) for each experiment. Included in the appendix are additional charts showing the mean and standard deviation for each criterion individually.

5.3.2 Discussion of Results.

Analysis of scores. The total scores show that Markov chains have the best performance, with the largest LSTM network in second place. The smaller LSTM, multi-layer perceptron, and convolutional networks all had relatively poor

Table 1. Criteria means for each experiment

Experiment Name	E	N	D	C	R	S	V	M	L
game_levels	0.337	1.000	0.024	0.956	28.521	0.020	58.594	1.000	23.500
markov-3	0.388	0.628	0.023	0.954	14.660	0.015	73.320	0.621	23.280
markov-12	0.378	0.638	0.022	0.943	15.440	0.016	71.911	0.636	23.000
markov-20	0.375	0.630	0.025	0.956	17.060	0.018	67.415	0.640	26.960
lstm-256	0.220	0.751	0.019	0.664	5.380	0.010	174.968	0.743	21.680
lstm-512	0.256	0.747	0.022	0.778	9.320	0.014	124.508	0.746	18.400
lstm-1024	0.232	0.877	0.026	0.877	15.960	0.018	70.886	0.774	18.880
mlp-256-12	0.243	0.560	0.014	0.882	5.140	0.011	124.273	0.717	20.860
mlp-256-20	0.671	0.723	0.007	0.837	4.020	0.002	668.832	0.596	17.320
mlp-1024-12	0.291	0.452	0.007	0.938	3.620	0.008	185.943	0.700	24.820
mlp-1024-20	0.130	0.504	0.007	0.940	1.840	0.011	185.411	0.777	20.520
mlp-longtrain-1024-20	0.345	0.609	0.006	0.865	3.120	0.004	371.693	0.674	23.680
conv-256-12	0.608	0.898	0.009	0.890	6.600	0.003	402.154	0.596	17.960
conv-256-20	0.329	0.508	0.019	0.788	6.520	0.010	123.641	0.668	24.320
conv-1024-12	0.252	0.451	0.011	0.835	3.820	0.010	142.937	0.708	20.580
conv-1024-20	0.268	0.453	0.009	0.737	2.380	0.006	272.442	0.691	21.040
conv-longtrain-1024-20	0.251	0.479	0.006	0.806	2.580	0.006	244.392	0.713	24.600
bigconv-longtrain-1024-20	0.192	0.412	0.007	0.925	2.180	0.008	198.652	0.739	21.520

Table 2. Criteria standard deviations for each experiment

Experiment Name	E	N	D	C	R	S	V	M	L
game_levels	0.133	0.000	0.013	0.060	28.737	0.013	24.959	0.000	13.853
markov-3	0.044	0.191	0.006	0.058	6.101	0.006	28.582	0.032	11.437
markov-12	0.065	0.205	0.007	0.072	7.441	0.006	32.402	0.046	11.841
markov-20	0.074	0.204	0.009	0.048	10.077	0.008	31.385	0.056	11.148
lstm-256	0.129	0.236	0.011	0.243	3.914	0.007	183.075	0.071	10.563
lstm-512	0.122	0.235	0.011	0.195	8.308	0.011	126.581	0.063	10.679
lstm-1024	0.129	0.166	0.012	0.157	12.522	0.008	53.466	0.075	9.397
mlp-256-12	0.084	0.169	0.009	0.213	2.885	0.007	66.162	0.046	8.630
mlp-256-20	0.121	0.187	0.006	0.179	2.328	0.001	446.233	0.029	17.484
mlp-1024-12	0.092	0.193	0.006	0.134	3.013	0.005	105.042	0.049	10.262
mlp-1024-20	0.085	0.150	0.009	0.215	1.302	0.011	145.391	0.038	8.003
mlp-longtrain-1024-20	0.115	0.193	0.004	0.220	1.657	0.002	415.943	0.048	10.924
conv-256-12	0.081	0.141	0.003	0.079	2.757	0.001	204.869	0.035	17.070
conv-256-20	0.059	0.192	0.006	0.156	3.780	0.004	73.301	0.041	10.177
conv-1024-12	0.089	0.199	0.007	0.203	1.808	0.006	110.189	0.051	9.018
conv-1024-20	0.062	0.182	0.008	0.333	1.482	0.004	195.742	0.039	7.605
conv-longtrain-1024-20	0.068	0.167	0.006	0.256	1.650	0.004	143.662	0.045	9.263
bigconv-longtrain-1024-20	0.076	0.152	0.008	0.171	1.410	0.005	159.995	0.044	9.100

performance. The best of those, in third place, was the the experiment conv−256−20, which, interestingly, means that the smaller convolutional network outperformed all of the larger configurations as well as smaller LSTMs.

