Reinforcement Learning as a Method of Dynamic Difficulty Adjustment in Video Games

Samir H. Patel
[1]University of Texas at Arlington
701 S Nedderman Dr, Arlington, TX 76019 shp6706@mavs.uta.edu

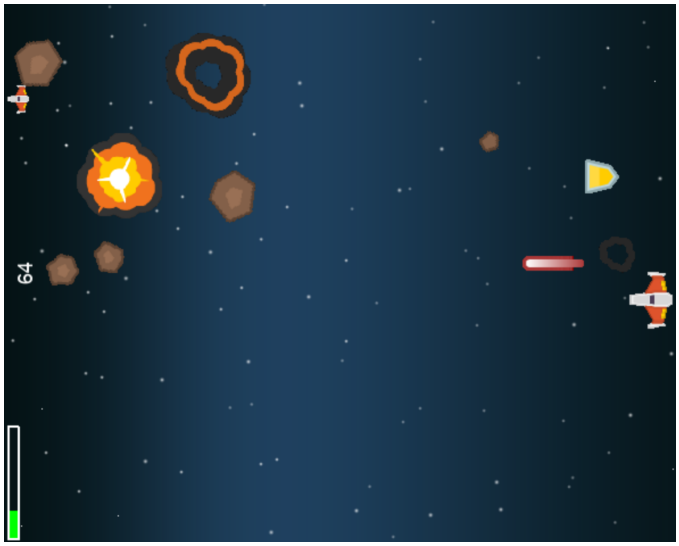

Figure 1: Screenshot of simple arcade style shoot 'em up in which we integrated reinforcement learning

Abstract

Reinforcement learning is a developed technology with lots of room to grow. Many of the current testing grounds for new reinforcement learning algorithms and techniques involve gaming. Reinforcement learning has shown great success in beating games and even human adversaries in games. In an alternate field, over the last 20 years viable methods for performing dynamic difficulty adjustment has also been a topic of research in which reinforcement learning has not been a part of. In order to rectify this we present an evaluation of reinforcement learning as a method of performing dynamic difficulty adjustment in a video game. Reinforcement learning would allow the developers to implement dynamic difficulty adjustment into their games without the need to code in low level details about changes in player skill level instead allowing the reinforcement learning agent to make these decisions itself. We will make this assessment through the use of a quantitative evaluation via a simple experiment pitting two version of the same game against each other, one with a reinforcement learning agent and one without. The viability of this method could open a whole new avenue for commercial application of dynamic difficulty adjustment, as well as as a new way to test reinforcement learning techniques.

1 Introduction

Currently there is a vast area of research focused on testing new reinforcement learning algorithms and techniques in video games. Many of these feature algorithms that teach agents to play games (Bellemare et al. 2012; Hu et al. 2020). Another subsection of research is focused on using reinforcement learning to beat humans in various games (Silver, Huang, and Maddison 2016; Ye et al. 2020). Many of these research efforts are important for pushing the boundaries of reinforcement learning, but are not focused on improving the games themselves.

Meanwhile in the world of video game based research there is research being done to find and test new techniques for dynamic difficulty adjustment (Hunicke and Chapman 2004; Sarkar and Cooper 2020; Xue et al. 2017; Zook and Riedl 2021). Dynamic difficulty adjustment is the process of changing the difficulty of the game based on the player. Dynamic difficulty adjustment has been a research topic for years and has many different viable methods to implement it.

However, due to the proliferation of reinforcement learning in the last 10 years its seems like the time has come to evaluate whether or not reinforcement learning is a viable technique for implementing dynamic difficulty adjustment in games. Additionally, this serves as another technique for testing reinforcement learning as well. So far most research involving human interaction with reinforcement learning agents involves getting the agent to defeat the human player (Silver, Huang and Maddison 2016; Ye et al. 2020). Now we want to expand that to see if the agent can sense more minute human patterns and see if the agent can adjust accordingly.

