Neural Wave Function Collapse: Generating Maps for Roguelike Games

James Garofolo and Jordan Lees
Henry M. Rowan College of Engineering
Rowan University
December 15, 2022

***Abstract*—We present a method of integrating neural networks with the image generation algorithm known as Wave Function Collapse. We use this method to procedurally generate video game maps in the style of the original Legend of Zelda. We also illustrate that, while subject to occasional errors, this algorithm's support of hard-coded rules allows for errors to be corrected during generation.**

I. Introduction

A. Roguelike Games

To begin, we will introduce the concept of *roguelike* games. *Rogue* was a video game that originally came out in 1980 for UNIX systems. The player would play a character exploring through procedurally generated levels of a dungeon. The original version of the game displayed its levels and characters purely as text in a terminal screen [1], [2]. The levels were simplistic because of the limited medium, but it became quite popular and inspired plenty of games. Games inspired by Rogue became so frequent that it turned into an entire sub-genre of video gaming, called *roguelike*. The most relevant feature of roguelike games to this project is the fact that every level is procedurally and randomly generated, and each time a game is started, the levels that are generated vary [3]. Because of the necessity of infinitely regenerable world maps, the need for procedural generation arises when developing these games.

B. Wave Function Collapse

Wave Function Collapse (WFC) is an algorithm developed by Maxim Gumin, inspired by the phenomenon in quantum mechanics by the same name. The algorithm was originally written in C# but the concept can be applied to any programming language. It is used to randomly generate images from square tiles whose features vary on a large scale, but are similar on a small scale to the images that are input into the program. It requires a set of predefined rules, defining which types of tiles are "allowed" to be near other types of tiles [4].

Before proceeding we should clarify the technical terminology used in this paper regarding WFC.

- "Tile" refers to a square element in a grid that can take any of a set of possibilities. For example, in the original Legend of Zelda, a tile could represent a stone wall, a tree, or a walkable path, or others.
- "Map" refers to a rectangular grid of tiles with a specified width and height. For example, we may run WFC with an output map size of 16x11 tiles, the same as Legend of Zelda, or 50x50, or 20x30.
- "Overworld map" refers specifically to the complete map of the original Legend of Zelda, excluding dungeons.
- "Collapsed" refers to a tile in a map whose state has been defined. This is analogous to the event in quantum mechanics when an observation "collapses the wave function" of a particle into a defined value.
- "Unknown" refers to a tile that has not yet been collapsed.
- "Possibilities" refers to the list of possible outcomes that a given unknown tile in a map could take.
- "Entropy" refers to the *number* of possibilities an unknown tile could take.
- "Rules" is a loose term referring to how WFC determines the state of possibilities for unknown tiles given its current state of defined tiles.

The program begins by creating a grid of tiles with a specified width and height, and sets their state to all be unknown. It then repeats the following main steps until all tiles in the grid have been defined:

1) Find the tile or tiles with the lowest entropy,
2) Collapse the tile with the lowest entropy,
3) Recalculate the state of possibilities for each tile.

Gumin's original version uses rules that only define what tiles are allowed to be directly adjacent to one another, but the algorithm does not need to be that restrictive and can use any method necessary to translate a grid full of both collapsed and unknown tiles into a grid of possibilities [4].

The original technique is powerful, but requires prior knowledge of all possible configurations of which tile is allowed to go next to another, and a potential game developer may not know ahead of time how they want tiles to be arranged. Additionally, since the relationship between each tile to each other tile has to be known, the number of rules needed for WFC likely increases exponentially with the number of tiles that exist in the game. The idea behind this project is to let a game developer create a limited number of *sample* maps that they want their game levels to look like, then use a neural network to analyze their sample maps and generate rules for WFC instead of needing to hard-code the rules.

C. Neural Networks

Neural networks are universal multivariate function approximators, which should be well equipped to handle this task.

