The benefits of credit assignment in noisy video game environments

Jacob Schoemaker[1] and Karine Miras[2]

[1]Rijksuniversiteit Groningen, Groningen, Netherlands
[2]Vrije Universiteit Amsterdam, Amsterdam, Netherlands
k.dasilvamirasdearaujo@vu.nl

Abstract

Both Evolutionary Algorithms (EAs) and Reinforcement Learning Algorithms (RLAs) have proven successful in policy optimisation tasks, however, there is scarce literature comparing their 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 solving EvoMan - a video game playing benchmark. We test the algorithms both with and without noise introduction in the initialisation of multiple video game environments. We demonstrate that EAs reach a similar performance to RLAs in the static environments, but when noise is introduced the performance of EAs drops drastically while the performance of RLAs is much less affected.

Introduction

Machine Learning has achieved considerable success creating autonomous agents to play video games - referred to as Computational Intelligence in games (Lucas, 2008). Creating an agent that presents human-level performance in games has applications such as automatically testing the difficulty of (procedurally generated) opponents (Promsutipong and Kotrajaras, 2017), or generating adaptively interesting opponents for single-player games (Moriyama et al., 2014). Furthermore, the varied nature of games provides a great testbed for the capabilities of machine learning algorithms because they allow us to test the performance and dynamics of an algorithm in a variety of situations.

Two prominent methods in this field are Evolutionary Algorithms (EAs) (Ishikawa et al., 2020; Hausknecht et al., 2014) and Reinforcement Learning Algorithms (RLAs) (Crespo and Wichert, 2020; Givigi et al., 2010; LeBlanc and Lee, 2021). In game playing, EAs evolve a population of policies through mechanisms called selection, mutation, and crossover. RLAs on the other hand function by training a single policy, updating the policy step-by-step through feedback given by the environment. The differences between EAs and RLAs are fundamental and influence trade-offs when choosing one over the other. RLAs and EAs receive feedback at different points in time. EAs receive eventual feedback at the end of an episode and thus the feedback is over their entire performance. In contrast, RLAs receive immediate feedback after every action, which allows them to assign credit for rewards to specific actions. Delayed rewards can be a problem for RLAs (Sutton, 1992) since they 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 since they do not receive their feedback until the end of an episode. However, this means that EAs are incapable of assigning credit to actions.

Although both EAs and RLAs have proven successful in policy optimisation tasks, there is relatively scarce literature assessing the strengths and weaknesses of their different characteristics. A few examples we are aware of are studies carried out by Rieser et al. (2011); Taylor et al. (2006); Drugan (2019). Therefore, this paper compares these classes of algorithms in two dimensions: nature (EA or RL) and complexity (simple and complex). As a testbed, we use EvoMan - a video game playing framework (da Silva Miras de Araújo and de França, 2016) created to facilitate experimentation using computational intelligence and to provide benchmarks. Importantly, no comparison of EAs with RLAs for the EvoMan benchmark has been found in the literature to this date.

In our experiments, we test these algorithms in four game environments twice: once with noise in the environment and once without noise. In particular, we seek to answer the following question: "What is the impact of noise on the performance of EAs and RLAs?". Furthermore, this paper also contributes with an extension of the EvoMan framework that supports the use of RLAs, and it provides a baseline for this class of algorithms within EvoMan games.

Related Work

Due to the usefulness of games as benchmarks for understanding the behaviour of Artificial Intelligence (AI) algorithms, they have been extensively used in prior AI research. The Arcade Learning Environment (ALE) has been used as a benchmark for performance of Computational Intelligence since it was first introduced by Bellemare et al. (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 so 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 and 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 scenarios have been emulated in a public domain clone of the original game Evoman (da Silva Miras de Araújo and de França, 2016). This clone serves as a testbed for one of these next-generation games, facilitating experimentation using computational optimisation techniques. EAs have yielded a good degree of success playing EvoMan games (da Silva Miras de Araujo and de Franca, 2016; Ishikawa et al., 2020).

EAs and RLAs have previously been compared in Rieser et al. (2011) and Taylor et al. (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. They 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.

