An Exploration Of Improving Evolutionary Search With Deep Learning For Procedurally Generated Video Game Levels

School of Computer Science & Applied Mathematics
University of the Witwatersrand

Lior Becker
School of Computer Science & Applied Mathematics
University of the Witwatersrand
Johannesburg, South Africa
2333263@students.wits.ac.za

Dr Steven James
School of Computer Science & Applied Mathematics
University of the Witwatersrand
Johannesburg, South Africa
steven.james@wits.ac.za

***Abstract*—Video games are expensive and difficult to make, due to the developers being required to make every piece of content within the game. Procedural content generation (PCG) can be used to overcome these challenges, as procedural content generation is an algorithmic means of producing content, which can be done significantly faster than creating content manually. While there are traditional algorithms that are frequently used for procedural content generation, machine learning approaches are an attractive alternative, as they have potential to create high quality content. While there have been several different approaches PCG using machine learning, they all have draw backs, such as being too slow or requiring too much training data. Therefore, we propose that by combining several models, we are able to mitigate the drawbacks, resulting in high quality levels being generated fairly quickly. We have used a genetic algorithm (GA) to generate high quality levels. These levels were then used as a data set for training a deep neural network (DNN), which resulted in models that were unable to create levels to the same standard as the GA, but showed potential for future investigation.**
[1]

I. Introduction

The development of video games is both expensive [Rayna and Striukova 2014] and time consuming. Higher budget video games often cost over a million dollars to develop [de Pontes and Gomes 2020] and take several years to complete, due to the fact that every piece of content is created by the developer. Procedural content generation (PCG) is a method of overcoming these challenges, by generating content algorithmically [Togelius *et al.* 2011]. PCG methods allow for content to be generated faster using fewer resources.

While there are algorithm based methods, such as using Perlin-noise for generating terrain [Rose and Bakaoukas 2016], this paper will be focusing on a machine learning approach to PCG for video game levels. The reason for this is that machine learning approaches have the ability to create diverse and high quality levels [Muir and James 2022]. This will allow game developers to create superior levels, while only taking a fraction of the amount of time.

The goal of any level designer is to create diverse and fun levels, which can be difficult for procedurally generated levels to achieve. While genetic algorithms are capable of generating diverse levels, the generating process can be time consuming [Muir and James 2022]. Alternatively, trained reinforcement learning agents can generate levels quickly by iteratively improving on randomly generated levels [Khalifa *et al.* 2020; Muir and James 2022]. However, training these agents takes a significant amount of time, and requires a reward function that is specifically designed for the type of game the agent is learning to make [Muir and James 2022]. This paper will therefore be attempting to overcome the limitations of both genetic algorithms and reinforcement learning, by using deep learning and deep reinforcement learning in order to create an agent that can mimic the output of the genetic algorithm in a fraction of the time.

This report is structured as follows: Section II provides essential background information such as how the various models and methods are trained and used. Section III discusses and analyses related work in the field. Section IV details the structure of each model and how they will be tested. Section V provides an in-depth discussion and analysis of the results collected. Section VI lays out the draw backs and limitations of the research that has been performed, while section VII provides potential solutions to these issues and ideas as to what needs to be researched. Section VIII concludes this document and provides a summary of the main points in this paper.

II. Background

A. Procedural Content Generation

Procedural content generation in video games is an algorithmic means of creating video game content [Togelius

[1]The full implementation of the models can be found on https://github.com/2333263/Deep-Learning-PCG

et al. 2011]. Togelius *et al.* [2011] also proposes rules that an algorithm must follow to be considered PCG, the most important of these rules is that it may not take players actions into account when the generation is occurring. Togelius *et al.* [2011] further states that random generation can be considered PCG if there are rules put in place to ensure the content is not truly random.

B. Genetic Algorithms

Genetic algorithms (GA) are currently a very popular method of procedural content generation. A GA is a method of machine learning that was inspired by Charles Darwin's theory of natural evolution [Beasley *et al.* 1993]. Potential solutions or outputs of this algorithm are called individuals [Muir and James 2022]. Individuals encode their solution in a structure known as a gene [Beasley *et al.* 1993]. In a given iteration of this algorithm, there are many individuals, together these individuals are called a population or a generation.

The algorithm begins by generating a population of individuals with randomly generated genes [Beasley *et al.* 1993; de Pontes and Gomes 2020]. The quality of the individuals are determined by a function known as a fitness function [Muir and James 2022]. Individuals are then chosen to be the basis of the next population. The probability of being selected is proportional to an individual's fitness [Liapis *et al.* 2014]. The selected individuals undergo a procedure called cross-over, where segments of these individuals are exchanged [de Pontes and Gomes 2020; Muir and James 2022; N. Ferreira *et al.* 2014]. Replicas of these exchanged genes are created. However, during a process known as mutation, segments of these replicated genes are randomly altered [Muir and James 2022]. These modified genes become the individuals that make up the current population, which is evaluated by the fitness function. This process is then repeated until the fitness of all individuals in a population meet a sufficiently high level, where the best performing individual of that population is chosen as an output. [Muir and James 2022]

There are several factors that can affect the quality of the output of the genetic algorithm such as, size of a population, how many genes are crossed-over, how many genes are mutated, and whether or not there is elitism. [Muir and James 2022] Elitism is when high performing individuals of the previous population are brought into the current population so that the genes of these individuals are not lost in the newer generations [de Pontes and Gomes 2020; Muir and James 2022].

C. Reinforcement Learning

An alternative and more recent approach to PCG is to use reinforcement learning (RL) agents, which takes a trial and error approach to learning [Sutton and Barto 2018]. In RL, an agent has the ability to interact with an environment [Muir and James 2022]. This environment is typically a Markov decision process $\langle S, A, P, R, \gamma \rangle$ [Khalifa *et al.* 2020; Muir and James 2022], where (i) S is a list of all possible states of the system. (ii) A is the set of actions the agent can make. (iii) P is the probability of an agent moving into a specific state after performing an action; this is known as the transition function. (iv) R is the reward function, that specifies if an agent receives a punishment or a reward for performing a given action in a specific state. (v) γ is used to discount future rewards [Muir and James 2022].

The agent then interacts with the environment. After every action it makes, it either receives a reward or a punishment. It will then use this feedback to learn a policy π [Sutton and Barto 2018]. Sutton and Barto [2018] define a policy as a mapping from the current state of the environment to the possible actions the agent can make. The agent will then use this policy as a guide when making decisions in an attempt to find the overall best actions, thus maximising the cumulative discounted reward. By doing this the agent will not only reach the goal, but also do so in the most efficient way possible. These best actions are known as an optimal policy π^* [Muir and James 2022].

D. Deep Neural Network

A very popular method of implementing machine learning algorithms is to use neural networks, more specifically deep neural networks. Traditional neural networks are comprised of layers of artificial neurons [Skansi 2018] that compute the summation of several weighted inputs. An activation function is then applied to the computed value [Skansi 2018]. This value then acts as the input values for the next layer of nodes [Skansi 2018].

A deep neural network functions similarly to a traditional neural network. However, instead of having only two layers, an input and an output layer, there are one or more hidden layers between the input and the output layer [Li 2017]. Furthermore, the output of each layer acts as the input into the next layer. The benefit of having more layers is numerous, such as the network being able to compute more complex functions [Skansi 2018], as well as having more complex layers that function differently from normal layers.

E. Deep Q networks

Deep Q Networks, also known as DQNs is a specific type of DNN designed for Deep Reinforcement Learning [Sutton and Barto 2018]. This network takes over the role of both the value function and the agent. It does this by taking the state of the world which it can then use to estimate the value of the current state. After which, the estimate is used by iterating through several more layers of the network before outputting to several nodes. These nodes represent which actions the agent can take and the output is the probability of taking said action. Thus an action can be selected by taking the argmax of the output nodes. Furthermore, these networks specialize in taking spatial information into account [Sutton and Barto 2018]. They are trained using the a standard Q-Learning algorithm, which has been modified to work with the DNN [Mnih *et al.* 2013]. The modification being that the reward from the environment is used to calculate a semi-gradient in order to back propagate though the network [Mnih *et al.* 2013]. However, training

using a single episode can cause instability [Mnih *et al.* 2013]. The solution to this instability is batching the episodes which are then sampled from during training [Mnih *et al.* 2013; Sutton and Barto 2018]. Using multiple networks can also help stability. The first network learns through back prorogation, while the second network's outputs are used to calculate the gradient for the back propagation [Mnih *et al.* 2013]. After every few iterations of training, the weights are copied over from the training network to the other network.