They consist of a series of linear transforms, separated by nonlinear activation functions. This combination of linear and nonlinear elements allows the networks to develop arbitrary levels of complexity, with the number of neurons in each layer and the number of layers in the network both contributing to the capabilities of such a network to learn. Increasing these parameters can assist the network in modeling more intricate functions, but they can also easily lead to over-fitting. While there are no widely accepted methods that guarantee choices for these parameters that will not over-fit, common rules of thumb are to limit the number of hidden layers to two, and to constrain the number of neurons in each hidden layer to be somewhere between the number of neurons in the previous layer and the number in the next layer. [5]

Discrete data points like tile IDs in a video game are somewhat of a challenge to represent for neural networks. Unlike continuous data, an increase in data point index has no consistent meaning for the change in the quality of the input. Additionally, inputs that are arbitrarily assigned to high indices may not warrant high amplitude outputs, forcing patterns learned by the network to be much more complex than necessary for the amount of information being represented. This challenge is often encountered in the field of natural language processing, where letters in an alphabet or words in a language are considered as discrete tokens that are operated upon based on their position in written text. A naive but functional solution to this problem is the use of one-hot vectors that encode which token from the lexicon is being represented, allowing the data to be trivially separable. However, this solution can quickly cause memory issues as the lexicon size increases, requiring anywhere from 16 to 64 bits of data to be allocated per element. A more elegant and memory efficient solution to this problem is to use dense trainable embedding vectors, which represent tokens at unique points in a hyperspace. [6] This solution is powerful, not only because it can use far less memory to represent the same amount of information, but also because it can learn to group tokens that cause similar patterns to emerge closer to one another than tokens that cause drastically different patterns.

When designing a network, the activation and loss functions are key selections to be made. A popular function for separating hidden layers is the Rectified Linear Unit, or ReLU, function. This function's computational simplicity aids in reducing inference time, and is well-suited for gradient descent, as the maximum of its derivative is 1. Output layers must have their activation functions more carefully selected, so as to provide an output that is compatible with the loss function for the model. Classification tasks tend to use cross-entropy loss, as it forces outputs toward zero for false classifications and toward one for true classifications. This loss is frequently paired with a softmax activation function, which forces the output vector to adopt the qualities of a probability distribution, as well as amplifying high values and attenuating low ones for more definable boundaries. While the binary nature of outputs is an attractive quality of cross-entropy, the amplification and attenuation caused by the softmax activation function would reduce the average entropy of the network's output, which is not ideal for applications that rely on nonzero entropy. Similar binary output qualities can be achieved by training a network using binary cross-entropy loss, which is designed for classification tasks in which a data point may belong to more than one class at once. For the network to satisfy the input requirements of this loss function, the inputs must be constrained on [0,1], but do not necessarily have to resemble a probability distribution. The sigmoid activation function satisfies this constraint for any input amplitude, while being completely data-independent, making it ideal for this high-entropy use case.

II. Methods

A. Data Preparation

To extract data for the neural network, we began with a PNG image of the entire overworld game map from the original Legend of Zelda, stitched together into one contiguous image, as well as a CSV file with the hexadecimal ID of each tile [7]. We created a script that loads both files and finds every tile that was used in the overworld, extracting a 16x16 PNG image for each type of tile for use in reconstructing maps later. Before proceeding, we needed to reconstruct our own version of the overworld map, because every version we could find online had the bottom of each 16x11-tile map cut off by a few pixels. This would cause issues with training, so after extracting the sprite for each tile ID, we used the CSV data to construct an image of the complete map without the pixel errors.

After this, we found that there were many gaps in the hexadecimal IDs of the tiles. There were several hexadecimal values in the sprite sheet that were not used in the overworld, which when translated into one-hot vectors, would lead to wasted space. In order to save memory and storage space when training the model, we manually changed the hexadecimal IDs of each tile so that there were no gaps. Since only 90 unique tiles were used in the overworld, we reduced the one-hot length of 144 (8f was the highest index in the original table [7]) to a one-hot length of 90.

The script then splits the overworld map into many small windows to create training data. For each window, it simultaneously outputs a binary Numpy file for training, and an image associated with that section of the map so we could visually confirm that the data was valid. The binary Numpy files contained an array of indexes, each index representing the numerical value of the hexadecimal IDs from earlier.

At first we split the overworld by one 16x11 window at a time, but this resulted in a total of 128 maps, which is not nearly enough to train a neural network. To obtain more data, the script was modified to traverse the full map with a varied stride, similar to 2D convolution. In fact, some of the math used came from references in how to calculate the output vector of a convolutional operation. In the end, the script was always run with a stride of 1, meaning that each window is 1 tile to the right of the previous, and once it reaches the end of a row, the next row is 1 tile below the next. The size of these windows was configurable, depending on what input size the neural network would be trained with.

