General Deep Reinforcement Learning in NES Games

David Gregory LeBlanc[†,*], Greg Lee[†]
[†] Acadia University

Abstract

The techniques involved in general game playing with Artificial Intelligence (AI) have advanced to meet the challenges of the most popular video game and board game domains. Until recently, the video game domains used as testbeds have been relatively simple. That is, the most complex console-wide domain that has been solved using Deep Reinforcement Learning is the Atari domain, which is decades behind modern video game domains. This work explores a more complex domain (the Nintendo Entertainment System, or NES) and the associated difficulties in developing deep reinforcement learning agents for it. To understand these difficulties, we trained agents on NES games with little to no expert knowledge provided to the agents. After developing some understanding of the challenges of the domain, we suggest areas on which to focus to solve this domain, work which will hopefully lead the field to solving ever more complex environments both real world and theoretical. This paper determines some of the necessary changes in hyperparameters and reward functions to solve the NES domain compared to the more popular Atari domain while using game pixel data as the only inputs to the agents' neural networks.

Keywords: Machine Learning, Reinforcement Learning, Games, Deep Learning

1. Introduction

Deep Reinforcement Learning (Deep-RL) is one of many powerful tools in the machine learning arsenal, capable of training agents with very little user generated data; for many domains all that is required from the user is an adequately specified reward function. This generality makes Deep-RL appropriate for tackling domains with highly variable environments that present state spaces as large as are found in video games; in an average Super Mario Bros. level the player character may be at any one of 3000 x-positions at any time within a few thousand timesteps, during which they may take any of 16 actions that can change the state. These environments are analogous to real world problems because of these complexities.

Reinforcement learning has been successfully applied in the Atari domain to the extent that DeepMind's Agent 57 [1] can outperform humans across all games in its library. The next natural step is the next generation in video games – the NES. From a hardware perspective, NES games support resolutions of 256x240 (up from 160x192 for Atari), roughly 8 times more input combinations, and storage/game sizes of up to 1MB, instead of the 16kB Atari games could utilise; an approximately 60 times increase of hardware complexity.

Aside from these quantitative differences, many games within the NES domain are far more complex than those in the Atari domain. This is where video games go beyond simply requiring twitch reflexes (where computers excel), and move on to requiring critical thinking skills. This translates to the goals of players moving from easily modelled ideas of getting the highest scores to more abstract concepts such as finding the end, exploring the game's environments, or optimizing any one of the many in game statistics. We define "solving" an environment or domain as creating an AI agent that consistently achieves superhuman performance in that environment or domain.

[*] 143807L@acadiau.ca

The results of Deep-RL are highly dependent on the given reward function; the perfect learning algorithm combined with an ill-defined reward function never converges to desired behaviours. In the past (i.e. with Atari games), basing the reward functions on in-game score was sufficient, given that maximising the score would represent the fulfillment of player goals. Since most NES game designers eschewed score because of the complexity of the intended player goals, this metric is no longer helpful. For the purposes of advancing reinforcement learning, this is a plus as many real world problems similarly lack a descriptor like score.

Thus, it follows that one of the crucial steps in solving this complex domain lies in determining appropriate reward functions, which is what we do in this paper. Our most successful agent is capable of completing the first level of Super Mario Bros. within seconds of a human expert time. This paper also shows how reward functions in the NES domain must be structured so that they avoid problems of *reward farming* (undesirable agent behaviour that is rewarded) by selecting features from the game that correlate with human player goals while punishing waiting behaviours with a time penalty. In examining different styles of reward function, we also show how the -greedy exploration strategy is ineffective for providing agents with samples of sufficient quality for behavioural convergence.

The rest of this paper is structured as follows: first, we explain the sources of chosen techniques as well as what new areas are being explored in this work in "Related Work" (Section 2), followed by "Our Approach" (Section 3). We then show the results of our experiments across Pong, Arkanoid, Mega Man, and Super Mario Bros. in Section 4. Finally, we discuss the key contributions of the work and areas for future work in Section 5.

2. **Related Work**

One of the largest inspirations for this research is the work of Murphy [2], as it explores the NES domain with success. However, it requires human-expert gameplay as training data for its algorithm to learn. It also differs in that the trained agents receive the system's memory as inputs rather than this work's focus on pixel-data. It may be worth combining the Deep-RL techniques here with these ideas in future work.

Agent57 from DeepMind [1] shows how Deep-RL may be used to solve the Atari domain, which more closely resembles the methods used here in a different domain. That is, Agent57 manages superhuman performance across the Atari library with a variety of reward functions. The principle difference with this work is that we test agents in the NES domain while Agent57 focuses solely on Atari games.