III. Related Work

Currently there are several different approaches to PCG, such as using Cellular Automata to efficiently generate caves [Johnson *et al.* 2010] or using various noise based techniques for terrain generation [Rose and Bakaoukas 2016]. However evolution-based approaches such as genetic algorithms are a very popular method of PCG. Several studies have used genetic algorithms [de Pontes and Gomes 2020; Muir and James 2022; N. Ferreira *et al.* 2014], however, they all took different approaches, which resulted in vastly different outcomes of the level generation. N. Ferreira *et al.* [2014] had four different populations, each of which handled a different part of the world generation, such as floor height, placement of floating blocks, placement of enemies, and placement of coins. In doing this, the overall quality of the levels generated were particularly high. Diversity and quality of levels were also prioritized using a direct fitness function which allowed N. Ferreira *et al.* [2014] to forgo the requirement of simulation in order to check level quality. Furthermore, each fitness function was made to measure different aspects of the level generated. The first fitness function was used to measure the unpredictability of the ground; this was known as entropy. Too low of an entropy and the ground would be flat without many jumps, but too high of an entropy and the ground would be impossible to traverse. The other three fitness functions all measure a form of sparseness, which is calculated by the average distances between these items generated. These items are: floating blocks, enemies, and coins. Too sparse and the player will rarely ever see these items and enemies, not sparse enough and the level becomes cluttered and potentially impossible to complete. While this did generate very high quality levels this method requires considerable fine-tuning of the fitness functions.

de Pontes and Gomes [2020] used one population which is used to generate chunks of the level. This method was chosen so that generation can happen in real time, and this also allowed de Pontes and Gomes [2020] to have a single fitness function. However, this fitness function was quite complex, having to take into account the feasibility of a required jump, the feasibility of the player being able to continue progressing if the height of the ground were to change, how large the holes in the ground are, and how dense the enemies and power up positions are from one another. This real time generation came at the cost of lower quality levels.

Based off of the outcomes of these papers, we can see that given the right fitness function, GAs can generate high quality levels, but those levels take a significant amount of time to generate.

An alternate approach would be to use generative adversarial networks (GAN). GANs are a method of generative machine learning, that uses two neural networks. One network generates the content while the other evaluates how similar the content is to pre-existing content [Creswell *et al.* 2018]. Giacomello *et al.* [2018] used this method in order to generate levels of the video game *DOOM*. They did this by generating an image of the level, which then acted as a height map which was converted into a playable level. By doing this they were able to generate very high quality levels. However, in order to train their networks, they needed over one thousand levels from the original game. They supplemented this data set, by rotating each level four times, thus using over four thousand levels to train the network. Another draw back is that the levels that were generated were made to mimic the training data set, meaning they were only unique as a result of the large data set.

Another method of PCG would be to frame the level generation as a Markov decision process [Khalifa *et al.* 2020] which would then allow for traditional reinforcement learning as well as deep reinforcement learning techniques to be applied in order to generate levels. Several studies have implemented and expanded upon this idea [Gisslén *et al.* 2021; Khalifa *et al.* 2020; Muir and James 2022]. Muir and James [2022] used reinforcement learning as a method to expand upon genetic algorithms, which is discussed below. Khalifa *et al.* [2020] used the current layout of the world as the state. The reinforcement learning agents would start at a certain position in the world and would be given a list of actions they could perform. These actions include moving to a different position in the world as well as editing the actual world. A separate system, known as the reward function, would then check the new world and if the quality of the world had improved, it would issue the agent a reward. Similarly, the reward function would issue a punishment if the agent made the level worse in any way. These rewards and punishments were used to train the agent. Three different agents were then trained and tested, each with different capabilities for moving around the world. Note that the agents were comprised of DQNs, which means that this method of training can be considered deep reinforcement learning. The first agent could not move and could only edit the cell they were in; they would then be moved at the start of each iteration of the world generation. The second could only move in straight lines, editing the blocks as it moved over them, and the third had unrestricted movement and could also edit blocks as it moved. These agents performed well, generating very high quality levels, however, they required complicated and resource heavy reward functions in order to perform the training. Furthermore, the levels generated were quite easy and took many iterations to generate.

Khalifa *et al.* [2022] would go on to expand this idea by applying both genetic algorithms and the DQNs in order to generate maze levels. Khalifa *et al.* [2022] would alternate between generating a few levels using the GA before training

the DQN to mimic these levels. Furthermore, Khalifa *et al.* [2022] found that the levels generated by training this method did not match the final outputs of the GA, but instead created it own level. Khalifa *et al.* [2022] proposed that this could be due to the noisy data generated by the GA. However Khalifa *et al.* [2022] only ever tested this method while the GA was generating levels instead of batching a few levels at time. Furthermore, this model was only ever tested on a maze game which is a fairly simple environment. It is possible that a more complex environment could allow the model to see patterns more clearly and produce better levels.

Gisslén *et al.* [2021] took a different approach to training the agents. Instead of having one agent that was trained using a complicated reward function, they instead had two agents that took on different roles. The first agent would generate the level, and the second would then play through the level. The generator would get a reward or punishment based on the performance of the player agent, as well as an auxiliary input which ensures that the agents were not over-training to the current level being generated. Training in this way ensures that the levels generated are possible to complete, as they are being simulated at each step of the generation process. Gisslén *et al.* [2021] then tested two types of training method using the idea of two agents. The first method was competitive, meaning that the generator would receive a negative reward when the solver was successful in making progress to completing the level and a positive reward was provided when the solver was unsuccessful in progressing towards completing the level. The second method was cooperative, meaning a positive reward was given to the generator when the solver performed well and a negative reward was given when the solver performed poorly. Testing these methods brought them to the conclusion that the quality of the levels depended on the value of the auxiliary input. Furthermore, this auxiliary input needed to be tuned, not only for every type of game being generated, but also the method they used for training.

Muir and James [2022] expanded upon GAs by using them to generate a data set of levels. They would then train reinforcement learning agents to mimic the levels within the data set. This was done with the intention to generate high quality and diverse levels, as a GA would, in a significantly shorter amount of time. Training the RL agents in this manner also allowed them to forgo the need for a complex reward function, as the reward was determined by how similar the agents output was relative to the data set. However, Muir and James [2022] admits that their approach was naïve and simple, as they used a K-nearest neighbour approach to find which previously seen state is most similar to the current state. After which the agents would then manipulate the level based off of rewards that were recorded by this similar state. Khalifa *et al.* [2020] proposed an alternate approach where the reinforcement learning agents are built on deep neural networks. These networks then learn the policy through reinforcement learning, meaning that during generation time, there is no need to search for similar levels. This is a more sophisticated approach and has the potential to improve the method used by Muir and James [2022].

IV. Methodology

A. Introduction

This section begins with a discussion and an explanation of all models that were tested. Furthermore, this section concludes with the methodology used to test said models.

B. Level Representation

All generated levels have the same structure, a 2 Dimensional binary array of size 30 blocks by 45 blocks. Each block can take on the value 0 or 1, where 0 is an empty space and 1 represents a solid block. As a note, the positions used for the starting and ending goals are $(27, 0)$ and $(27, 44)$ respectively.

C. Genetic Algorithm

In order to create a base data set, a genetic algorithm [Gad 2021] with the following hyper parameters was used:

- Number of Parents: 16
- Population Size: 50
- Number of Elitism Individuals: 15
- Crossover Type: Two Points
- Parent Selection Type: Roulette Wheel
- Mutation Type: Scramble
- Probability of Mutation: 10%
- Parents Carried Over: 6

The fitness function focuses on the solvability of a particular level as no developer creates levels with the intention of them being impossible to complete. Therefore the A^* algorithm [Beukman *et al.* 2022] is used to determine if there is a path from the starting position to the goal position. With this focus in mind, when looking at the levels the genetic algorithm is able to generate, they fall into three distinct categories. Firstly, invalid levels, which are levels where either the starting or goal position is not an empty block. Secondly, a unsolveable levels, which are levels that are valid however there is no valid path from the start position to the end position. Finally, solveable levels, which are valid levels that have a path from the start position to goal position. For each of these categories fitness is determined differently. Invalid levels have the minimum fitness of -120, as to discourage the GA from creating them. Unsolveable levels have the potential to evolve into solveable levels over the course of the generation process, therefore the fitness is determined by the negative euclidean distance from the end goal to the node that A^* found to be the closest to the goal. This encourages the GA to improve the path, eventually making the level solveable. Finally if the level is solveable fitness is determined using the following formula:

$$f(l) = |P(l)| + \Delta \times |N(l)| \tag{1}$$

Where $P(l)$ is the path from the start node to the end node. Δ is a constant difficulty modifier and $N(l)$ is a list of all nodes explored by the A^* algorithm. This encourages the levels generated to be more difficult and thus making more interesting levels.

