Using Deep Q-Learning in Games to Teach about Machine Learning

Braden Wilson
Under the direction of Philip Tan
Game Lab, Massachusetts Institute of Technology

Research Science Institute
July 27, 2019

Abstract

While traditional education with rigorous definitions may suffice for many topics, machine learning is complex enough to require a deeper understanding in order for students to have a proper intuition about its precise functionality. In this case, alternative methods should be employed to convey the general functionality of machine learning models to less technical audiences. This work investigates the applications of Deep Q-Learning to a platform game as a visualization technique for machine learning concepts. Deep Q-Learning as an algorithm is applied and optimized in order to provide the most effective analogy for the training of a machine learning model.

Summary

Machine learning is an extremely important technology in the modern world, and it holds some unintuitive properties that need to be well understood in order for it to be applied effectively. This paper investigates the development of a game that uses machine learning algorithms to teach high school students about how the algorithms themselves function. This is achieved through a machine learning algorithm that learns to play a platform game, and through this training process it demonstrates how machine learning algorithms behave and learn.

1 Introduction

A machine learning algorithm, as put by Naqa and Murphy, is "a computational process that uses input data to achieve a desired task without being literally programmed to produce a particular outcome" [1]. While this is a reasonable definition of machine learning, it simply does not capture what the process actually is. Defining statistical models and regression techniques certainly gives an understanding of some of the mechanics of machine learning algorithms, but it cannot give the same level of intuition as a visualization of the algorithm. A potentially useful facet for this visualization is in games. Machine learning has been used since its inception to play games [2], and even now it serves as a challenging standard for machine learning capabilities, with companies like Google DeepMind pushing the boundaries of machine learning through games. Games are not only useful for testing the limits of machine learning algorithms, but also for visualizing them. Training an algorithm to play a game is a concrete and visual process that can serve as an ideal tool in the increasingly prominent field of machine learning education.

Machine learning's great potential as a data analysis tool has brought it to the forefront of a variety of fields from natural language processing, computer vision, aerospace engineering and finance to computational biology and medical diagnosis [1]. As machine learning algorithms continue to find applications in industry and become responsible for increasingly important decisions, it is crucial that they are implemented properly to ensure their accuracy and reliability. In the case of this project, games can serve as a source of understanding and visualization for machine learning in order to give potential developers and users of these algorithms a better intuition of how they work and the potential dangers that come with them.

In this paper we investigate the effectiveness of machine learning algorithms in playing games and their potential as an educational tool to visualize machine learning. This is achieved through a game targeted for high school students that demonstrates machine learning in a simple environment. This environment is a *platformer*, which is a type of game with two-dimensional graphics in which the player controls a character that must jump between solid platforms on the screen in order to reach an objective. In this game, the player designs levels by placing blocks that form obstacles, platforms, and objectives. The player can then take a set of such levels and train a machine learning algorithm to play them. The machine learning algorithm controls the character on screen and learns how to play the levels, clearly visualizing its training process. It learns to play these levels through the use of a reinforcement learning algorithm called Deep Q-Learning.

Deep Q-Learning is a *Deep Reinforcement Learning* algorithm used in a wide variety of applications, including Google DeepMind's software that learns to play games such as Atari Breakout, a game in which the player has to use a paddle to bounce a ball against a layer of bricks. Deep Q-Learning uses a Markov Decision Process in combination with a neural network in order to determine the optimal action for any situation. This is achieved through a *policy function* determined by a neural network that gives the optimal action in any state. Using this method, Deep Q-Learning is able to learn to play almost any game [3].

Markov Decision Processes, also called *Markov Chains*, have been applied in many different situations ranging from training self-driving cars to composing music with algorithms. In general, a Markov Chain is a stochastic model of a sequence of possible events, called *states*, each with a probability of occurring determined by the previous state. In this particular application, a Markov Chain is being used in the form of a feedback system of *rewards* and *penalties* applied to an *agent*, which is the character in this case, that operates in an *environment* and needs to choose *actions* to move through a series of states to reach a predefined final state. The agent achieves this by using a mix of experimenting with random actions and exploiting past experiences to learn the optimal move for each situation. The solution is reached when the agent finds an optimal sequence of actions to maximize the *accumulated* *sum of rewards* [4].