AlphaStar [3] and OpenAI Five [4] have created superhuman agents for Starcraft II and Dota 2, respectively. Both of these games are more sophisticated than most of the NES library in terms of image resolution, possible outputs, and, arguably, goal complexity. However, both AlphaStar and OpenAI accomplish this performance using expert knowledge by training their agents against hand-made strategies. In contrast, we are more interested in using as little expert knowledge as possible with the aim of discovering reward functions that can apply across large sections of the video game domain.

AlphaGo and AlphaZero, which surpassed human masters in Go and Chess, respectively, show how AI may be used to solve complex domains. They differ from this work in that they operate in domains with fewer time steps than the NES domain, as a game of go or chess has a number of turns on the order of 100. In contrast, a complete playthrough of a shorter NES game like Super Mario Bros. contains 17000 time steps (5 minutes of gameplay) if played with world record speed. A more casual playthrough may go on for hours [5]. Many of these timesteps do not carry as much weight as a single turn in a game of chess, but instead an agent must be able to carry out one strategy over hundreds or thousands of time steps in small increments.

3. Our Approach

To train NES agents, we use Deep-Q learning, due to its previous success in the field of game-playing. Deep-Q involves modeling the Q, or state-action value function of the given environment, picking the action with the best predicted reward at each step. Aside from the basic implementation, we also include the enhancements of double-Q learning [6] and Prioritised Experience Replay [7] (PER). These have been shown to reduce overestimation and improve sample efficiency, respectively. Algorithm 1 shows the basic steps of this process.

Algorithm 1 Enhanced Deep-Q Algorithm

```
Require: Default priority $\beta$, batch size $N$, prioritisation factor $\eta$, discount rate $\gamma$, target
    network update parameter $u$
 1: Initialise neural network [Model] to model $Q_{eval}(s,a)$, accepting environment states.
 2: Initialise secondary network [TargetModel] to model $Q_{target}(s,a)$
 3: Initialise prioritised experience replay [Memory]
 4: for each episode do
 5:     Initialise environment for state S
 6:     while S not a terminal state do
 7:         $A \leftarrow \underset{a}{\text{argmax}} Q(S,a)$
 8:         Take action A, observe R, S' from environment
 9:         Store experience in Memory, S, A, R, S', with priority $\beta$
10:         Sample batch of size $N$ from Memory
11:         for sample in batch do
12:             $target \leftarrow reward + \gamma Q_{target}(s', \underset{a}{\text{argmax}} Q_{eval}(s,a))$
13:             Update network towards $target$
14:             $Priority \leftarrow [r + \gamma \max_{a'} Q_{target}(s',a') - Q_{eval}(s,a) + \epsilon]^{\eta}$
15:             update priority in Memory
16:         end for
17:         if u steps have passed since last TargetModel update then
18:             Update $Q_{target}$ model weights to $Q_{eval}$ weights
19:         end if
20:     end while
21: end for
```

The Deep-Q algorithm is modelled by a convolutional neural network (CNN) with the following architecture, which has been adapted from DeepMind's original Atari agent architecture, described in Table 1. [8]. Each of these layers use a *ReLU* activation function, and the model uses the Adam optimizer with mean squared error loss. Note that the input layer is linked to the resolution of the game; this is because we wish to train our agents using only image data as inputs.

Table 1. An overview of the network architecture

Layer Type	**Description**
Input	One input for each pixel in preprocessed image
2D Convolutional	32 filters, 8x8 kernel, stride 4
2D Convolutional	64 filters, 4x4 kernel, stride 2
2D Convolutional	64 filters, 2x2 kernel, stride 1
Flatten	Prepare for 1D layers
Dense	512 units
Output	One output for each selected button combination

To make the image data as learning-friendly as possible, a variety of preprocessing techniques and environmental constraints are implemented. Through OpenAI Gym Retro [9], NES games are emulated step by step, starting at some predetermined point in the game past menu navigation so the agent may focus on learning to play the games rather than learning peripheral skills such as game setup. Typically, the agent will be provided with a terminal signal for the current episode when it reaches what a human would consider a fail-state (e.g. "game over", losing a life). A proven technique called *frame-skipping*[8] is also introduced just before the emulation level, where we provide the agent with with one out of each n frames. Increasing the n value results in dividing the state space n times, but risks losing information necessary to make rational decisions in the game

The pixel data received from this emulation comes in three colour channels (256 x 240 x 3), which we condense to a single greyscale colour channel with a smaller resolution, by scaling the entire frame by some factor, typically 2 or 3 depending on the experiment. Furthermore, the value of each pixel after this process is scaled to a value between 0 and 1. All of this helps in reducing the state space available to the agent, which is especially important given that the state space for any given game from the NES library prohibitively large due to the free rein. Additionally, we perform these state space reductions to make room for the principle enchancement to the data: *frame-stacking*.

In frame-stacking a circular buffer of frames is stored which is fed to the agent, with the goal of imparting a sense of time to the agent. This buffer stores the past m frames which have been seen by the agent, with the most recent frame pushing the least recent from the buffer. Without frame stacking, a Deep-RL agent will fail to converge on an effective strategy on even the most simple video games such as Atari's Pong. This also means increasing the amount of input data m times, where m is the number of frames stacked.