During the course of the generation process, every time there is a positive change in the max fitness of a population, that

individual is saved. Once the max fitness passes a specific threshold, the best individual is also saved and the generation for a given level is complete. These saved individuals then make up the evolutionary path the GA used to generate a given level, and are thus used as training data for the other models. Finally, the actual generation is as follows:

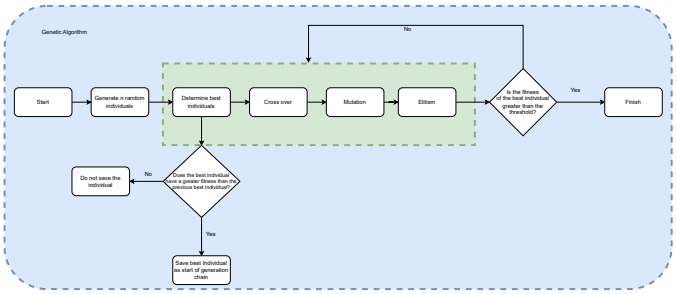

Fig. 1. The Process of generating levels using the GA.

For a full sized version of this process, see appendix A.

D. Standard Deep Neural Network

This network took a similar shape to that of an auto encoder. The DNN was symmetric around the center layer. It was comprised of five fully connected layers, with the following node counts:

- Layer 0 or Input Layer: 1350 Nodes
- Layer 1: 750 Nodes
- Layer 2: 500 Nodes
- Layer 3: 750 Nodes
- Layer 4 or Output Layer: 1350 Nodes

All layers used the ReLU activation function expect for the output layer which used a sigmoid as each block in the level can only take on the values 0 or 1. This DNN has no convolutional layers and thus the input and output are a flattended version of the level. The network also used Mean Square Error (MSE) to calculate loss. Finally, an input target pair was comprised of a level and the next level in the chain of generation. The network was trained on all 1260 levels generated by the GA. Note that 20% of these levels were used for validation data and 20% of the levels were used as testing data.

E. Convolutional Neural Network

A major drawback of standard DNNs is that they cannot take positional information into account when producing an output. However, positional information is vital in *Mario* levels, as the position of a single block can change the path to complete the levels entirely. Thus to overcome this limitation a Convolutional Neural Network was also tested. The structure is as follows:

- Layer 0 or Input Layer: CNN Layer with 128 Kernels of size 3x3
- Layer 1: Max Pooling Layer with kernel size of 2x2
- Layer 2: CNN Layer with 256 kerenels of size 3x3
- Layer 4: Max Pooling Layer with kernel size of 2x2
- Layer 5: CNN Layer with 512 kerenels of size 3x3
- Layer 6: A Flattening layer to bring the output into a 1 Dimensional Array
- Layer 7: 750 Fully connected nodes
- Layer 8: 500 Fully connected nodes
- Layer 9: 750 Fully connected nodes
- Layer 10 or Output Layer: 1350 Fully connected Nodes

Again, all layers used a ReLU activation, with the exception of the output layer which used sigmoid instead. MSE was also used for the loss. The input into this network, however, is the full 2 Dimensional level and its output is once again a flattened version of the level. Training pairs are therefore made up of the 2 Dimensional current level and a flattened version of the next level in the generation chain. This model was trained using the full 1260 level with the same 60-20-20 split for training validation and testing levels.

F. DQN

The CNN can potentially struggle with making too many changes to a level at a time, to counter this Khalifa *et al.* [2022] proposed using DQN instead. Where a small chunk of the level is given to the network as input and the network outputs what action needs to take place at the center square in the chunk. For this network the chunks are of size 12x12 and the possible actions are $[0, 1, 2]$ where 0 changes a block to an empty square, 1 changes the block to a solid object and 2 tells the network to make no change. The network is structured as follows:

- Layer 0 or Input Layer: CNN Layer with 32 kernels of size 12x12
- Layer 1: CNN Layer with 64 kernels of size 9x9
- Layer 2: CNN Layer with 128 kernels of size 3x3
- Layer 3: A Flattening layer to bring the output into a 1 Dimensional Array
- Layer 4: 256 Fully connected nodes
- Layer 5 or Output Layer: 3 Fully connected nodes

All layers uses ReLU activation with the exception of the output which uses a softmax function. For loss Categorical Cross entropy is used. A training pair consists of a 12x12 chunk of a level with the target being a one hot encoded array, where the position of the 1 corresponds to the action needed to have the center block be the same as the next step in the generation loop. This model was trained on all 1260 levels, however, there was an imbalance in the data, where the action of "do nothing" had many times more training pairs than either of the other actions, thus a second model was trained where half of the "do nothing" pairs were not used in any training. The standard 60-20-20 split was used for training testing and validation data.

G. Stacked Input DQN

An issue with only using small chunks of a level is that the network cannot see how the change it applies effects the level as a whole. Therefore, while the changes it performs has the potential to be good locally, globally the change could be detrimental, as a single change could make a given section

more interesting but make the level impossible to complete overall. A potential solution to this would be stacking the input to the network. For this model our top layer contains the entire level while the second layer contains the same 12x12 chunks that were used in the previous model, however, this chunk is padded with 0s so that it is the same size as the original level. The rest of the network is identical to the standard DQN that was used in the previous model. Furthermore, all layers use ReLU except for the softmax on the output layer. Categorical Cross entropy is once again used for loss. The issue with stacking the input is the extra memory it requires to store the training data. Due to a limitation in the amount of compute available, this model was only trained using 800 levels instead of the full 1260 levels. Finally, the class imbalance still exists, so two models were trained, one keeping every training pair, and the other not using half of the training pairs which had "do nothing" as the action.

H. Multi-Input DQN

The stacked DQN has two major flaws. Firstly, the full level and the chunks require different convolutions for the network to understand what type of changes need to be made. Secondly, padding the chunks increase memory usage of the network substantially. Furthermore, due to CNN layers being spatially invariant, the network does not know which block in the level it is editing. The solution to this is to have two different CNNs, whoes output gets concatenated along with the x and y location of where the edit is taking place. This concatenated output is then run through a fully connected layer before outputting an action. The structure of all the networks is as follows:
Network 1: Input is the entire level:

- Layer 0 or Input Layer: CNN Layer with 32 kernels of size 12x12
- Layer 1: CNN Layer with 64 kernels of size 9x9
- Layer 2: CNN Layer with 128 kernels of size 3x3
- Layer 4 or Output Layer: A Flattening layer to bring the output to a 1 Dimensional Array.

Network 2: Input is the chunk of the level:

- Layer 0 or Input Layer: CNN Layer with 32 kernels of size 12x12
- Layer 1: CNN Layer with 64 kernels of size 9x9
- Layer 2: CNN Layer with 128 kernels of size 3x3
- Layer 4 or Output Layer: A Flattening layer to bring the output to a 1 Dimensional Array.

Network 3: Input is the concatenated output of network 1, network 2 and an xy coordinate of where the network is editing.

- Layer 0 or Input Layer: 256 Fully Connected Nodes
- Layer 1 or Output Layer: 3 Fully Connected Nodes

All layers use ReLU except for the final output layer which uses a softmax function instead. The network was trained using Categorical Cross entropy as its loss function. Instead of training each network individually, the loss was only calculated once from the final input and the update was propagated through all networks. This network was trained on the full 1260 levels.

All neural networks underwent the same general algorithm for generating levels. Firstly the current level, was edited to be the correct shape for the network. This level was then inputted into the network in order to get the changes that needed to be applied to the level. These changes would then get applied to the current level. This was repeated several times.

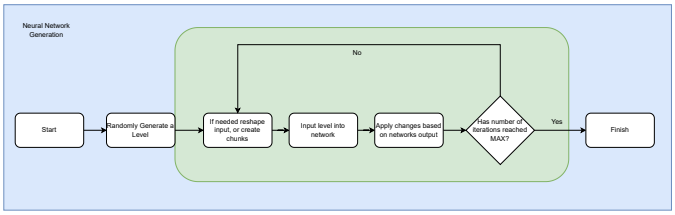

Fig. 2. The Process of Generating a Level using a Neural Network

For a full sized version of this graph see appendix B.

