Generalizing Artificial Intelligence to Play Super Mario World

Bridging Genetic Algorithms and Supervised Learning

Benjamin Spiegel, Chris Yu

May 15, 2019

1 Abstract

Generalizing artificial intelligence to perform tasks on which it was not trained has been a historically difficult task, especially if the type of problem being solved, in our case, the gameplay of any level of a classic game, is not trivial to begin with. While there is plenty of work on developing artificial intelligence that can play video games, existing research applications to Super Mario World have fallen short due to developed bots' inability to play levels it has never seen before at a human level or above. Currently, our approach implements a generic Neuroevolution of Augmenting Topologies (NEAT) algorithm, which mainly consists of training a bot that has a neural network using a genetic algorithm. We will discuss this in greater detail in the Technical Approach section, but in general, the NEAT algorithm evaluates the fitness of certain policies based on how far to the right of the level that policy achieves (i.e. how long it survives and makes progress). Our ultimate goal is to attain a human level of success on complex levels on which the bot was not trained, showing that we have successfully designed an algorithm that achieves some degree of generalization, which we try to do via two main methods; supervised learning and cross-breeding. Both methods produce bots that demonstrate generalized behavior, but only cross-breeding shows that it is possible to produce bots of high fitness on levels to which the bots had low exposure. In the end, we were able to produce agents that competently play levels they have only seen a few times.

2 Introduction

Although the creation of a game-playing agent may not be a particularly pressing issue, if we are successful in creating a general solution that can play multiple levels in Super Mario World while only training on one, our resulting algorithm should be of interest to anyone attempting to generalize artificial intelligence, as the same algorithms that are used to train gaming bots can often be trained on other tasks that are on similar levels of complexity. In the big picture, fully generalized artificial intelligence has been a major goal in the field since its inception. Although we only seek to attain a small degree of generalizability, existing research in the industry proves that it is a difficult problem with very high utility, as much of existing artificial intelligence lags behind people on human tasks due to an inability to adapt or model generalized inferences.

As far as our problem goes, bots trained on Deep Q-Networks, among other deep learning architectures, have been successful at playing levels in Super Mario World, but only ones they have been trained on. It is also possible to create a bot that plays completely optimally on a given level based on hardcoded moves performed in a frame-perfect manner. This is the basis of Tool-Assisted Speedrun (TAS) bots, which are accepted as being the pinnacle of superhuman gameplay. Obviously, TAS bots are even more restrictive than the class of reinforcement learning bots, since a TAS bot for one level may not even successfully complete any other level. In the end, current methods fall short because they do not create adapting bots after training.

Primarily, our work will be using the NEAT algorithm as a basis in our attempt to create a bot that can complete any Super Mario World level with minimal distinct training levels. Our first method involved taking a lot of data from relatively successful NEAT agents, in other words, the decision they made (in terms of which buttons to press) given an input image. This large quantity of data was then fed in as inputs to a fully connected two-layer supervised learning algorithm that produces an agent that makes decisions on which buttons to press, given input images of the same format. There was a lot of attempts to achieve better results via input pruning, tuning hyperparameters, and adjusting the architecture, which we will discuss in detail in the Technical Approach section.

Another idea we had was to see how well agents generated by NEAT could perform on levels that were different but with similar features to the

level they were trained on. We hypothesized that supplementing agents that were trained on one level with training done on another level would yield agents that were successful at both levels. We took a population of relatively successful agents that were trained on one level and ran them on a second level to assess their baseline fitness. Then we continued training on this second level using the NEAT algorithm to see how long it would take for agents to successfully beat this second level. We refer to this method as cross-breeding.

The number of generations it takes for agents to successfully complete the second level is a measure of the success of this method. Under ideal circumstances, we would like to create bots that can fully complete levels (achieve maximum fitness by making it all the way to the right of a level) to which they have had limited exposure. Another metric of success we used was fitness, which was measured by how far right in the level that a given bot could progress. Obviously, the most successful bots achieved much higher fitness numbers than our initial bots.

3 Related Work

Genetic Algorithms use the principles of biological evolution to produce agents that can solve complex tasks. Generations of randomly generated agents are subjected to a fitness test and then culled and re-breeded. The fitness test is an evaluation of the performance of an agent with respect to the level. This process continues for many generations until agents can only make marginal improvements to the fitness test, signifying that the algorithm is close to the optimal solution that is within the capabilities of the operating structure of the agent. Neuroevolution of Augmenting Topologies (NEAT) was an improvement to previous Genetic Algorithms because it provided a better way to preserve network structures through to offspring. In previous algorithms, gene crossover would often lead to damaged offspring because structures in individual agents succeeded in different niches. Simply combining the structures often broke the underlying strategies used by the individual structures.