Lastly, and most importantly, we seek to optimize the reward function.. We limit the reward to the value range $[-1.0, 1.0]$ (large values can produce undesirable behaviour), and impart some constant penalty (negative reward) per frame such as -0.1 or 0.01 depending on how sparse the reward signal is otherwise expected to be. This is because with a sparse reward signal, there is a risk that the time penalty becomes greater than any positive rewards, resulting in defective agents. We can also safely assign a penalty of -1 whenever the agent receives a terminal signal, as this usually means that the agent has failed to complete the game as shown in Section 4.

4. **Experiments**

4.1. **Hyperparameter Selection**

Before experimenting with reward functions, we determined hyperparameters through a mixture of analysis of other experiments and empirical observation. The learning rate, variables (for the -greedy strategy), and Prioritised Experience Replay hyperparameters we utilise are similar to those chosen for training agents exclusively in the Atari domain. The hyperparameters with the largest impact on learning were frame stacking and frame skipping m and n values, hence both were tested iteratively in Pong, due to its simplicity among games in the video game domain (it lacks some possible confounding variables found with agents in the NES domain, such as increased reward variance). Figure 1 shows that stacking with $m = 4$ is most effective, as lower values do not converge to as high a reward, while higher values introduce more noise.

Figure 2 shows that agents trained with a frame skipping $n = 3$ are the quickest at converging. This is because higher values skip too high a sample variety for the agent, while lower value provide the agent with too many similar samples.

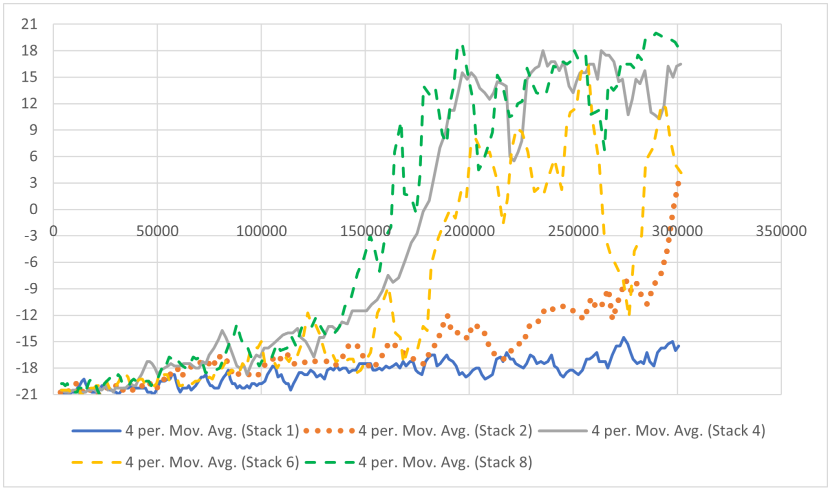

Figure 1. Results from training agents in Pong for 300 000 steps (reward against timesteps), varying frame stack m value.

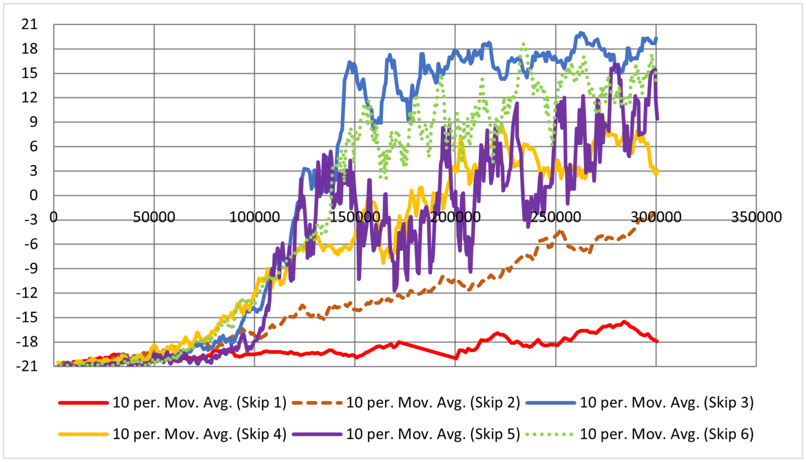

Figure 2. Results from training agents in Pong for 300 000 steps (reward against timesteps), varying frame skip n value.

4.2. **Arkanoid (NES) vs Pong (Atari)**

Our approach was first tested on Atari Pong, following DeepMind's work on Atari. It follows that one of the first games from the NES library to test Deep-Q is Arkanoid, since Pong and Arkanoid are games where the player controls a paddle and tries to prevent a ball from moving past it.Figure 3 shows the visual similarities between the two games.

