The benefits of Credit Assignment in noisy video game environments

Bachelor's Project Thesis

Jacob Schoemaker, s3793435, j.t.schoemaker@student.rug.nl
Supervisor: Prof Karine Miras

Abstract: Both Evolutionary Algorithms (EAs) and Reinforcement Learning Algorithms (RLAs) have proven successful in policy optimisation tasks, but there are very few comparisons of their relative strengths and weaknesses. This makes it difficult to determine which group of algorithms is best suited for a task. This paper presents a comparison of two EAs and two RLAs in Evoman, a new benchmark domain. It compares the algorithms both when there is no variation between environments, and when noise is introduced in the initialisation of the environment. We conclude that whilst EAs reach a similar performance to the RLAs in the static environments, when noise is introduced the credit assignment done by RLAs gives them an edge in modeling the underlying environment.

1 Introduction

Machine Learning has found some success in video games with the goal of creating autonomous agents to achieve a specific task, referred to as Computational Intelligence in games (Lucas, 2008). Two prominent methods in this field have been the application of evolutionary algorithms (Ishikawa, Trovões, Carmo, França, & Fantinato, 2020; Hausknecht, Lehman, Miikkulainen, & Stone, 2014) and Reinforcement Learning (Crespo & Wichert, 2020; Givigi, Schwartz, & Lu, 2010; LeBlanc & Lee, 2021).

Creating an agent that can achieve human-level performance in games has uses such as testing the difficulty of (procedurally generated) opponents automatically (Promsutipong & Kotrajaras, 2017), or creating a more interesting opponent for single-player games (Moriyama et al., 2014). Furthermore, the varied nature of games provides a great testbed for the robustness of AI algorithms. They allow us to test the performance and dynamics of an algorithm in a variety of situations. If an algorithm manages to reach robust performance in multiple diverse game environments, it could be indicative of performance in real-world scenarios with similar properties.

Evolutionary Algorithms (EAs) evolve a policy through selection, mutations and crossover. A group of individuals is generated and evaluated in the environment by playing an entire episode and getting assigned a fitness based on its performance. The individuals with the best fitness are then selected from the group, and their genomes are used to generate more individuals for the next generation. Some individuals might be subjected to random, small changes to their genomes, which allows the population as a whole to explore the state space.

Reinforcement Learning Algorithms (RLAs) on the other hand function by training a single individual to learn the task at hand by means of exploration, and updating their policy through feedback given by the environment. After every action, the RLA agent gets the new state and some reward value. The agent attempts to learn what action gives the most (eventual) reward in a given state by assigning credit to the state-action pairs that lead to the largest amount of reward. For small problems, the entire state-action representation can be stored, but for problems with a large state space, a function approximator such as a neural network is often used.

This paper will explore these different types of algorithms in two dimensions: The nature of the algorithm, and the complexity of the algorithm. To achieve this, two algorithms of both classes will be used, one simple one and one complex one. As the testbed we have used Evoman, a clone of MegaMan II created by da Silva Miras de Araújo & de França (2016) to facilitate experimentation using computational optimisation techniques, such as evolutionary algorithms which have yielded some degrees of success (da Silva Miras de Araujo & de Franca, 2016; Ishikawa et al., 2020).

The algorithms will all be trained/evolved for an approximately equal amount of time in the environment before evaluating their final performance. The algorithms will be trained in each environment twice: once with the initial position of the enemy being the exact same between runs, and once with the initial position being randomly offset by a couple of pixels, to test their robustness to noise in the environment. The goal is to investigate whether credit assignment, as seen in RLAs, leads to more robust performance than algorithms which are incapable of assigning credit to actions, e.g., EAs.

The motivation for this comparison stems from the differences between EAs and RLAs, and the trade-offs and benefits that these approaches have. Whilst there are quite some differences between EAs and RLAs, in this paper we will be focusing on credit assignment.

RLAs and EAs receive feedback at different points in time. EAs receive eventual feedback at the end of an episode, and thus receive feedback over their entire performance. In contrast, RLAs receive immediate feedback after every action, which allows them to assign credit for rewards to specific actions.

In Evoman, the only way to damage the enemy is by shooting them, and as a projectile takes time to travel, this will lead to delayed reward. Delayed reward is a difficult problem for RLAs (Sutton, 1992), as RLAs rely on assigning credit to the actions most influential to the reward they gained. This is often addressed using discounted rewards, where some fraction of the total reward in the previous time step is added to the reward of the current time step.

To EAs delayed reward poses no problem as they don't receive their feedback until the end of an episode, and as such always receive the total reward gained during the episode, called their fitness. However, this means that EAs are incapable of assigning credit to actions, which could mean they evolve policies that are less robust to noise in the environment.

This paper seeks to determine whether the benefits gained from a a robustness against delayed reward weighs out the benefits of assigning credit to specific state-action pairs, and addresses the question "Does credit assignment result in better performance in noisy video game environments?". Furthermore, it also contributes with an extension to the Evoman testbed to facilitate the training of RLAs, and provides a baseline for RLAs in the Evoman environment.

2 Related Work

Due to the usefulness of games as benchmarks for understanding the behaviour of AI algorithms, they have been used extensively in prior AI research. The Arcade Learning Environment (ALE) has been used as a benchmark for performance of Computational Intelligence since it were first introduced by Bellemare, Naddaf, Veness, & Bowling (2013). It consists of over 50 games originally designed for the Atari 2600, each of which provide a setting interesting enough to be representative of a real world scenario, free from the experimenter's bias as it has been created by a third party. Atari games are also simple enough that it is possible to emulate them much faster than real time. With advances being made toward beating the human benchmark performance on the ALE, most notably by the AI agent Agent57 by Badia et al. (2020), researchers have been looking towards games from the next generation of consoles, the Nintendo Entertainment System (NES) (LeBlanc & Lee, 2021; Murphy, 2013).

One such game is Mega Man II, a challenging platforming game developed by CapCom in the 1980s for the NES. This game includes several one vs one combat scenarios with various mechanics. These fights have been emulated in a public domain clone of the original game, Evoman, created and described in da Silva Miras de Araújo & de França (2016). This clone serves as a testbed for one of these next-generation games. It is designed to facilitate the easy learning using computational optimisation techniques, such as evolutionary algorithms which have yielded some degrees of success (da Silva Miras de Araujo & de Franca, 2016; Ishikawa et al., 2020).