This paper presents a game with an implemented reinforcement learning agent. The goal of this agent is to detect changes in player skill and adjust the difficulty of the game accordingly to enhance player enjoyment of said game. This technique would allow game developers to implement dynamic difficulty adjustment without having to chart out and define every change in skill. Instead by implementing reinforcement learning they can allow the agent to make these decisions for itself. Overall this paper will contribute:

- An assessment of the viability of reinforcement learning as a technique for dynamic difficulty adjustment.

In order to make our assessment we will be doing a quantitative assessment of our agent by having a base version of the game without the reinforcement learning agent and testing them both against bots. By measuring our own metric for player engagement we will be able to make a cursory evaluation of the promise of this method.

Next, we will discuss some related works within the fields of reinforcement learning and dynamic difficulty adjustment. From there we will look at a detailed problem solution which shows the tools used, a breakdown of our implementation, the methodology of our evaluation, and the results. Finally, we will conclude by discussing the results, analyzing what they mean, and considering some future avenues to expand upon this work.

2 Related Works

The work this paper presents involves two different facets of current AI research. The first of which is reinforcement learning, specifically in the area in which it interacts with games. The other is dynamic difficulty adjustment which is a research problem in the realm of AI and video games.

2.1 Reinforcement Learning and Gaming

Games whether they be digital or physical remain a hotbed for reinforcement learning research. When it comes to physical games in 2017 DeepMind's, AlphaGo, became the first AI to defeat a human player in the Chinese board game Go (Silver, Huang and Maddison 2016).

In the digital domain a popular testing strategy for reinforcement learning is Atari games. In their work Bellemare et al., created a framework capable of testing and benchmarking reinforcement learning algorithms and feature selections against various Atari games (Bellemare et al. 2012). Meanwhile, other research has been pushing the boundaries further such as Hu et al., who introduce an alternative to deep q learning with the multi-view deep attention network and achieving a much more stable model than an alternative deep q network for playing Atari games(Hu et al. 2020).

Additionally, research in this area has moved to more complex game types such as the work of Ye et al., authors used deep reinforcement learning to train AI to play multi-player online battle arena (MOBA) games. The authors showed success having tested their model against professional players and achieving a 99.81% win rate against them (Ye et al. 2020).

Our project sets its self apart from these previous efforts via our goal. While some of the work was related to training AI to play games (Bellemare et al. 2012, Hu et al. 2020), our work is meant to work with a human player. In conjunction, while the others do involve humans playing against AI (Silver, Huang, Maddison 2016; Ye et al. 2020), these work aim to have the AI defeat the human. The goal of this project is to create a reinforcement learning model that will temper itself based off of the abilities and skill of the player.

2.2 Dynamic Difficulty Adjustment

Another area of research related to ours is dynamic difficulty adjustment. Dynamic difficulty adjustment aims to adjust the difficulty of a game in order to math the skill level of a player.

Sarkar et al., explored the differences between two methods of performing dynamic difficulty adjustment and how interweaving or using them alone can affect balancing the game (Sarkar and Cooper 2020). In other works. Zook et al. introduced a new method of detecting changes in player skill level for dynamic difficulty adjustment by introducing tensor factorization (Zook and Riedl 2021). In their work Xue et al., used probabilistic graphs to model long term player skill progression in order to define more long term changes to game difficulty (Xue et al. 2017). Hunicke et al. uses statistics about player health and inventory levels in oder to determine changes in difficulty (Hunicke and Chapman 2004).

All of these papers involve various methods of performing dynamic difficulty adjustment. Our project aims to expand upon these works by exploring the efficacy of reinforcement learning as a method for dynamic difficulty adjustment.

3 Problem Statement

In order to expand the field of dynamic difficulty adjustment we propose integrating a reinforcement learning algorithm into a game to tests it's efficacy as a method of dynamic difficulty adjustment. We will implement reinforcement learning using Pythons pytorch library into a simple shoot 'em up game (kidscancode 2017). We expect the game to be able to adjust its difficulty in such a way that the player is challenged whilst still being engaged than when compared to the regular version of the game.

By engaged we mean the player should have a high average score per second while playing the game. By challenged we mean the agent should maintain some consistency in the player's average score per second no matter their skill level. We will be using a quantitative evaluation in order to analyze this by tracking the players score per second across both versions of the game.