To keep the comparison as direct as possible, we base the reward function on the score in Arkanoid, which is improved by destroying blocks at the top of the screen with the ball. Because the NES has many inherent added difficulties to learning (e.g. the higher state / input space, goal complexity), we give the Arkanoid agent 5 million learning steps compared to 300 000 learning steps for Pong. There is room for improvement for agents (beyond 5 million steps), but training was halted there as returns were diminishing with further training.

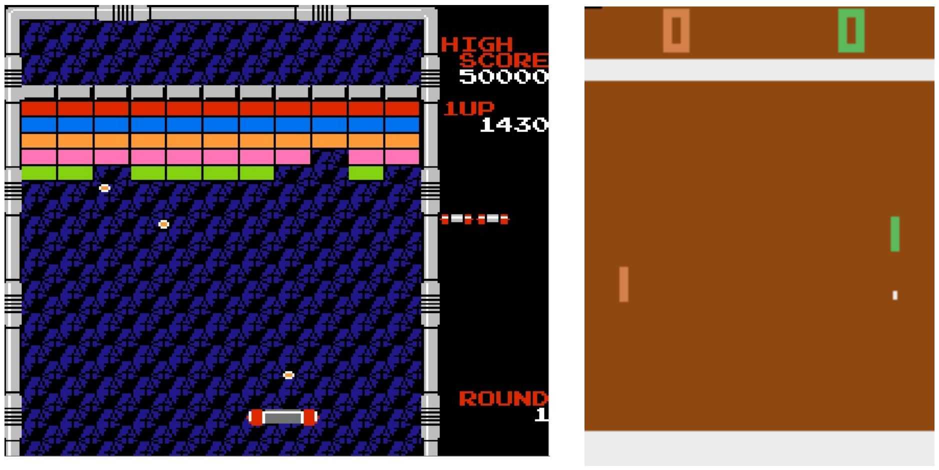

Figure 3. Screenshots of Arkanoid (left) and Pong (right).

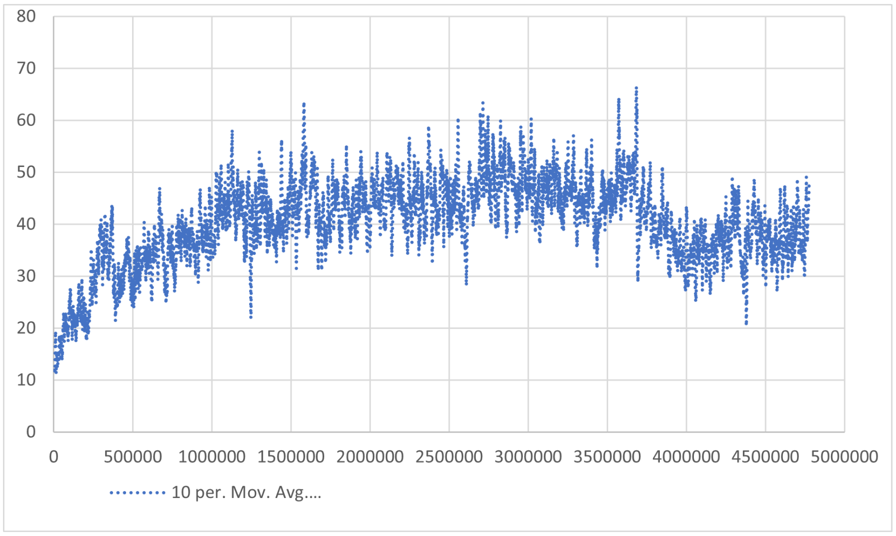

Figure 4. The agent's achieved reward per episode across 5 million steps in Arkanoid

Figure 4 shows that the Arkanoid agent's performance is noisy, even after averaging the reward across multiple episodes. This is particularly apparent when comparing the results to the Pong agent shown in Figure 5. The Pong rewards are stable throughout its episodes. That is, the agent's in game performance is qualitatively similar enough between episodes to observe a clear improvement over time. With the Arkanoid agent however, the noise is such that when observing a single episode, it is unclear whether the achieved reward is close to the average or some extreme high / low and thus the amount of learning is much more difficult to discern.

There is also a difference in the overall shape of the agent's learning curves; the Pong agent converges to a value close to the maximum score possible after only 80 000 steps, following a steep increase in achieved reward (6 times the preceding improvement) while the Arkanoid agent sees only a modest increase (2-3 times preceding improvement). This 80 000 step mark roughly lines up with the end of the exploration phase (-greedy strategy hits its minimum).