At every time step, the agent enters a state and needs to choose an action from a fixed set of possible actions. The decision of which action to take depends only on the current state, and not on historical states. This means that once a policy function is determined, it only needs to take the current state as input in order to determine the optimal action. Performing an action a at time t will result in a transition from the current state s at time t to a new state $s' = T(s, a)$ at time $t + 1$, and a reward $r = R(s, a)$ to be collected by the agent as a result of its action. The function T is called the *transition function*, since it determines how each state progresses to the next state based on the chosen action. R is the *reward function* that gives the agent feedback on how well the action taken moves closer to the winning state. If the agent takes some action sequence $a_1, a_2, a_3, ..., a_n$, then the resulting total reward for the sequence is $A = R(s_1, a_1) + R(s_2, a_2) + \cdots + R(s_n, a_n)$. The goal is to find a policy function π that maps each state to the optimal action that should be taken to maximize the total reward A. This means that the optimal action a to take in each state s is $a = \pi(s)$. Once an effective policy function is determined, the network can follow it blindly to reach the optimal solution:

$$a_1 = \pi(s_1),\ s_2 = T(s_1, a_1),\ a_i = \pi(s_i),\ s_{i+1} = T(s_i, a_i),\ i = 2, 3, ..., n-1 \tag{1}$$

Following the policy function in this way guarantees the optimal reward and therefore the optimal solution, but finding an accurate policy function is rarely straightforward. Companies like Google DeepMind have found a new way to derive a policy function using a function $Q(s, a)$ called the *best utility function* or *best quality function*, defined as $Q(s, a) =$ the maximum total reward after choosing an action a in state s. Once $Q(s, a)$ is determined, the policy function can just return the action a_i that maximizes $Q(s, a_i)$. This best quality function can be determined using a recursive property of it demonstrated by *Bellman's Equation*:

$$Q(s, a) = R(s, a) + \max_{i=1,2,...,n} Q(s', a_i), \tag{2}$$

where $s' = T(s, a)$. The quality function $Q(s, a)$ can be approximated using a neural network N that takes the state s as input and returns a vector of *q-values* corresponding to the n possible actions: $q = (q_1, q_2, ..., q_n)$, where q_i approximates $Q(s, a_i)$ for each action a_i. This neural network provides a *derived policy*:

$$\vec{q} = (q_1, q_2, ..., q_n) = N[s] \tag{3}$$

$$j = \underset{i=1,2,...,n}{\arg\max}(q_1, q_2, ..., q_n) \tag{4}$$

$$\pi(s) = a_j \tag{5}$$

Using this derived policy π will give the optimal action to take at each state [4]. In this paper, the most effective way to train the neural network N in order to ensure that it converges to an accurate derived policy function will be investigated in order to develop an efficient algorithm to play games while simultaneously creating an environment to visualize and teach about machine learning as a whole.

2 Methods

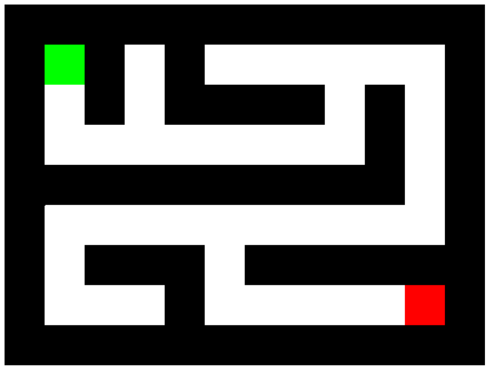

Figure 1: A simple maze designed using the prototype.