EAs and RLAs have previously been compared in Rieser, Robinson, Murray-Rust, & Rounsevell (2011) and Taylor, Whiteson, & Stone (2006).

Rieser et al. (2011) compared SARSA (RLA) and a simple, binary GA (EA), and found SARSA to perform significantly better than the GA when there was uncertainty in the environment. It also found that the more uncertain the environment became, the more of an advantage SARSA gained.

In contrast, Taylor et al. (2006) found NEAT (EA) to perform better than SARSA in a partially observable task with noisy sensors. When the environment was made fully observable however, SARSA outperformed NEAT.

Since the Evoman Environment is fully observable, and the sensors have no noise, these results indicate that we can expect RLAs to perform better in the noisy environments. In addition, we can expect EAs performing on-par or better than RLAs in environments without noise.

3 Environment

The EvoMan environment is a reimplementation of the boss fights of the game MegaMan II introduced by da Silva Miras de Araújo & de França (2016). It serves as a test ground for EAs and has been adapted into an OpenAI Gym (Brockman et al., 2016) environment for this paper to fascilitate RLAs. The goal of the player is to deplete the energy of the enemy by shooting at it. Meanwhile the player has to avoid the enemy and its projectiles (up to 8 at a time), which will deplete the player's energy.

The player (or agent) can take 5 actions

- Move right
- Move left
- Jump
- Shoot
- Release[*]

The environment returns 20 values every timestep, representing the current state of the environment. These values consist of the following:

[*]Release is equivalent to a human player letting go of the jump button. This allows for an early cut-off to the upwards momentum, resulting in a lower jump

- Enemy's x position relative to the player
- Enemy's y position relative to the player
- The direction the player is facing (represented as 1 or 0)
- The direction the enemy is facing (represented as 1 or 0)
- Hostile projectile's x position relative to the player (8×)
- Hostile projectile's y position relative to the player (8×)

3.1 Enemies

For this paper a subset of 4 enemies was used, as they were deemed to be have the most diverse set of game mechanics. All enemies deal a small amount of damage to the player every timestep the player is in contact with the enemy. A visual example of each of the enemies is available here.

3.1.1 AirMan

AirMan was chosen because it is very easy to beat, and thus makes for a good baseline benchmark.

AirMan starts off on the side of the arena opposite the player. He shoots out 6 projectiles in a specific pattern around the arena, which hover for a little bit, and then move away from AirMan. As the projectiles move away from AirMan, the player is moved in the same direction. AirMan repeats this move 3 times, before moving to the other side of the screen, and repeating the move 3 times, before moving back to his original position. The projectiles shot by AirMan block the player's projectiles, and damage the player a small amount when they hit the player.

This makes for an easy enemy, as the target is mostly static, and thus easy to aim at, and the projectiles are always in the same location and thus an easy avoidance move can be found.

3.1.2 BubbleMan

BubbleMan was chosen as it has a special peculiarity in its arena which makes for an interesting case.

BubbleMan starts at the opposite side of the arena as the player. He shoots two bubbles that move in a zig-zag motion towards the opposite side of the screen, and 6 bullets that go just over the player's head if the player does not jump. BubbleMan then moves forward by around 70% of the screen, and shoots another 6 projectiles. He then moves the last 30% and turns around, after which he repeats this same pattern, but mirrored. In BubbleMan's arena, gravity is reduced, making the player jump much higher. The top of the arena is lined with spikes which kill the player instantly on touching them.

The difficulty of this environment is jumping over the projectiles, whilst releasing the jump in time in order to avoid the spikes which line the top of the arena.

3.1.3 FlashMan

FlashMan was chosen as it is not particularly difficult to beat, but very difficult to get a high score on.

FlashMan starts off at the opposite side to the arena as the player, and gradually approaches the player. Its arena is made of multiple platforms of different heights, making it impossible to just walk left or right. Every once in a while FlashMan will completely freeze the player and his own movements, and proceed to shoot a large amount of projectiles right in front of him. The player and enemy stay frozen until the bullets have disappeared.

The difficulty in this enemy is making sure the player is in the right place at the right time. The player needs to line themselves up with the enemy to be able to hit them with their projectile, but if they are lined up with the enemy at the wrong time, they could sustain a lot of damage whilst being powerless to avoid the projectiles in real-time.

3.1.4 HeatMan

HeatMan was chosen as it has an interesting trait, and was found to be difficult to optimise for by da Silva Miras de Araujo & de Franca (2016).

HeatMan starts off at the opposite side of the arena to the player. He shoots multiple projectiles at the player which remain on the ground for a small amount of time, damaging the player if they step on them. When the player hits HeatMan, he turns into a ball for a moment, during which they can not be damaged by the player. Whilst in this ball form, HeatMan rapidly moves to the other side of the arena, damaging the player if they come in contact with the player. HeatMan will then resume his normal attacks from the other side of the screen, mirrored.

The difficulty of this enemy comes from the fact that the player needs to both jump over existing, static particles, and needs to dodge an enemy that is invulnerable during its time of movement.

3.2 Evaluation function

The evaluation function was chosen such that the total reward gathered by an RLA over a single episode would be equivalent to the fitness assigned to an EA playing the exact same episode.

For the RLAs, the reward returned at each time step is the damage done to the enemy multiplied by a hyperparameter $e\_weight \times damageDone$, minus the damage taken by the player, multiplied by a hyperparameter $p\_weight \times damageTaken$, leading to Equation 3.2. For the EAs, these rewards are summed over the course of a full episode to determine the fitness of the individual (Equation 3.3).

During training, e_weight and p_weight were both set to 0.5, as this was empirically found to yield the best reward out of the tested values

$$[e\_weight, p\_weight] = [\alpha, 1 - \alpha] \tag{3.1}$$

where $\alpha = [0.0, 0.1, ..., 1.0]$.

This leads to the eventual equations

$$reward_n = 0.5 \times DD_n - 0.5 \times DT_n \tag{3.2}$$

$$fitness = \sum_{n=0}^{N} 0.5 \times DD_n - 0.5 \times DT_n \tag{3.3}$$

Where DD = damage dealt to enemy, DT = damage taken by player, n = the current timestep, and N = length of the episode.