I. Experimentation Methodology

1) Genetic Algorithm: The GA acts as the baseline for this research and thus several tests were run to act as a comparison. The first test was the generation of the first 1260 levels, during which several statistic were monitored such as average number of generations required to generate a level, average amount of time needed to generate a level and average fitness over time. The second test which was run generated another 1000 levels, where the number of generations between changes of fitness was tracked.

2) Neural Networks: All Neural network models underwent the same testing. All models had to generate a level using a specified amount of iterations through the network. These specified amounts were $1, 2, 3, 4, 5, 10, 100, 1000$. During these, the average fitness over time was tracked. There is a comparison between the final output of these iterations and the original level, showing the common trends that these models follow during generation.

V. Results and Analysis

A. Genetic Algorithm

The first thing to note about the genetic algorithm is that not every level is equal, as some levels take substantially more generations before they meet the criteria to be solveable. The threshold for stopping was a solveable level. Due to this, the amount of times a step in the generation process is saved also varies greatly, this makes it very difficult to explore the various statistics that were monitored over time. However, as figure 3 shows, that the amount of times a step in the generation process was saved falls into a gaussian distribution between 8 saves and 27 saves. Therefore, we can split future metrics by the particular number of saves a level had.

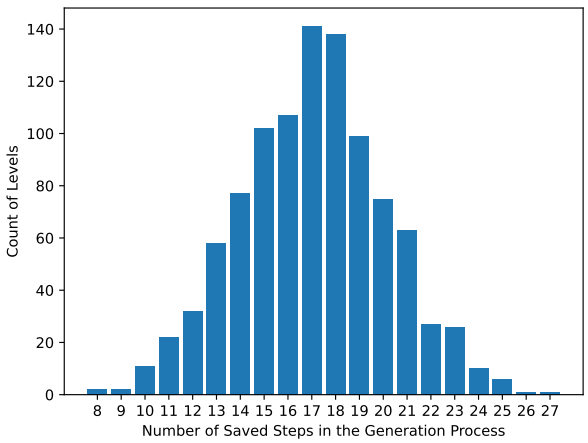

Fig. 3. A Bar Graph Showing Counts of Levels vs How many Generations it Took to Generate

All future graphs will only show bins that have 40 or more levels in them, but for the full graphs see appendix C and appendix D.

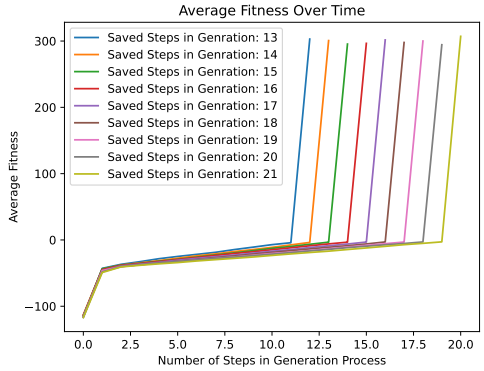

Fig. 4. Line Graphs Showing the Average Fitness Over Time

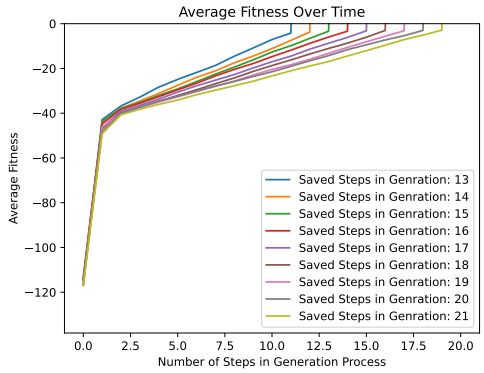

Fig. 5. Line Graphs Showing the Average Fitness Over Time Cropped to Only Show Values Below 0

Figure 4 and 5 show the average fitness over time. All levels start with in an invalid state, as they all have a fitness close to -120. However, within the first few generations the level becomes valid, which is indicated by the sudden jump up to a number close to 0. Once the level is valid, the GA begins making the level solveable, which is indicated by the slight increase in fitness over time. What is interesting here is that there are no noticeable jumps before the fitness reaches 0, this implies that the GA tends to make changes from the left side of the level and moves to the right over time, which aligns with how fitness is measured. Lastly, the levels all reach a solveable state where the fitness jumps to measuring difficulty instead. Due to the stopping threshold being solveable, the fitness stops improving. If the threshold were to be higher and allow the genetic algorithm to generate levels for longer, the fitness would follow a similar pattern to when the levels were becoming solveable, resulting in a gradual increase in fitness over time. All levels reaching a fitness greater than 0 implies that GAs are good at generating levels that are humanly playable. The problem is generation time, on average the GA took 675.879 seconds and 3852 generations to create a playable level. This roughly translates to about 11 minutes per level, which is longer than the few seconds a player is willing to wait between playing levels. Furthermore, this time will only increase as levels need to be more complex as providing the GA with more options means more mutation possibilities, which means longer generation time. Another factor that could increase generation time is size of levels, the increase in the possible positions available for the GA to cross over, once again increases probability of the GA making changes that do not impact in the fitness and thus increase generation time.

B. Deep Neural Methods

The graphs showing the levels generated are structured as follows: First the original level which was created using a random number generator is displayed. Following this is the level after all iterations of generation have been completed. Lastly, a difference map, where all changes from the original level are indicated in red. Also note that the starting and end location are marked with *Mario* and a small flag respectively.

1) Standard Neural Network: After only a single iteration of generation, the neural network shows a lot of potential. Both figure 6 and 7 show that the level instantly becomes valid and a path from the starting position to the goal position is clearly starting to form. Furthermore, figure 8 shows that there are iterations with much fewer changes being applied to the level. However, the model still tends towards creating solveable levels after a few iterations. These solveable levels, tend to have a low fitness, with an average of 259. This implies there are minimal obstacles for the player to interact with and this is clearly shown by figure 9. This is substantially lower than the average level produced by the GA having a fitness close to 300, which is a more complex and interesting level.

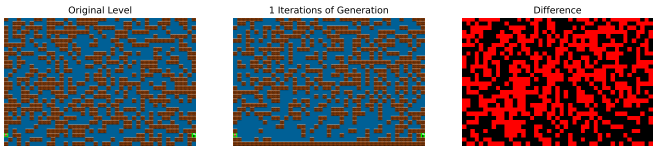

Fig. 6. The Level Generated after 1 Iteration of Generation using a Standard Neural Network

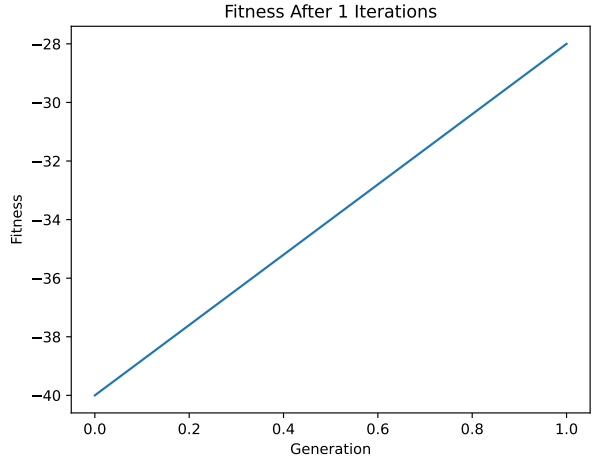

Fig. 7. The Fitness of a Level after 1 Iteration of Generation using a Standard Neural Network

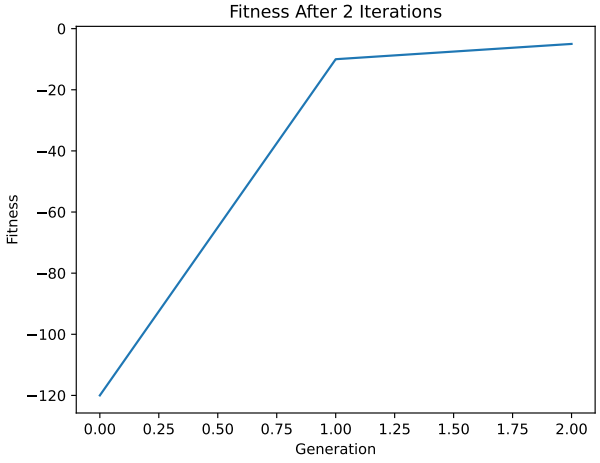

Fig. 8. The Fitness of a Level after 2 Iterations of Generation using a Standard Neural Network

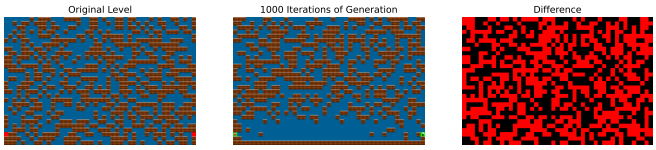

Fig. 9. The Level Generated after 1 000 Iteration of Generation using a Standard Neural Network

A possible explanation is that the network does not take spacial information into account. As the network is fully connected, there are specific output nodes associated with each block in the level. Due to the network not having any convolutional layers, the network learns to set certain blocks to specific values, in this case 0, rather than use actual spatial information to decide on what changes need to be made.