In order to allow the network to handle inputs from the beginning and middle of the WFC algorithm, the network has to be trained on inputs in which not all of the tiles have been collapsed yet. To accomplish this, a series of corruptions of each input window were made, each with a varying number of tiles replaced with the index associated with unknown tiles, which was chosen to be one higher than the largest valid tile index. For the best results, the number of windows in this series should be at least equal to the number of tiles in the input window. When this requirement is satisfied, linearly varying the tile replacement probability from 0 to 1 along the sequence produces an adequate distribution of inputs for the network to be able to handle any concentration of known and unknown tiles. A consequence of this method of training is that windows with different desired outputs can be made to have the same input by replacing all of the differing tiles with unknown tiles. If this issue is left unaddressed, the network could learn to classify less common tiles as not possible in the given position because it is less likely to incur a penalty for doing so than it is for classifying them as equally possible as more likely tiles. A simple solution to this problem would be to detect these samples, and label both output tiles as desired positive outputs for the given sample. However, the algorithm for doing so grows with $O(n^2)$ time complexity, where n is the number of output maps in the algorithm. Because n can easily tend toward the hundreds of thousands for certain network shapes, locating overlaps across the entire dataset is not always feasible. To circumvent this, the networks were pre-trained without overlap detection to ensure that they have seen the entire dataset, and then fine-tuned with mini-batches of data that had overlap detection run within the confines of the mini-batch. This technique sacrifices a bit of entropy, but transitions the time complexity from $O(n^2)$ to $O(b*s^2)$ where b is the number of batches and s is the batch output size. The loss of added entropy is loosely associated with the size of s, so the model will generally perform better with it large, but it can be made as small as necessary in order to ensure that the model will finish training in a reasonable amount of time.

B. Rule-Generating Model

For its input, the model should take in a matrix of indices representative of the tiles placed at each space in the map grid, with any currently unknown tiles having the index after the largest valid index in the tile list. These indices can be converted by the model into dense embedding vectors that can be interpreted by a fully connected layer. Because the ideal dimensionality of embedding vectors with respect to the size of their lexicon is unclear, the dimensionality for this application was chosen to be the lowest number of bits necessary to represent the highest index in binary. More formally,

$$n_{dim} = ceil(log_2(max\_id)) \tag{1}$$

The intuition behind this choice is that, at its simplest, the embedding vector could represent all of the tokens separable by binary-coding them. While this is likely not how the embedding space would actually be structured, it is a strong guarantee of satisfactory complexity for the given task, and provides enough of a reduction in data dimensionality over the one-hot vector case that it is not of much worth to reduce further. Once the tiles have been embedded in hyperspace, the resulting tensor can be flattened and passed through a fully connected network for inference. To avoid overfitting, this network was chosen to have two fully connected layers, separated by a ReLU activation function, with a ratio of neurons of 2:1 from the input layer to the hidden layer. The outputs of the second fully connected layer are then constrained by a sigmoid function, reshaped into vectors of possibilities for each tile in the lexicon, and returned to be trained by a binary cross-entropy loss function. To accelerate the training process, GPUs both in personal desktop computers and on Google Colab's servers were used to perform the network inferencing calculations whenever possible. In inference time, these tile vectors would be thresholded by the mean of their elements, guaranteeing that at least one tile was deemed possible in any given location.

Several shapes of inputs and outputs were considered when designing the model. The first shape had the model take in a matrix the size of one Legend of Zelda overworld screen (16x11), and output a vector of tile possibilities for each of those tiles in the same inference, with the possibilities of collapsed tiles being ignored, and those of unknown tiles being considered. This shape was made to learn patterns based off the entirety of the contents of any 16x11 screen. The next network shape would have a defined square of inputs, chosen to be smaller than the room size, and made to infer tile possibilities for every tile for which it took in an input. The intuition behind this model was that it would learn smaller patterns from the maps, which could then be pieced together to make new larger patterns. The last network shape to be explored would have a similar input to that of the previous network shape, but would only be allowed to infer possibilities for one tile at a time. This model was intended to achieve a similar goal to the previous one, but made under the hypothesis that a field of view that exceeds the network's range of influence would allow it to make decisions that flow more elegantly off of the decisions of previous iterations of the algorithm. While the first two network shapes tended to infer rather quickly, the last required many more calculations per wave-function-collapse step, so its inferences were run on GPU hardware wherever possible.

C. Neural Network Compatibility in WFC

When creating a Wave Function Collapse implementation for this project, the goal was to make a class that works generically, meaning that different "driver programs" could use different methods to generate rules. It exposes a few functions that lets the user control how it gets its rules. The driving program can either tell it to run its next step with an explicitly stated set of new possibilities, then output what tiles it has collapsed in total; or it can specify a list of rules as callback functions which get executed in order.