Environments

The EvoMan framework[1] is a reimplementation of the final-stage games of the video game MegaMan II, introduced by da Silva Miras de Araújo and de França (2016). Originally, the framework did not allow querying the states of the environments nor updating the agent's controller during the episodes but only at their end.[2] In the current paper, EvoMan has been adapted into an OpenAI Gym environment (Brockman et al., 2016), solving these limitations and thus supporting the use of RLAs. The code for the adapted framework is available here.

The player (or agent) can take 5 actions: *move right, move left, jump, shoot, and release*[3]. It is possible to take multiple actions simultaneously, e.g., go right and shoot. 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 deplete the player's energy.

[1]The documentation of the framework is available here.

[2]Additionally, the baselines available for EvoMan were only EAs.

[3]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

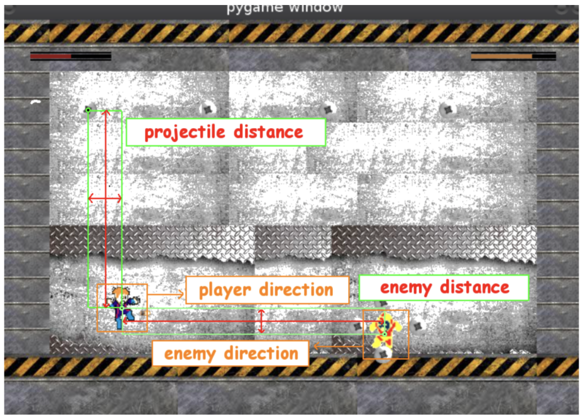

Figure 1: Sensors of the player agent.

The environment returns 20 sensor values at every timestep, representing the current state of the environment (Fig. 1). These values consist of the following:

- 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×)

Enemies

From EvoMan, 4 environments (enemies) were chosen for experimentation. These environments were chosen because they were deemed to present the most diverse set of game mechanics. Examples of learned agents playing against each of the enemies are available here. Notably, all enemies deal a small amount of damage to the player at every timestep the player is in (body) contact with the enemy.

AirMan AirMan was chosen because it is very easy to beat, and thus constitutes a good baseline benchmark. It is considered easy because the target is mostly static, and thus not challenging to aim at, and because the projectiles are always in the same location so that an avoidance move can easily be discovered.

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

FlashMan FlashMan was chosen because, though it is not particularly difficult to defeat, it is very difficult to obtain a high score on. 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.

HeatMan HeatMan was chosen because it was found to be difficult to optimise for (da Silva Miras de Araujo and de Franca, 2016). The difficulty of this enemy comes from the fact that the player needs to both jump over static particles and to dodge an enemy that is invulnerable during its time of movement.

Evaluation functions

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 Eq. 2. For the EAs, these rewards are summed over the course of a full episode to determine the fitness of the individual (Eq. 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:

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

where $\alpha = [0.0, 0.1, ..., 1.0]$. This led to the equations

$$reward_n = 0.5 \times DD_n - 0.5 \times DT_n \tag{2}$$

$$fitness = \sum_{n=0}^{N} 0.5 \times DD_n - 0.5 \times DT_n \tag{3}$$

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

Noise

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

Algorithms

We use four different algorithms: 2 EAs and 2 RLAs - one simple and one complex from each class. All algorithms have been trained using a total budget of 2.5×10^6 timesteps. In preliminary testing, it was found that in most environments a player would either die or win within about 250 timesteps. For the EAs, which used a population and offspring size of 100 and were evolved for 100 generations, this came out to about $100 * 100 * 250 = 2.5 \times 10^6$ timesteps. Therefore, this value was also used as the (maximum) amount of timesteps for the RLAs. All algorithms optimize neural network controllers that receive the 20 sensors as inputs and that output the actions for the agent to take.

Evolutionary Algorithms

Both EAs perform neuroevolution of controllers for the player agent.

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. The GA evolves a population weights that plug into a fixed-topology neural network consisting of 20 inputs, followed by a fully connected hidden layer of 50 neurons, followed by a fully connected output layer of 5 neurons.

The initial population μ is generated with random values for each of the network's weights. At each generation: first, λ pairs of parents are selected via k-tournaments with $k = 2$; second, one child is generated per pair using whole arithmetic recombination, and each weight of the child has a 20% chance of being mutated with the increment of a value taken from a normal distribution with a mean of 0 and a standard deviation of 1; finally, a pool with $\mu + \lambda$ is formed, from which μ survivors are selected via fitness proportionate selection.