2) CNN: The benefit of adding convolutional layers to the network is that the network can now take positional information into account when making changes to the level. After the first iteration, the level becomes valid as shown by figures 10 and 11. However, upon examining figure 12, a pattern in the model's behaviour emerges. Instead of making changes that improve the quality of the level, the model removes blocks directly in the path of the player. This results in levels with extremely low fitness as the player only has to walk in a straight line to complete the the level. This is further confirmed by figure 13. A possible reason for this is that the actual structure did not change much from the Standard Neural Network, with the exceptions of the convolutions taking more information into account in a given iteration. This would mean that the two models converge to the same output with the CNN converging faster.

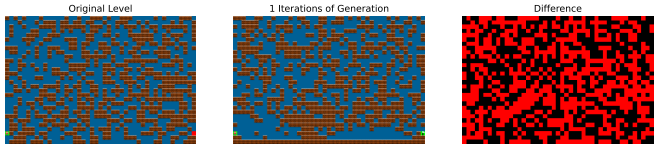

Fig. 10. The Level Generated after 1 Iteration of Generation using a Convolutional Neural Network

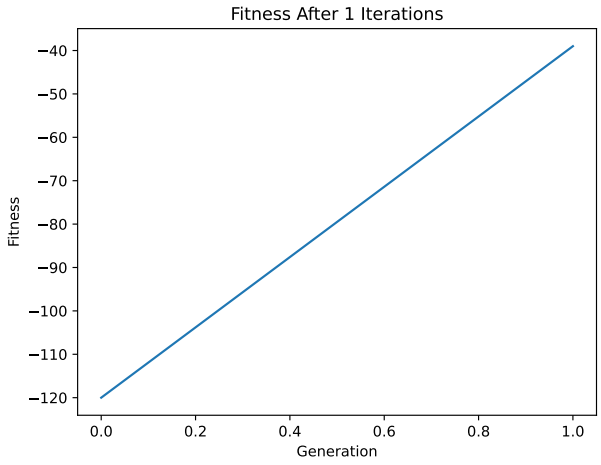

Fig. 11. The Fitness of a Level after 1 Iteration of Generation using a Convolutional Neural Network

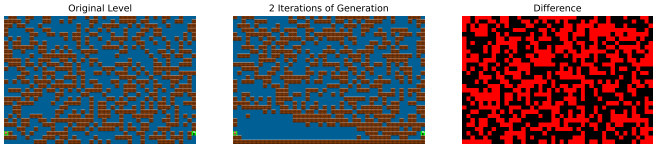

Fig. 12. The Level Generated after 2 Iterations of Generation using a Convolutional Neural Network

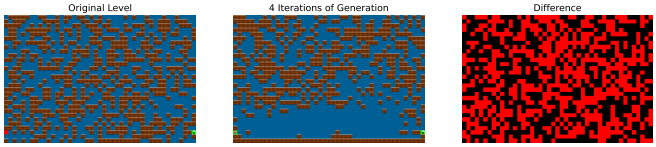

Fig. 13. The Level Generated after 4 Iterations of Generation using a Convolutional Neural Network

3) DQN: The DQN appears to make no changes to the level such as in the case of figure 14, or minimal changes such as in the case of figure 15. The DQN only has 3 actions it can perform, "change block to solid", "change block to air" and "do nothing". A pattern emerges when analysing the training data. Due to the minimal amount of changes that take place between one step of the generation process and the next, the data becomes unbalanced, favouring the "do nothing" action. This causes the model to become overfit to the "do nothing" action.

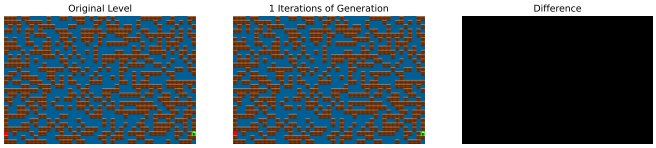

Fig. 14. The Level Generated after 1 Iteration of Generation using a DQN Trained on an Unbalanced Dataset

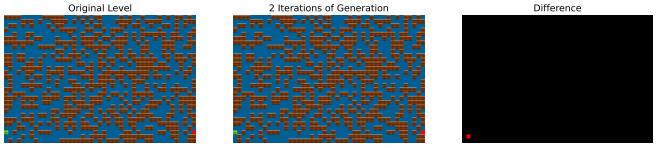

Fig. 15. The Level Generated after 2 Iterations of Generation using a DQN Trained on an Unbalanced Dataset

To overcome this limitation a second DQN was trained. For this model, half of all training pairs that had the target of "do nothing" was unused. This resulted in an even split between all three actions in the training data. Figures 16 and 17 indicates a recurring pattern, where the model oscillates between iterations of significant modifications to the level and iterations of reverting these changes. However, these changes that the model reverts are not changes that affect the path that the player would interacts with, but instead are changes to surrounding blocks that do not directly affect the fitness of the level. This is confirmed by figure 18. This model also does not follow the same generation trends as the other models. This is evident by the model never completing the floor. In addition the model often changes the starting and goal positions to be a solid block, which makes the level invalid. A potential reason for this could be that the network is only seeing a small chunk of the level and is therefore unaware of where it is editing and how the changes affects the level overall. Thus it makes general changes rather than changes that have a positive effect to the fitness.

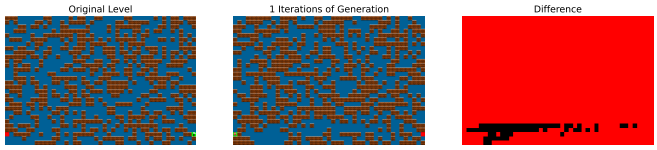

Fig. 16. The Level Generated after 1 Iteration of Generation using a DQN Trained on Balanced Dataset

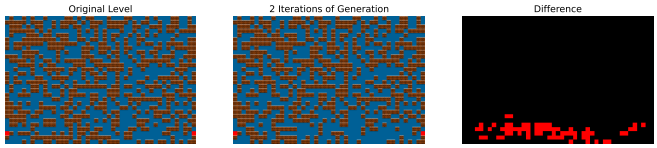

Fig. 17. The Level Generated after 2 Iterations of Generation using a DQN Trained on Balanced Dataset

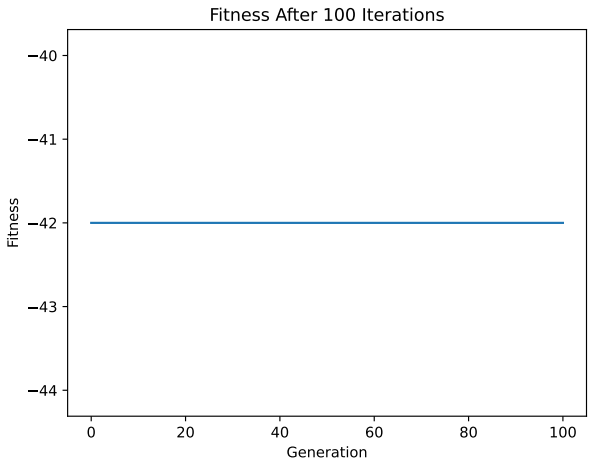

Fig. 18. The Fitness of a Level after 100 Iteration of Generation using a DQN Trained on a Balanced Dataset

4) Stacked DQN: As previously shown, a single chunk of the level is not enough information for the network to correctly understand where it is editing and what effect the edits will have to the overall fitness. A potential solution to this is to include more information, specifically the entire level. In order to achieve this the chunks need to be padded to be

the same size as the level and are then stacked along side a copy of the current level. Resulting in the doubling of memory usage during training and generation. Once again two models were trained, one with a balanced data set and one with an unbalanced data set, which will be discussed in that order. Due to compute limitations these models were trained on 800 levels as apposed to the usual 1260 levels.

The model has a much better understanding of the level as a whole and performs better overall compared to the equivalent unstacked model. This is shown by figure 19, as both the start and goal positions are converted to be empty spaces and the floor is filled in. Furthermore, this model shows potential, as after several iterations the levels generated require several jumps to complete, such as in figure 20. However, the model never creates solveable levels. In the case of figure 20 the model never adds in a block that would make the first jump possible for the player, and thus the level is unsolvable.