The process of designing this game began with a paper prototype. Many of the mechanics of the game, including the machine learning algorithm, could only be implemented when the game was run digitally, but the paper prototype still gave some insight into the interface and flow of the game. The next step was to create a simple digital interface to design levels. Since the mechanics of a platform game are relatively complicated and would impede testing the general concept of the game, the prototype began instead as a maze designing game (Figure 1). The interface found itself identical either way, but when designing a maze there is only one block available to be placed and the level is much less difficult for the algorithm to learn. The final game was written in Python with Pygame [5], but this prototype was written in Java with Processing, since it was much easier to get a graphical prototype up and running quickly in such a framework. This meant that the maze designed by the user had to be exported to a file that could then be imported by a Python script to train the algorithm with.

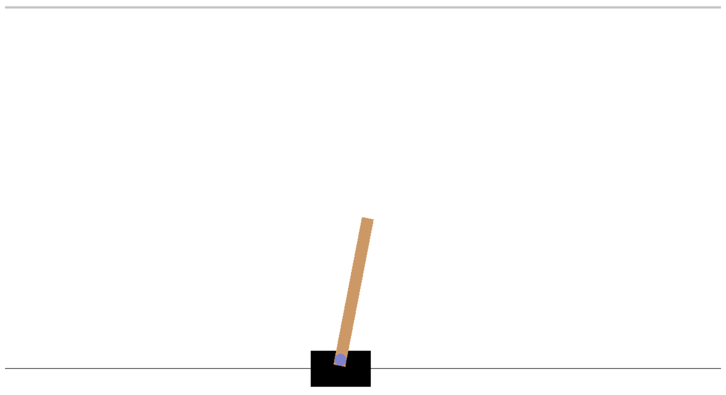

Figure 2: CartPole being played by the Q-Learning algorithm.

The Q-Learning algorithm was written in Python using PyTorch [6] and then tested on some sample mazes. Once it was properly functioning, the algorithm was generalized to work for more games than just a simple maze by testing it on a variety of games provided by OpenAI Gym [7]. The first game to test it on was "CartPole-v0" (Figure 2), a simple game in which the algorithm has to learn to move a cart left and right to keep a stick balanced on top of it. From there, the algorithm could be generalized to work on any game in the OpenAI Gym library. At this point once the platform game was developed, the algorithm could be easily modified to work on it and the game would be complete.

The next step was to write the game itself. Since the algorithm was written in Python, the most logical graphics framework to design the game in was Pygame. The level designer in the game looked as shown in Figure 3:

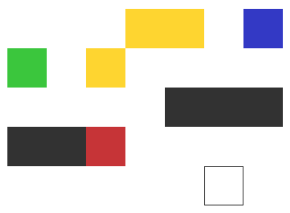
Figure 3: In-game level designer

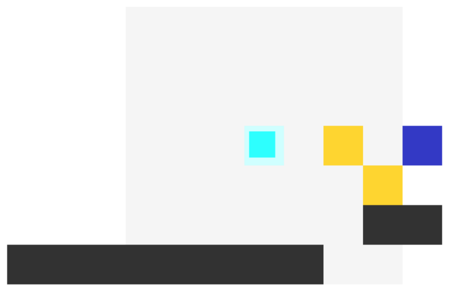
Figure 4: The algorithm training on a level

In Figure 3, the black blocks represent solid platform blocks, the green block represents the starting location of the character, the blue block represents the objective to get to, the red block represents "lava" that kills the character on contact, and the gold blocks represent "coins" that provide the character with extra points in order to guide the algorithm.

The player can design a set of levels, and then train the algorithm on said levels over the course of some number of games. Each game is an *epoch* of training for the neural network, composed of a sequence of *episodes*, where each episode is a taken action paired with its resultant state and reward that gets fed into the neural network as training data.