Looking at the indvidual criteria provides some more insight. The LSTM was able to ensure that a larger percentage of its walkable space was in the largest component (that is, it kept the level well connected), but generated levels with much smaller amounts of walkable space compared to the Markov chains. As a result, the largest walkable spaces of each were of similar size.

The Markov chains are far better than any other model at ensuring that doors are placed correctly, with C values very close to the original game. In fact, the markov−20 experiment tied with the game levels. Other models with good C values obtained a good score simply because they generated an abnormally small number of doors.

The LSTM had the highest value of M, implying that it did the most direct copying. This is supported by additional observations explained below.

Qualitative Observations. In addition to the scores, manual inspection of the generated levels can reveal some insights.

Fig. 5. Total combined score for each experiment.

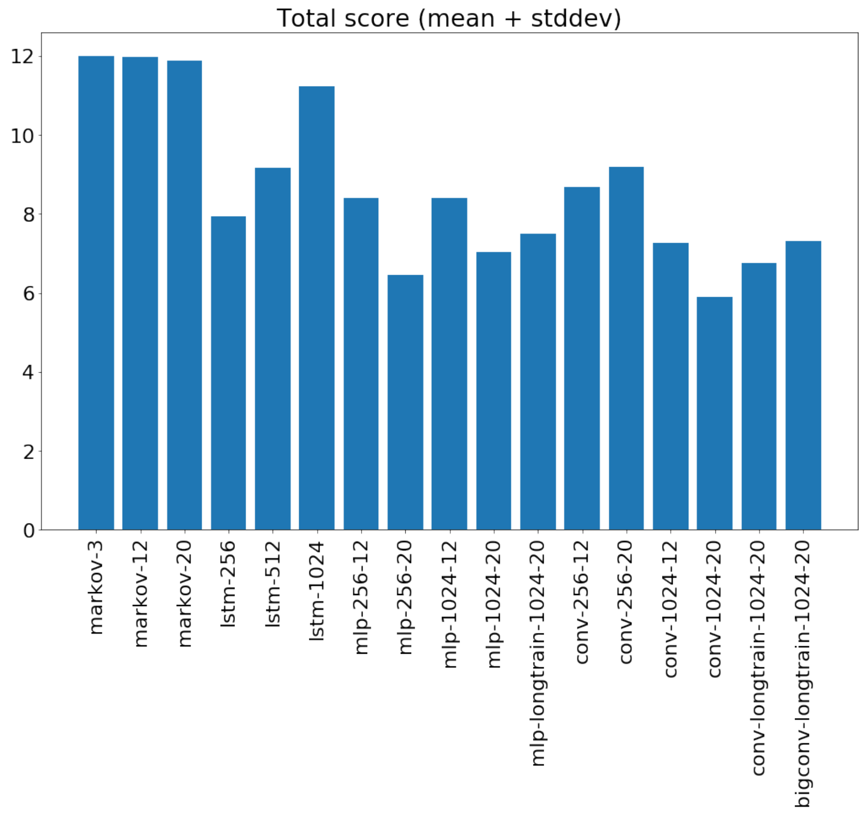


Fig. 3. Total score for each experiment when comparing the mean with the orignal levels.

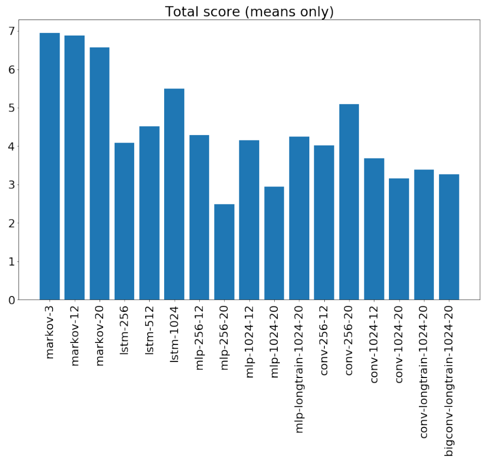
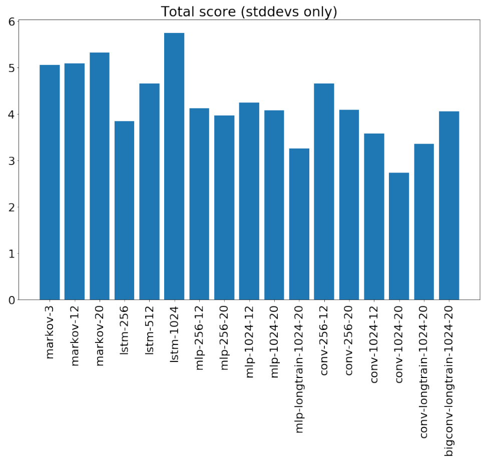


Fig. 4. Total score for each experiment when comparing the standard deviation with the orignal levels.