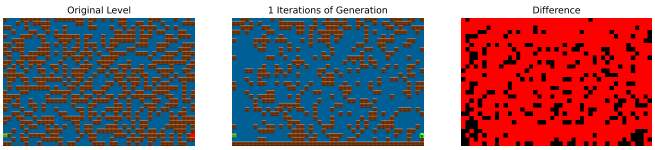

Fig. 19. The Level Generated after 1 Iteration of Generation using a DQN with Stacked Input Trained on an Unbalanced Dataset

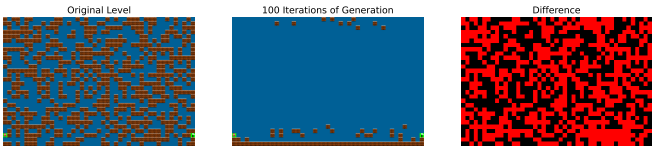

Fig. 20. The Level Generated after 100 Iterations of Generation using a DQN with Stacked Input Trained on an Unbalanced Dataset

It is possible that the unbalanced training data set could be the cause of these issues. In order to test this a second model was trained, this time with a balanced data set. Figures 21 and 22 show that this model alternates between iterations of mass change followed by iterations of reverting many of the previous changes. Figure 23 shows that the model reverts all changes that have a direct positive impact on the fitness. Overall, this model performs worse compared to its unbalanced counter part. This could be due to the reduced data set, as not using half of all training pairs that contained the "do nothing" action could have reduced the already shrunken data set too much. Thus causing the model to underfit.

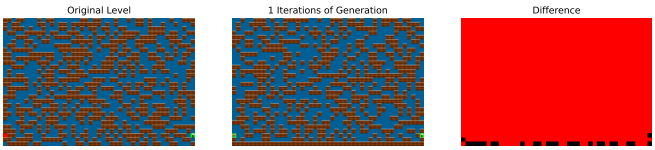

Fig. 21. The Level Generated after 1 Iteration of Generation using a DQN with Stacked Input Trained on a Balanced Dataset

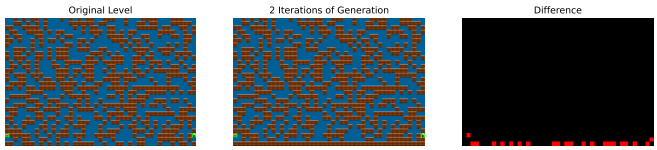

Fig. 22. The Level Generated after 2 Iterations of Generation using a DQN with Stacked Input Trained on a Balanced Dataset

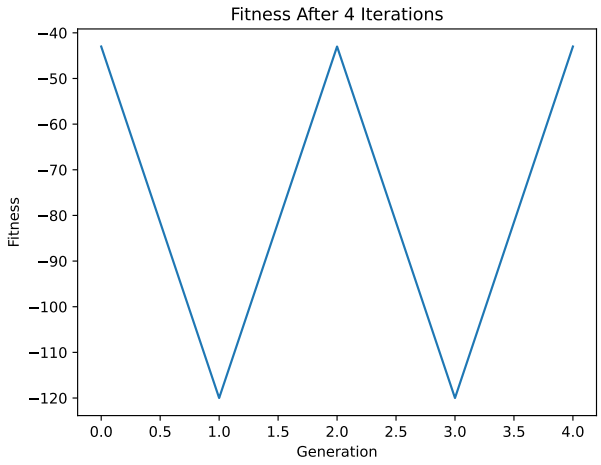

Fig. 23. The Fitness of a Level after 4 Iterations of Generation using a DQN with Stacked Input Trained on a Balanced Dataset

5) Multiple Input: The multiple input model combines all the improvements made to the other models, including balancing the training data, using chunks as well as the full level, but allowing them to have different convolutions as well as explicitly telling the model where the changes are occurring. Figures 24 and 25 show that the model alternates between periods of mass change and periods of reverting those changes. This is further shown by figure 26. Figure 25 also shows that changes that are reverted by the model, are changes that do not directly affect the path of the player. This model also struggles with creating valid and solevable levels, often placing blocks in the starting or goal positions and never clearing enough of a path for the player to be able to reach the goal. This could possibly be due to the data that was used to train the model. As this is the balanced data set, a significant portion of the training pairs were unused. Due to the complexity and size of the of the model, there is a distinct possibility that there was not enough data for the model to train on, causing it to underfit.

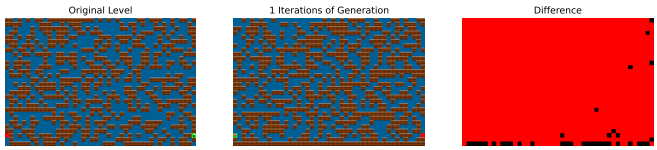

Fig. 24. The Level Generated after 1 Iteration of Generation using the Multiple Input Model

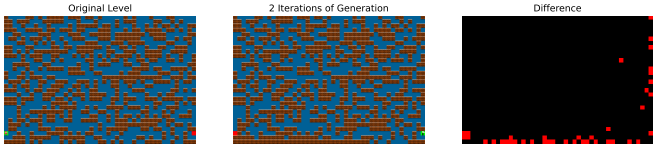

Fig. 25. The Level Generated after 2 Iterations of Generation using the Multiple Input Model

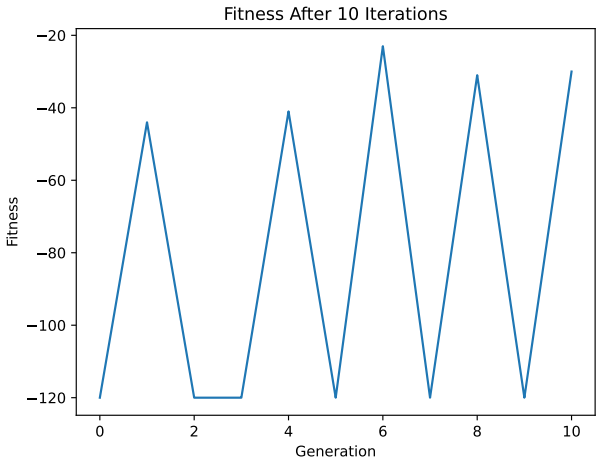

Fig. 26. The Fitness of a Level after 10 Iterations of Generation using the Multiple Input Model

VI. Limitations

Compute was a major limiting factor. Due to GA's having on average 3852 generations to create a level, it becomes infeasible to store every generation of every level in memory when training other models. To overcome this, only the generations that have a change in the fitness are saved. However, as indicated in figure 27, there is an exponential growth in how many generations need to occur before the next step in the generation process is saved. Figure 4 implies that this should not be an issue as the fitness gradually increases rather than having large jumps. However, what these figures do not show is that there are subtle changes to the levels that do not directly affect the fitness. As these changes grow in number, it eventually becomes impossible for clear patterns to emerge during generation, thus greatly hindering the performance of the other models. There are two possible solutions to this, namely a form of iterated learning where the models are trained on only a few levels at a time. Alternatively Khalifa *et al.* [2022] trains the models as the GA generates new levels. Both of these methods would overcome this limitation and could show far better results than only storing a portion of the generations.

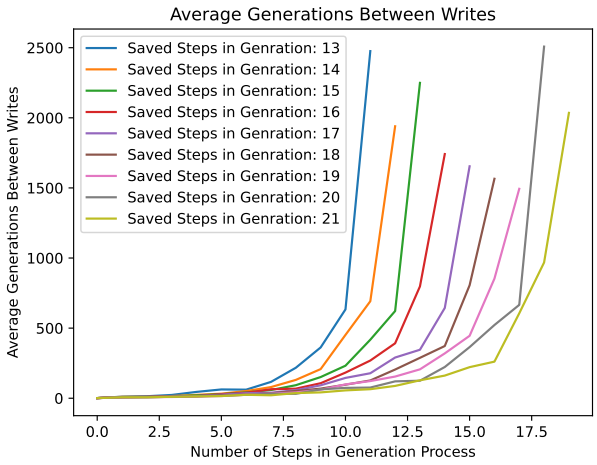

Fig. 27. Line Graphs Showing the Average amount of Generations Between Saves in A Generation Process

Please see appendix E for full version of this figure.