The callback function gets provided the WFC's current list of collapsed tiles, and uses whatever logic it desires to output a list of possibilities for the remaining unknown tiles. The purpose of this technique is that it allows a game designer to add hard-coded rules *in addition* to those generated from the neural network. For example, they may wish to require doors in a certain area or require walkable paths to be a certain width. We did not write any custom hard-coded rules for the project, but the capability is present.

As the WFC instance loops through its list of rules, it uses a boolean AND operation between its previous list of possibilities and the new list of possibilities, allowing subsequent rules to act like a 'filter' on previous rules. However, in the case where a rule causes a tile to have zero entropy, it will disregard the last rule because then there would be some tiles that could not be defined. By using custom rules, small errors could be fixed with additional rules set up by the developer, such as random blocks in a location that does not make sense.

III. Results

One pattern that was universal to all of the architectures tested was the fact that the networks generalized very well to data they had not seen. Tile possibility vector accuracies were commonly in the mid-to-high 90% range. While it is possible that the network simply learned that a majority of tiles are almost never used, the networks were shown to be able to use quite a few of the more rare tiles in situations similar to those in which they were seen in the game. This is a result that one would expect not to occur if zero-bias were affecting the evaluation accuracy. While tile vector accuracy is not necessarily indicative of fitness for the given application, it is a promising result in terms of the architecture's ability to learn the data.

Assessing the model's fitness to generate new Legend of Zelda maps is a more qualitative process. To do so, each of the models was used as the only rule in a WFC algorithm, and a series of maps was generated and inspected for similarity to the game, logicality of block placement, and traversability in the game environment. The first model to be tested was the model that processed all tiles in the map at once, a result of which is shown in Fig. 1.

The map in this figure was generated by the model but is exactly equal to one present in The Legend of Zelda. Subsequent generations show a similar pattern, with differences being constrained to single tiles worth of information, almost always in ways that are illogical for tiles to be placed in the game, like having two different color palettes of floor or wall. While this is promising data in terms of a model's ability to learn this kind of information, it is not ideal for the application of generating new content for a game. These data lead to the idea to create and test the model that attended to sections of a map at a time, a result of which is shown in Fig. 2.

This map was generated on a 20x20 tile grid, because the training method for sliding windows is not conducive to confining the map generation to the 16x11 rooms seen in The Legend of Zelda. To make it, the model's 7x7 input window was convolved across the map with a stride of 3 in each direction, and its output was stamped out on the tiles to which it was attending. Because the model's area of attention completely encapsulated its area of effect, it was unable to take into account the context into which it was generating a pattern. As a result, the model produced small squares of generally coherent output, which do not tile together well at all. These data lead to the idea of reducing the area of influence of the model to a single block, while leaving the area of attention alone. The results of this model shape are shown in Fig. 3.

This map was generated in a similar way to the previous one, with the single output nature necessitating a stride of 1. Any input tiles that exceeded the dimension of the map were replaced with immutable unknown tiles to maintain the input shape of the model. The wave-function collapse algorithm was also allowed to collapse as many as 10 higher entropy tiles at a time in the beginning of generation, as a way of automating the addition of variety to the maps. The figure shows the model to be much more capable of generating naturally flowing shapes and structures in the style of those seen in the game, while

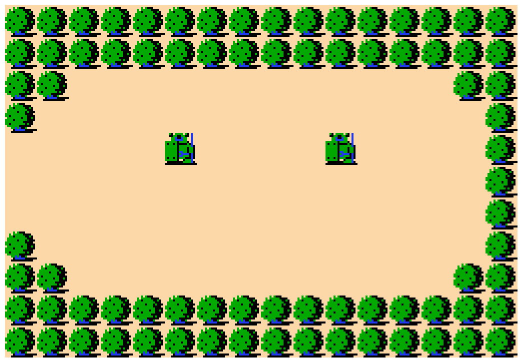
Fig. 1. Generation made by all-to-all network.

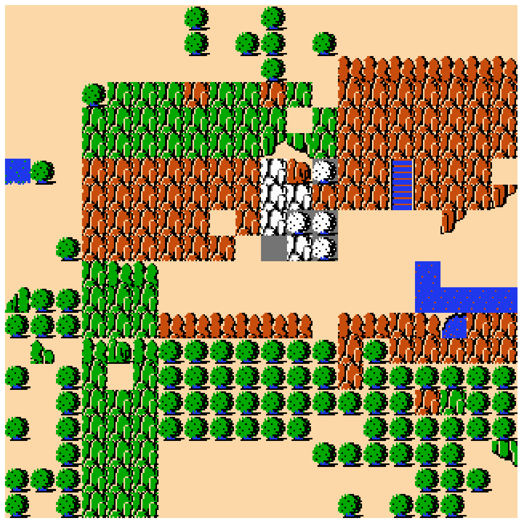
Fig. 2. Generation made by section-to-section network.