3.3 Noise

Noise can be introduced into the environment by having the position of the enemy randomised at the start of each episode. When noise is enabled, the starting x position of each enemy takes one of 4 values, sampled from a uniform distribution. The

Enemy	Default	Noisy options
FlashMan	640	640, 500, 400, 300
AirMan	588	630, 610, 560, 530
HeatMan	588	640, 500, 400, 300
BubbleMan	635	640, 500, 400, 300

Table 3.1: Starting x positions for each enemy. In a non-noisy the default position is always used. In a noisy environment, the starting position is one of the 'Noisy options', sampled from a uniform distribution.

random starting locations are displayed with the default starting location in Table 3.1

4 Algorithms

In this paper we will compare four different algorithms in different variations of the EvoMan environment. We have chosen for 2 EAs and 2 RLAs, a simple and an advanced one from each category to see how they compare. All algorithms have been trained for 2.5×10^6 'Timestep equivalents'. In preliminary testing, it was found that in most environments a player would either die or win within about 250 timesteps. Thus, for the evolutionary algorithms, which both used a population size of 100, and both evolved for 100 generations, this came out to about $100*100*250 = 2.5 \times 10^6$ timesteps, which was taken as the amount of timesteps for the RLAs to train for.

4.1 Evolutionary Algorithms

4.1.1 Genetic Algorithm

For the simple EA, we used a self-designed EA, which we will refer to as "Genetic Algorithm" (GA) in this paper for simplicity sake. Our genetic algorithm works by generating a population of Neural Networks (NNs), and testing how well they perform in the environment you are trying to optimise for. The first generation has the weights for the NNs randomised, and each subsequent generation the population is composed of combinations of the 'good' genomes (determined by some selection function) from the previous generations, with random mutations. In this way, they mirror how evolution works in real life. The pseudocode used for our algorithm can be found in algorithms A.1 and A.2. Some short descriptions for the functions are given in Table A.1.

The initial population is generated with random values for each of its weights, and during the mutation each weight has a 20% chance of changed by a value taken from a normal distribution with a mean of 0 and a standard deviation of 1.

Our GA differs from 'simple genetic algorithms' found in literature in that each neuron takes a continuous input, and give a continuous output, whereas the ones found in literature take in and output binary values.

For this paper we used a genetic algorithm consisting of 20 input neurons, a densely connected layer of 50 hidden neurons, followed by a densely connected layer of the 5 output neurons.

4.1.2 NeuroEvolution of Augmenting Topologies

For the advanced EA, we used NeuroEvolution of Augmenting Topologies (NEAT), introduced by Stanley & Miikkulainen (2002). It is genetic algorithm which evolves its own topology together with the weights, with the goal of evolving the simplest topology to solve the task at hand. To achieve this goal, every individual starts off as a simple perceptron, and gains hidden neurons over time. In addition to the mutation, crossover and selection seen in most genetic algorithms, NEAT also makes use of speciation to protect topological innovation.

To evolve a topological structure, NEAT encodes its genome as a list of *connection genes* and *node genes*. The node genes encode the inputs, hidden nodes and outputs, with the connection genes encoding the connections between nodes, and their weights.

The weights of the connection genes mutate like any other genetic algorithm, with each weight either being randomly adjusted or not. It is also possible for a new connection to be added between two unconnected nodes, which is initialised with a random weight.

A node can be added by disabling an existing connection, and replacing it with a new node with two connections, connecting the originally two connected nodes to the newly added node. The connection *into* the newly added node is given a weight of 1, with the *outgoing* connection inheriting the weight of the original connection. This serves to minimise the initial effect of the mutation.

Any newly generated node or connection is given an incrementally assigned innovation number, which serves as a historical marker used in crossover and speciation.

During crossover, genes with the same innovation number will be chosen randomly from either parent. Genes that don't appear in the other genome's genes are either disjoint, or excess. A gene is disjoint if it does not appear in the other genome's genes, but its innovation number is within the range of innovation numbers in the other genome. A gene is excess if it does not appear in the range of innovation numbers in the other genome. All disjoint and excess genes of the *parent with the higher fitness* are included in the offspring.

The genomes in a population are divided into species based on the differences in network topologies and weights. The compatibility distance between two genomes is represented by their compatibility distance δ, which is calculated as

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} \tag{4.1}$$

where E represents the number of excess genes, D the number of disjoint genes, and $\overline{W}$ the average weight differences of all matching genes. c_1, c_2, and c_3 are coefficients adjusting the importance of their relevant factors. N is the total number of genes in the largest genome, which normalises for genome size.

A species is created by evaluating each genome sequentially. Its compatibility distance to the first member of a species is calculated. If it is beneath a certain threshold, it is part of that species. If a genome is not selected to be part of any existing species, a new species is created with it as its first member.

Selection is done mostly within a species. Each individual i is assigned an adjusted fitness f'_i, calculated as

$$f'_i = \frac{f_i}{S} \tag{4.2}$$

with S being the amount of individuals in the species i belongs to. This penalises species that grow too large, to make sure proper speciation can occur.

Every species is assigned an amount of offspring proportional to the sum of the adjusted fitness of all individuals in the species. The worst individuals of the species are then eliminated, and the remaining individuals are used to generate the specified amount of offspring. The next generation consists purely off the generated offspring.

4.2 Reinforcement Learning Algorithms

4.2.1 Deep Q-Networks

For the simple RLA, Deep Q-Networks (DQNs) were used, introduced by Mnih et al. (2013). DQNs are a Neural Network extension to Q-learning, with the high level idea to make Q-Learning problem look like a supervised learning problem. It employs two important ideas for stabilising Q-learning.

- Use a replay buffer, which stores a large amount of state transitions, which minibatches can be sampled from and trained on.
- A secondary copy of the NN is kept and updated less frequently which is used to compute the target values. This is to keep the target function from changing too quickly, and avoid chasing a moving target.

The algorithm used is described in Algorithm A.3, with the topology of the used network consisting of the 20 input neurons, followed by a densely connected layer of 64 neurons, followed by a densely connected layer of 32 (2^5) output neurons. All neurons in the network use ReLU activation. Each output neuron corresponds to a set of actions.