4 Problem Solution

In order to explore this problem we must first define what reinforcement learning is. According to Russel and Norvig, reinforcement learning is a system in which an agent interacts with its environment, gains rewards from those interactions, and uses those rewards to determine future actions (Russel and Norvig 2021). For the rest of this section we will first explore the toolset that we will be using to implement the reinforcement learning algorithm. Then we will look at the setup of the program. After this we will discuss the evaluation method to test the efficacy of our system. Finally, we will take a look at and discuss the results of our evaluation.

4.1 Tools

In order to implement a reinforcement learning system we need to decide three things:

- A programming language

- A game to implement the algorithm in
- A reinforcement learning library: either original or off the shelf

The obvious solution to this was to use python. Python has various reinforcement and machine learning libraries to choose from. Additionally, python also has game development libraries with many open source examples.

Pygames is one of the most popular game development library for python. For this project we decided it best to stick with a simple game, after looking through various examples of open source games we decided on a simple shoot em' up game developed by the kidscancode organization for their pygame tutorial series (kidscancode 2017). The simplicity of the game is a benefit because it is easy to implement the needed changes and also the state and action space is sparse, meaning there is not a lot we need to account for in terms of states and actions. We will discuss the specifics of the changes made in the next section. The game features the player strafing from left to right whilst trying to shoot oncoming meteors before they damage you enough that your shields lower and you die. The game can also spawn powerups to help the player, one gives the player more health, whilst the second powerup allows the player to shoot two shots rather than one. The objective of the game is to gain a high score. The current 'AI' in the game is simple and has the meteorites spawning with random speeds as well as randomly chosen powerups spawning randomly.

Finally, for the reinforcement learning library we chose to use an off the shelf solution. This one done mainly to speed up development time and focus on designing the algorithm instead. Ultimately we chose to go with pytorch, a deep learning library developed by Facebook, since it implements many optimization algorithms itself. This leaves us with more time to worry about the design decisions such as choosing how to represent our state space or what actions our agent will take, rather than worry about implementing low level code for the various algorithm involved in a reinforcement learning model. The specific optimization algorithm and the overall structure of the reinforcement learning algorithm we be detailed in the next section

4.2 Program Setup

In order to setup the program for reinforcement learning we must setup some classes to help implementation. Firstly, we need a class to hold the agent, handle interactions between the agent and the model, and makes decisions on when to start training. For our implementation we will call this the Agent class. Secondly, we need a class that implements the model itself from pytorch Module class. Since pytorch is a deep learning library and implements Q-learning we will call this class DeepQ. Finally, we need a class that handles the training of the model, we will call it Trainer. Before we will dig a little deeper into each of these classes let's discuss the reward.

Reward An important part of any reinforcement learning algorithm is defining the reward function. We need to design this to reward the algorithm for achieving the goal. But, what is our goal? As stated in the previous section the point of the game is for the player to achieve a high score. Since we want to challenge the player while maintaining their enjoyment of the game it seems logical to reward the algorithm based off the player score per second. We could base it purely off the players score, but we also want to make sure the player is constantly being challenged. To implement this we simply award the algorithm the score per second if it is above one, and the negative second per score otherwise. The pseudocode for this can be found in figure 3.

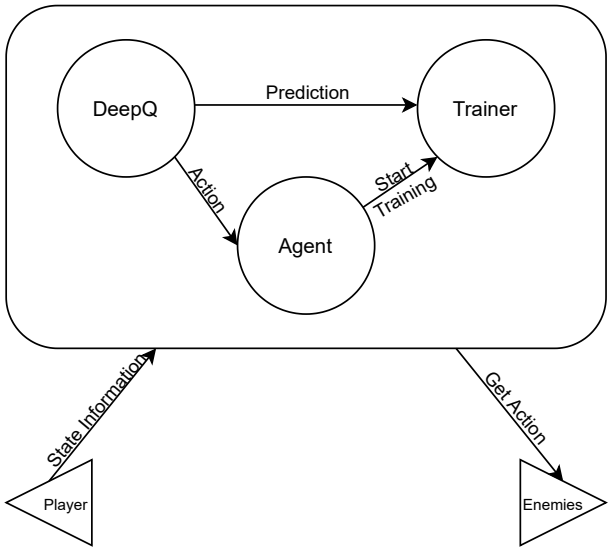

Figure 2: Diagram showing the layout of the program and interactions between the classes

```
if score per second > 1
    reward = score / second
else
    reward = -second / score
```