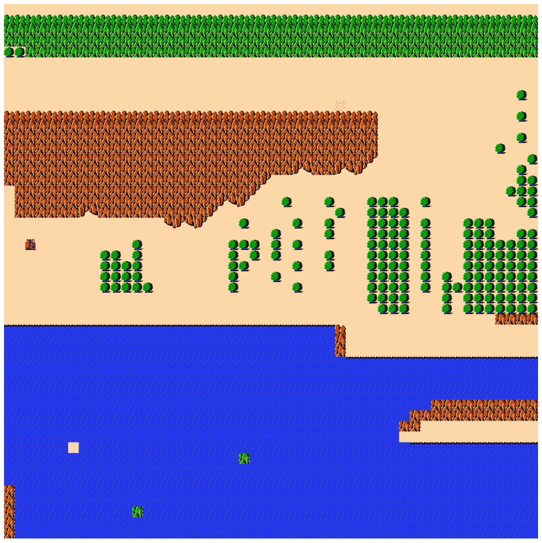

Fig. 3. Generation made by section-to-one network.

not being confined to reproducing exact patterns from the training set. While the single-output-window model produces promising results, its behaviors are not ideal. It tends to suffer from mode collapse if artificial variety is not introduced, and is very capable of filling an entire map with wall tiles. While this problem could be mitigated by the insertion of key features in the map prior to generation, this requires excess effort that the developer would ideally not need to put in. Adding artificial variety by increasing the collapse limit is a semi-viable solution, but it tends to cause single tiles that make no sense in the given context, like the sand and wall blocks in the middle of the ocean in Fig. 3. These errors can be stopped proactively or corrected retroactively by hard-coded rules, but once again, the ideal algorithm would not require this of developers. It was hypothesized that a larger input window would combat mode collapse, as the dataset would contain less samples that are entirely made of one tile, but experimental results quickly disproved this hypothesis.

IV. Future Work

Map generation from this technique is promising, but will require refinement before it can be used in a proper game. To reduce the single-tile noise, the WFC algorithm could be made to keep track of tiles that were collapsed from high-entropy, and re-evaluate what they should be, in case they no longer fit in the complete map. Additionally, the model could be fine-tuned to generate one or multiple room sizes, and then use that fine-tuning to build a map room-by-room, rather than all at once. This may lead to an increase in variety, and a similar connected-segments feel to that of the original game. Additionally, there are a number of possible ways to combat mode collapse. Multiple networks of different shapes could be used to provide different insights into the patterns present in the game. These networks could also be trained on more selectively pruned datasets, and with a loss function that promotes entropy by punishing false negatives more heavily than it punishes false positives. Lastly, a pseudo discriminator model could be made to assess fitness of the model's decisions and help fine-tune its generations using reinforcement learning.

V. Conclusion

We presented a new method of procedurally generating video game maps, by integrating neural networks with the image generating algorithm Wave Function Collapse. Our first method led to outputs that seemed overfit to the training dataset. Our second method led to more varied results, but the segmented nature of the output was incompatible with a coherent video game map. Our third method led to the most convincing looking maps, in which the network gets fed a square matrix of tiles and only predicts one at a time. It still suffers from occasional noise and tends to make large swaths of walls or ocean, but the results are promising and do appear to mimic the style of the Legend of Zelda game map. With some refinement, this method could be successfully integrated into a roguelike game's map generation.

References

[1] P. Kuittinen, "Rogue - exploring the dungeons of doom (1980)," 6 2001.
[2] E. Staff, "The making of: Rogue," 7 2009.
[3] C. Dotson, "What is a roguelike? the beginner's guide," May 2020.
[4] M. Gumin, "Wave Function Collapse Algorithm," 9 2016.
[5] "155 - how many hidden layers and neurons do you need in your artificial neural network?," Sep 2020.
[6] "Embedding¶."
[7] A. Sweigart, "8-bit nes legend of zelda map data," Dec 2012.

Appendices

Source code on GitHub: https://github.com/James-Garofolo/neural_wfc