4.2.2 Proximal Policy Optimization

For the advanced RLA, we used Proximal Policy Optimization (PPO), introduced by Schulman, Wolski, Dhariwal, Radford, & Klimov (2017). PPO is similar to TRPO in that it uses a trust region to avoid making too large of an update to the network at any one update, to avoid taking a bad step, which could ruin any further gathered data. It does this by clipping the commonly used objective function

$$\hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right] \tag{4.3}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, such that

$$L(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), t - \epsilon, 1 + \epsilon)\hat{A}_t) \right] \tag{4.4}$$

where ϵ is a hyperparameter. This removes the incentive to move r_t outside of the interval $[1 - \epsilon, 1 + \epsilon]$ (Schulman et al., 2017). The advantage $\hat{A}_t$ is calculated as

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{4.5}$$

$$\text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{4.6}$$

Besides using this clipped objective function, PPO also uses N (parallel) actors to gather data and accumulated updates, as seen in A2C (Mnih et al., 2016). This improves training stability by exploring different parts of the environment at the same time. This leads to the algorithm described in Algorithm A.4. The optimisation of L is performed using Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions introduced in Kingma & Ba (2014)

The topology of the Actor network consists of the 20 input neurons, followed by a densely connected layer of 64 neurons, which is densely connected to 5 output neurons, each controlling one of the 5 possible actions in the environment. The topology of the Critic network mirrors that of the Actor network, but it only has a single output neuron, which gives the expected value of the current state. All neurons in both networks use Tanh activation.

5 Methods

For collecting the data, each algorithm described in Section 4 was trained on each environment described in Section 3 twice, once with noise, and once without noise. Every 2.5×10^4 timesteps/1 generation, the progress was evaluated by stopping learning, and evaluating the agent for 25 episodes. For the Evolutionary Algorithms (EAs), the best performing individual from the population was used to gather this data. These 25 results are then averaged to get the performance of the agent for that timestep. The training/evolution was repeated 50 times for each algorithm to account for variability, and show the reliability of each algorithm.

Since the original Evoman framework as implemented by da Silva Miras de Araújo & de França (2016) is limited in scope, due to only being capable of playing an entire episode before providing feedback, this paper introduces a re-implementation of the Evoman framework as an OpenAI Gym environment (Brockman et al., 2016). This allows us to gather the rewards, and advance the environment, one time step at a time, as is required for RLAs.

All the code used for all the experiments can be found here.

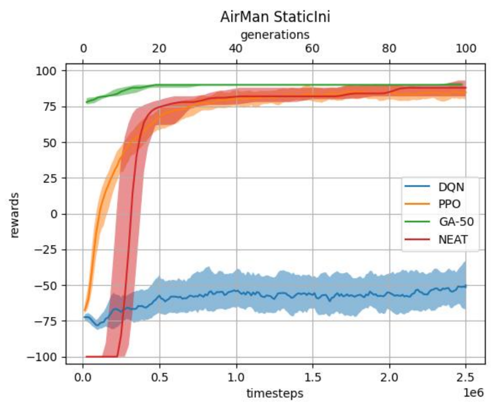
6 Results

6.1 Static Initialisation

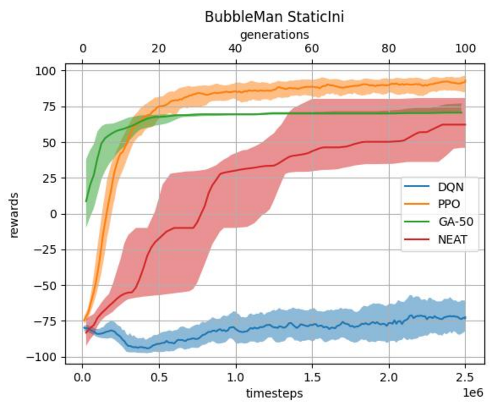
Looking at Figure 6.1, the first thing we see is that DQN performs poorly in all environments. This could be explained by the fact that DQN takes long to converge, and in the ALE has shown to require 1×10^7 to 4×10^7 timesteps to reach a policy that has better performance than random actions (Schulman, 2017). Due to universal poor performance of DQN it will not be discussed in further detail.

6.1.1 AirMan

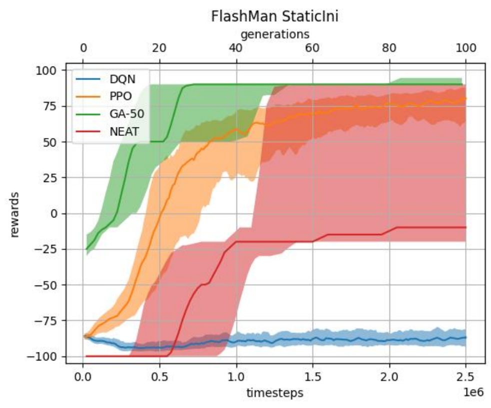
As expected from the easy nature of the environment, all algorithms quickly learn how to beat AirMan consistently, as seen in the narrow bands of the IQR in Figure 6.1a. NEAT consistently seems to evolve a strong policy after around 20 generations. PPO steadily learns over time and manages to beat the enemy consistently after approximately 2×10^5 timesteps. After this it slowly improves its score over time to approach the performance of GA-50 and NEAT. GA-50 does extremely well right off the bat, and we hypothesise this is because the best individual is evaluated, and with how simple the environment is, out of a hundred random initialisations for the first generation, it is likely that one of them happens to start with good parameters right from the start. This is unlikely to be the case for PPO as it only generates a single individual per run, and unlikely to be the case for NEAT as all NEAT individuals start off as a single-layer perceptron, and hidden neurons are necessary for reaching a decent score in the environment.



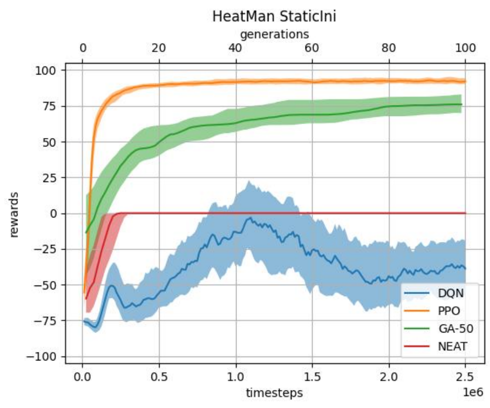
(a) AirMan



(b) BubbleMan



(c) FlashMan



(d) HeatMan

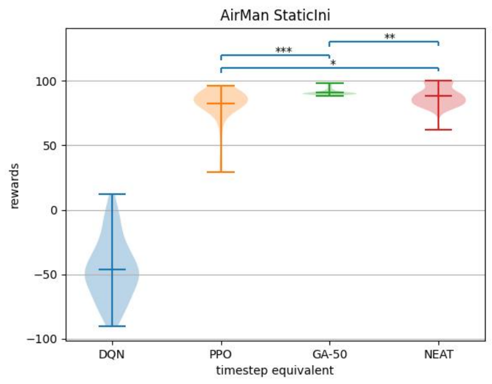
Figure 6.1: IQR of rewards gained by the algorithms over time in the environments with a consistent initial position during 50 training/evolution sessions. Rewards over 0 indicates a win, rewards lower than 0 indicate a loss.

