Better First Person Shooter Bots Via Reinforcement Learning

Cardiff University | CM3203 | 40 Credits

Author
Gregor Webster
Supervisor
Frank Langbein

Abstract

This project's focus was the building of better FPS bots through the use of reinforcement learning, and specifically the area of sample biassing as a means of achieving that improvement. To find out if sample biassing was a useful improvement, data needed to be gathered on both unbiased and biassed programs.

To that end, a simple reinforcement learning algorithm was adopted and adapted to generate the data needed. After six runs, three unbiased and three with varying levels of bias, sufficient data had been gathered to establish the basic effects of biassing. The goal of the project was to establish if reward biassing was a useful tool for FPS bots, either in terms of providing an overall improvement, or simply by improving some desirable aspect of the bot, even if it was detrimental to other areas. Thus, for the project to be a success the baissed data would have to show that biassing could provide at least one useful benefit that unbiased sources could not provide.
After analysing the data, the biassing proved to have a mix of positive and negative effects on the resulting bot, notably it had adopted a preference for high-risk high-reward strategies which lead to the program having a greater number of high scores, but a lower average score. This could be adapted into a tool which would suit certain problems better than an unbiased one.

Overall, the project was deemed a success, but more data is still required to help quantify the effects at differing levels of bias.

Acknowledgements

I would like to thank my supervisor, Frank Langbein, for his exceptional support and guidance over the course of this project and my group project last year. Without him I would have never stopped trying to isolate bugs, and thus missed the bigger picture of the project.

I would also like to thank my mother and father for taking the time and effort to read over my report, giving me ample feedback on flow and greatly accelerating the process of polishing this report.

Furthermore, I would like to thank Mathew, who gave up four hours to read me my paper, which proved invaluable for the editing and polishing process.

Table of Contents

Table of Figures

Table of Abbreviations

FPS
: First person shooter – ii, 1, 3, 4, 8, 30

PvP
: Player versus player – 1

PvE
: Player versus environment – 1

PvPvE
: Player versus player versus environment – 1

POV
: Point of view – 3

DQN
: Deep q network – 5, 6, 7, 9, 10, 11, 12, 13, 14, 28, 30, 32

GPU
: Graphics processing unit – 5

CPU
: Central processing unit – 5

NaN
: Not a number – 14

Inf
: Infinity – 14

R2D2
: Recurrent Replay Distributed DQN – 6

ape-X
: Distributed Prioritised Experience Replay – 6

PPO
: Proximal Policy Optimization – 6

API
: Application Programming Interface – 5

GUI
: Graphical User Interface – 4

DirectML
: Direct Machine Learning – iv, 5

1 Introduction

Since the beginning of the industrial revolution, humans have been automating the boring and difficult aspects of our lives to free us to do what we want. What started with very specific and repetitive tasks, like milling, spinning, and weaving quickly turned into things no-one could have imagined. Then came electricity and it revolutionised the world by making it possible to power machines from hundreds or thousands of kilometres away, and automation was suddenly all around us in our everyday lives. Then came computers, initially cumbersome and expensive, but amazing for their ability to perform calculations and arithmetic faster than any human. Finally, we come to the most recent major expansion in automation, machine learning.

Machine learning allows us to step back from figuring out how to solve a problem and instead let a computer. This is a huge step forward, and it did not take long before machine learning could solve problems so vast and data intensive that no human could accomplish them. The only remaining area we have yet to automate is identifying problems, and that might not be a good idea (what if some algorithm decides humans are a problem?).

The pairing of machine learning and videogames in general is still a relatively unexplored area, and reviews of the current state of the machine learning video game pairing have called for greater use of machine learning in video games (Edwards et al. 2010) both to improve the bots that players play with in order to increase the quality of player experience, and because video games present an untapped set of environments for testing new machine learning concepts (François-Lavet et al. 2022) emphasis on section 9.1. Specifically, first person shooters present an opportunity to provide new visual learning machines with a sandbox for testing without the potential consequences of placing a visual learning machine in the real world. Obviously, video games cannot perfectly simulate the real world, but every day they get closer, and at a certain point they may be able to provide realistic enough simulations for a seamless application of its learning to the real world.

The problem I chose “better first person shooting bots via reinforcement learning” is not some revolutionary step forward, but it is a good place to start. Better FPS bots is a problem that keeps changing, and so it is impossible to nail down any single solution. Individual FPS’s focus on different key elements, some are PvP focused while others are PvE or even PvPvE. Some are focused on slow and methodical gameplay like stealth and assassination, while others are fast paced ‘run and gun’ style games where staying still is a bad idea. In short, there is no one solution to the problem, so instead we

turned to reinforcement learning. With reinforcement learning, we can build a program that, given a few rules that assign a value to each action based on the outcome of taking that action and sufficient data on its environment, can learn to play any game reasonably well. After some alterations, the focus of the project shifted towards research, so a better description of the project could be “Does biassing sample data make better first person shooter bots via reinforcement learning”. After collecting the data and performing some analysis, the results of this project suggest that under certain circumstances, it does.

1.1 Report Breakdown

The following is a quick breakdown of the sections in order and a brief description of what is covered in that section.

1. Introduction:

Introduces the problem and explains the report.

2. Background:

Elaborates on the context of the problem, specifically what is required to create a solution, including examples of how others approached the problem.

3. Methodology:

Discusses the evolution of the project and what it eventually became, covering each major turning point and focusing on alterations that had a major impact on the end result.

4. Implementation:

Focuses on the alterations to the base code required to create a program capable of providing the data necessary to determine if the change made is beneficial.

5. Results:

Describes the tests run on the program and the data obtained from both the biassed and unbiased sampling in addition to covering the trends in the data and speculating on the causes. Ending in a group of collective conclusions based on the notable differences between all datasets.

6. Evaluation:

Evaluates the overall performance of the biassed sampling V.S. unbiased sampling and reviews the potential use cases for biassing. Also reviews the project overall and how successful it was.

7. Future Work:

Focuses on both the extensions available for this project and why they might be worth attempting, and other related areas in which some improvements might be possible.

8. Conclusion:

Quickly overviews what has happened in the report and the overall conclusion on the problem.

9. Reflection on Learning:

Takes a step back from the report to reflect on myself and what I have learned, both in terms of the skills I have learned and applied, and the personal growth I have experienced.

2 Background

2.1 Requirements

A machine learning program has two major requirements, knowledge of the environment, and a series of rewards for causing certain changes in that environment. Without knowledge of its environment, the program lacks context. In short it is unable to learn what to do in each situation because it does not know it is in that situation. Without a series of rewards for causing certain changes in the environment the program cannot learn as it has no frame of reference for beneficial or unbeneficial actions.

2.1.1 Environment

The environment of an FPS is mainly communicated through visual feedback from the POV of the controllable entity, and thus the program will need access to that visual feedback so it can understand its environment. This can be simplified slightly, as certain visual feedback consists of numbers displayed on a screen (Ammo, Health, Stamina…), which can just be passed to the program directly without the program having to figure out how to read, but the most important elements of the visual input will still be too chaotic for standard machine learning and require layers of evaluation before a meaningful decision can be reached. This is best accommodated with deep learning, a form of machine learning which, through the use of a many-layered network of interconnected nodes similar to neurons, useful information can be extracted from unstructured data and parsed into a meaningful decision.

2.1.2 Rewards

The rewards of an FPS are slightly more difficult, as they are often communicated implicitly, and so the program will require a series of rules relating to certain changes in the environment to provide it a frame of reference for beneficial or unbeneficial actions. This is the basis for reinforcement learning. Also, rewards in video games, especially FPSs are often locked behind patterns of behaviour rather than individual actions, so the program will need to learn that, while certain actions do not immediately give rewards, they might still be good because they set up future rewards also known as delayed gratification. This problem was addressed in the paper Learning from Delayed Rewards (Watkins 2022) where the Q-learning algorithm was proposed. The algorithm itself is well explained in the last 19 minutes of this video (Klein 2022) on machine

learning. In short, Q-Learning allows rewards from actions to propagate backwards so actions that lead to actions with rewards are considered rewarding in and of themselves.

2.2 Tools

Meeting all of the requirements will require the use of certain select tools which have been designed to address the challenges inherent in problems like this one. Fortunately, these tools already exist in the form of code libraries and research platforms.

2.2.1 ViZDoom and Pogamut

ViZDoom is perfect for setting up, running, and passing data between an instance of FreeDoom and a Python program (it also has in-built support for C++ and Julia). It describes itself as a “Doom-based AI Research Platform for Reinforcement Learning from Raw Visual Information” (ViZDoom. 2022). It is based on ZDoom (Gimmer and Acheron 2022), a Source port (Source “is a 3D game engine created by Valve” (Source - Valve Developer Community. 2022)) of the original Doom (as well as other titles that ran on the Doom engine), and it makes it possible to pass frame data from the game to python and confirm when certain rewards should be distributed, as well as handling starting and stopping the game when the program desires. ViZDoom also has many scenarios which provide unique demonstrations of machine learning in action.

The only other tool vaguely similar to ViZDoom is Pogamut, a “Java middleware that enables controlling virtual agents in multiple environments provided by game engines. [. . .] Pogamut provides a Java API for spawning and controlling virtual agents and GUI (NetBeans plugin) that simplifies debugging of the agents.” (Pogamut - virtual characters made easy | About. 2022). Pogamut is limited in the amount of control and detail the agent has access to, making it difficult but not impossible to create a true machine learning (Berg 2022). Even without these limitations Pogamut lacks the ease of use that ViZDoom provides, and Java is generally considered harder to program in than Python (though it has other advantages that python lacks).

Few if any tools like ViZDoom and Pogamut exist, so they sit in classes of their own, where the only viable alternatives are to build an entirely custom FPS and use it as the environment, or to alter an existing FPS to function as required.

2.2.2 TensorFlow, PyTorch, and Lasagne/Theano

TensorFlow is “an end-to-end open source platform for machine learning.” (TensorFlow. 2022). With TensorFlow, DQN’s can be built from scratch or from a save file, and with

Keras, which "... is a deep learning API written in Python, running on top of the machine learning platform TensorFlow" (Keras documentation: About Keras. 2022), the process of teaching that DQN can be performed rapidly and efficiently.

PyTorch is "an optimized tensor library for deep learning using GPUs and CPUs." (PyTorch documentation — PyTorch 1.11.0 documentation. 2022). Pytorch can accomplish just as much TensorFlow, and in general they are not that dissimilar. TensorFlow is older and deeper but is less focused on python and thus can be slightly more difficult to work with, while Pytorch is younger and shallower, but it has a dedicated following and is focused on Python, which makes it slightly easier to work with.

Lasagne "is a lightweight library to build and train neural networks in Theano." (Welcome to Lasagne — Lasagne 0.2.dev1 documentation. 2022). Theano is "...a Python library that allows you to define, optimize, and efficiently evaluate mathematical expressions involving multi-dimensional arrays" (Theano. 2022). Lasagne focuses on speed and efficiency at the detriment of ease of development, in contrast TensorFlow is easier to develop in, but is not quite as efficient.

2.2.3 DirectML

While TensorFlow alone can make use of spare GPU computing to accelerate operations by using the GPU to perform the DQN updates (being setup to do a lot of very small calculations very quickly makes GPUs perfect for this role), unfortunately my GPU is not recognised by TensorFlow. To compensate for this DirectML was installed (Direct Machine Learning (DirectML). 2022), which allows for a wider range of GPUs to participate in machine learning, including mine.

2.3 Prior Research

Understanding the problem and the tools required to solve it are both important but can be enhanced with the knowledge of prior research surrounding other attempts to solve similar problems. This research details parts of the problem that have yet to be solved and warn of potential weaknesses in certain solutions that might not cause outright failure but reduce efficiency or optimality.

2.3.1 Model-Free Reinforcement Learning

DeepMind and the DQN:
Projects like Google's DeepMind, which had similar requirements to this one, lead to the creation of the DQN or deep Q network, a useful technique which combines all the

above requirements into a single machine learning program (Deep Reinforcement Learning. 2022).

DQN's are model-free, meaning they estimate the best action based on prior data rather than using that to build a model of the environment and using that to inform decisions. DQN's need to be custom made for every application, since the number of nodes per layer and layers total change based on the problem's exact requirements.
Google's DeepMind was tested on a variety of Atari 2600 games to see how it stacks up against previous machine learning algorithms (Mnih 2022). The paper eventually concluded that in most cases the DQN based DeepMind was better, but it also raised several other less important points that are relevant to this paper. Two notable points were that the DQN was able to perform at or above human level for a good portion of the games chosen, and that certain games, Montezuma's Revenge especially, completely defeated both learning algorithms because obtaining any points in the game requires long term planning, so a learning machine going in blind would find it almost impossible to earn any points before dying.