VII. Future Research

The methods that were proposed in this paper do have merit and potential. However, this potential was not fully realised and thus further research into this field is necessary. This research should focus on several key areas, the first of which should be the limitation on compute. Khalifa *et al.* [2022] used a form of iterated learning that could be the basis for improving upon the method proposed. Another area that could be improved is the model architecture. *Mario* is a fairly complex environment as every block placed has an indirect impact on the path the player must travel in order to complete the level. Standard deep neural network architectures struggle with these implicit connections between blocks. A possible solution would be to use the multi-head attenuation of a transformer as they are specifically designed to handle implicit long distant connections such as those between block placements in a *Mario* level. Alternatively, using networks designed for generation, such as a GAN could possibly improve the performance even further. Research into improved fitness functions would allow for the GA to generate superior levels, which could have more distinct patterns for the downstream models to learn, thus improving overall performance of all models. Finally, retesting these models with an improved data set could potentially result in better performance over all.

VIII. Conclusion

Genetic algorithms clearly have the ability to produce high quality and interesting levels, as long as they have a good enough fitness function and are given enough time to produce the levels. Deep learning showed some potential in improving upon the generation process. In particular the stacked DQN model produces almost solveable levels that were fairly interesting. However, all deep learning based models failed to reach an acceptable standard for the levels produced, even if the generation took a fraction of the time of the GA. The

levels were either, invalid, unsolveable or uninteresting to play. This poor performance could be attributed to the lack of compute forcing testing to be done using a flawed data set. There is a distinct possibility that standard deep learning and deep reinforcement learning are inappropriate mediums for generative models and thus more complex models such as transformers and generative adversarial networks should be considered and explored.

REFERENCES

[Beasley *et al.* 1993] David Beasley, David R Bull, and Ralph Robert Martin. An overview of genetic algorithms: Part 1, fundamentals. *University computing*, 15(2):56–69, 1993.

[Beukman *et al.* 2022] Michael Beukman, Christopher Cleghorn, and Steven James. Procedural content generation using neuroevolution and novelty search for diverse video game levels. In *Proceedings of the Genetic and Evolutionary Computation Conference*, July 2022.

[Creswell *et al.* 2018] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, 2018.

[de Pontes and Gomes 2020] Rafael Guerra de Pontes and Herman Martins Gomes. Evolutionary procedural content generation for an endless platform game. In *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 80–89, 2020.

[Gad 2021] Ahmed Fawzy Gad. *PyGAD: An Intuitive Genetic Algorithm Python Library*, 2021.

[Giacomello *et al.* 2018] Edoardo Giacomello, Pier Luca Lanzi, and Daniele Loiacono. Doom level generation using generative adversarial networks. In *2018 IEEE Games, Entertainment, Media Conference (GEM)*, pages 316–323, 2018.

[Gisslén *et al.* 2021] Linus Gisslén, Andy Eakins, Camilo Gordillo, Joakim Bergdahl, and Konrad Tollmar. Adversarial reinforcement learning for procedural content generation. *CoRR*, abs/2103.04847, 2021.

[Johnson *et al.* 2010] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, New York, NY, USA, 2010. Association for Computing Machinery.

[Khalifa *et al.* 2020] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. PCGRL: procedural content generation via reinforcement learning. *CoRR*, abs/2001.09212, 2020.

[Khalifa *et al.* 2022] Ahmed Khalifa, Michael Cerny Green, and Julian Togelius. *Mutation Models: Learning to Generate Levels by Imitating Evolution*, 2022.

[Li 2017] Yuxi Li. Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274, 2017.

[Liapis *et al.* 2014] Antonios Liapis, Georgios Yannakakis, and Julian Togelius. Constrained novelty search: A study on game content generation. *Evolutionary computation*, 23, 03 2014.

[Mnih *et al.* 2013] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning*, 2013.

[Muir and James 2022] Nicholas Muir and Steven James. Combining evolutionary search with behaviour cloning for procedurally generated content. In *EPiC Series in Computing*. EasyChair, 2022.

[N. Ferreira *et al.* 2014] Lucas N. Ferreira, Leonardo Pereira, and Claudio Toledo. A multi-population genetic algorithm for procedural generation of levels for platform games. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 07 2014.

[Rayna and Striukova 2014] Thierry Rayna and Ludmila Striukova. 'few to many': Change of business model paradigm in the video game industry. *Digiworld Economic Journal*, (94):61, 2014.

[Rose and Bakaoukas 2016] Thomas J. Rose and Anastasios G. Bakaoukas. Algorithms and approaches for procedural terrain generation - a brief review of current techniques. In *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, pages 1–2, 2016.

[Skansi 2018] Sandro Skansi. *Introduction to Deep Learning: from logical calculus to artificial intelligence*. Springer, 2018.

[Sutton and Barto 2018] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Togelius *et al.* 2011] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N Yannakakis. What is procedural content generation? mario on the borderline. In *Proceedings of the 2nd international workshop on procedural content generation in games*, pages 1–6, 2011.