The LSTM's had a clear trade-off between stability and overfitting. With small amounts of training, the LSTM outputs almost entirely garbage data - its text output could not be interpreted meaningfully as a level. By training the model further, it eventually reaches a point where properly formatted levels are output; however, by the time it reaches this point, the levels generated tend to be nearly exact copies of original game levels. This is true for each LSTM size. Because it tends to shift the position of some things in the level a bit, the LSTM experiments have an artificially low M criterion value (yet it is still the largest!). I attempted to reduce this overfitting by introducing the use of dropout in the training process, with no success. In each model I trained, by the time the model learned how to output syntactically correct levels, it was already copying large portions. The main difference between the layer sizes for LSTM was in how well it could generate well structured walls. The smaller sizes produce similar output to the larger ones, but with walls that do not stay flat, and doors that are placed incorrectly.

With the exception of conv−256−20, all of the non-recurrent networks produced extremely simplistic levels, with only a small number of large, simple, rectangular rooms. In contrast, the conv−256−20 experiment acheived results similar in style to that of the LSTMs and Markov chains.

Other Observations. Even after training, the LSTM models would often output garbage data in between the production of valid level data. I was able to obtain 50 levels from the noise by writing a program to continuously generate text with the model and extract any levels that it finds. The LSTM ended up taking a very long time to train (hours) and to generate with (minutes). This is expecially significant in comparison to the Markov chains, which train in about a minute and generate levels in a few seconds.

6 FUTURE WORK AND CONCLUSIONS

6.1 Future Work

I focused on generating only the level geometry and placed only simple wall and floor tiles. To move toward a more complete level generation, future work may involve including the placement of enemies, decorations, and more. For example, a second Markov chain could be trained and used to place enemies onto a map generated by this paper's level geometry generator.

Of the non-recurrent architectures, the smaller convolutional network had the best performance. This suggests that further research in this area may be promising. The convolutional network architecture used in this project was a small variation on a example model used on 32×32 MNIST data. Someone with more experience in convolutional network design will likely be able to acheive even better performance.

Another possibility for future work is in aleviating the overfitting issues of the LSTM. I hypothesize that the large training time and network size needed to get good output syntax from the character based model is leading to the overfitting. So instead, an LSTM architecture that uses the matrix or tensor formats, like the other models, wherin the layout and formatting of the level data is not the LSTM's responsibility, should be implemented. Without the need to learn syntax in addition to the properties of the level design, the LSTM should require less training time and fewer nodes, and therefore, avoid the overfitting that is occurring now.

Lastly, the scoring methodology used is rather simplistic. If further work is done, a method to weigh criterion based on their design importance should be used to produce more meaningful scores for the generated levels.

6.2 Conclusions

This project provides a comparison of the performance of multiple machine learning algorithms for level generation. I chose to use Wolfenstein 3D as the game to generate levels for, and it provided a training set of 47 64×64 levels. A number of variations of Markov chains, LSTMs, multi-layer perceptrons, and convolutional networks were trained and used to generate 50 levels. Each experiment's output was compared to the original game levels using numerous criteria which were then used to construct a combined score for each each one. From the results, Markov chains had the best performance; however, the LSTM and convolutional networks also had promising results. This suggests improvements in the LSTM or convolutional architectures as possible future work.

REFERENCES

[1] A Beginner's Guide to Recurrent Networks and LSTMs, (N. D.). Retreived March 20, 2017, from: https://deeplearning4j.org/lstm

[2] Abu-Mostafa Y., Magdon-Ismail M. and Lin, H. e-Chapter 7: "Neural networks,", in *Learning From Data*, 2012. Retreived April 29, 2017, from: Amlbook.com.

[3] Compton K. and Mateas, M, Procedural Level Design for Platform Games. in *Papers from the 2006 AIIDE Workshop*, (2006).

[4] Dahlskog, Togelius, and Nelson, Linear levels through n-grams. in *Proceedings of the 18th International Academic MindTrek Conference*, (2014).

[5] Havoc's Wolf3D Editor. Retreived April 29, 2017, from: http://hwolf3d.dugtrio17.com/index.php?section=hwe.

[6] Hochreiter, S. and Schmidhuber, J. "Long short-term memory" *Neural Computing* 1997.

[7] Hoover, A. K., Togelius, J., and Yannakakis, G. N, Composing video game levels with music metaphors through functional scaffolding. in *Proceedings of the ICCC Workshop on Computational Creativity and Games*, (2015).