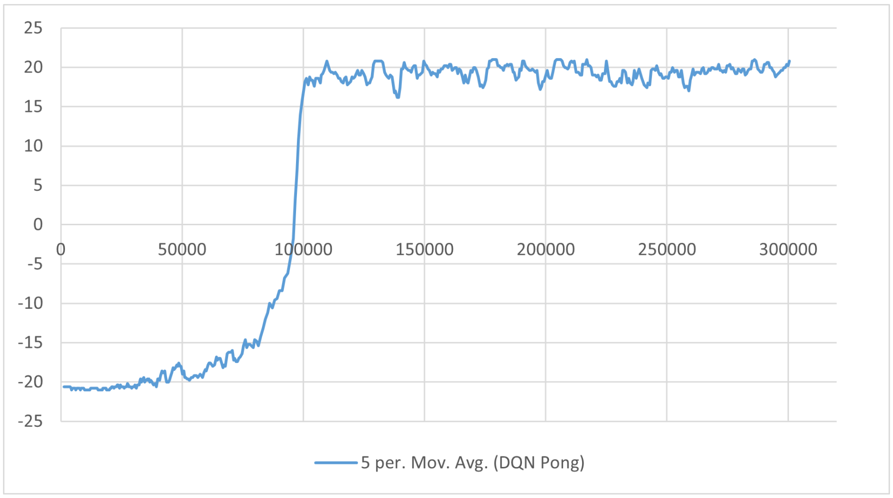

Figure 5. The agent's achieved reward per episode across 300 000 steps in Pong

Thus, the exploration stage for the Pong agent is entirely successful in providing the agent with sufficient training data to navigate the environment with minimal exploration from that point on. There is a sharp increase in the Arkanoid agent's reward around the same point, but the agent has not solved the environment, achieving only about 10% (see Table 2) of human performance on the first level alone, thus, the -greedy exploration strategy is not as effective at finding the global maximum in Arkanoid as it is in Pong.

Table 2. Human and agent performance in Pong and the first level of Arkanoid

	Human	Agent Best
Pong	21	21
Arkanoid (First Level)	1435	148

Some of the difference in performance between Pong and Arkanoid may be accounted for by the increase in visual noise presented to the agent; Arkanoid features a patterned background, flashing life display, and animated enemies which may cause the convolutional layers in the CNN to not discover the most important visual features such as the ball and paddle.

4.3. **Mega Man (NES)**

Mega Man is an example of a game from the NES library where game complexity increases from previous generations. The player is given a more sophisticated control scheme and must navigate a level with multiple unique obstacles. Figure 6 shows the overall structure of one level (the player starts in the bottom left and must make their way to topmost right room), which demonstrates how a successful player must learn and utilise a number of different strategies to complete the level.

Unlike Pong and many other Atari games, Mega Man does not progress without appropriate player input. So, without appropriate inputs, the agent can fail to trigger anything in the environment at all. This leads to the question of how to develop a suitable reward function. Mega Man has a score counter that increments by accomplishing *some* of goals surrounding finishing the level.

Therefore, basing a reward function on the score and applying a time penalty that may encourage the agents to seek out the rewards is appropriate. Figure 7 shows the performance of an agent through 2.2 million learning steps.

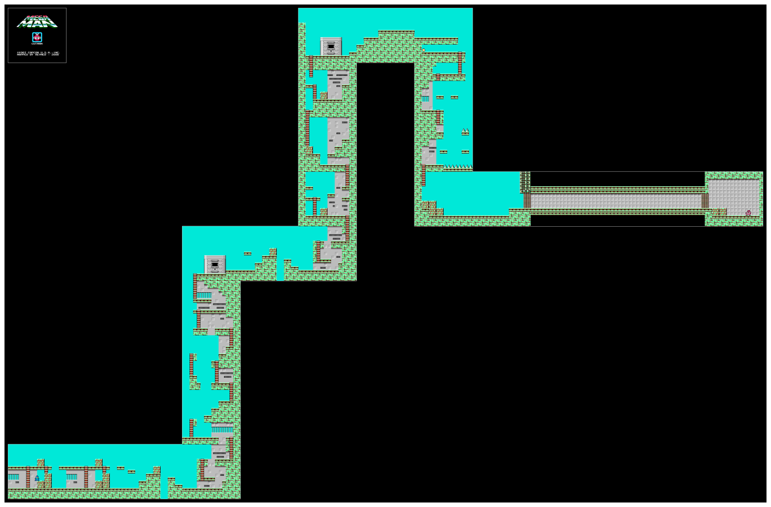

Figure 6. A stage in Mega Man

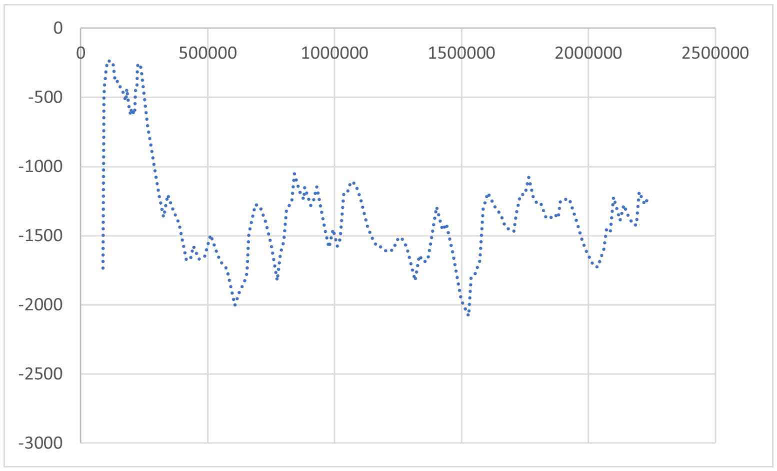

Figure 7. Agent rewards in Mega Man over 2.2 million timesteps.