The NEAT algorithm is designed to breed networks with simple causal connections with growing complexity over generations. The first generation of agent networks are single mutations to the fully unconnected base network that consists solely of input and output nodes. Part of the NEAT algorithm

involves the re-breeding and mutation of agent networks. There are two ways the structures of these networks can mutate:

1. A new connection can be made between between two nodes that were previously not connected. This connection may be positive or negative.

2. A new node may form and sever a current connection, inserting itself between the previously connected nodes.

Positive connections relay the input signal to the receiving node and negative connections relay the negated input signal to the receiving node. Combinations of the two above possibilities of mutation result in intricate networks in the long run. The rationale behind this additive approach is that solutions to problems need not be more complex than what the problem necessitates. The first successful agents that NEAT creates are also the simplest agents generated that can perform the task. This is why NEAT can yield simple solutions to complex problems.

The success of using NEAT to generate agents to play levels in Super Mario World has been demonstrated before, however existing research only examines the generated agents on the level they were trained on and does not rely on other machine learning tools.

4 Technical Approach

4.1 Formal Description

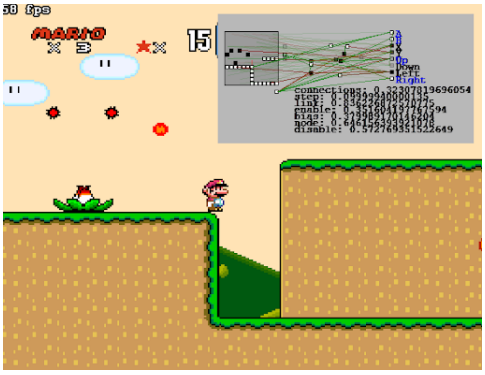

Figure 1: An example of what an input looks like at some point in the level

For the base NEAT algorithm, the inputs were in the form of image data grabbed from the emulator of the blocks in a six-block manhattan radius of Mario, or in other words, a thirteen-by-thirteen block with Mario in the center. To simplify the format of the input, it was stored as a 1D array of 170 elements ($13 \cdot 13 + 1$ extra element that is always set to 1). Each element stores an integer, in which 1 represents a block that Mario can stand on, 0 represents empty space that Mario can pass through, and -1 represents an enemy taking up that block.

The output labels to this problem are unique in that they are not one-hot, as is often the case in classification problems; at each point in time, the agent can choose to press any number of buttons on the controller, or none of them. When applicable, this output is provided in the form of a 1D array of 8 elements (one for each button on the controller), with entries of 1 meaning that the button corresponding to that element is pressed, and entries of 0 meaning that the button corresponding to that element is not pressed.

4.2 Supervised Learning

Our first method involved collecting a lot of examples from fairly successful NEAT agents (in this case, fairly successful was defined as achieving a fitness greater than or equal to 2000 on a given training level), and using them as inputs to train a supervised learning network to produce labels that would, hopefully, retain the good decisions made by the many agents we drew data from. Since our implementation learned on TensorFlow, one of our difficulties was in creating the full pipeline, from receiving inputs from the emulator, outputting input data from the NEAT algorithm defined in Lua, using that input data to train a supervised learning agent in Python, to outputting those weights so that Lua could run the agent in the emulator. This was necessary because the libraries that make supervised learning very efficient are not well-defined in Lua, and Python does not easily interface with our emulator.

In general, our supervised learning algorithm sends the data through a two-layer neural network connected with rectified linear units (ReLU), taking in random batches of a predefined size. Multi-label classification was implemented via a threshold on output logits, in which logits above a defined threshold are mapped to 1, otherwise mapped to 0. We tested this method with a variety of hyperparameters, including different learning rates, different hidden dimension sizes, and adjusting the number of epochs. Our

final bot was ran on 10 epochs, with a learning rate of 0.01, a hidden size of 20, a batch size of 100, and a threshold of 0.5. The trained weights were then sent back into a Lua algorithm that made decisions in the emulator.

We noticed a lot of repeat input/label pairs, as well as a lot of duplicate inputs mapped to different labels. This is not surprising, since we are taking in data from multiple NEAT agents who see the same level, and there are obviously multiple ways to complete levels in Super Mario World. Consequently, we tried to prune our inputs to the supervised learning architecture by only maintaining the input/label pairs that were the most populous. We tested our implementation on both pruned and unpruned inputs.