6.1.2 BubbleMan

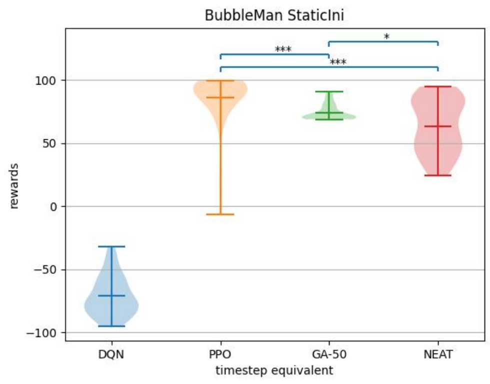
In the BubbleMan environment we consistently see a steady increase of performance until leveling out just under the max reward of 100 for PPO (Figure 6.1b). GA-50 once again starts off strong, but levels out below the performance of PPO. It seems to be incapable of evolving past this point, possibly explained by the fewer hidden neurons it has compared to PPO. NEAT performed the worst out of these three algorithms during the time provided, though it is clearly still improving at the time of cut-off, so it likely would have reached a better score if allowed to run until convergence. NEAT is also by far the least consistent, which is likely explained by its lack of a complex topology starting out. Due to this is it reliant on chance to find a decent topology before it is actually capable of evolving good weights for the topologies it evolves.

6.1.3 FlashMan

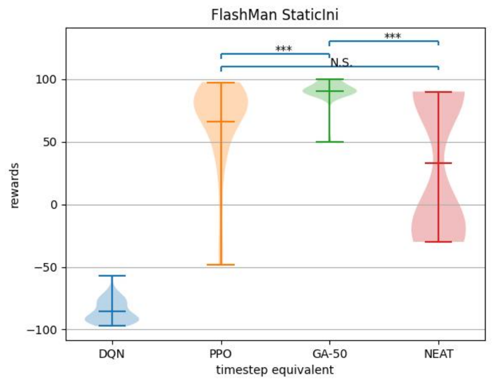
In the FlashMan environment, PPO learns a bit slower than in the AirMan and BubbleMan environments, but ends up consistently learning a good policy within 10^6 timesteps, as seen in Figure 6.1c. It is also still improving at the cut-off time, so it is likely it could achieve even higher scores if allowed to run until convergence. GA-50 also manages to consistently evolve a good policy. NEAT shows a very large IQR, with the 50th percentile towards the lower end. This seems to indicate an overall low performance, but the violin plot in Figure 6.2c tells a different story. 24 of the 50 runs actually reach a



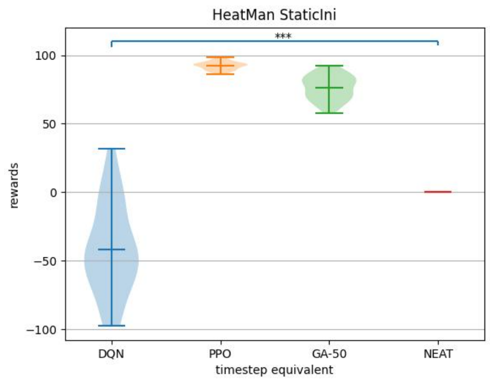
(a) Airman



(b) BubbleMan



(c) FlashMan



(d) HeatMan

Figure 6.2: Rewards gained by 50 runs the algorithms in the environments with a consistent initial position after 100 **generations/**2.5×10^6 **timesteps. The plots are annotated with significance markers calculated according to a Wilcoxon Rank-Sums test. N.S. = Not significant,** $* = p < 0.05$**,** $** = p < 0.01$**,** $*** = p < 0.001$**. Unlabeled means** $p < 10^{-15}$**.**

final score of 90. This seems to indicate that there is a single connection that is essential to reaching a good score, and unless that connection is evolved, a good score can not be achieved in this environment. GA-50 and PPO don't suffer from this limitation as they start off with the necessary hidden neurons, and only have to find the correct weights for reaching a good score.

6.1.4 HeatMan

In HeatMan we observe a very clear difference between the different algorithms (Figure 6.1d). We can see that PPO very easily and consistently learns an almost ideal policy, and seems to reach convergence at approximately 10^6 timesteps. GA-50 steadily evolves over time, and whilst consistently beating HeatMan, it does not seem to reach convergence within 100 generations. HeatMan is the only environment where DQN actually seems to improve over time, but after 1.1×10^6 timesteps it seems to regress in performance again. We are not sure why this happens. NEAT seems to evolve to a reward of 0 and seems to get stuck here. After looking at the amount of time the evaluated episodes ran for, we see all of them ran until the environment stopped for lasting too long (1500 timesteps). This indicates that NEAT consistently evolved avoidance behaviour and got stuck in this policy.

6.2 Random Initialisation

As we look at Figures 6.3 and 6.4, we once again see DQN failing to make any meaningful process towards solving any of the environments. We chalk this up to the same explanation given in Section 6.1.

6.2.1 AirMan

In the environment with random initial positions for AirMan, the 3 AIs manage to consistently reach a policy that beats AirMan, however they are less consistent with their score than in the case of static initial positions, as see in Figure 6.2a. GA-50 manages to find a good policy from the start, but does not seem to improve much thereafter. NEAT managed to find a policy to beat AirMan consistently after approximately 30 generations, but seems to mostly stagnate after this, and does not reliably find a policy where it rains a reward higher than 20. PPO improves steadily during training and whilst seemingly still improving at the cut-off point, gains significantly better rewards after the allotted timeframe, as seen in Figure 6.4a.