Figure 3: Reward Function

4.3 Agent

As stated before the Agent class handles all of the interactions between the game and the model. This means it retrieves state information for the model, calls on the model to predict new actions and decided when and how training occurs. Like with any AI problem in order to implement this into the game we need to discuss the state space and the actions the AI can take.

States Our state space should constitute information that we believe the model would find relevant towards achieving our goal. Since our goal is for the game to be enjoyable to the player, the state space consists of information regarding the player. In the game the player has only a few actions they can actually take they can move to the left, to the right, and shoot.

From these actions it is important to take the direction the player is moving as it can indicate to the game where to spawn meteors next. Additionally, we want to have information as to the players health since the agent can also spawn health powerups. Additionally, we want to include where on the map they currently are. Since the player can only move in the x direction we do not need to take into account their position on the y axis. This leaves us with a final state representation of $\{playerhealth, playerlocation, playerxdirection\}$.

Actions There are few different ways that we can go about adjusting the difficulty of the game. Firstly we can adjust the speed in both x and y direction of incoming meteors currently in the game this is done via random selection between a set of minimum and maximum values for each. Next, we can control the flow of powerups and the type of powerup, of which there are two, the player receives. Since the x and y speed are bounded by a minimum and maximum we chose to allow the agent to adjust the boundaries rather than just change the speed.

Possible actions for the speeds are plus one, minus one, and no change. Since we are adjusting the minimum and maximum for these values the program still selects the speed randomly, just between those values. While this is not ideal this was done to keep all meteors from moving at the same speed without unnecessarily bloating the output layer of the network. Without this we would need a separate x and y velocity action for each of the seven meteorites which would require an output layer with a size nearing 9000.

For the two powerup controls we can simply use a boolean one or zero. This leaves us with six variables in any possible action and a total of 324 possible actions for the agent to take. Additionally, the agent facilitates exploration via random choices. For the first 50 games there is a chance that any action taken is random. According to Russel and Norvig, forcing the agent to not always make the greedy choice can help by making it less likely the model will end up not finding the optimal policy (Russell and Norvig 2021).

Training While the agent does not handle the low level details of training it does handle how the it is trained. After every frame the agent starts a single training step involving the last move and next move. Additionally, at the end of each game the agent begins the process of long term training in which the agent looks over a random sample of multiple previous moves. For the long term training we do 1000 iterations of training. If there have yet to be 1000 moves made we take all the moves that we have and use them. Note the memories of previous moves carries over from game to game. The specifics of the training algorithm will be discussed later.

4.4 DeepQ

The DeepQ class implements the base class of pytorch, nn.Module, which is the base of any reinforcement learning model in pytorch. Since pytorch is implemented using neural networks our initialization function takes in the size

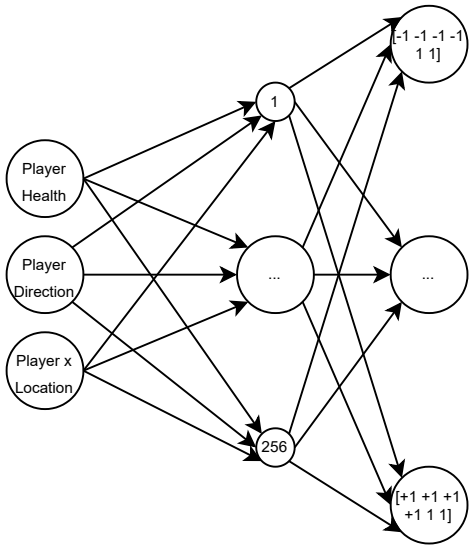

Figure 4: The Neural Network for our model

of the input layer, the size of the hidden layer and the size of the output layer.