4.3 Cross-Breeding

To cross-breed agents, we took a generation of agents that were successful at playing the level they were trained on, and simply ran the generation on a new level. To do this, we modified the existing NEAT training interface to run existing pools of agents on our level of choice. We found that a decent selection (about 25%) of agents performed reasonably well (achieved a fitness of at least 2000) by the 35th generation. This is when we started to evaluate and train agents on a new level. Running the agents on the new level was pretty cut and dry without much overhead.

5 Evaluation

As stated previously, our goal was to create agents that could competently navigate through levels that they had not had very much exposure to before. In using supervised learning, we were hoping that the many good behaviors of fit NEAT agents could get generalized into one agent that would then be able to solve levels more generally, and thus levels that have not yet been seen. With cross-breeding, we are hoping to demonstrate that it is possible to produce agents that perform well on a level of low exposure, given that we already have agents that perform well on another level.

Ideally, the new agents produced by our methods would reach higher fitness scores on new levels when compared to NEAT agents on new levels. For cross-breeding, we should be able to create agents that can complete the testing level given a significantly smaller number of generations in training.

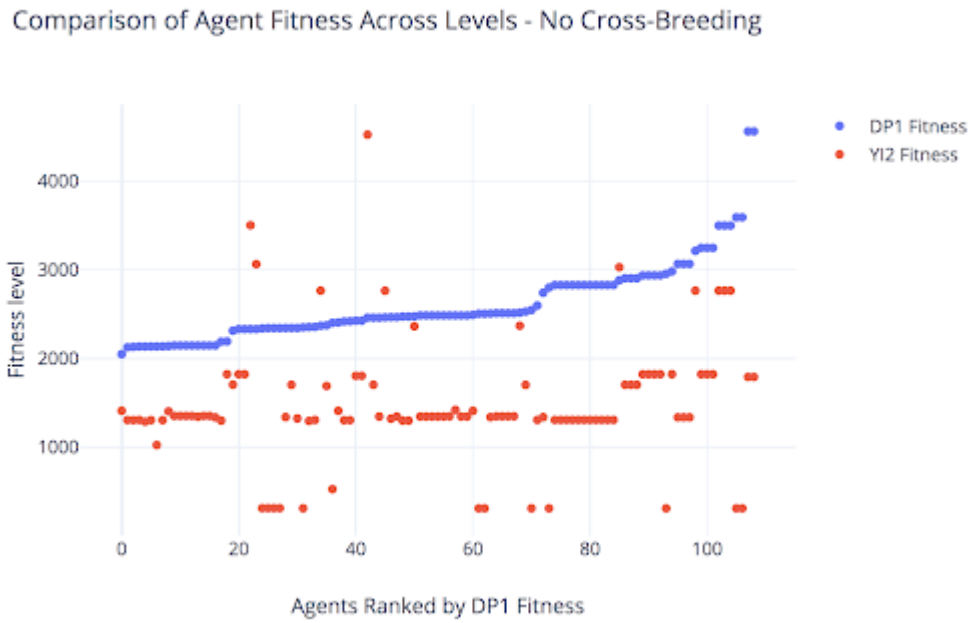

Figure 2: Running agents from Donut Plains 1 on Yoshi's Island 2

This graph shows the comparison of the fitness achieved by agents on two levels. The population of agents were trained on level Donut Plains 1 (DP1) for 68 generations and the agents that achieved a fitness above 2000 were then ran on a new level, Yoshi's Island 2 (YI2). In a majority of cases, the agents performed worse on YI2 than DP1, this is likely because the agents found themselves in unique situations for which they did not have any similar experience in. In most cases, agents achieved a fitness level of 1000 or more, demonstrating some knowledge transfer. Some agents even outperformed their run on DP1.