6.2.2 BubbleMan

Against BubbleMan, NEAT fails to find a policy sufficient for winning. This could be explained by a lack of neurons evolving to be able to determine when the agent should interrupt the jump. GA-50 seems just about able to evolve a policy that wins from BubbleMan most of the time. This could be explained in two ways. Either GA-50 takes most of its damage from the projectiles shot by, or contact with, BubbleMan, or GA-50 manages to avoid the spikes a bit over half the time, and jumps into the spikes during the other evaluations. PPO once again steadily and reliably improves over time, and nearing the end of the training period finds a policy that seems to be capable of defeating BubbleMan consistently.

6.2.3 FlashMan

In the FlashMan environment, it seems PPO is the only agent making any progress on learning the environment. NEAT evolves a little between 20 and 40 generations, but this seems to more be learning to shoot in the right direction, than actually avoiding and attacking the enemy. GA-50 does not make any progress whatsoever, and seems to just be stuck with policies that are about as effective as random button pressing.

6.2.4 HeatMan

PPO managed to quickly and reliably find an almost optimal policy against HeatMan. Besides that the story is very similar to that of BubbleMan (see Section 6.2.2).

7 Discussion

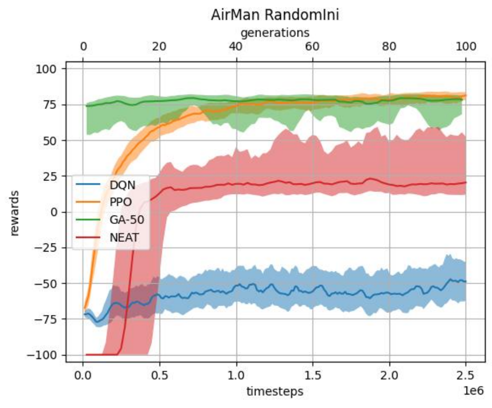
Whilst random initialisation had very little negative effect on either of the reinforcement learning algorithms, it clearly had a large influence on the evolutionary algorithms, which performed significantly worse in the noisy environments.

This could indicate that the EAs evolution led to over-fitting for the environment they were evolving for. This is not unexpected in environments where the only variation is caused by the actions of the agent. In the environments with noise, each individual only ever encountered one of the initial positions during the evaluation step, which likely contributed to the incapability of the EAs to generalize.

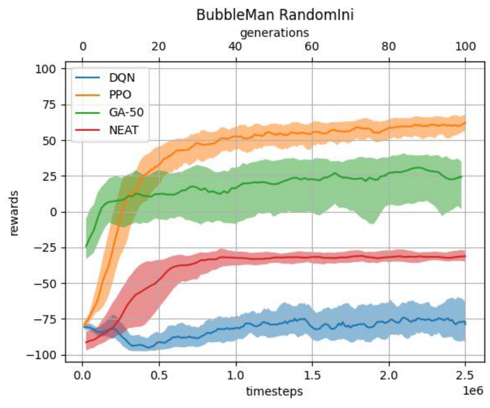
PPO on the other hand seems to be able to learn effective policies whether the initial conditions are are noisy or not (DQN did not generate any effective policies, though it's learning did not seem affected by the switch to noisy environments). This suggests that credit assignment leads to a more generalized understanding of the underlying mechanics of the environment by the agent. Thus we conclude that credit assignment does result in better performance in noisy video game environments.

This could be due to the fact that credit assignment leads to learning what actions work well in what states. Since the initial state does not matter to the individual states, it could be that credit assignment more effectively filters out the influences of the initial states. This could also be caused by the fact that in the RLAs, a single individual encounters all different starting positions, and can learn from all of them, whereas each EA individual only ever encounters one starting position.

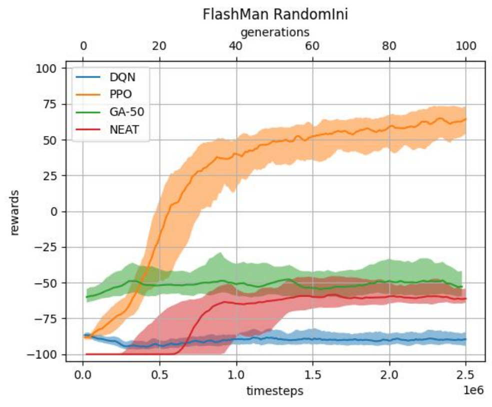
It is noteworthy that despite all the algorithms



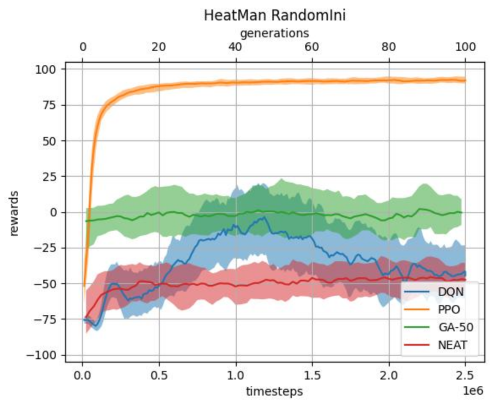
(a) AirMan



(b) BubbleMan



(c) FlashMan



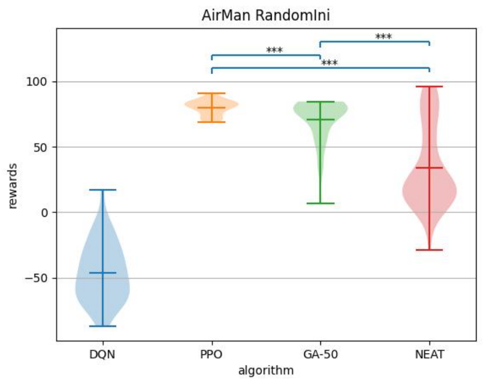
(d) HeatMan

Figure 6.3: IQR of rewards gained by the algorithms over time in the environments with a consistent initial position during 50 training/evolution sessions. Rewards over 0 indicates a win, rewards lower than 0 indicate a loss.