For our algorithm the input layer is equivalent to the number of variables in any given state, as discussed earlier this is three. For the hidden layer we may select a value we chose 256. Finally, for the output layer we want this number to be equivalent to the total number of possible actions, 324. Figure 4 visualizes the connections between these layers.

The reason for having the number of output states equal to the number of total number of possible actions is so when our model outputs its predictions we can simply take the index of the output node with the largest value and use it to access a constant array that includes all the possible actions giving us our next action. The initializer creates a linear transform between the input state and the hidden state. As well as between the hidden state and output state. These transforms will be used to compute our forward function.

Forward Function In addition to the initializer we must also implement nn.Module's forward method. This represents the forward function of our model. To compute this we take a value x and send it through the first linear transform between the input and hidden layer. Then we get the rectified linear unit of that value and send it through the second transform between the hidden and output layer.

4.5 Trainer

The training algorithm implements a single iteration of the training loop. It also handle back propagation and the optimization of the parameters for which we use pytorchs implementation of the Adam algorithm. Back propagation is also handled by pytorch.

When called the algorithm takes the state, action, current reward, the next state and whether or not the game is over. All of these are converted into tensors which are the base format in which pytorch takes data. Then we call the model

to make a prediction based on the current state, this prediction is used as a target value for the training. Then we iterate through the previous states making a new prediction based of the Q learning rule as described by Russel and Norvig (Russel and Norvig 2021).

After this we call the loss function for which we use a simple mean squared loss between the predicted Q value and the value calculated via the Q learning rule and start back propagation all of which pytorch handles. Finally, we call our optimization algorithm to take a single step.

Adam Pytorch implements a variety of methods for optimization. For our program we chose to use their implementation of Adam. Adam computes first-order gradient-based optimization. Our reasoning for choosing Adam was due to its high level of efficiency and because it works well with sparse gradients (Kingma and Ba 2015). The fact that Adam works well with sparse gradients is important to us since the speed of the meteors is randomly generated we feared the model might have trouble tuning its weights.

5 Evaluation

While the ideal way to test our implementation would be qualitatively, through a user study, due to time constraints we alternatively chose to do a quantitative test. In which we test our implementation against the regular version of the game. Through both we measure the output of the score per second and graph it for a total of 100 games. We also keep track of the overall average score per second and graph it as well.

5.1 Methodology

For the evaluation we had to make some edits to the way the game functions in both the regular version and our version. For simplicities sake we will refer to the version of the game with an integrated reinforcement learning model as RL and the version without the reinforcement learning model as NRL.

First, we lowered the number of player lives to one instead of three. Additionally, we increased the number of frames per second from 60 to 480 in order to decrease the amount of time it takes to run. To ensure we could calculate the score per second correctly we count the number of frames as a float and divide it by 60. Ensuring that the number of seconds is equivalent to 60 frames of gameplay.

Finally, to facilitate the testing we replaced the player with a bot. The bot simply strafes from left to right and shoots on every nth frame. The n is decided randomly at the start of the game either choosing one, two, or three. Put simply the bot shoots either every frame, every other frame, or every third frame. We did this to simulate players with lower skill levels, since a bot that shoots less is less likely to hit something and gain high score.

Since the goal of the game is to get a high score we can track that as a measure of how well the bot is doing. In order to measure the engagement the bot has and ensure it is constant throughout the game we divide the score by the number of seconds the games been running. This is because if the bot can maintain a high score per second we can assume that overall it is shooting enough of the meteorites and is properly engaged in the game. As stated earlier we track this for around 100 games for both NRL and RL and keep track of the overall average score across all the games.

While using a bot is not entirely ideal we can still get an idea of how well the RL versions agent can handle changes in behavior and adjust itself to ensure the maximization of player challenge and engagement.