NeuroEvolution of Augmenting Topologies For the complex EA, we utilized NeuroEvolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen, 2002) - we consider it more complex because it evolves not only the weights but also the topology of neural networks. To achieve this goal, every individual starts off as a simple perceptron, and may grow into a more complex network. It makes use of speciation to protect topological innovation. Networks receive 20 inputs and have its final layer with 5 outputs.

Reinforcement Learning Algorithms

Both RLAs perform deep Reinforcement Learning.

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 mini-batches 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 topology of the used network consists of the 20 inputs, followed by a fully connected layer of 64 neurons, followed by a fully 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.

Proximal Policy Optimization For the complex RLA, we used Proximal Policy Optimization (PPO), introduced by Schulman et al. (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:

$$\hat{E}_t\left[r_t(\theta)\hat{A}_t\right] \tag{4}$$

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

$$L(\theta) = \hat{E}_t\left[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), t-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{5}$$

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:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{6}$$

$$where \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{7}$$

Besides using this clipped objective function, PPO also uses N (parallel) actors to gather data and accumulated updates (Mnih et al., 2016) - therefore, we consider it more complex than DQN. This improves training stability by exploring different parts of the environment at the same time. The optimisation of L is performed using Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions (Kingma and Ba, 2014).

The topology of the Actor network consists of the 20 inputs, followed by a fully connected layer of 64 neurons, which is fully connected to 5 output neurons[4]. 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.

[4]Note that PPO needs fewer outputs than DQN because PPO supports continuous outputs. This is relevant because the agent can take simultaneous actions.

Experimental setup

Each of the four algorithms was trained on each of the four environments twice: once *with* noise and once *without* noise. At every 2.5×10^4 timesteps (1 generation - or stage), the progress was assessed by pausing the learning process and evaluating the agent for 25 episodes. For the EAs, the best performing individual from the population was used as the policy for this assessment. These 25 values were then averaged to get the performance of the agent for that stage/generation, and these are the values shown in the plots of the next section. The experiments were repeated 50 times (runs) independently for each algorithm.[5]

Results and Discussion

Figures 2 and 3 show comparisons among the final policies of each experiment. Figure 4 shows the results of the policies throughout the learning stages/generations. The main observations are I) The introduction of noise had a dramatically negative impact on both EAs in every environment, whilst the impact on the RLAs was either insignificant or much milder - the performance of EAs dropped severely in the face of noise while the performance of the RLAs was much less affected (Fig. 2); II) DQN performs poorly in all environments in comparison to the other algorithms. This is not surprising, since it has been shown before that DQN takes an extremely high number of steps (1×10^7 to 4×10^7) to find a solution better than random for the benchmark ALE (Schulman, 2017).

Environments without noise

AirMan As expected from the easy nature of this environment, all algorithms (but DQN) quickly learn how to beat AirMan consistently. NEAT evolves a strong policy after around 20 generations, and PPO steadily learns over time and manages to beat the enemy consistently after approximately 2×10^5 timesteps. Thereafter, PPO slowly improves its score over time to approach the GA's and NEAT's performance. GA does extremely well right from the beginning of the evolutionary process, and we hypothesise this is because in such a simple environment, out of a hundred random initialisations for the first generation, it is likely that one of them happens to start out with good parameters. This is unlikely to be the case for the RLs since they have a single solution per run, and unlikely to be the case for NEAT because all NEAT individuals start off as a single-layer perceptron.

[5]All the code to reproduce the experiments can be found here.