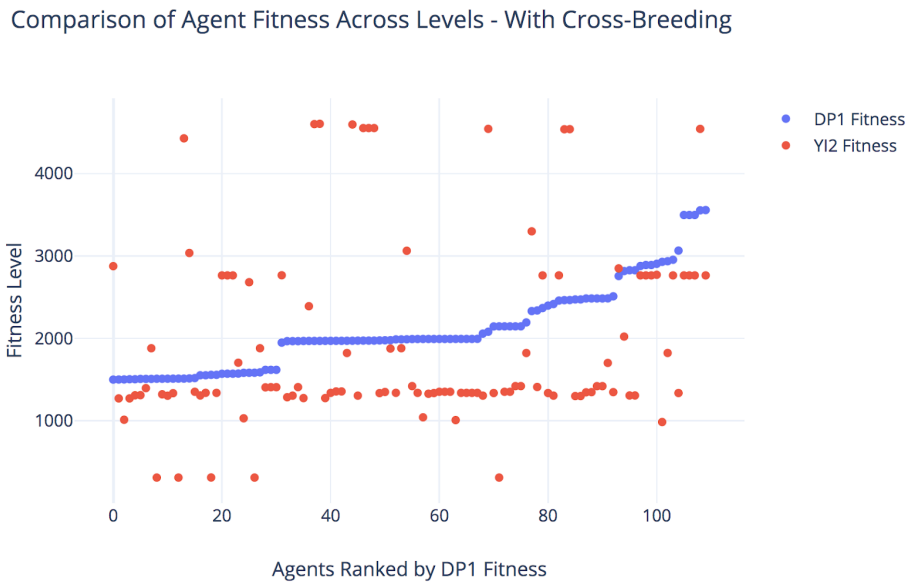

Figure 3: Crossbreeding Donut Plains agents on Yoshi's Island 2

This graph shows the fitness of a generation of descendants from the previous graph that were crossbred on YP2 for only two generations. The resulting agents still performed well on DP1 though with a lower fitness on average, but a much larger percentage of agents completed YI2. Two generations of training yielded much success on levels to which the agents had little exposure.

Here are the results of some of the bots that we trained, as well as where we started out; check the descriptions for more information on each clip: `https://drive.google.com/drive/folders/1oQB3djV167ur9jBiiRI3OHyZcpk5pHBI?usp=sharing` For our supervised learning model, although our final trained bots had decent general behavior (mostly running to the right and spin jumping a lot), it was not enough to solve levels competently. In particular, it struggled when confronted by enemies, which may be a result of not having enough inputs of successful agents handling enemies and enemy edge cases.

For the cross-breeding technique, agents with no additional training performed with mixed results. Some agents performed with higher fitness scores on the new level than on the level they were trained on and some agents performed with a lesser fitness score. There was an agent who performed in the 2000-2500 fitness bracket on the original level who won the new level.

Agents demonstrated skills with respect to dealing with enemies but they struggled with niche layouts they had not seen before, often stopping or jumping in place at the end of their run. Few agents died at the hands of an enemy and most runs were halted after inactivity. Agents who resumed training on the new level continued to increase in fitness over the course of multiple generations with about another half-dozen agents being able to solve the new level after only four generations. These agents were capable of playing both the original and new level proficiently.

6 Conclusion

In summation, we used different methods in our attempt to create a generalized bot to play Super Mario World levels to which it had little or no exposure. Our first method with supervised learning was not very successful. Although it did produce an agent that had generic behavior, that behavior was not enough to progress very far through other levels, or handle enemies well. Cross-breeding showed more promise, proving that we can efficiently create viable agents on levels of low exposure, given that we already have agents that are successful on a known training level.

Since our results have been largely preliminary, there is a lot of work that can be done where we left off. First of all, while supervised learning has not produced good results yet, it is possible that we have simply not tuned to the right parameters, or we need to consider modifying the training architecture to be more expressive. This was a concern because our model must output multi-hot labels, which means that our less than impressive results may be a by-product of lack of expressivity in the model. Another concern about this is the loss function, which is currently a basic implementation of sigmoid cross-entropy with multiple classes. Lastly, our current method with input-pruning could be improved, and we could also prioritize collecting inputs with enemies nearby.

On a longer term, we should also explore even more methods in producing generalized agents. In particular, methods that take in multiple inputs at once should be considered, since multiple frames may inform an agent better due to implicit rate-of-change information that is not present in a still. This could be particularly good for handling enemy movement, and may be more along the lines of reinforcement learning, or a combination of reinforcement and supervised learning.

Ultimately, our goal is to show that the methods we have implemented and adjusted will produce agents with good generalized behavior that results in a degree of competence in a given task (in our case, playing Super Mario World). To show that we have attained our goal in general, we would have to find more problems to test our methods on, whether it be on other level-based games, or more concrete applications.

7 Sources

Charniak, Eugene. Introduction to Deep Learning. The MIT Press, 2019.

SethBling, director. MarI/O - Machine Learning for Video Games. YouTube, YouTube, 13 June 2015, www.youtube.com/watch?v=qv6UVOQ0F44.

SMW Central - Your Primary Super Mario World Hacking Resource, www.smwcentral.net/?p=main.

Stanley, Kenneth O., and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies." Evolutionary Computation, vol. 10, no. 2, 2002, pp. 99–127.