[8] Johnson, J. C., torch-rnn: Efficient, reusable RNNs and LSTMs for torch. (2016). Retreived April 29, 2017, from: https://github.com/jcjohnson/torch-rnn.

[9] Karpathy, A., char-rnn: Multi-layer Recurrent Neural Networks for character-level language models in Torch. (2015). Retreived February 1, 2017, from: https://github.com/karpathy/char-rnn.

[10] Karpathy, A., CS231n Convolutional Neural Networks. (N. D.). Retreived April 29, 2017 from: http://cs231n.github.io/convolutional-networks/

[11] Karpathy, A, The Unreasonable Effectiveness of Recurrent Neural Networks. (2015). Retrieved February 1, 2017, from: http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[12] Snodgrass, S. and Ontañon, S, A hierarchical mdmc approach to 2d video game map generation. in *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, (2015).

[13] Snodgrass, S. and Ontañon, S, Generating Maps Using Markov Chains. in *Artifical Intelligence and Game Aesthetics: Papers from the 2013 AIIDE Workshop*, (2013).

[14] Summerville, A. and Mateas, M, Super Mario as a String: Platformer Level Generation Via LSTMs. (2016). Retrived April 29, 2017, from: arXiv:1603.00930v2

[15] Togelius, J., Yannakakis, G. N., Stanley, K. O. and Browne, C., Search-Based Procedural Content Generation: A Taxonomy and Survey. in *IEEE Transactions on Computational Intelligence and AI in Games*, (2011), 3 (3), 172-186. doi: 10.1109/TCIAIG.2011.2148116