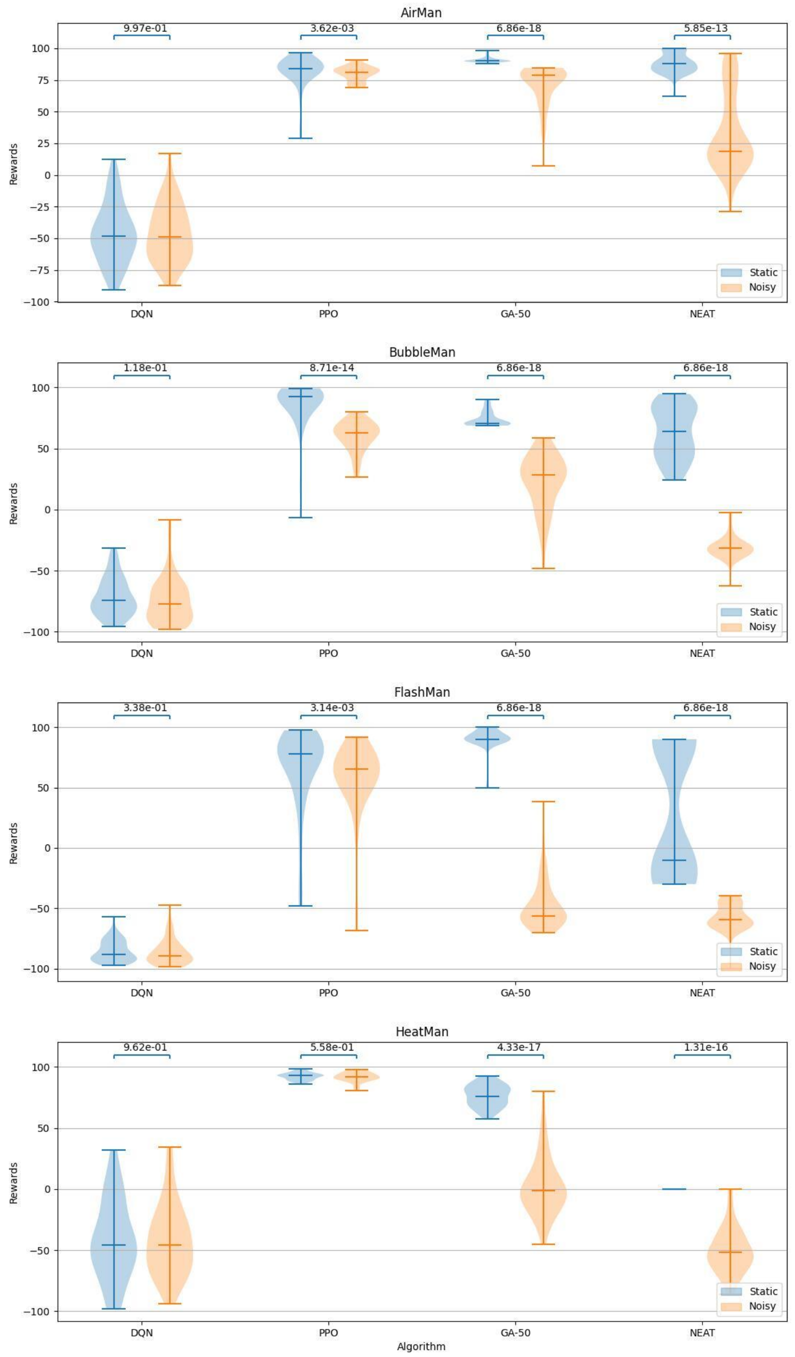

Figure 2: Rewards of final policies averaged among the independent runs - comparison between environments *with noise* (Noisy) and *without noise* (Static). A reward value above 0 means the amount of player-energy left when the player won, and a value below 0 means the amount of enemy-energy left when the player lost. For the EAs, the rewards of all individuals in the final population within each run were averaged to obtain the 'final policy' reward of the run. The plots are annotated with $p-values$ resulting from Wilcoxon Rank-Sums tests.

BubbleMan In the BubbleMan environment we consistently see a steady increase of performance until leveling out just under the maximum reward of 100 for PPO. GA once again starts off well, 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