Not only are the rewards negative from the time penalty, but the agent actually appears to learn how to reduce this reward from episode to episode. One of the actions that increases the score counter is defeating the constantly respawning enemies in the game. Unfortunately for the agent, the reward from this is quite low given that enemies take some time to defeat. Since enemies are the first accessible score generator for the agent in this particular level, the agent learns a policy to defeat and respawn the enemies reasonably efficiently.

Fighting the enemies presents some risk to the agent (it can lose the episode from taking too many hits), and this is what causes the episodes to actually end. By the end of these training sessions it is clear that this is the only thing the agent will ever accomplish, even though it is far from the true goal of the game: reaching the end of the level.

We describe this brand of defective behaviour as *reward farming*, which happens when the reward function does not properly incentivise agents towards reaching the true goal. This is an illustration of why it is difficult to design reward functions as environments become more complex, and in the next experiment we can explore intelligent alternatives that result in more desirable agent behaviours.

4.4. **Super Mario Bros. (NES)**

Many games in the NES library can be completed by following the principle of 'move to the right'. This strategy works for most 'sidescroller' games. For this reason, in Super Mario Bros., positive reward is granted for scrolling the screen right (accomplished by moving the player character right) while still penalizing the agent for time taken. Figure 8 shows the agent's performance over 5 million steps using a Deep-Q CNN.

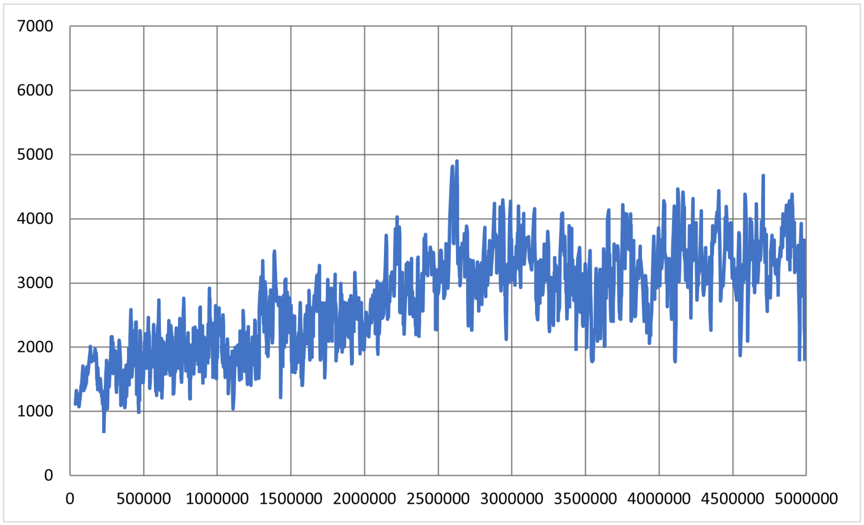

Figure 8. Agent rewards in Mario without a time penalty over 5 million steps

There is noise comparable to that of the agent playing Arkanoid, but there is no dip in achieved reward towards the end of training. Another similarity between the two results is the two graphs show a decrease in rate of learning towards the end of training (i.e. the reward curve flattens) despite the distance from the global maximum (which would be found towards the end of both games).

However, comparing the two agents may be unfair due to the difference in game genre. As Mega Man is within the same genre as Super Mario Bros., we can directly compare the efficacy of the two reward functions. Firstly, we see a positive change in reward over time instead of the negative change shown in Figure 7, which is desirable. Despite the noise, we still see a more consistent gradient with the Mario agent over the Mega Man agent. Therefore, we conclude that the Super Mario Bros. agent's reward function better allows the agent to learn how to optimise its behaviour. We also do not observe any short-term reward farming from the Super Mario Bros. agent.

However, there are oddities in the agent's behaviour that suggests reward farming on a long-term scale. One of the most prevalent oddities is the agent's tendency to stop moving at a particular point in level 1-1, near the end. While the agent eventually moves on with the level, this may be an indication that it is attempting to run out the in-game clock so that it may restart the level, bringing it to a state where it can comfortably achieve reward.