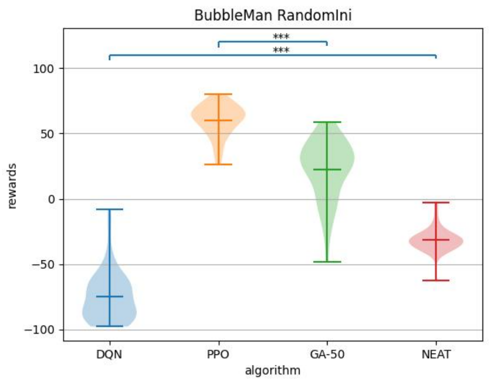
being trained on the same amount of data, both EAs trained in much less time than the RLAs. The RLAs needed about 10× the amount of time to train despite training on the same amount of data. This indicates that the evolution step of the EAs used in this paper is much cheaper than the learning of the RLAs used in this paper. This means it could be beneficial for the EAs to be evaluated for multiple episodes to give them an opportunity to be evaluated against multiple initial positions, whilst still taking a similar amount of time to train as the RLAs.

8 Conclusion

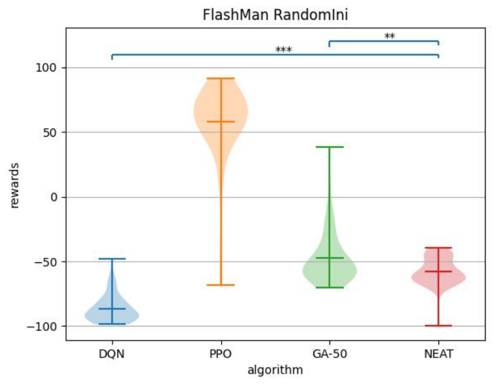
In this paper we have shown that RLAs are less strongly influenced by noise being introduced into their environment than EAs are. This indicates that, in 2D videogames, credit assignment leads to more general policies. This aligns with what was found in Rieser et al. (2011) and Taylor et al. (2006), and indicates that with a limited amount of data, RLAs are likely to achieve higher performance in noisy environments than EAs.



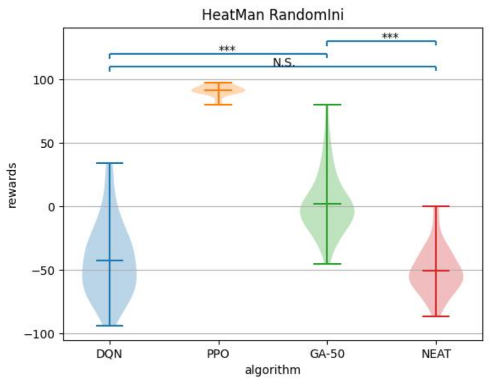
(a) Airman



(b) BubbleMan



(c) FlashMan



(d) HeatMan

Figure 6.4: Rewards gained by 50 runs the algorithms in the environments with a consistent initial position after 100 generations/2.5×10^6 timesteps. The plots are annotated with significance markers calculated according to a Wilcoxon Rank-Sums test. N.S. = Not significant, $* = p < 0.05$, $** = p < 0.01$, $*** = p < 0.001$. Unlabeled means $p < 10^{-15}$.

9 Future Work

9.1 Convergence

Due to time restrictions, the algorithms in this paper were only ran for a set amount of time. Some algorithms did not reach convergence in the time given by this paper. It would be interesting to see the difference in performance if the algorithms are allowed to run to convergence, rather than with a comparable amount of data. This would also allow a comparison of the convergence time to be drawn, and see if the conclusions from this paper also apply to simpler RLAs, such as DQN.

9.2 More games

As mentioned in Section 1, games make for a great test-bed due to their varied nature. Whilst this paper has laid a baseline for one game, a comparison for multiple games should be performed to gain further information about how this performance generalises for the presented algorithms as a whole.

9.3 Other types of noise

This paper has only explored noisy initialisation of the environment, with an otherwise deterministic environment with flawless sensors. It would be interesting to see this experiment performed in non-deterministic environments, or an environment with noisy sensors. This could make it harder for RLAs to assign credit to the appropriate actions.

References

Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z. D., & Blundell, C. (2020). Agent57: Outperforming the atari human benchmark. *CoRR*, *abs/2003.13350*. Retrieved from https://arxiv.org/abs/2003.13350

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Crespo, J., & Wichert, A. (2020, Apr 06). Reinforcement learning applied to games. *SN Applied Sciences*, *2*(5), 824. Retrieved from https://doi.org/10.1007/s42452-020-2560-3 doi: 10.1007/s42452-020-2560-3

da Silva Miras de Araújo, K., & de França, F. O. (2016). An electronic-game framework for evaluating coevolutionary algorithms. *CoRR*, *abs/1604.00644*. Retrieved from http://arxiv.org/abs/1604.00644

da Silva Miras de Araujo, K., & de Franca, F. O. (2016). Evolving a generalized strategy for an action-platformer video game framework. In *2016 ieee congress on evolutionary computation (cec)* (pp. 1303–1310). doi: 10.1109/CEC.2016.7743938

Givigi, S. N., Schwartz, H. M., & Lu, X. (2010, Jul 01). A reinforcement learning adaptive fuzzy controller for differential games. *Journal of Intelligent and Robotic Systems*, *59*(1), 3–30. Retrieved from https://doi.org/10.1007/s10846-009-9380-4 doi: 10.1007/s10846-009-9380-4

Hausknecht, M., Lehman, J., Miikkulainen, R., & Stone, P. (2014). A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(4), 355-366. doi: 10.1109/TCIAIG.2013.2294713

Ishikawa, F., Trovões, L. Z., Carmo, L., França, F. O. d., & Fantinato, D. G. (2020). Playing mega man ii with neuroevolution. In *2020 ieee symposium series on computational intelligence (ssci)* (pp. 2359–2364). doi: 10.1109/SSCI47803.2020.9308303

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

LeBlanc, D. G., & Lee, G. (2021). General deep reinforcement learning in nes games.

Lucas, S. M. (2008). Computational intelligence and games: Challenges and opportunities. *International Journal of Automation and Computing*, *5*(1), 45–57.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., . . . Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*, *abs/1602.01783*. Retrieved from http://arxiv.org/abs/1602.01783

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Moriyama, K., Branco, S. E. O., Matsumoto, M., Fukui, K.-i., Kurihara, S., & Numao, M. (2014). An intelligent fighting videogame opponent adapting to behavior patterns of the user. *IEICE TRANSACTIONS on Information and Systems*, *97*(4), 842–851.