The algorithm itself was straightforward to adapt for this game. The most significant change was the shape of the input and output of the network in order for it to interact properly with the the game. Additionally, a program had to be written to interface between the algorithm and the game in the most efficient way possible. Firstly, the screen was split up into a grid of square cells, each the size of one block. Even with the screen being inputted as a low-resolution grid instead of as a high-resolution image of the screen, it was still a fairly large array of data to be feeding into the neural network in real time as the game was played. Also, because the position of the character on the screen did not align with the grid, it would need to be added as another element of the input in addition to the contents of the grid. In order to mitigate these issues, the way the input was passed into the neural network was entirely refactored. Instead of passing in the entire screen, only a square segment of the screen spanning 7x7 cells centered around the player was passed in each time, shown as a light grey square around the character in Figure 4. The center of this square was calculated as the cell that the character was most contained in, shown as a light blue cell behind the character in Figure 4. This way, not only would the input size be reduced, but the position of the player would also be passed in implicitly as the center of the grid, instead of as a separate input value. Another way the interface between the algorithm and the game was optimized was through an improvement of frame rate. To ensure training could run as efficiently as possible, Pygame's frame rate was uncapped so the limiting factor for the rate of training became how quickly the algorithm could process what action to take on each frame. Updating the screen every frame was also slowing down the training of the algorithm, so the graphics were reduced to only update every third frame in order to improve the speed without causing any noticeable drop in frame rate.

Once the algorithm was functioning well, it needed to be optimized through a series of tests with varying *hyperparameters* of the algorithm. Hyperparameters in this context refer to the parameters dictating how the algorithm itself functions, such as how likely it is to pick a random action or how many hidden layers and nodes it has in its neural network. This was tested over a series of trials on four and eight levels at a time. The exact four level designs used to test the algorithm are shown in Figures 5-8. In order to train with eight levels at a time, four more levels were added that were just reflections of the first four levels about the vertical axis.

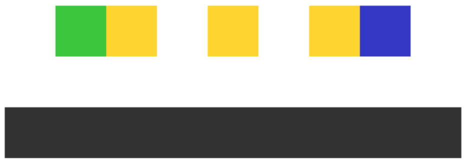

Figure 5: Level 0

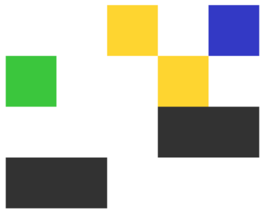

Figure 6: Level 1

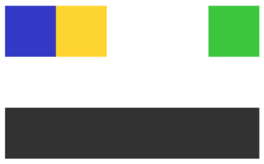

Figure 7: Level 2

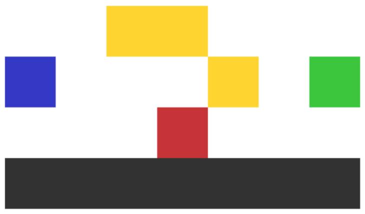

Figure 8: Level 3

3 Results

The Deep Q-Learning algorithm was tested for 1000 games in eight trials with different parameters. These trials were split into two groups, with the first group training the algorithm to learn to play four levels, and the second group providing the algorithm with eight levels to learn. The complexity of the neural network driving the algorithm was also varied between one or two hidden layers with the same number of nodes as the input layer. The complexity of a neural network is a measure of how many nodes it has in its hidden layers, with more nodes being more complex. In the context of this paper, a neural network with one hidden layer is said to be *simple*, and a neural network with two hidden layers is said to be *complex*. The way that inputs were passed into the neural network also led to a large amount of *useless* data. The character takes up less than a full grid cell on the screen, so the cell it is most contained in is given to the neural network as its location instead of the actual position of the character. This means that when the character moves, it may not move enough to change cells, meaning that its position that is inputted to the neural network will not change. This may cause the algorithm to learn that moving will occasionally not do anything. Since the neural network is trained using each individual move as data points, a lot of the data points that the network trains on will have nothing happening even if the neural network tried to move the character on that turn. For the duration of this paper, such a data point will be called a *useless* data point. A data point that is not useless is *useful*. The effect of this is tested by removing these useless data points in some of the trials.

In the first group of trials, the algorithm is tested on four levels. In each of the following graphs, The overall win rate over time is plotted in blue, and the win rate on levels 0, 1, 2, and 3 are plotted in orange, green, red, and purple respectively.