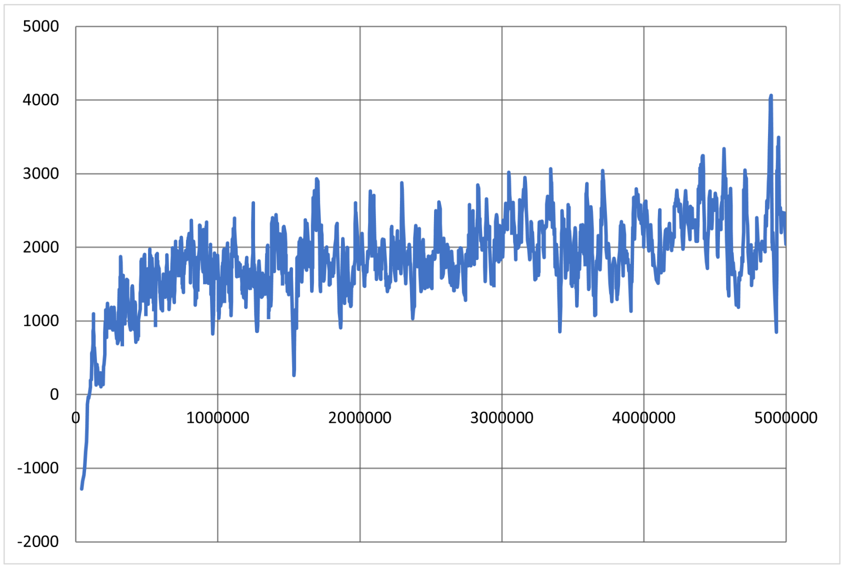

Figure 9. Agent rewards in Mario with a time penalty of -0.1 per frame over 5 million steps

Another piece of evidence towards this hypothesis is that in any episode where the agent reaches the second level, it will quickly run out of lives, potentially because it cannot achieve reward as effectively as it can in the first level. An alternate explanation to the behavioural oddities is that the agent is confounded by the differences between the first and second levels. Figure 10 shows screenshots from the first and second levels before preprocessing. Note the difference in colour palette and level layout which may be confounding the agent.

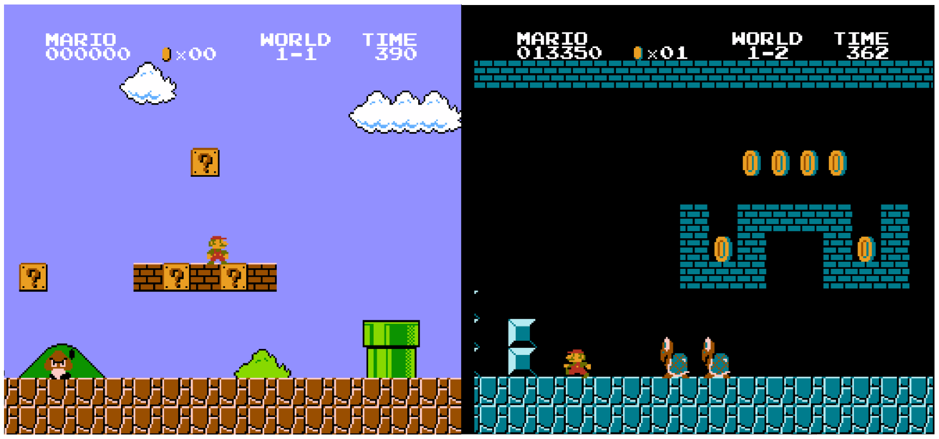

Figure 10. Screenshots of the first two Super Mario Bros. levels prior to preprocessing.

In either case, this is a sign that the agent is having difficulties generalising its positive behaviours across levels in Super Mario Bros. Within the first level, the agent's behaviour is competitive with human times on its best attempts. We observed a practised human time of 27 seconds for the first level, while our agent managed to complete the same level in 30 seconds. For perspective, a more novice human would complete the level with a time of 40-60 seconds. In experimenting, we also see the influence of the time penalty, as tests without it result in agents that complete the level less often and with a best time of 213 seconds.

It may be that further tuning of the penalty alone could result in agents that surpass the human time. Without the time penalty, there are more oddities in the agent, as it tends to stop at multiple points in the level for extended periods of time. The penalised agent also lessens the noisiness of the achieved reward; it manages to score a reward variance 37% lower than the unpenalised agent.

On the attempts that are on pace with human performance, the agent occasionally makes it halfway through the second level, especially towards the end of the training session. This suggests that the agent may be capable of completing larger portions of the game if it were provided with a combination of a longer training time and a more sophisticated neural network architecture.