5.2 Results

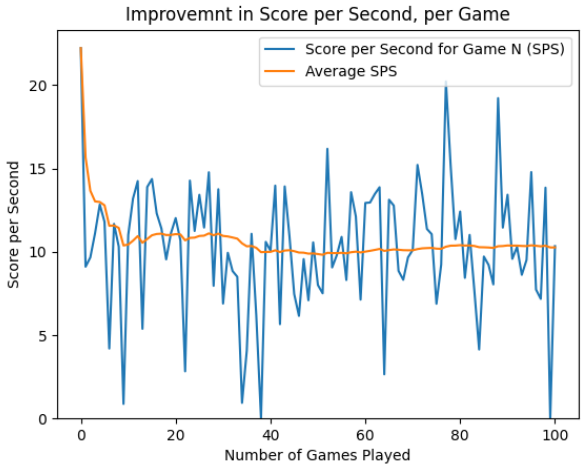

Figure 5: Performance of the bot without the reinforcement learning agent

Figure 5 shows the results for the NRL across 100 games. The blue line indicates the average score per second for every individual game played. The orange line indicates the running average of the score per second of all the games played. As expected the results are inconsistent due to the fact that the enemy AI relies solely on random events. For the NRL the bot averages a 10.86 score per second ratio the standard deviation and variance for the NRL can be found in table 1.

Figure 6 shows the results for the RL across 100 games. Unexpectedly the results are wildly inconsistent we can see in table 1 that the variance of the average score per second is over 300. While some consistency issues were expected since elements of the agent in the RL version are still randomly selected, the consistency of the RL falls well behind that of the NRL.

All of this indicates lack of stability in our reinforcement learning model. All of that being said the overall average score per second for the bot in the RL is quite a bit higher than that of the bot in the NRL with a 36.58 score per second ratio. As with the NRL the rest of the statistics for this bot can be found in table 1.

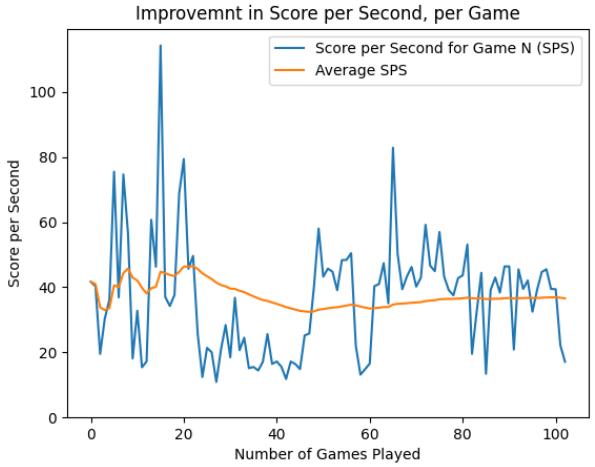

Figure 6: Performance of the bot with the reinforcement learning agent

	Non-RL	RL
Mean	10.26	36.58
Standard Deviation	3.89	17.75
Variance	15.16	315.11

Table 1: Basic statistics for both the Non-RL and RL version of the game

6 Conclusion

6.1 Limitations

As stated in the previous section the RL did not perform as expected. From table 1 we can see that the standard deviation and the variance for the data set are very high showing our results to be inconsistent and unstable. There could be a few reasons for this, one problem could be the actions themselves.

To avoid overly repetitive gameplay we had the agent change the minimum and maximum bounds for the speed of the meteorites rather than changing the speed of the meteorites. What this means is the speed of any meteorite is ultimately random but chosen from a specific range that the agent can change. The alternative would have had every meteorite move in a singular direction at a singular speed so the tradeoff seemed beneficial at the time. We did try to account for this by using the Adam optimization algorithm since it works well with sparse gradients (Kingma and Ba 2015).

Another issue could be that the model just was not being fed enough state information to make any conclusions about the correct next action. We only included three variables in the state. It could be possible that including different variable, maybe the rate the player shoots at or time in-between successful shots from the player might help. Or perhaps it could be a mixture of this and the randomness compounding on each other.

Alternatively, the issue could stem from a lack of available actions for the agent to take. The AI could have also been coded to control the position in which an individual meteorite spawns. Much like with controlling the speed of the individual meteorites this was not integrated in order to to keep the size of the output layer down. This would have the additional downside of introducing another level of randomness to the system.