The first trial that the algorithm was tested on had four levels, one hidden layer, and all the input data included:

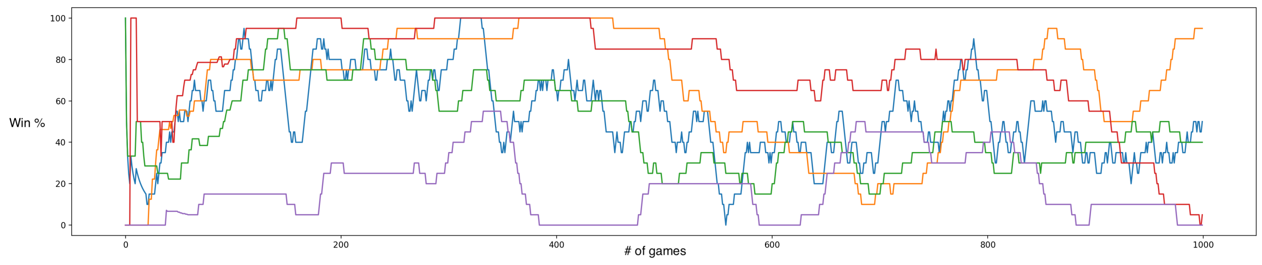

Figure 9: four levels, one hidden layer, all input data

The second trial tested the algorithm again on four levels with all input data included, but in this trial the neural network was tested with two hidden layers:

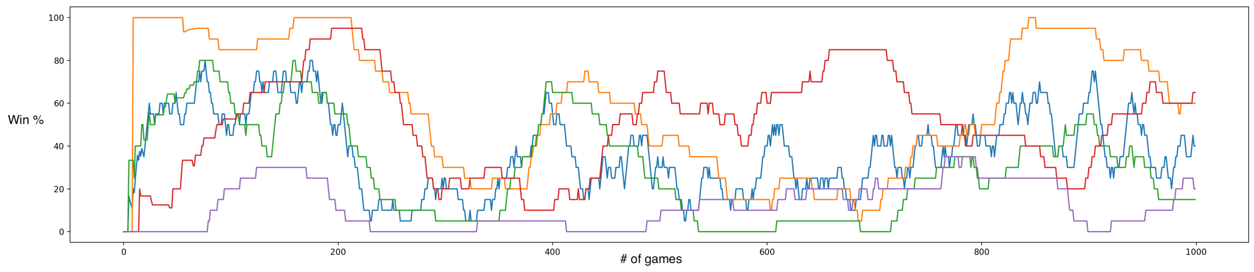

Figure 10: four levels, two hidden layers, all input data

The third trial trained over four levels with a single-hidden-layer neural network, and only the useful input data was included:

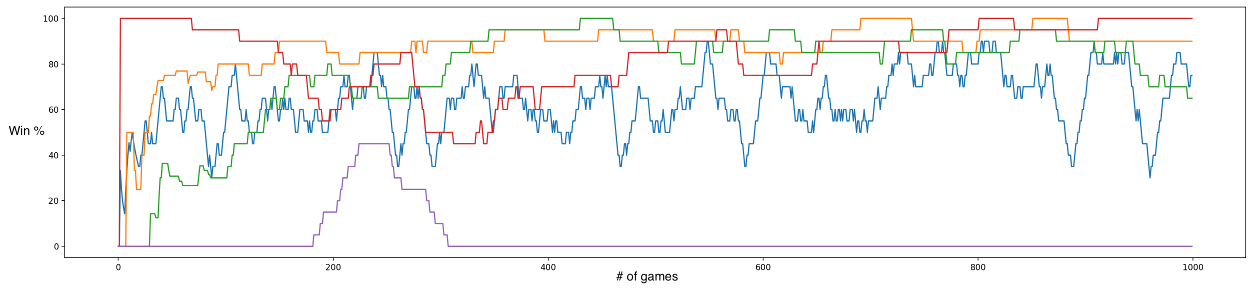

Figure 11: four levels, one hidden layer, restricted input data

The fourth and final trial over four levels trained a neural network with two hidden layers on input data restricted to only useful data points:

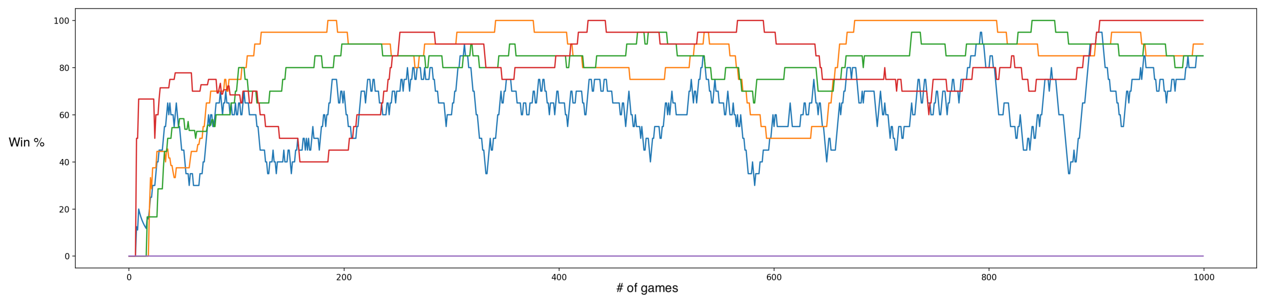

Figure 12: four levels, two hidden layers, restricted input data

The second group of four trials used eight levels to train each neural network, again varying the number of hidden layers and the included input data.

The fifth trial trained a neural network with one hidden layer over eight levels with all input data included:

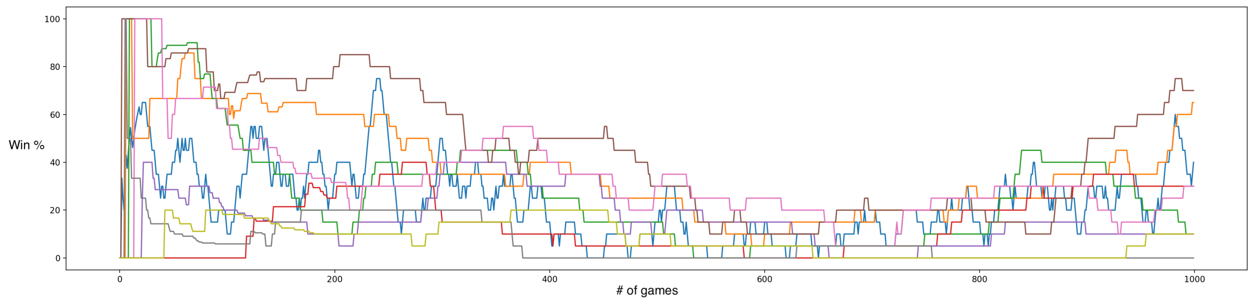

Figure 13: eight levels, one hidden layer, all input data

The sixth trial used eight levels to train a neural network with two hidden layers over all input data:

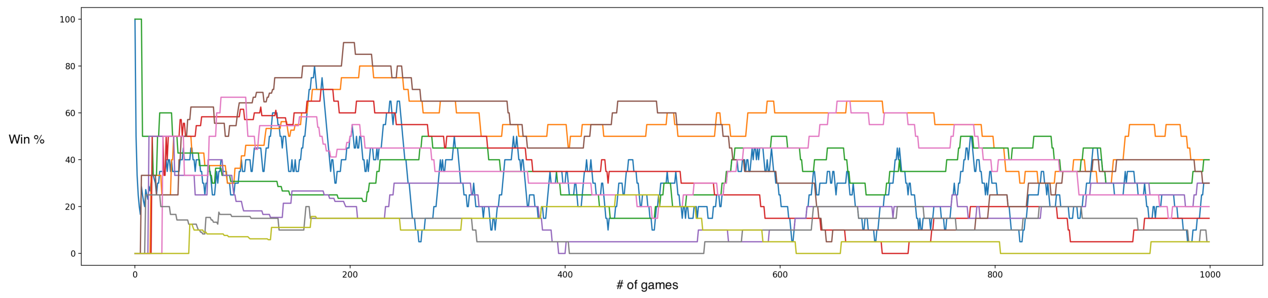

Figure 14: eight levels, two hidden layers, all input data

The seventh trial trained over eight levels with one hidden layer and input data restricted to useful data points:

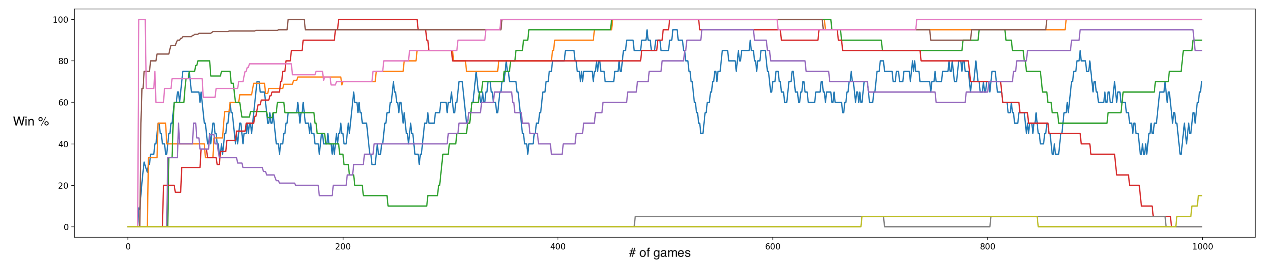

Figure 15: eight levels, one hidden layer, restricted input data

The eighth and final trial used eight levels to train a neural network with two hidden layers over restricted input data:

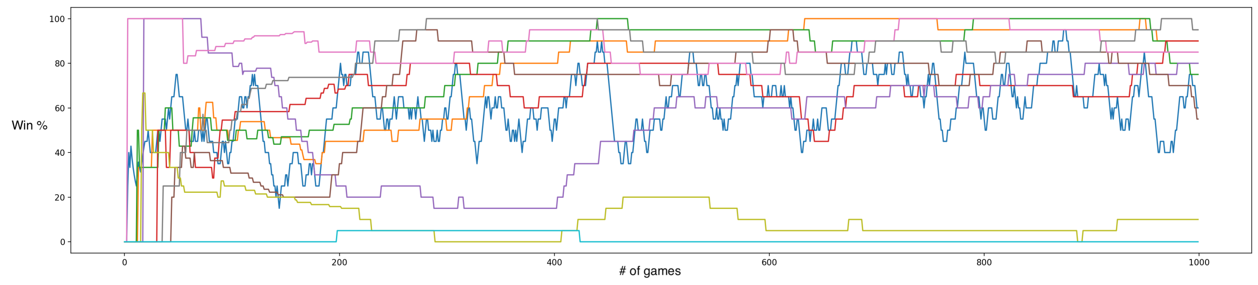

Figure 16: eight levels, two hidden layers, restricted input data

4 Discussion

Three factors affecting the functionality of the algorithm were tested: the number of hidden layers in the neural network, the amount and quality of input data provided to the network, and the number of levels to be learned at once.

In theory, a smaller neural network with fewer hidden layers would learn faster, while having more layers would allow the network to effectively learn more levels. Based on the results, it seems that different numbers of hidden layers are useful for different situations, though in general neural networks with only a single layer tend to perform better at learning this game. The first two trials compared the efficiency of one versus two hidden layers while being trained on four levels with all input data, and it is clear that the single-layered neural network (Fig. 9) performs better. They start fairly similarly, but over time the more complex network (Fig. 10) starts to regress due to the quantity of useless data being inputted to it. In trials three and four when this useless data is removed (Fig. 11,12), the single- and double-layered networks both greatly improve, but neither seems to perform better than the other. The strength of the more complex two-layered network is revealed in trials five and six, where the more complex network (Fig. 14) seems to better accommodate the higher number of levels than the simple network (Fig. 13) did. Curiously, the single-layer neural network in trial seven (Fig. 15) where the useless data is removed seems to perform better than the two-layer network in trial eight (Fig. 16) did. The simple neural network clearly struggled to learn eight levels when a lot of the data was useless, but it seems that providing it with only useful data allowed it to learn eight levels fairly effectively, despite its lack of depth in its hidden layers.