One aspect of the agent's behaviour that differs from typical human behaviour in the game is that it utilises a very risky strategy; it plays close to many of the hazards in the level (e.g. jumping close to pits and enemies) while also avoiding any of the various powerups located in the level. This means that in just a few inputs, the agent is at risk of losing the game due to its proximity to the hazards without the safety the powerups grant. A human might pick up these powerups and avoid these obstacles to increase their chances of making it farther in game. Because the agent is most rewarded for high amounts of short term rewards, this strategy is reinforced as the agent does not need to slow down to avoid obstacles and collect powerups. However, because this strategy often places the agent near failure states, the -greedy exploration strategy occasionally chooses doomed actions that would be insignificant with a safer gameplay strategy. This may be remedied by instead adopting a Boltzmann exploration strategy so that the agent could learn that some actions will never be worth exploring given a dangerous state (for example, the agent would not halt a jump halfway across a pit). Therefore, -greedy exploration is incompatible with this style of reward function, but the right scrolling reward function may not be optimal.

The agent's risky behaviour resembles human expert gameplay, where safe strategies are ignored in the interest of saving time on each level (which is also reflected in its best level time). This style would be a poor choice for a human to try on learning the game, so it may be the case that learning a safer strategy would result in a more well-rounded agent. Perhaps the best agent would start out using safe strategies and as training progressed, shift towards learning riskier, but more rewarding, strategies.

5. **Conclusion and Future Work**

Deep-Q may be suitable for creating agents that compete with humans in the NES domain with little to no expert knowledge. The chosen reward function for the agents is key to their success, and some reward functions are applicable across significant portions of the NES library. In particular, we have found that a scroll-based reward function is effective for games in the sidescroller genre. NES games typically scroll some small number of pixels at a time, which in theory could be identified by calculating if the current frame is at least mostly a translation of the previous frame. This could be done either by taking the previous image and testing for similarities pixel by pixel or by maintaining a separate machine learning agent that could recognise scrolling. Combining scrolling recognition with a constant time penalty also appears to be key to the success of the agents, as without it they engage in more undesirable behaviours and there is more noise in acquired reward.

-greedy is not suited to the larger environments that the NES library presents for a number of reasons. Firstly, it does not include any mechanism for risk-avoidance, which, paired with the agents' tendency to follow a risky strategy makes for a situation where at many moments the agent can lose the current episode with but a few poor rolls of the -greedy dice.

A more suitable exploration strategy would learn that some actions are always poor choices, favouring longer episodes that could reveal larger portions of the environment to the agents.

The scroll-based reward function achieves higher reward because it models the steps required to accomplish the games' goal; reaching the end of every level which is invariably located to the right of the player's starting position. It is important that the reward function is at least principally based on these required steps, as we observe that when proximal goals (steps that can make the end goal easier to reach) are the primary basis to the reward function, we obtain agents with reward farming behaviour. However, an agent may benefit from an auxiliary reward function based on proximal goals as they can shape the agent's gameplay strategy into appropriately safe territory. For example, our Super Mario agent may have been able to proceed farther into the second level if it had been rewarded for picking up powerups during the first level. This would be distinguished from the primary reward function in that the amount of reward obtainable would be limited to avoid reward farming.

Reward functions based on score (when a score counter is present) may be effective for a smaller subset of games on the NES platform. Agents with these reward functions do struggle to perform well past the initial sections of a game, therefore we would expect the agents performance to be best in games with a short length. This would include many sports games within the NES catalogue, as they often feature a small number of unique screens and sprites (e.g. Tennis [NES]). Note that score in these games does not simply follow the intermediate goals of the game, it is indeed the only goal of the games.

To achieve superhuman performance for the entirety of lengthier games such as Super Mario, the agents require some adjustments. This would include training for extended periods of time beyond 5 million learning steps, as well as more sophisticated neural network architectures. We also see that additional work may be necessary for enabling agents to generalise their knowledge across different areas within the same game, shown by the agent having difficulty in the second level of Super Mario after near human performance in the first level.

References

[1] Deepmind. *Agent57: Outperforming the Human Atari Benchmark.* 2020. URL: https://deepmind.com/blog/article/Agent57-Outperforming-the-human-Atari-benchmark.

[2] T. Murphy VII. "The First Level of Super Mario Bros. is Easy with Lexicographic Orderings and Time Travel ...after that it gets a little tricky". In: Jan. 2013.

[3] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, and et al. *Grandmaster level in StarCraft II using multi-agent reinforcement learning.* 2019. URL: https://www.nature.com/articles/s41586-019-1724-z#citeas.

[4] OpenAI et al. "Dota 2 with Large Scale Deep Reinforcement Learning". In: (2019). arXiv: 1912.06680 [cs.LG].

[5] *How long is Super Mario Bros.?* URL: https://howlongtobeat.com/game.php?id=9371.

[6] H. van Hasselt, A. Guez, and D. Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[7] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. *Prioritized Experience Replay.* 2016. arXiv: 1511.05952 [cs.LG].

[8] D. Technologies. *Playing Atari with Deep Reinforcement Learning.* 2013. URL: https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning.

[9] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman. "Gotta Learn Fast: A New Benchmark for Generalization in RL". In: *arXiv preprint arXiv:1804.03720* (2018).