IX. Appendix

A. The Process of Generating Levels using a Genetic Algorithm

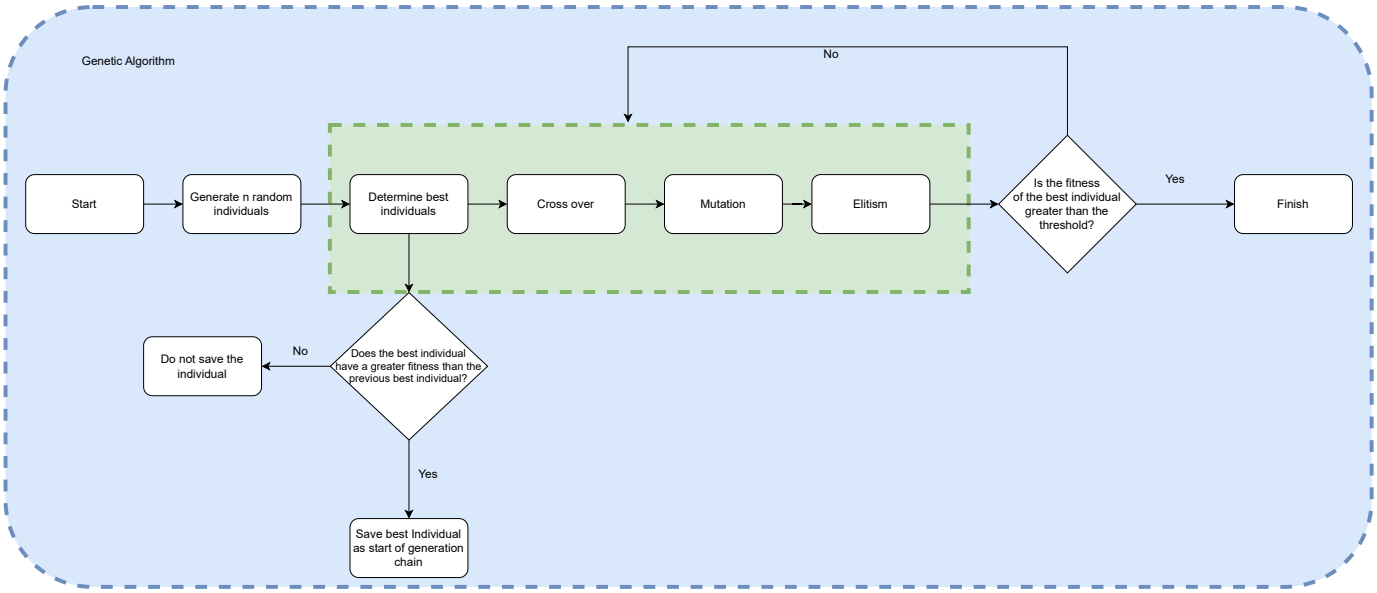

Fig. 28. The Process of Generating Levels using the GA

B. The Process of Generating Levels Using Neural Networks

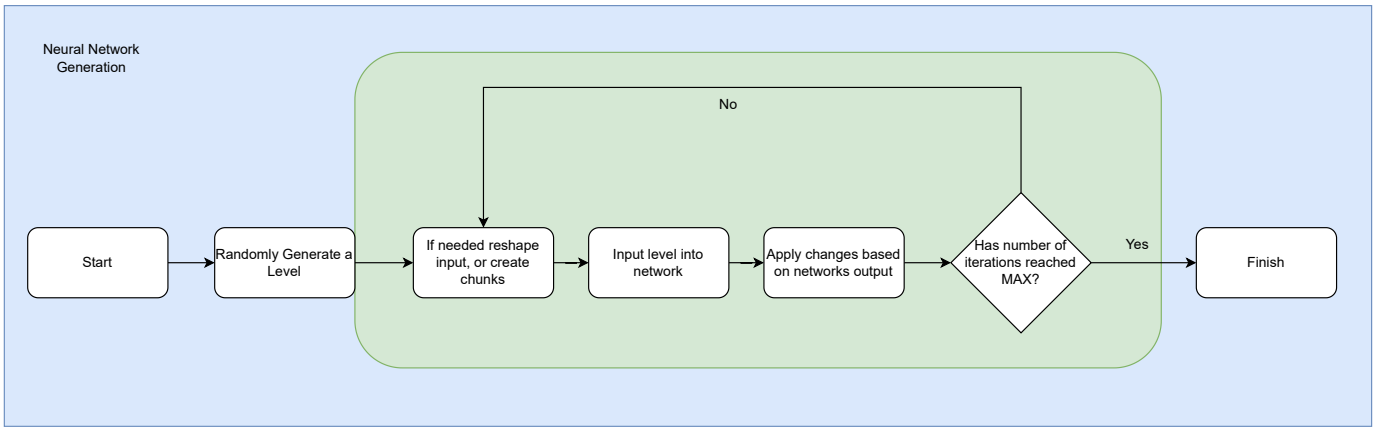

Fig. 29. The Process of Generating a Level using a Neural Network

C. Fitness Over Time of Levels Generated using a Genetic Algorithm

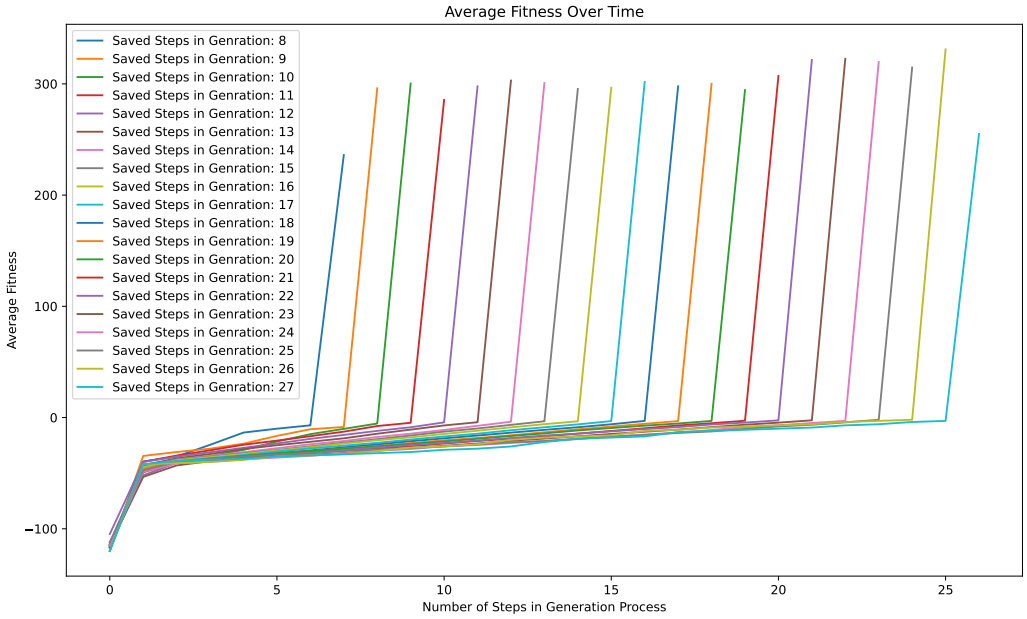

Fig. 30. Average Fitness Over Time

D. Clipped Fitness Over Time of Levels Generated using a Genetic Algorithm

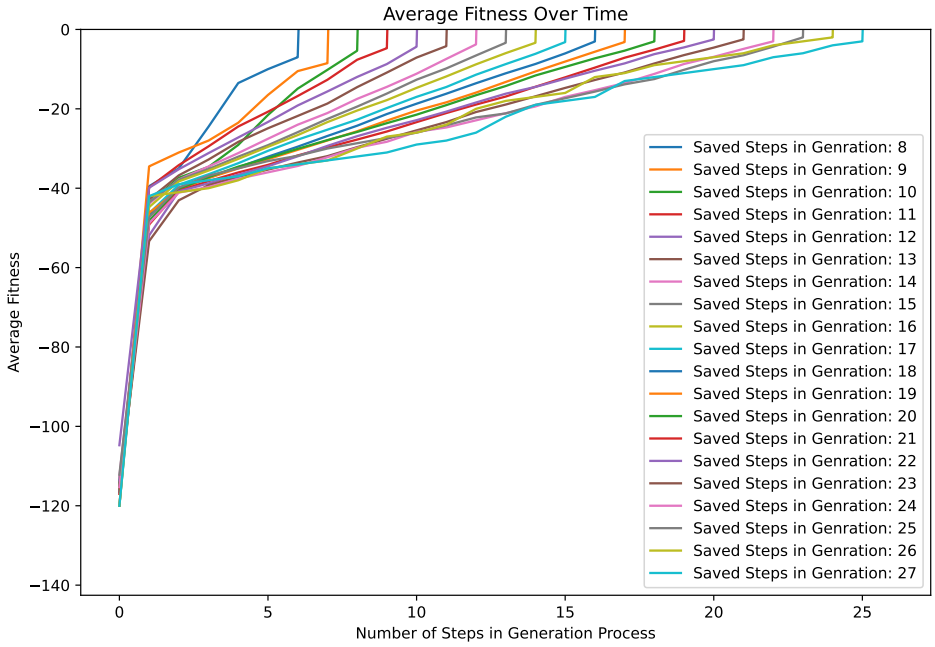

Fig. 31. Average Fitness Over Time

E. Average Generations Between Saving

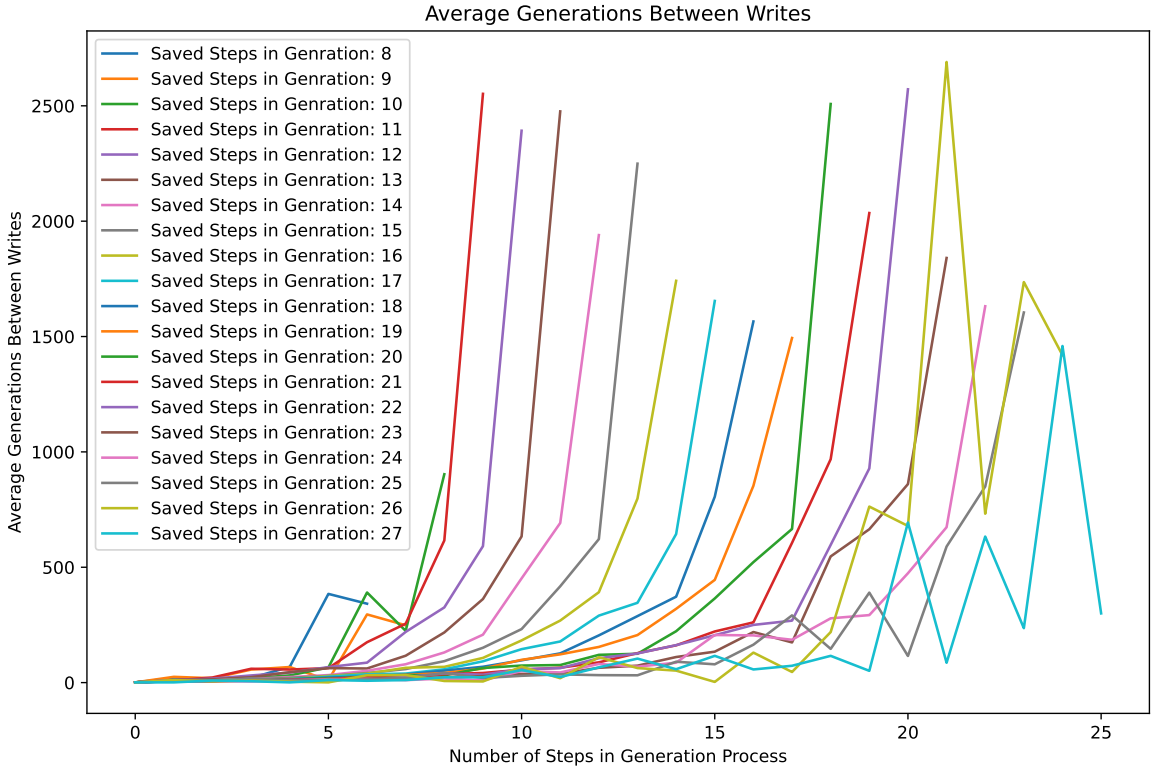

Fig. 32. Line Graphs Showing the Average amount of Generations Between Saves in A Generation Process