The amount of useless data being fed into the algorithm is clearly a major factor in its ability to learn. This is particular obvious across two sets of trials: First, between the second and fourth trials (Fig. 10, 12), a complex neural network learns to play four levels, and it significantly improves when the useless data is removed. Also, on trials five and seven (Fig. 13, 15), a simple neural network learns eight levels and undergoes a substantial improvement when the inputs are limited to useful data. This is likely because a simpler neural network is better suited to learn fewer levels, while a more complex one is better suited for more levels. When a simple neural network tries to learn many levels, like in trial seven, it already struggles to learn. When the useless data is also included in trial five, the delicate learning process can no longer function. The second and fourth trials have a similar problem with a complex neural network learning only four levels, and it has too much difficulty learning when useless data is included.

An important standard at which this project must be evaluated is its usefulness as an educational tool to visualize and teach about machine learning. While it certainly does do these things to some extent, the difficulty with which the algorithm learns to play the levels designed by the player often obfuscates some of the learning goals of the program. A simpler game would allow for less customization in the level design, but it would also lead to the algorithm learning more quickly and in a more straightforward manner that would likely improve its capacity as an educational tool.

5 Future Work

There is a lot of room for future development in terms of improving the algorithm, the structure of the game, and the game's overall usefulness in education. For future improvement of this game, one may consider a more sophisticated restructuring of the neural network driving the algorithm. While varied numbers of hidden layers were tested, these layers themselves were still simple linear fully-connected layers with the same number of nodes as the input layer and using the basic ReLU activation function. These aspects, as well as others, should be investigated in the future in order to further optimize the algorithm. By improving the rate at which the algorithm is able to learn, the game can be made more complex while still having the algorithm function. This would provide the player with a more sophisticated environment to experiment and test the algorithm in, allowing for deeper learning opportunities. Also, fleshing out the game and making it more interesting to play will motivate more people to play it and learn about machine learning, so it would certainly be a valuable future endeavor. Overall, this work shows that games can be an effective way of visualizing and teaching about machine learning, but more work must be done in optimizing this particular algorithm and improving the game overall in order to have a more effective learning tool.

6 Acknowledgments

This research would not have been possible without my mentor, Philip Tan, or my research supervisor, Sarah Verrilli. I would also like to thank Jenny “the Great” Sendova for being a wonderful tutor and guiding me in my research, all the Research Science Institute staff that helped the development of this paper, Basheer AlDajani for working alongside me for these four weeks of research, and everyone that helped edit including Gabriel Mintzer, Ersin Arioglu, and Osama Dabbousi, and the Center for Excellence in Education and their sponsors for giving me the great opportunity to conduct this research.

References

[1] I. El Naqa and M. J. Murphy. *What Is Machine Learning?*, pages 3–11. Springer International Publishing, Cham, 2015.

[2] M. Bowling, J. Fürnkranz, T. Graepel, and R. Musick. Machine learning and games. *Machine Learning*, 63(3):211–215, Jun 2006.

[3] A. Coudhary. A hands-on introduction to deep q-learning using openai gym in python. `https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/`, retrieved 2019-07-20.

[4] S. Zafrany. Deep reinforcement learning for maze solving. `https://www.samyzaf.com/ML/rl/qmaze.html`, retrieved 2019-07-20.

[5] P. Shinners. Pygame. `http://pygame.org/`, 2011.

[6] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[7] OpenAI. Getting started with gym. `https://gym.openai.com/docs/`, retrieved 2019-07-20.