R2D2, ape-X, and PPO:
Many other variants of Q-learning exist, for instance R2D2 and ape-X which use distributed architectures, i.e., they separate the process of data collection and learning by placing experiences in a list and then learn from batches of that data, to create solutions. Both also adopt several other changes to improve upon basic DQN's, notably prioritised experience replay (Horgan 2022) ((Samsami and Alimadad 2022) section 3.5, page 9).

PPO is another type of model free reinforcement learning, which has two notable differences to a standard DQN. First it searches a policy gradient for an optimal action rather than having state action pairs which means it can operate on continuous action spaces. Also, it uses advantage learning, which is an expansion on Q-learning which can learn from cases where actions cause very small state changes faster than Q-learning and does everything else at essentially the same speed (Proximal Policy Optimization. 2022).

Duel DQNs:
"[The] dueling network represents two separate estimators: one for the state value function and one for the state-dependent action advantage function. The main benefit of this factoring is to generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm." (Wang 2022). Duel DQNs (not to be confused with double DQNs) are a unique variety of DQN's which combine the basic Q-learning and advantage learning, which when combined allow "... the dueling

architecture [to] learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way" (Wang 2022). The only disadvantage of duel DQNs is that they are more complex than standard DQNs and thus take more effort to alter.

2.3.2 Model-Based Research

MuZero:
Google DeepMind's MuZero is a relatively new player on the reinforcement learning scene, being released in December of 2020. It is the most recent step in a series of model-based reinforcement learning algorithms, which started back in 2016 with AlphaGo, an algorithm for playing Go. MuZero is especially impressive for two reasons. First, it learns to play games without any outside help, not even the rules of the game. Second, its ability to outperform model-free reinforcement learning in video game environments, which is notable because building a model based on visual input had previously proved extremely difficult. MuZero solved this by ignoring most of the environment and simply focusing on the most important details "After all, knowing an umbrella will keep you dry is more useful to know than modelling the pattern of raindrops in the air." (Schrittwieser 2022). A MuZero variant known as EfficientZero is currently the state-of-the-art model-based reinforcement learning algorithm (Ye 2022).

2.3.3 Sampling Research

Another experiment was performed on Atari 2600 games with a DQN (Anenberg 2022), but instead of evaluating the DQN itself, the question posed by the paper was whether alternatives to uniform random sampling would allow the DQN to perform better. Of note in this paper is the final alternative tested, reward-based sampling, which ran a uniform random sample across the pool of rewards rather than the pool of actions. The paper concluded that none of the proposed alternatives were better than uniform random sampling.

3 Methodology

3.1 Initial plan

The initial plan developed to solve the problem was simply to build a reinforcement learning algorithm, run it against an FPS, and collect some data to prove it was learning to play the game. This technically met the definition of a solution but was not that

interesting and had been done many times before, and there was still plenty of time left in the project to shift to a more comprehensive solution.

3.1.1 Shoulders of Giants

Deciding which tools to use required evaluating the relative strengths and weaknesses of the options available. First, picking whether to use ViZDoom, Pogamut, or build custom interaction software. Pogamut was quickly eliminated due to a combination of requiring Java and being designed for bots rather than true machine learning, making it more difficult to work with. After Pogamut, the idea of building custom interaction software fell by the wayside because despite posing an interesting challenge and potentially allowing the involvement of more modern FPSs, it would take a great deal of effort and time to build, and that would likely cut into the time available for building and teaching a machine learning algorithm. In contrast ViZDoom had been custom built with machine learning in mind, and it had libraries of examples and custom scenarios to learn from.

After deciding to use ViZDoom to handle many of the more arduous elements required for the program to interact with the game, and thus deciding to use FreeDoom as the FPS to teach my program to play, a group of particularly interesting example programs linked on the ViZDoom site seemed a good place to start (M. Kempa 2016). While most of the examples were somewhat helpful in understanding the finer details of ViZDoom integration, three stood out. learning_tensorflow, learning_pytorch, and learning_theano matched the basic plan for a solution, that is they were all reinforcement learning machines built to play a custom doom scenario called simple and learn to optimise it. This presented a bit of a problem, since using one of them as a basis for my own work felt like it would make the project far too simple and meant accepting the learning machine as it was, and thus it would be less customisable, but ignoring them would mean spending time remaking what had already been done. In the end the decision came down to the fact that taking another's code and moving it onward to new heights felt more exciting. Choosing to use one of them also settled the project as a ViZDoom based DQN reinforcement learning algorithm with no room to swap to a fancier DQN variant or perhaps try a model-based learner, but that was a sacrifice that made sense. The decision between the three different bases also proved surprisingly simple, after some research and basic runs Theano was discounted as quite unwieldy, and the Pytorch implementation used a duel DQN which was more complex than the project needed to be, leaving learning_tensorflow as the code I would build my project out of.

3.2 Optimisation Plan

Instead of attempting to reinvent the wheel, the goal of the project shifted to attempting a more difficult scenario and attempting to optimise the learning machine itself. Unfortunately, the code in question was not well explained and contained only superficial comments detailing what it did and lacked an explanation of how it did it. This meant spending time figuring out what was going on so it could be adapted to suit the new task. Deciphering another's code is like solving a jigsaw puzzle, while it is simple to understand each piece/line in and of itself, its place in the larger design is often a mystery until its neighbours have been placed. However, with some time the program's basic flow became apparent.

3.2.1 Some initial inspiration

After running the demonstration code on a new scenario (deadly_corridor) some notable things happened that would influence decisions later in the project. First, after a particularly long test, it became apparent that the program seemed to get stuck after a while and was unable to improve beyond a certain point. There were several theories as to why this might be; the program was trying to pick between too many combinations of actions (where it had once had 8, now it had 128); the program was taking to many random decisions for such a complex task to be accomplished (starting at a 100% chance and ramping down to 10% after a few epochs); and the program was being bogged down by bad data (the scenario chosen opened with the actor in imminent danger and even good player would have to be lucky to escape alive). Of these three the first two had obvious solutions but solving the issue of the program being bogged down by bad data would take a more creative approach.

3.2.2 The optimisation goal

Taking the idea of optimisation forward meant establishing a baseline for performance and then testing alterations to see the speed at which the program ran, and how effective a solution the program came up with. With so much alteration possible and so much data required to properly examine each alteration, the optimisation plan had to be refined down to a single optimisation tested in multiple ways to be completed in time. The goal chosen was related to the prior issues with bad data due to poor luck. After pondering the issue, the idea was conceived to bias some of the sample data fed to the learning algorithm towards higher reward actions to compensate for poor luck, which became the focus of the project. Establishing if this was a good solution would take a lot of testing since learning machines are inherently unpredictable and so even the unbiased case would need several runs to come to a useful baseline, so creating a program capable of providing that data became the goal.

3.2.3 Base Program

This is a basic breakdown of the base program. (M. Kempa 2016)

- If loading: load saved DQN and initialise where left off
- Else: Initialise the game, DQN, and a couple other elements from scratch
- Run the learning loop for desired number of epochs*
- For each epoch in the learning loop:
    - Run games till epoch ends
        - For each game:
            - Choose action
            - Perform action
            - Observe reward
            - Store data related to the action in a list
            - Learn from random sample of list*
            - Repeat until game ends
    - Test the program at the end of each epoch
- Save DQN*
- Watch the program play game*

*If certain conditions are met

Note: This is obviously a massive oversimplification

3.2.4 Goal Program

This is a basic breakdown of the goal program.

- If loading: load saved DQN and experiment data, and initialise where left off
- Else: Initialise the game, DQN, and a couple other elements from scratch
- Run the learning loop until program exit*
- For each epoch in the learning loop:
    - Run games till epoch ends
        - For each game:
            - Choose action
            - Perform action
            - Observe reward
            - Store data related to the action in a list
            - Learn from (potentially) biassed sample of list*
            - Repeat until game ends
    - Save DQN and experiment data at the end of each epoch
- Watch the program play game*

*If certain conditions are met

Notable differences:

- In the base program the load function does not work (this may be due to an issue with my version of TensorFlow. Two different functions in TensorFlow think they are on mismatched versions?)
- Loading will require pulling in prior experiment data to then have it save correctly later
- Loading was supposed to update epsilon to keep continuity (see section 4.2)
- Because of the extended nature of experimental trials, learning was passed an extremely large number for number of epochs to run, allowing to run over several days without interaction
- Saving both the DQN and data was moved into the learning loop for safety
- Changed the type of save and load, since TensorFlow does not like fully saving active DQN's (which incidentally fixed the issue of the base program not loading properly)
- Biassed a certain amount of the data passed to the learning function to have higher than average rewards (though a version of the program without this change was kept establishing a baseline)
- The removal of the testing section, since it was not needed and took time away from learning

Hidden changes:

- A new, more difficult scenario was chosen
- This necessitated the program be capable of taking more actions, originally it moved from 8 to 128, but that was changed to 64 after the removal of one action
- Two additional rewards, a reward for surviving and a reward for killing enemies, were planned to be added on top of the two existing rewards, one based on forward progress and one lump sum for completing the level (see section 4.2)
- Epsilon decay was slowed down, but epsilon minimum was also lowered, with the goal being to increase the length of time the program explored all available actions, but also have the program make fewer random decisions later on, lowing the odds the random action lead to the end of an otherwise good run

4 Implementation

The changes made to the base program come in two loose groups separated by the complexity of the required change, straightforward and complicated.

4.1 Straightforward Changes

The straightforward group ironically contains most of the important changes, like the implementation of the biassed sampling, which only required altering one function so

that instead of returning 64 randomly sampled data points, it instead returned 64 data points of which some fraction was randomly sampled from the whole group. The rest were randomly sampled from the data points which had a reward greater than the average reward across the whole list. The only difficulty encountered with this was the discovery that if the program got very unlucky it could crash because there were not enough points above the average to provide the full portion, which was easily solved with an if statement.

Other straightforward changes included moving the save function so that it was run after every epoch rather than after the program was done learning, both to back up the learning in the case of a crash, and because the program had been changed to run learning until the user closed it rather than to stop at some pre-assigned point. This saving move only required minor changes, so instead of saving the whole DQN as in the base program, now it only saves the weights the DQN is based on (this is because TensorFlow does not allow the saving of a DQN that is currently open, but the weights of the DQN are the only important part). Another straightforward change was the removal of the testing process, which was nice for a program made as a demonstration of what was possible, but for a program designed to record large amounts of data about the learning process, was just an unnecessary slowdown.

Finally, one change was made to the way the program was rewarded, however this required altering a different file, as the rewards were split across a .CFG (a config file used by ViZDoom as an intermediary between the scenario file itself and any program) and the .WAD (the file that contains the scenario and all its associated features). Editing the config file was extremely simple and only required adding one line which rewarded the program for surviving (in a scenario called deadly corridor, survival seems a fair goal).

4.2 Complicated Changes

While thankfully most of the changes to the program proved straightforward, three proved far more complicated than initially expected. The first was the saving and loading of the data necessary to evaluate the performance of the program. Because of a fundamental misunderstanding of how some tensors were stored, the program kept crashing due to mismatched element types in a list. It took a long time to fix this issue as some of the tensors were 2 dimensional but only contained one element while others were one dimensional but contained multiple elements. This combined with another misunderstanding of how the timing code was stored and thus needed to be saved/loaded required several days to work.

The second was an attempt to maintain continuity between runs that ended up being abandoned. To retain continuity if the program was closed, epsilon (the value which determines the chance the program will take a random action) needed to be saved and loaded, but whenever this was attempted, the secondary DQN used for training cause a crash due to a mismatch with the main DQN. This made no sense and seemed to occur whenever the first action chosen by the program after starting was not randomly selected. After trying several different possible fixes, the idea of retaining epsilon continuity was abandoned because it was not essential for the project to go ahead, but as a result in the event the program is closed and reopened, it will start with a 100% chance to pick a random action which will again scale down to the epsilon minimum value.

Finally, a second change to the rewards system was attempted to reward the program for defeating enemies, however this change, unlike the one above, would require editing the .WAD file, and without a proper ZDoom level builder this was not possible, and learning the ins and outs of the Source engine would have taken too long, so after trying to manually alter the .WAD file, this idea was also abandoned.