Murphy, T. (2013). *The first level of super mario bros. is easy with lexicographic orderings and time travel... after that it gets a little tricky.*

Promsutipong, P., & Kotrajaras, V. (2017). Enemy evaluation ai for 2d action-platform game. In *2017 14th international joint conference on computer science and software engineering (jcsse)* (pp. 1–6). doi: 10.1109/JCSSE.2017.8025906

Rieser, V., Robinson, D. T., Murray-Rust, D., & Rounsevell, M. (2011). A comparison of genetic algorithms and reinforcement learning for optimising sustainable forest management. In *Proc. 11th int. conf. geocomput.* (pp. 20–24).

Schulman, J. (2017). *Deep reinforcement learning bootcamp lecture 6: Nuts and bolts of deep rl experimentation.* AI Prism. Retrieved from https://youtu.be/8EcdaCk9KaQ

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, *abs/1707.06347*. Retrieved from http://arxiv.org/abs/1707.06347

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99-127. doi: 10.1162/106365602320169811

Sutton, R. S. (1992). Introduction: The challenge of reinforcement learning. In R. S. Sutton (Ed.), *Reinforcement learning* (pp. 1–3). Boston, MA: Springer US. Retrieved from https://doi.org/10.1007/978-1-4615-3618-5_1 doi: 10.1007/978-1-4615-3618-5_1

Taylor, M. E., Whiteson, S., & Stone, P. (2006). Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *Proceedings of the 8th annual conference on genetic and evolutionary computation* (pp. 1321–1328).

A Algorithm Details

A.1 Genetic Algorithm

Function	Description
GenPop()	Generate a population of neural networks with all weights set to a random value, return the population
Eval(pop)	Evaluate each individual in pop, return the fitnesses
Crossover(pop,fit)	See Algorithm A.2
Mutate(pop)	Mutate each individual in the population, return the mutated population
Select(pop,fit)	Select X-1 individuals from pop with the chance for each individual being proportional to their fit, return selected individuals
Best(pop,fit)	Select and return the highest fitness individual in the population.
Doomsday(pop,fit)	Get rid of the worst 25% and replace them with a 50/50 mix off fresh genomes and copies of the best genome, return the new population
Tour(pop,fit,N,T)	Select T random individuals from pop, take the one with the highest fitness. Repeat N times. Return the individuals

Table A.1: Function Descriptions

Hyperparameter	**value**
Population size	100
Crossover Prob.	100%
Mutation Prob.	20%

Table A.2: Hyperparameters for the Genetic Algorithm

Algorithm A.1 Genetic Algorithm

```
Require: $N > 0 \wedge G > 0 \wedge I > 0$
  $pop \leftarrow GenPop()$
  $pop\_fit \leftarrow Eval(Pop)$
  while $G > 0$ do
    $next\_pop \leftarrow Crossover(pop, pop\_fit)$
    $next\_pop \leftarrow Mutate(next\_pop)$
    $next\_pop\_fit \leftarrow Eval(next\_pop)$
    $pop \leftarrow pop + next\_pop$
    $pop\_fit \leftarrow pop\_fit + next\_pop\_fit$
    $pop, pop\_fit \leftarrow Select(pop, pop\_fit) + Best(pop, fit)$
    if No improvement for I generations then
      $pop, pop\_fit = Doomsday(pop, pop\_fit)$
    end if
    $G \leftarrow G - 1$
  end while
```

Algorithm A.2 Crossover(pop,fit)

```
Require: pop is an array, $T \geq 2, N \geq 1$
  $new\_pop \leftarrow []$
  while $new\_pop.length < pop.length$ do
    $parent1, parent2 \leftarrow Tour(pop, fit, 2, T)$
    for N times do
      $\alpha \leftarrow Uniform(0, 1)$
      $offspring \leftarrow \alpha * parent1 + (1 - \alpha) * parent2$

      $new\_pop \leftarrow new\_pop + offspring$
    end for
  end while
  return $new\_pop$
```

A.2 Deep Q-Networks

Hyperparameter	value
N	6×10^6
ϵ	†
γ	0.99
Learning Rate	3×10^{-4}
Mini-batch size	32

Table A.3: Hyperparameters used for DQN

Algorithm A.3 Deep Q-Learning, where θ and θ^- describe some neural network; $\phi()$ is a function that pre-processes the frame;

```
Initialize replay memory D with capacity N
Initialize $Q$ with random weights $\theta$
Initialise $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1, ..., M do
    Get initial state $s_1$
    $\phi_1 \leftarrow \phi(s_1)$
    for t = 1, ..., T do
        With prob. $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \underset{a}{\operatorname{argmax}} Q_\theta(, a)$
        Execute $a_t$, observe $r_t$ and $s_{t+1}$
        $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample a random mini-batch of transitions
        $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        if episode terminates at step $j + 1$ then
            $y_j \leftarrow r_j$
        else
            $y_j \leftarrow r_j + \gamma \max_{a'} \hat{Q}_{\theta^-}(\phi_{j+1}, a')$
        end if
        Perform a gradient descent step on
        $(y_j - Q_\theta(\phi_j, a_j))^2$ with respect to $\theta$
        Every C steps, reset $\theta^- \leftarrow \theta$
    end for
end for
```

†ϵ is linearly reduced from 1 to 0.05 over the first 10% of learning, and kept at 0.05 afterwards

A.3 Proximal Policy Optimisation

Hyperparameter	value
M	64
T	2048
N	1
K	10
ϵ	0.2
γ	0.99
Learning Rate	3×10^{-4}

Table A.4: Hyperparameters used for PPO

Algorithm A.4 Proximal Policy Optimisation

```
for iteration = 1, 2, ... do
    for actor = 1, 2, ..., N do
        Run policy $\pi_{\theta_{old}}$ in env N for T timesteps
        Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
    end for
    Optimise $L$ wrt $\theta$, with K epochs using a mini-
    batch of size $M \leq NT$
    $\theta_{old} \leftarrow \theta$
end for
```

A.4 NEAT

Hyperparameter	value
Population Size	100
Compatibility Threshold	3
Excess coefficient	1
Disjoints coefficient	1
Weights Coefficient	0.5
Link Insertion Prob.	0.5
Link Removal Prob.	0.5
Node Insertion Prob.	0.2
Node Removal Prob.	0.2
Weight Mutation Prob.	0.8

Table A.5: Hyperparameters used for NEAT