APPENDIX A ADDITIONAL FIGURES

Fig. 6. Criterion E

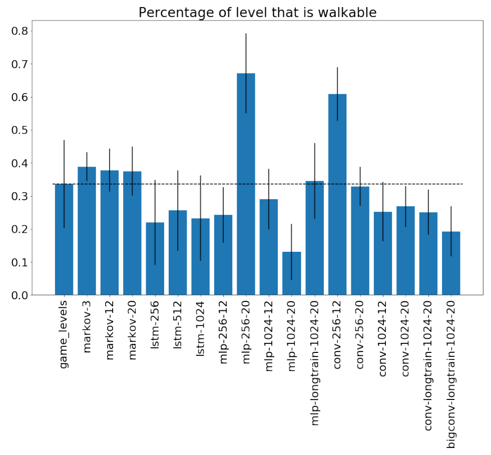


Fig. 9. Criterion C

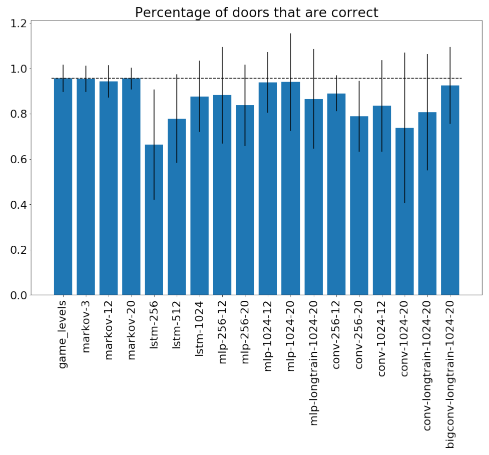


Fig. 7. Criterion N

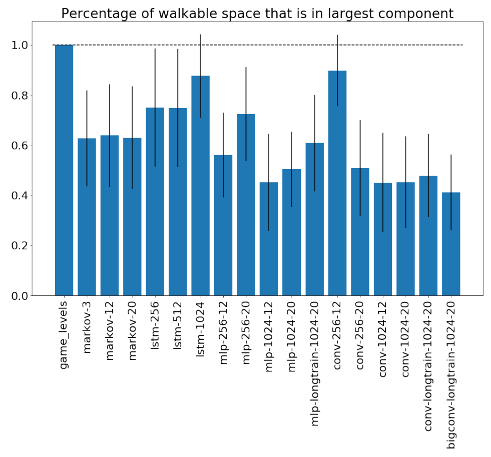


Fig. 10. Criterion R

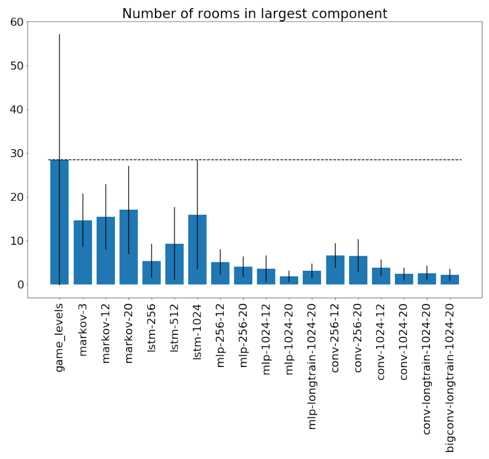


Fig. 8. Criterion D

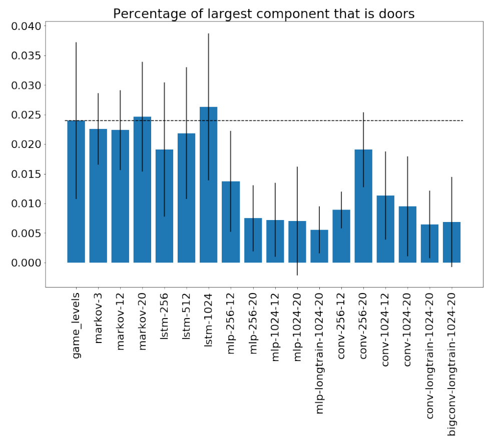


Fig. 11. Criterion S

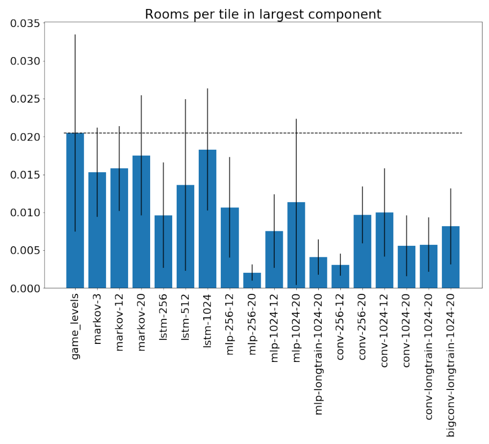


Fig. 12. Criterion V

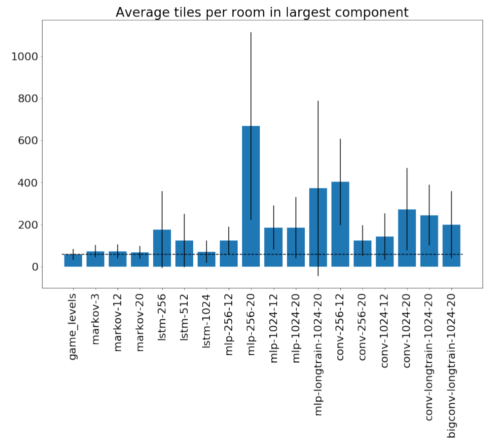


Fig. 13. Criterion M

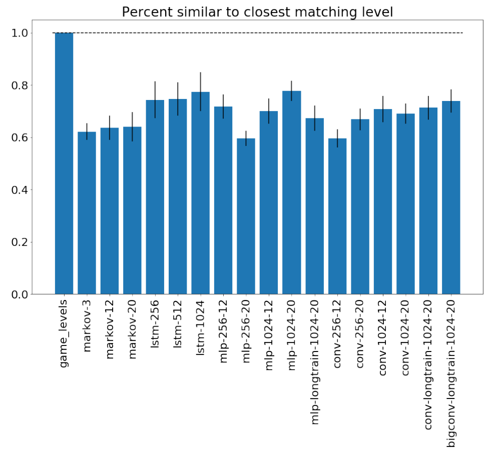
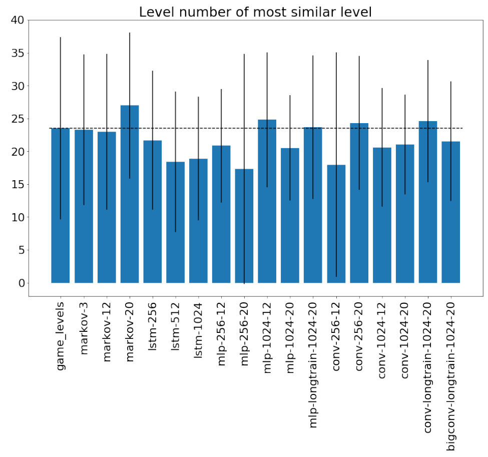


Fig. 14. Criterion L


Level number of most similar level
game_levels
markov-3
markov-12
markov-20
lstm-256
lstm-512
lstm-1024
mlp-256-12
mlp-256-20
mlp-1024-12
mlp-1024-20
mlp-longtrain-1024-20
conv-256-12
conv-256-20
conv-1024-12
conv-1024-20
conv-longtrain-1024-20
bigconv-longtrain-1024-20