6.2 Discussion

Setting aside the model lack of stability it was not a total loss. If we dig into the datasets a little deeper we find a couple of interesting facts. As can be plainly seen in figures 5 & 6, the overall average score per second is much higher for the RL than the NRL variant. In fact the individual score per seconds for each game in the RL set never dip below the overall average for the NRL set. Additionally, in 70% of cases the RL variant performed better than the best individual game for the NRL variant. So while the RL agent did not perform exactly as expected, we think the numbers indicate that at the very least reinforcement learning is method for dynamic difficulty adjustment that should be looked into more in the future.

Since we defined the point of dynamic difficulty adjustment as being the ability of the agent to both keep a player adequately challenged and engaged with the game. We can say the RL implementation succeeded at providing engagement since the score per second was consistently higher than the NRL version. But, since the variation was so dramatic it would seem the NRL failed at providing an adequate challenge to players of all skill level.

6.3 Future Work

In the future we believe that there are few things that should be worked on. Specifically for this project it remains to be seen if the implementation could be improved upon, it may be that fixing either or both of the issues highlighted above could lead to more promising results.

Additionally, running user tests to see if the any of this is noticeable to the a human player is important. While the numbers indicate a significant increase in player score per second it could be the introduction of a human being over a bot with a set pattern could lead to different much different results for the RL.

Finally, I also think in the future we should see reinforcement learning used as a method of dynamic difficulty adjustment for a more complex games to see if the method can scale.

6.4 Summary

In summary, the RL agent did not work entirely as intended. That being said it did not fail in every regard and still shows promise as a method of dynamic difficulty adjustment. Furthermore, there are plenty of avenues through which continued work on this agent can perhaps improve upon its design. More importantly, there are a plethora of opportunities to build upon this work.

7 References

Bellemare, M. & Naddaf, Y. & Veness, J. & Bowling, Mi. (2012). *The Arcade Learning Environment: An Evaluation Platform for General Agents*. Journal of Artificial Intelligence Research. 47(1), 253-279.

Hu, Y., Sun, S., Xu, X., & Zhao, J. (2020). *Multi-View Deep Attention Network for Reinforcement Learning (Student Abstract)*. Proceedings of the AAAI Conference on Artificial Intelligence, 34(10), 13811-13812.

Hunicke, R., and Chapman, V. 2004. *AI for dynamic difficulty adjustment in games*. In Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence.

kidscancode. (2017). *schmup-14.py*. Github. https://github.com/kidscancode/pygame_tutorials/blob/master/shmup/shmup-14.py.

Kingma, Diederik & Ba, Jimmy. (2015). *Adam: A Method for Stochastic Optimization*. International Conference on Learning Representations.

Russell, Stuart & Norvig, Peter (2021). *AI: A Modern Approach*. Hoboken, New Jersey: Pearson.

Sarkar, A., & Cooper, S. (2020). *Evaluating and Comparing Skill Chains and Rating Systems for Dynamic Difficulty Adjustment*. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 16(1), 273-279.

Silver, D. & Huang, A. & Maddison, C. et al. (2016). *Mastering the game of Go with deep neural networks and tree search*. Nature 529, 484–489.

Ye, D., Liu, Z., Sun, M., Shi, B., Zhao, P., Wu, H., Yu, H., Yang, S., Wu, X., Guo, Q., Chen, Q., Yin, Y., Zhang, H., Shi, T., Wang, L., Fu, Q., Yang, W., & Huang, L. (2020). *Mastering Complex Control in MOBA Games with Deep Reinforcement Learning*. Proceedings of the AAAI Conference on Artificial Intelligence, 34(04), 6672-6679.

Xue, Su & Wu, Meng & Kolen, John & Aghdaie, Navid & Zaman. (2017). *Dynamic Difficulty Adjustment for Maximized Engagement in Digital Games*. In ¡i¿Proceedings of the 26th International Conference on World Wide Web Companion (WWW '17 Companion). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 465–471.

Zook, A., & Riedl, M. (2021). *A Temporal Data-Driven Player Model for Dynamic Difficulty Adjustment*. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 8(1), 93-98.