4.3 The Defect

Aside from the aforementioned difficulties, one other unintended interruption took time away from writing the code. This defect did not come about because of changes made while adapting the code, at least not any one specific change. To test the program after many of the major changes mentioned above, it was run to see if the changes implemented had worked, but after a while a strange pattern began to emerge. After a couple of epochs of good learning, the program's score would plummet to near the minimum and stay there. This made no sense as no changes to the way the program learned had been made outside the reward biassing, but the data that passed to the learning function had not been altered.

Investigating the issue was made extremely difficult by its inconsistency; sometimes the program ran fine; sometimes it failed. This could mean the drop-off happened when the epsilon values approached epsilon minimum (basically the program was being carried by the random choices and was never truly learning), but this did not make sense as sometimes the program succeeded just fine. Eventually, after watching the program's view while learning for multiple runs (a headache inducing process as the game was run as fast as it could be), it became apparent that at some point the program began to spin, always picking the turn left action. This was just as confusing as the initial discovery, as while turning is useful in conjunction with other actions like move forward and shoot, it alone is useless. Shortly after that discovery, it also became apparent that the constant turn emerged suddenly, one moment it would be fine, the next the program

would choose only to spin. After a particularly long test run over several days, it emerged that the program could not learn its way out of this behaviour. It would just spin endlessly.

Finally, after over a week spent focused on the problem, some useful information cropped up. When running a debugging version of the program, which printed the table generated by the program when passed the frame data for which action was best every time an action was chosen, the flaw became apparent. At some point, seemingly out of nowhere, the table's values suddenly dropped, spiked, and then all became NaN's over the course of 3 choices. This was the first useful breakthrough on the problem, but it resulted in no further understanding of why this happened. Clearly, some value involved with the learning went mad, and it took inspecting the thousands of numbers involved in making decisions to discover that a few instances of 'Inf' showed up extremely out of place.

At this point over two weeks had gone by and this bug was the only thing holding up data procurement, so rather than spending more time trying to solve the problem, a patch was added that would catch NaN's during learning and rollback to the copy of the DQN from the start of the epoch. This solution was not perfect, as if either the drop or spike occurred on the exact action which updated the copy, then the program would be irreparably damaged. Fortunately, this never occurred during the project.

5 Results

5.1 Tests

Over the course of several weeks, data was collected on three pairs of programs. Each pair consists of one biassed and one unbiased program. Data collected included the mean score, standard deviation, minimum, and maximum of every epoch for the entire course of the program. Sometime data was also collected on the third test to help evaluate if biassing accelerated or slowed the program notably. Alongside this data, some screenshots of the program playing the game and six short (<1min) videos are also available in the supplemental materials submitted showing the bots in action, which are helpful when attempting to understand what the program has accomplished.

5.1.1 75% test

The first major test run was between an unbiased version of the program and one in which 75% of the sample was biassed. Its results were as follows:

Mean score by epoch:

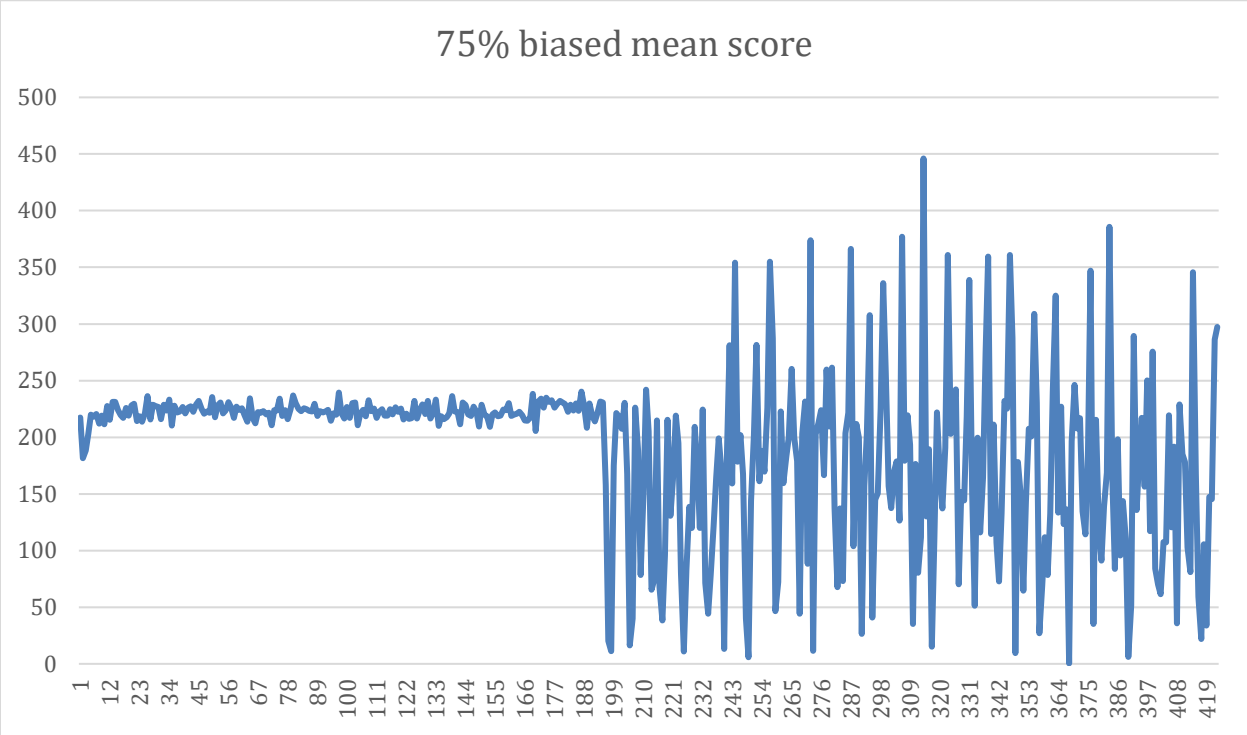

Figure 1– The mean score for each epoch of the 75% trial

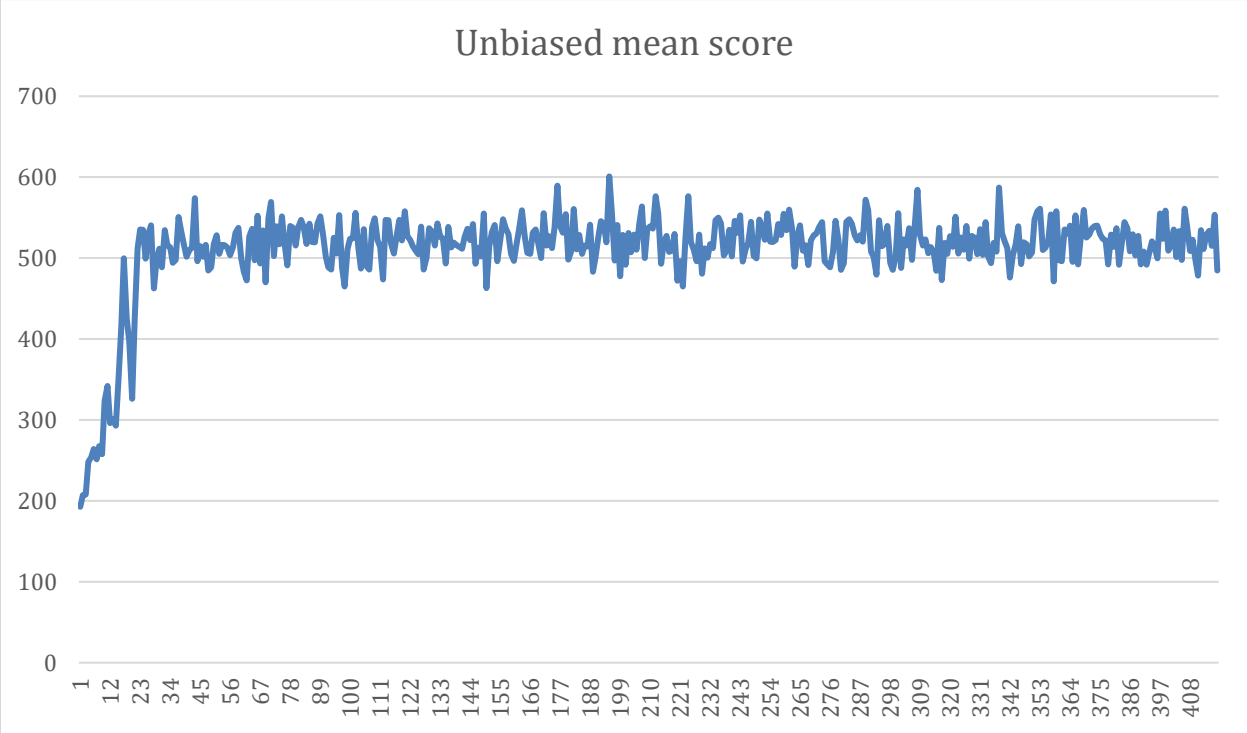

Figure 2 - The mean score for each epoch of the first unbiased trial

Maximum score by epoch:

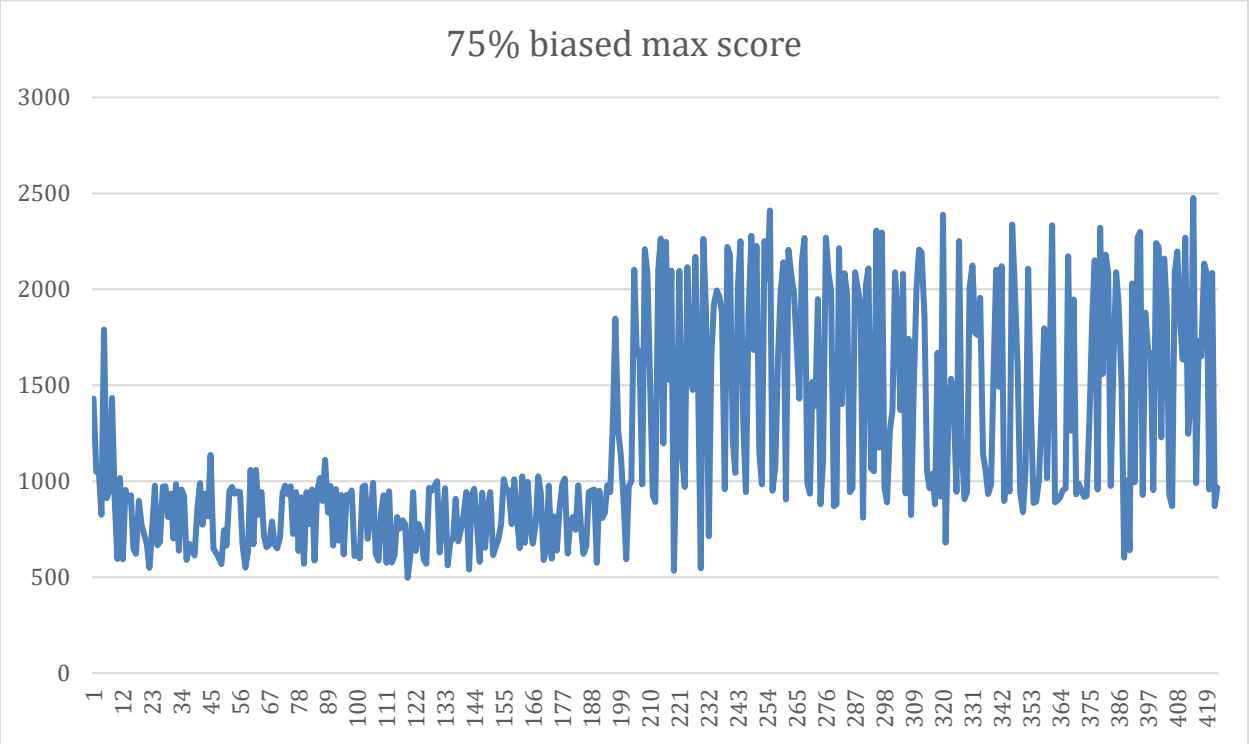

Figure 3 – The maximum score for each epoch of the 75% trial

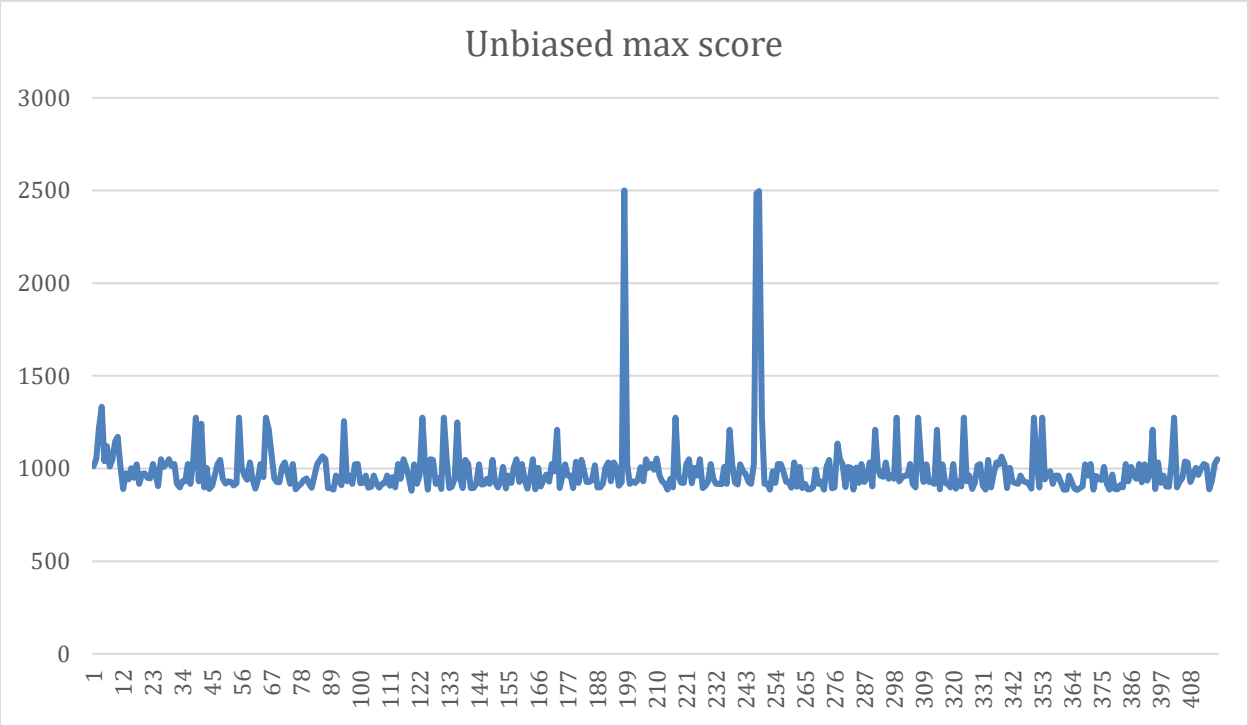

Figure 4 – The maximum score for each epoch of the first unbiased trial

Standard deviation by epoch:

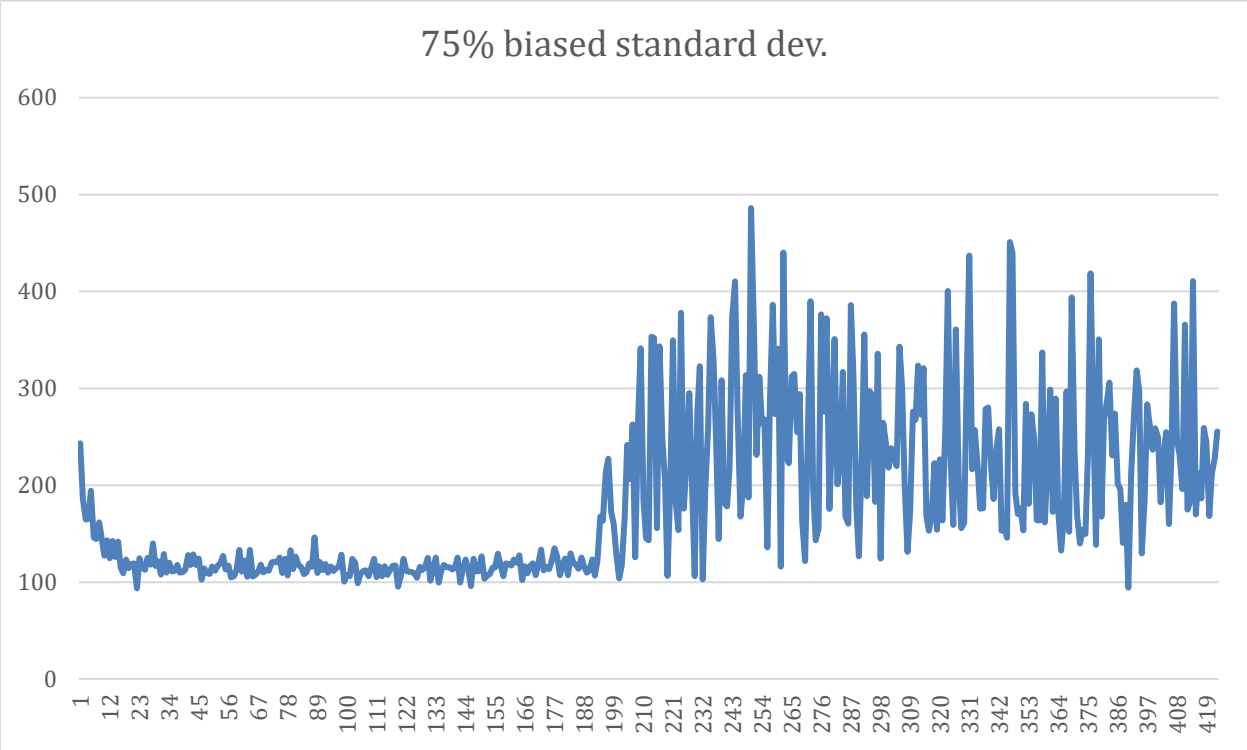

Figure 5 – The standard deviation for each epoch of the 75% trial

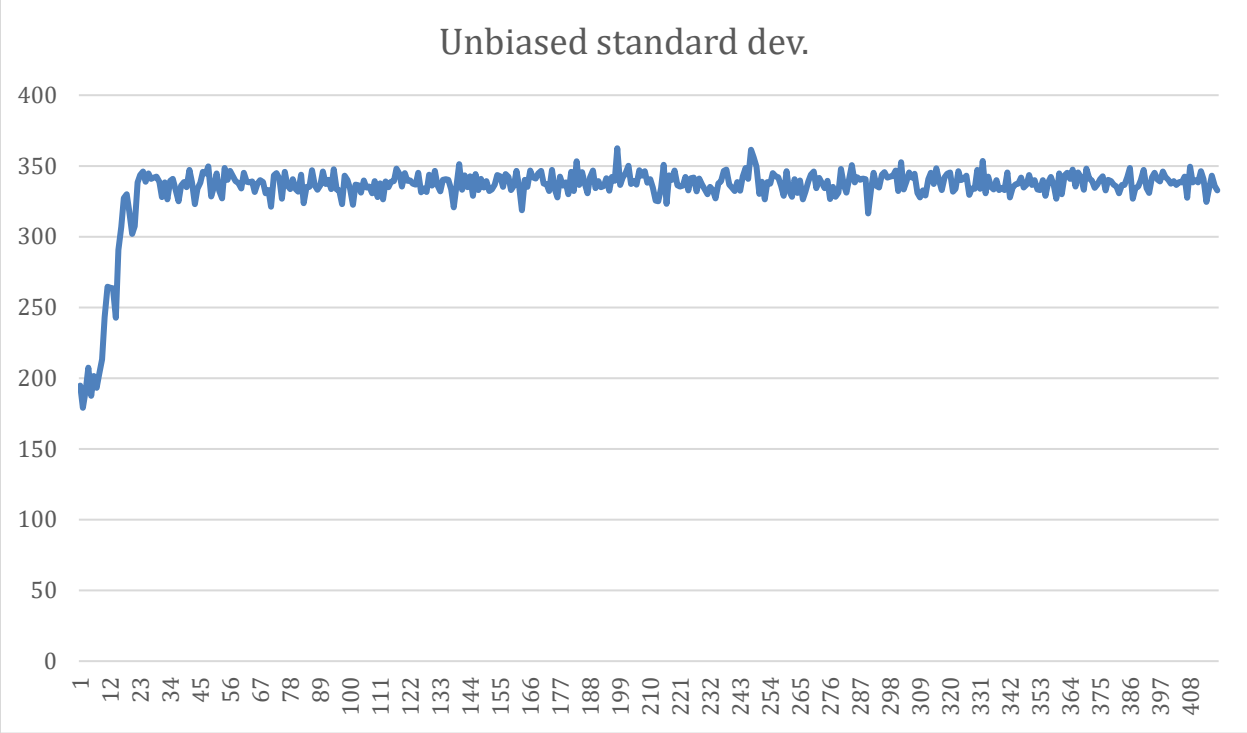

Figure 6 – The standard deviation for each epoch of the first unbiased trial

Of Note Are:

- The low mean scores for the first half of the biassed graph compared to the unbiased version
- The sudden explosion that seems to occur in all three biassed graphs vs the continuous plateau for the unbiased
- The great number of 1500+ maximum scores for the biassed vs the lonely two for the unbiased
- The Unbiased higher standard deviation on average, vs the biassed's chaotic but lower average

5.1.2 25% test

The first major test run was between an unbiased version of the program and one in which 25% of the sample was biassed. Its results were as follows:

Mean score by epoch:

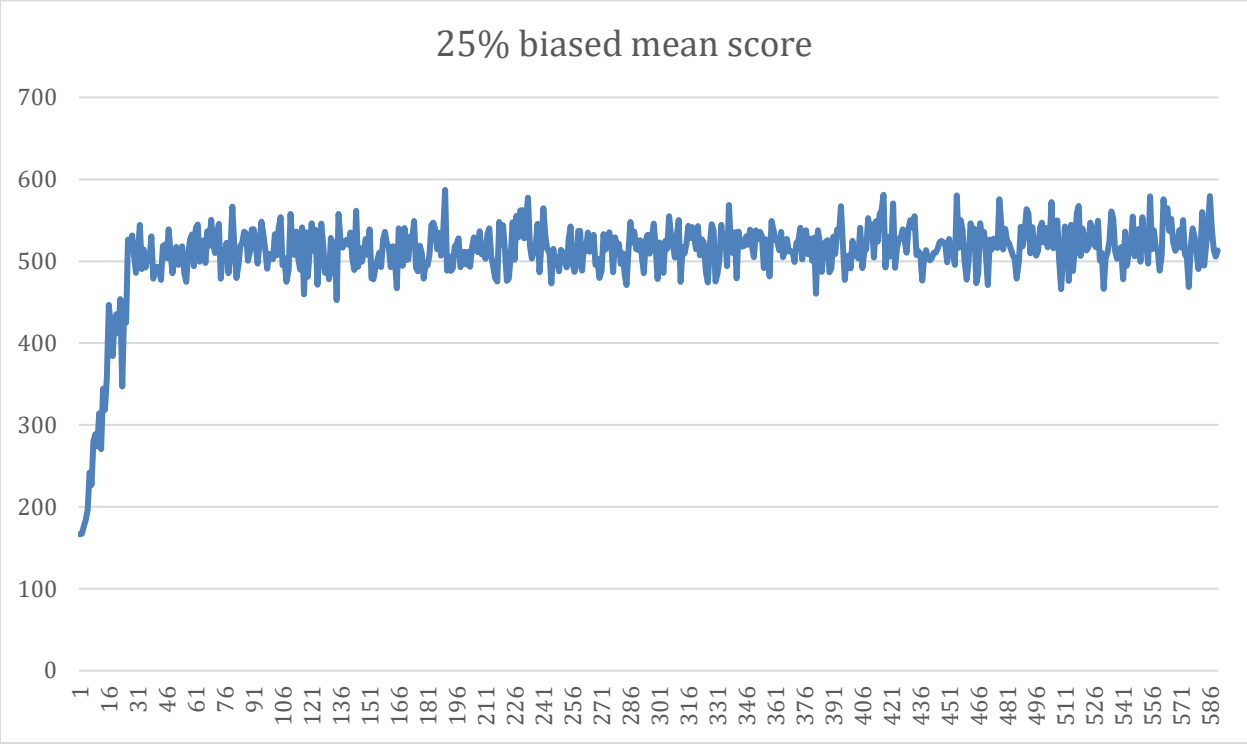

Figure 7 – The mean score for each epoch of the 25% trial

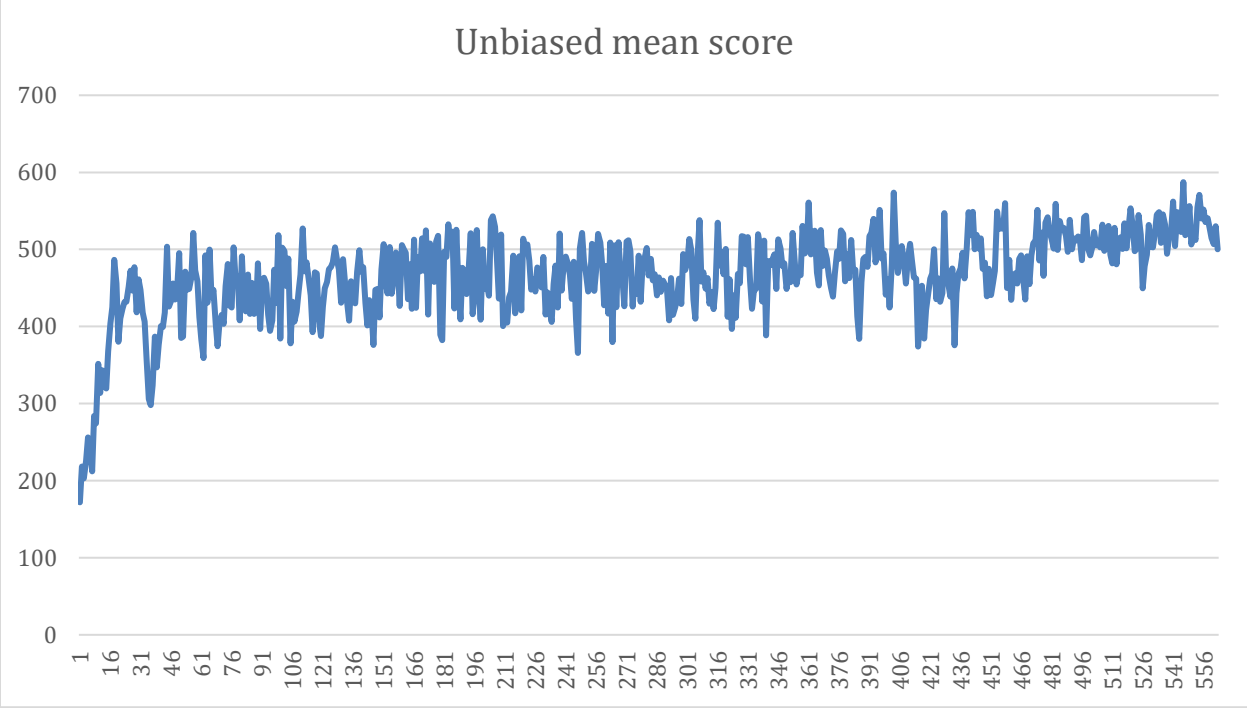

Figure 8 – The mean score for each epoch of the second unbiased trial

Maximum score by epoch:

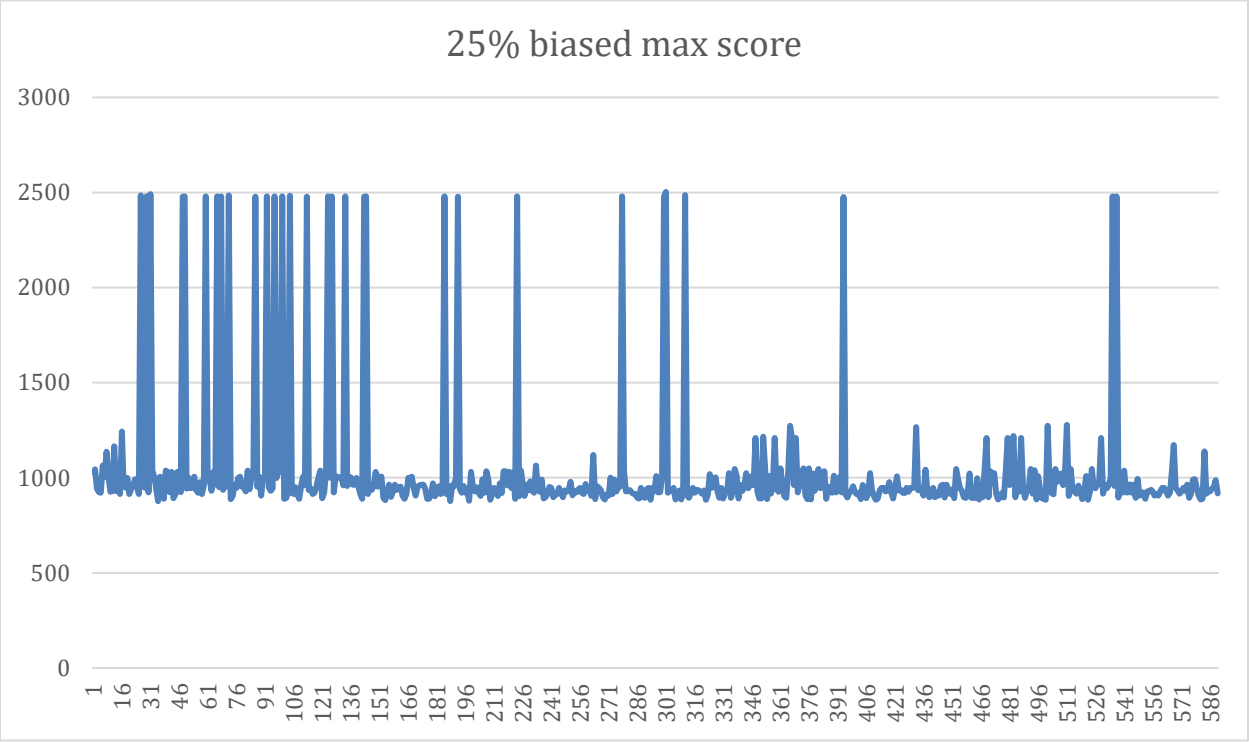

Figure 9 – The maximum score for each epoch of the 25% trial

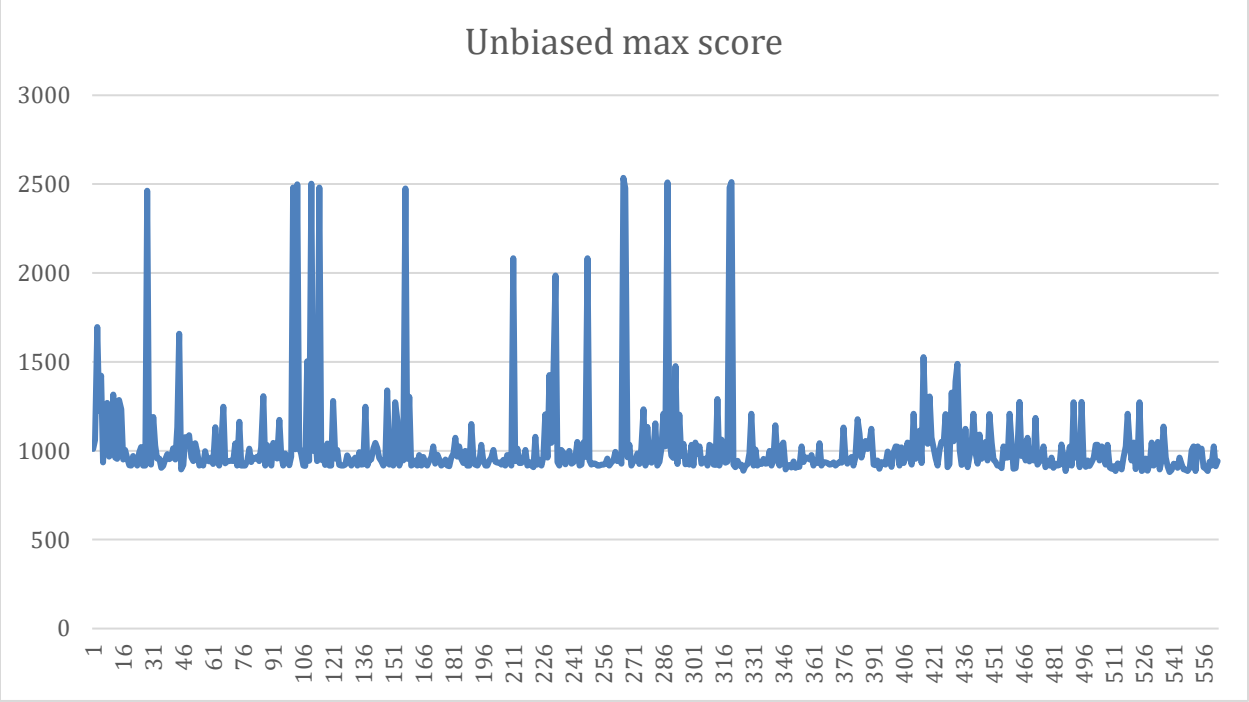

Figure 10 – The maximum score for each epoch of the second unbiased trial

Standard deviation by epoch:

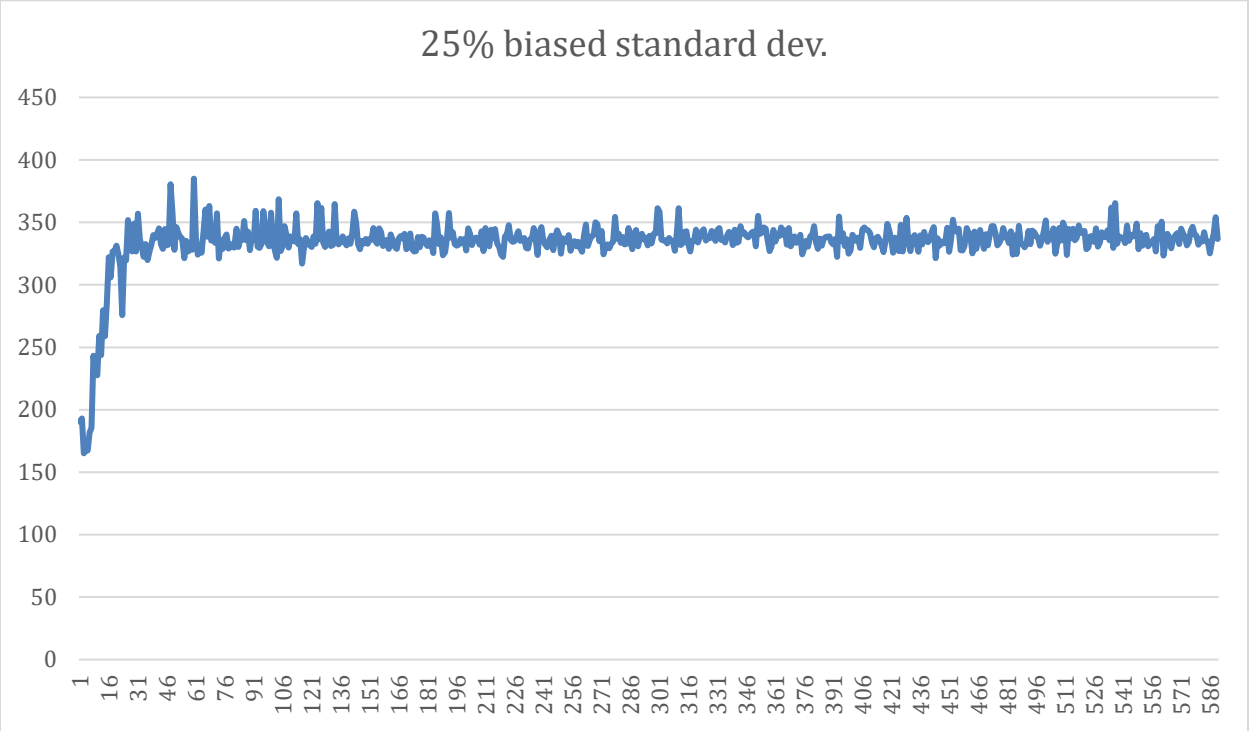

Figure 11 – The standard deviation for each epoch of the 25% trial

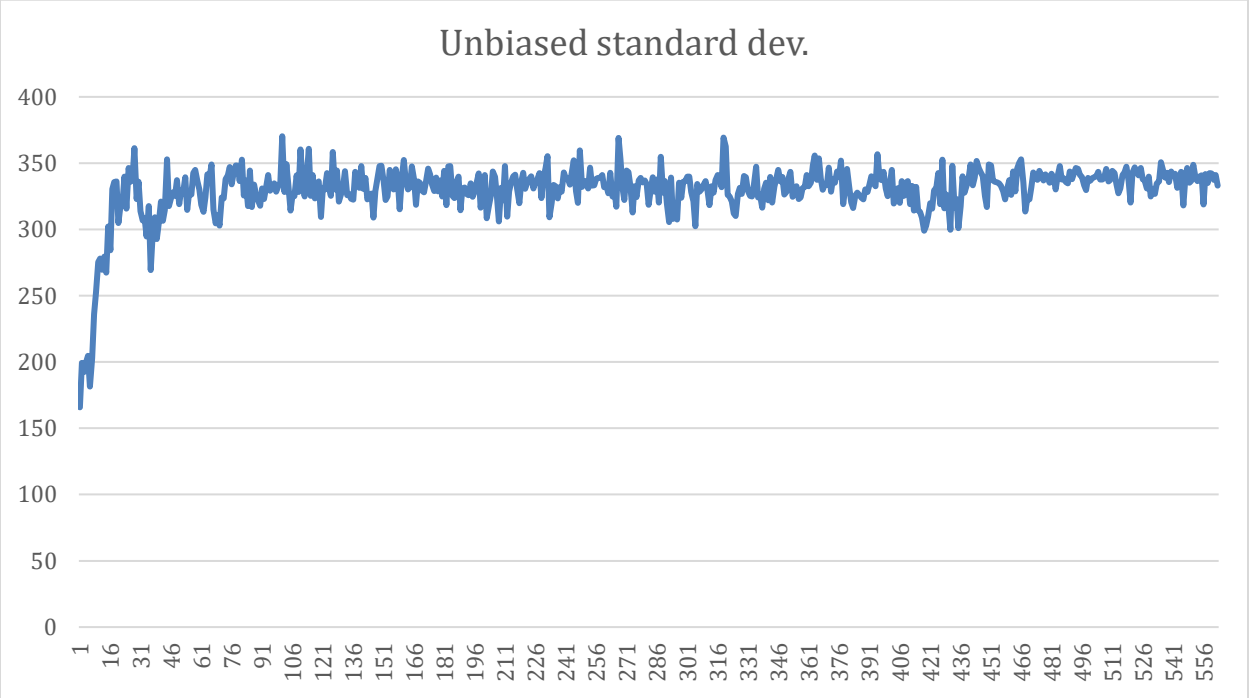

Figure 12 – The standard deviation for each epoch of the second unbiased trial

Of Note Are:

- The 25% biassed better having mean scores for most of the epochs against the unbiased mean scores
- The unbiased upwards trajectory at the end vs the 25% biases plateau
- The 25%'s greater number of 2000+ maximum scores, vs the unbiased
- The unbiased cluster of 2000+ maximum scores in the first two thirds vs the 25% biassed more spread 2000+s
- The very similar plateaus of both Unbiased and 25% biassed

5.1.3 100% test

The first major test run was between an unbiased version of the program and one in which 100% of the sample was biassed and included a timer to show epochs per minute. Its results were as follows:

Mean score by epoch:

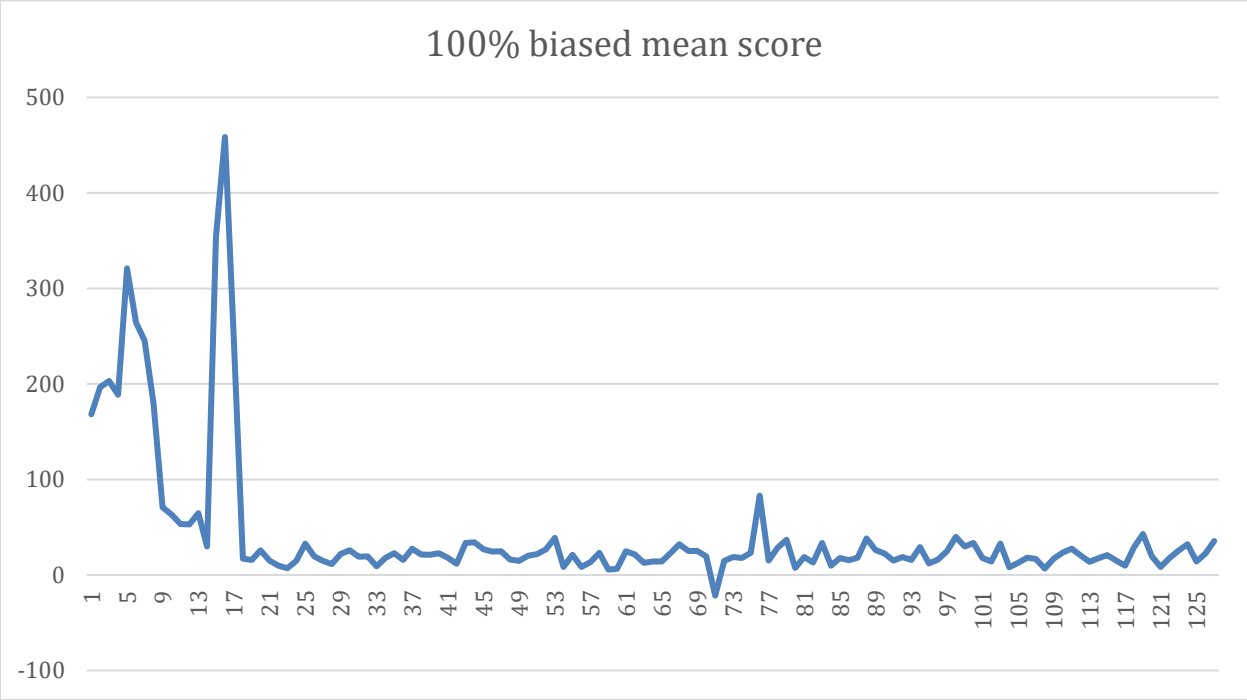

Figure 13 – The mean score for each epoch of the 100% trial

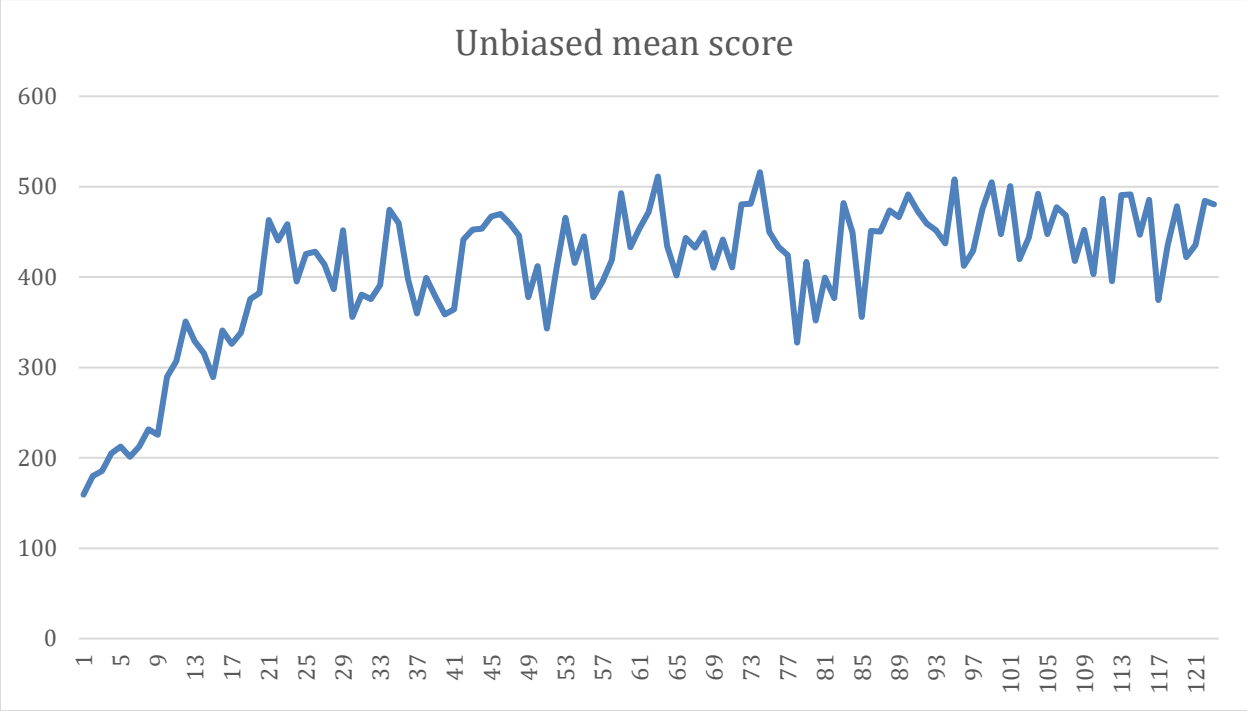

Figure 14 – The mean score for each epoch of the third unbiased trial

Maximum score by epoch:

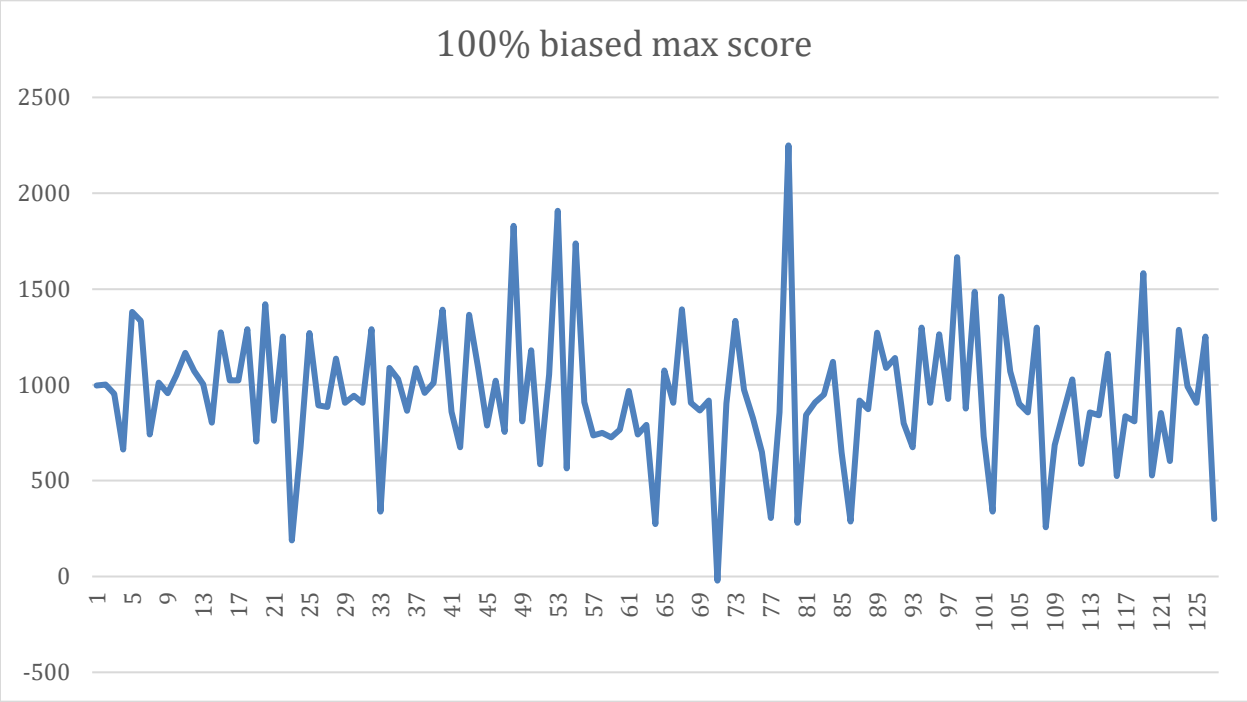

Figure 15 – The maximum score for each epoch of the 100% trial

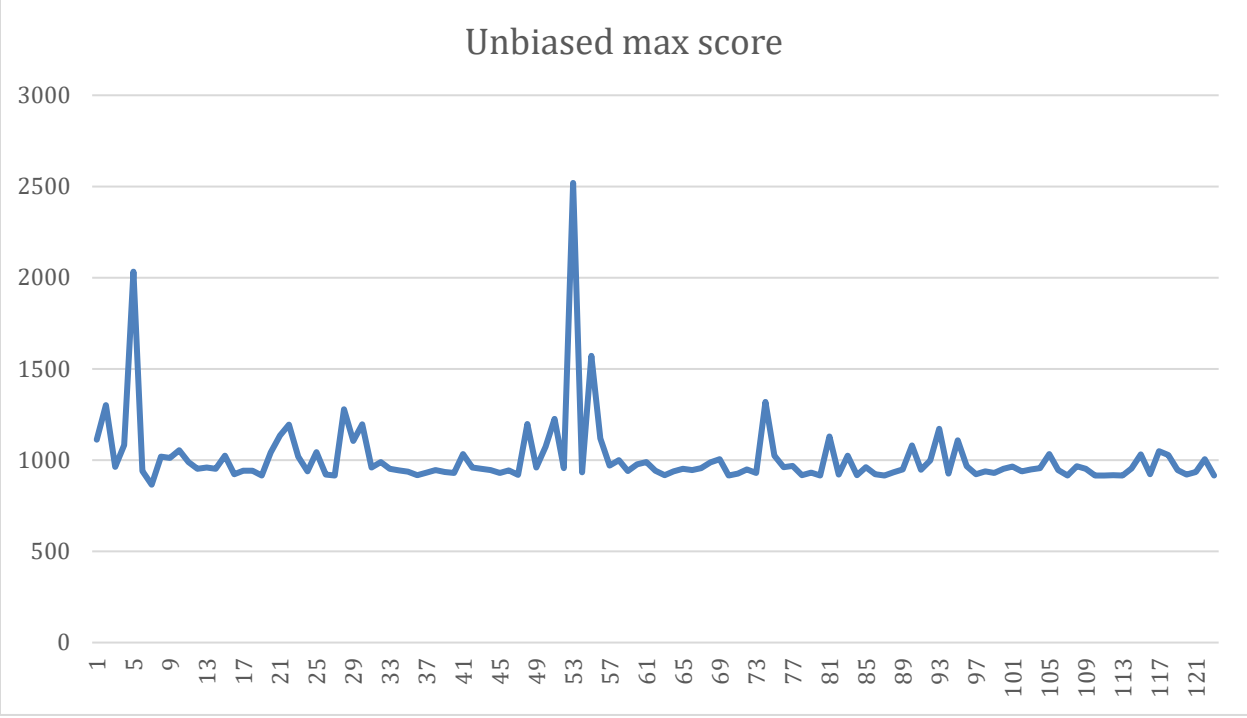

Figure 16 – The maximum score for each epoch of the third unbiased trial

Standard deviation by epoch:

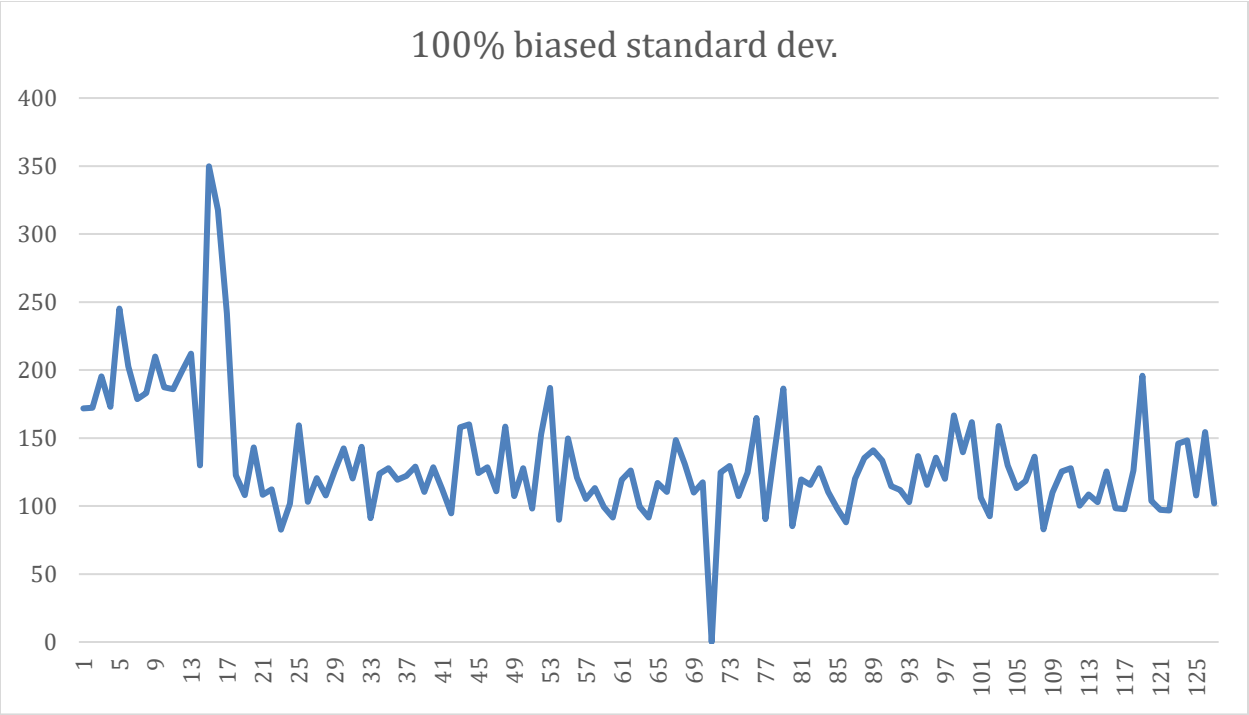

Figure 17 – The standard deviation for each epoch of the 100% trial

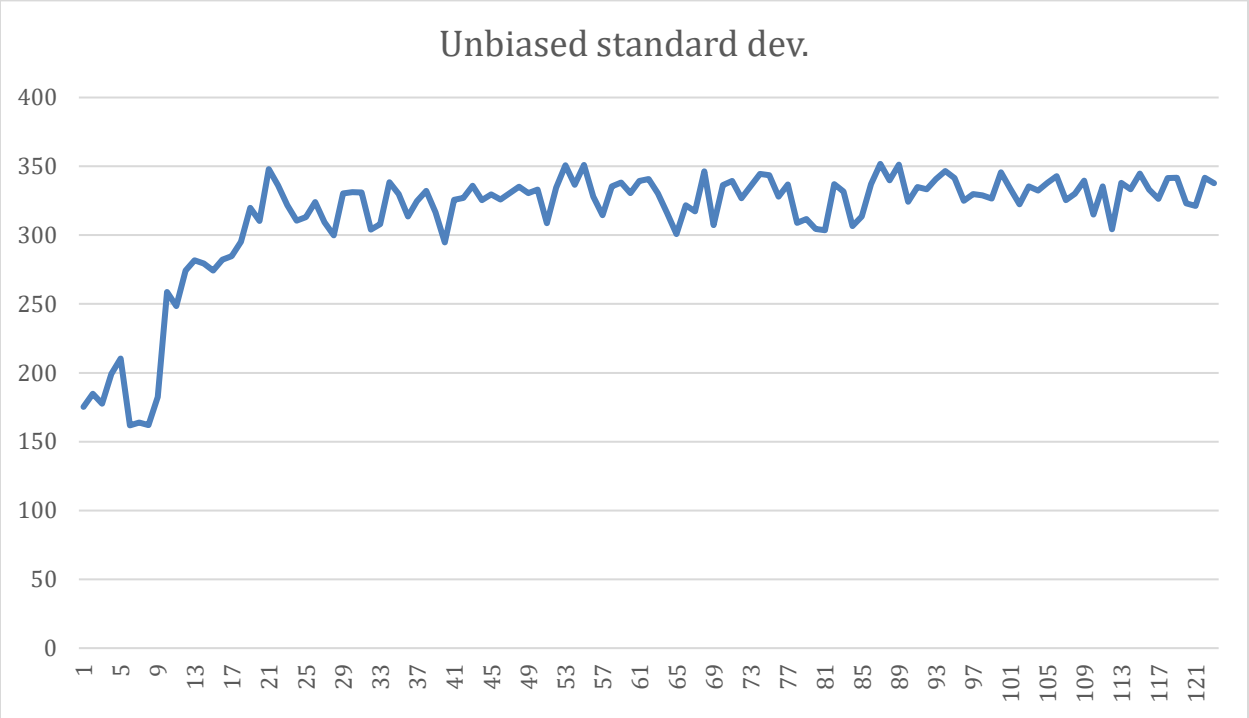

Figure 18 – The standard deviation for each epoch of the third unbiased trial

Epochs per minute:

- 100% biassed took 6.5645 mins per epoch
- Unbiased took 6.7722 mins per epoch

Of Note Are:

- The fact that after 16 epochs 100% biassed never rose above 100 mean score while at the same time unbiased never fell below 300 mean score.
- The similar average maximum score between 100% biassed and unbiased despite 100% biassed have a much greater variance
- The 100% biassed has a lower and much less consistent standard deviation while unbiased has a much higher but more consistent standard deviation
- The 100% biassed ran epochs slightly faster than unbiased

5.1.4 Common Characteristics

Since we have three sets of data for the unbiased version, it is simple to identify characteristics unique to the biassed versions, and with three different levels of biassing, it is possible to see some of the range of those characteristics.

Most notable amongst them are:

1. Something strange is going on with the 75% bias data
2. The slight but notable differences in all three unbiased runs, especially their mean and maximum data
3. High bias levels seem to relate to low average mean scores, but low bias levels outperformed no bias (at least for most of the 25% bias test)
4. Both biassed and unbiased maximum scores had similar averages, (excepting the 75% bias test), but higher biases had more variance
5. High bias levels have lower average standard deviation, but their deviation itself has more variance than unbiased and less biassed runs
6. Unbiased runs are slightly slower than biassed versions, seen both through the actual time test and the fact that other pairs of runs had the biassed version cover more epochs in the same amount of time

Addressing each one of these points in the order given:

First, the sudden and complete shift in the behaviour of the 75% bias test was almost certainly triggered by external factors. There is no concrete proof, but since no other dataset shows anything even remotely like this, it would make sense for some external factor to be at least partially responsible. Then again, trying to understand what a

learning machine is thinking, especially a deep one, is like asking a mediaeval peasant to watch and understand Fox news. The problem this presents is whether to use the data or not, but since data took so long to gather and there was a small chance it might happen again, it seemed sensible to include it, but to favour the other two biassed tests over this one.

Second, the differences between all three unbiased trials in the mean and maximum scores are slightly surprising given they were all run on the same settings. The main two differences are the lower average mean scores for both the second and third trials, even when accounting for the facts that the third trial is shorter and the second trial did seem to pick up at the end, but it was also longer than the first trial. The second main difference is the number of 1500+ values in the second trial compared to the other two. The first trial had 2 such values over 400 epochs, the second had 14 over 550 epochs (but none after the 350-epoch mark), and the third had 3 over 120 epochs. Combined these two oddities go to show the base level of randomness a learning machine adds to the mix, which is helpful for identifying what is significant and what is not.

Third, the connection between the higher bias levels and the lower mean scores makes some sense if the biassing is harming the machines' ability to perform consistently, but the fact that the 25% bias performed better than some unbiased data seems to discredit this, and instead imply that there is some ideal amount of biassing greater than 0% which improves the mean for this scenario. This better performance is more likely a fluke, as 25% bias may have outperformed its paired unbiased trial, but that unbiased trial was also outperformed by the first unbiased trial. Thus, while further data might push it over, current data suggests the higher 25% bias performance is not high enough to be considered significant and the original point that bias correlates to lower mean scores stands.

Fourth, the similar average maximum values for most of the trial (except the 75% test, though it did not stray that far from 1000) suggests that no amount of biassing affects the average maximum value. This is strongly supported by the data gathered, but on its own is not that interesting. However, the fact that all bias levels had higher maximum value variance (except the first half of 75% bias dataset) suggests that the biassing causes the DQN to change its behaviour to adopt a strategy with a similar average maximum outcome, but more highs and lows. This is discussed in greater detail in section 6.1.

Fifth, the higher bias levels have lower average standard deviation than the unbiased and 25% biassed datasets, but higher standard deviation variance presents yet another insight into learning achieved by high bias levels. It would make sense that the lower

average come because of being closer to the absolute minimum value (-100) so there would be less room for downward deviation (this is supported by the lower average mean scores), but the higher variance show that the program is still able to occasionally hit high values (this is supported by the similar average maximum values). Thus, the program may be lower on average, but it can still achieve high results.

Sixth, the unbiased programs seem to be slightly slower than the biassed versions. Why this happens is a mystery, but the outcome is that the biassed version can gather data slightly faster than the unbiased versions. This is very strongly supported by the data and is strictly good for the biassed versions as more data makes for a better learning machine.

6 Evaluation

6.1 Correcting the Imbalance

The original reason mentioned in section 3.2.2 for biassing the data was to correct an imbalance introduced by the extreme difficulty of the scenario. This did not work. High biassing levels showed weaker mean results than unbiased counterparts, but, by accident, biassing has shown to provide some benefits outside its original intended role. The higher variance for maximum values, specifically the increase in the number of extremely high-reward maximums suggest what has occurred is the biassed learning machine has learned to take more high-risk high-reward actions. This is extremely interesting as normally machine learning discourages this kind of behaviour if the average reward is lower than that of some other action, but sometimes it is better to take risks. This is not to say that biassing is always better (excepting the performance of the 25% bias data), as it clearly came with some major disadvantages in the field of mean score, which for many problems is the most important metric for success.

6.1.1 A Pinch of Salt

Due to a combination of time constraints and time lost, the pool of data to evaluate is sadly limited, and so, despite none of the analysis being technically incorrect, it should still be taken with a pinch of salt that further data might cast some doubt. This is in conjunction with the fact that, as mentioned above, machine learning is inherently unpredictable, so it is possible the trends observed here are the result of chance and not reflective of the actual impacts of biassing the sampling of data. It could also be the case that these two factors conspired to water down the potential benefits of biassing data and overplay its weaknesses. All this combined means a further examination of the

concept of biassing is a good idea, after all, reproducibility is one of the foundations of the scientific method.

6.2 Overall Goals

With the original aim of this project having fallen by the wayside, evaluating the success or failure of the project has become a far more tangled affair. Reinforcement learning was used to create a bot which played an FPS, but is it ‘better’? Certainly, the project took a program capable of playing a ‘simple’ scenario and set it up so it could play a more complicated one, but that answer ignores the spirit of the project and the intended meaning behind the word ‘better’. Interpreting the word better in the project title to mean better than the current industry standard seemed sensible, and despite that being an incredibly lofty goal, in a way, the project has probably advanced the field in some miniscule way. As mentioned in the background section, several other studies have occurred relating to using DQN’s in video games and even FPSs, and DQN’s running on FPSs have also been used in experiments on different sampling methods, but no study appears to have covered this exact idea of biassing sampled data, so it is possible it has been overlooked or ignored because its limitations are obvious to an expert. If this is the case then this project may have shone a spark of light on an area which might have untapped potential, and that seems a good enough reason to deem this project a success, whether the idea of biassing turns out to be useful, it is still better to be certain that it does not work than to leave it unexplored. After all, in the words of Edison “I have not failed 10,000 times. I have not failed once. I have succeeded in proving that those 10,000 ways will not work. When I have eliminated the ways that will not work, I will find the way that will work” (Furr 2022).

6.3 How Valuable is This?

This report scratched the surface of an infinitesimally small fraction of the wider field of machine learning, but given some time and further exploration, it may prove fruitful enough to assist in several areas both related and unrelated to FPSs and video games in general. A high-risk high-reward machine learning algorithm could revolutionise the field of speed-running in video games and might prove a useful tool in the arsenal of the programs used to tackle some of today's major computer science problems, but whether the idea of biassed sampling is even a significant improvement is yet to be seen. On a more positive note, projects like this are useful even if they end in failure simply because they help to refine the pool of candidate ideas for the advancement of machine learning, and thus combined they silently help to bring about the big breakthrough studies and papers, and every now and then, one of them proves to be a little breakthrough of its own.

7 Future Work

7.1 Direct improvements

The program at the centre of this project might have managed to output some useful data, but it also contains several flaws, bugs, and unimplemented features which could all be added to help create cleaner and more useful data, for example, the major bug focused on in section 4.4. Also, the unimplemented features like epsilon saving and the reward for killing enemies could be added, and new features like providing the learning machine with a wider range of actions and weapons could be implemented, though at a certain point other options are available for further exploration that might prove more fruitful than expanding the program.

7.2 Extensions

This project has proved interesting and, as mentioned above, further exploration in this area might provide some interesting insight into the potential of biassed sampling. If the time and resources were made available then redoing this project from the ground up, building the whole program from scratch and taking the time to identify and eliminate bugs to end up with a platform capable of pumping out data on biassed sampling for a proper study would in and of itself be a good idea, helping to establish a more concrete foundation and perform a more comprehensive analysis.

Assuming such a study results were like the results found here, other options become available, like performing a follow-up study to determine the situations in which biassed sampling performs best and, following on from that, attempting to build an algorithm that selectively employs biassed and unbiased sampling depending on the needs of the program at a given time. This could prove helpful as machine learning is used to tackle tasks which contain many different environments, some of which are better suited to biassed or unbiased sampling.

Outside the realm of sampling, two other areas seem ripe for exploration, including an exploration of the optimal ratio of columns and rows to problem complexity for deep learning, which also looked like an underexplored area of machine learning, and alternatives to epsilon greedy reinforcement learning, again seeing if a combination of other options with strengths and flaws might prove better in certain situations. For instance, while altering the epsilon function, the idea of a dynamic greedy epsilon function which raised and lowered the epsilon value based on the highest expected reward from a certain action was conceived. Basically, in situations where the program

sees no good options it is more likely to pick a random option, but situations where it has already found a good option are less likely to have random picks.

8 Conclusion

This report details the process of solving the problem of creating “better first person shooting bots via reinforcement learning”. This goal eventually turned into “Does biassing sample data make better first person shooter bots via reinforcement learning”, and after analysis of the data and discovering that doing so caused the program to take more high-risk high-reward actions, the answer was that under certain circumstances it seems to provide some benefits. Due to limited data, these findings are not concrete and further analysis is needed to shine light on the exact nature of biassed sampling, but there was enough to begin to hypothesise what areas might benefit from biassing. Overall, the project has shone a small ray of light onto a previously dark area, and emphasises that, with some time and work, useful tools might be derived using some form of biassing.

In conclusion, this report was successful in its goal to examine a new area of Reinforcement learning and thus open the possibility for new tools to be built that could make “better first person shooting bots via reinforcement learning”.

9 Reflection on learning

9.1 General Learning

At the start of this project I knew very little about the field of machine learning beyond the basic concept and what was explained in a couple of YouTube videos, so throughout the course of the project I have learned a lot about concepts like a DQN and model-free V.S. model-based and what they mean, as well as now having a fair understanding of the way ZDoom scenarios are created and operated, and How ViZDoom’s interaction layer provides a back and forth for programs and the game itself. I have also learned about how little machine learning is used in the video game industry despite the two seeming to be a great pair. Finally, I have come around to the idea of working with music in the background to help keep focus on the task at hand and avoid distractions.

9.2 Professional Growth

Alongside the lessons I have learned about machine learning, I have also taken onboard many professional skill I was aware of but had very little or no actually experience in, for instance in my attempts to decipher the base program I tried reading through the other work by the same coder in the examples folder to see if I could gain an insight into how they think and therefore how the base code might function. I also put into practise a policy of backing up my work to multiple files in different places to prevent any time loss on recovering or remaking files. Finally, taking on board a lesson from my group project where we had several supervisor meeting in which we failed to make good use of our time despite needing help in several areas, I also began to keep track of questions that occurred to me in the leadup to my supervisor meetings so I could make better use of the time available.

9.3 Personal Growth

Throughout my time spent studying in Cardiff I have completed many assignments and attended many lectures, and in doing so I have learned a lot, but this project has provided me the chance to turn inward and learn about myself. I have watched myself decipher another person's code. I have watched myself be forced to give up on ideas I was invested in. I have watched myself fight battles with bugs through stress and covid and life in general. Throughout previous group projects I learned how to work as part of a team and even when to take charge, but in this project, I was alone, and I had to learn how to turn those skills inward. I have been forced to accept I am not quite as good a worker as I thought, and I have had to learn to work around that flaw to bring this report together. But most of all, I have watched myself grow.

In short, I have learned how to be a slightly better version of myself.

References

These papers had a preferred Reference:

Gemrot, J., Kadlec, R., Bida, M., Burkert, O., Pibil, R., Havlicek, J., Zemcak, L., Simlovic, J., Vansa, R., Stolba, M., Plch, T., Brom C.: Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. In: Agents for Games and Simulations, LNCS 5920, Springer, 2009, pp. 1-15. (PDF)

M. Kempka, M. Wydmuch, G. Runc, J. Toczek & W. Jaśkowski, ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning, IEEE Conference on Computational Intelligence and Games, pp. 341-348, Santorini, Greece, 2016 (arXiv:1605.02097)

Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare and Joelle Pineau (2018), “An Introduction to Deep Reinforcement Learning”, Foundations and Trends in Machine Learning: Vol. 11, No. 3-4. DOI: 10.1561/2200000071.

The rest are in Cardiff Harvard:

Anenberg, B. 2022. Sampling Strategies for Deep Reinforcement Learning. Available at: https://anenbergb.com/assets/pdfs/deep_rl.pdf [Accessed: 26 May 2022].

Berg, A. 2022. If It’s Fun, It’s Fun : Deep Reinforcement Learning In Unreal Tournament 2004. Available at: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1567486&dswid=-9232 [Accessed: 26 May 2022].

Deep Reinforcement Learning. 2022. Available at: https://www.deepmind.com/blog/deep-reinforcement-learning [Accessed: 26 May 2022].

Direct Machine Learning (DirectML). 2022. Available at: https://docs.microsoft.com/en-us/windows/ai/directml/dml [Accessed: 26 May 2022].

Edwards, G. et al. 2010. IEEExplore Digital Library. Choice Reviews Online 47(11), pp. 47-6268-47-6268. doi: 10.5860/choice.47-6268.

Furr, N. 2022. How Failure Taught Edison to Repeatedly Innovate. Available at: https://www.forbes.com/sites/nathanfurr/2011/06/09/how-failure-taught-edison-to-repeatedly-innovate/ [Accessed: 26 May 2022].

Gimmer, D. and Acheron, X. 2022. ZDoom - About. Available at: https://zdoom.org/about [Accessed: 26 May 2022].

Horgan, D. 2022. DISTRIBUTED PRIORITIZED EXPERIENCE REPLAY. Available at: https://arxiv.org/pdf/1803.00933.pdf [Accessed: 26 May 2022].

Keras documentation: About Keras. 2022. Available at: https://keras.io/about/ [Accessed: 26 May 2022].

Klein, D. 2022. Lecture 10: Reinforcement Learning. Available at: https://www.youtube.com/watch?v=w33Lplx49_A [Accessed: 26 May 2022].

Mnih, V. 2022. Human-level control through deep reinforcement learning. Available at: https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf [Accessed: 26 May 2022].

Pogamut - virtual characters made easy | About. 2022. Available at: http://pogamut.cuni.cz/main/tiki-index.php?page=About [Accessed: 26 May 2022].

Proximal Policy Optimization. 2022. Available at: https://openai.com/blog/openai-baselines-ppo/ [Accessed: 26 May 2022].

PyTorch documentation — PyTorch 1.11.0 documentation. 2022. Available at: https://pytorch.org/docs/stable/index.html [Accessed: 26 May 2022].

Samsami, M. and Alimadad, H. 2022. Distributed Deep Reinforcement Learning: An Overview. Available at: https://arxiv.org/pdf/2011.11012.pdf [Accessed: 26 May 2022].

Schrittwieser, J. 2022. MuZero: Mastering Go, chess, shogi and Atari without rules. Available at: https://www.deepmind.com/blog/muzero-mastering-go-chess-shogi-and-atari-without-rules [Accessed: 26 May 2022].

Source - Valve Developer Community. 2022. Available at: https://developer.valvesoftware.com/wiki/Source [Accessed: 26 May 2022].

TensorFlow. 2022. Available at: https://www.tensorflow.org/ [Accessed: 26 May 2022].

ViZDoom. 2022. Available at: http://vizdoom.cs.put.edu.pl/ [Accessed: 26 May 2022].

Wang, Z. 2022. Dueling Network Architectures for Deep Reinforcement Learning. Available at: https://arxiv.org/pdf/1511.06581.pdf [Accessed: 26 May 2022].

Watkins, C. 2022. Learning From Delayed Rewards. Available at: https://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf [Accessed: 26 May 2022].

Welcome to Lasagne — Lasagne 0.2.dev1 documentation. 2022. Available at: https://lasagne.readthedocs.io/en/latest/ [Accessed: 26 May 2022].

Ye, W. 2022. Mastering Atari Games with Limited Data. Available at: https://arxiv.org/pdf/2111.00210.pdf [Accessed: 26 May 2022].