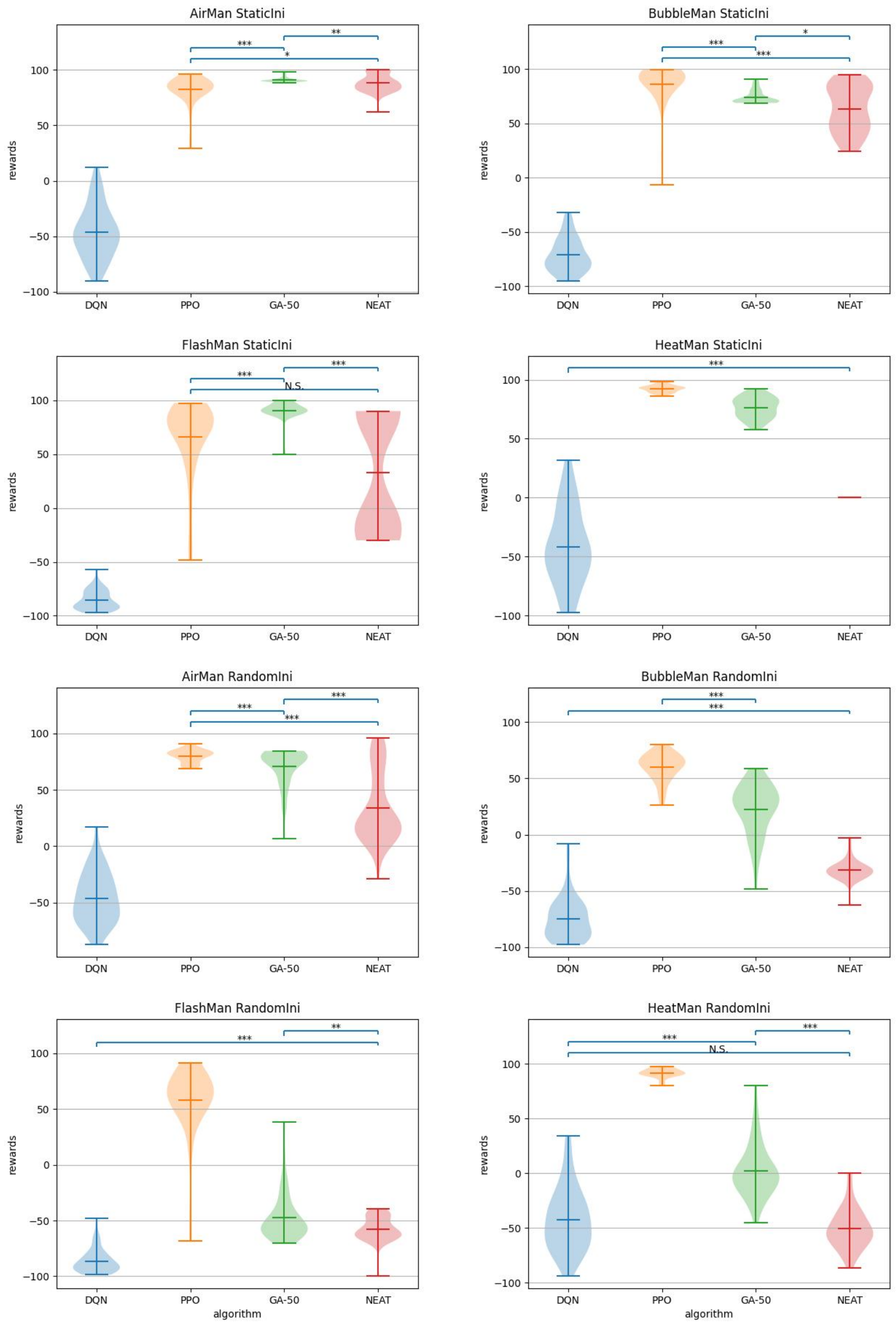

Figure 3: Rewards of final policies averaged among the independent runs for the environments *without noise* (StaticIni) and *with noise* (RandomIni). Definition of 'rewards' from Figure 2 applies. The plots are annotated with significance markers calculated with Wilcoxon Rank-Sums tests. N.S. = Not significant, $* = p < 0.05$, $** = p < 0.01$, $*** = p < 0.001$. Unlabeled means $p < 10^{-15}$.

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.

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. 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 also manages to consistently evolve a good policy. NEAT shows a very large IQR, with the 50th percentile towards the lower end of the search. This appears at first overall low performance, but in fact, 24 of the 50 runs actually reach a final score of 90 (Fig. 3). This suggests that there is a particular topological change 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 and PPO do not suffer from this limitation as they start off with multiple hidden neurons, and only have to find the correct weights for reaching a good score.

HeatMan In HeatMan we observe a very clear difference between the different algorithms. We can see that PPO very easily and consistently learns an almost ideal policy, and reaches convergence at approximately 10^6 timesteps. GA 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 improves over time, but after 1.1×10^6 timesteps it regresses in performance. We are not sure why this happens. NEAT evolves to a reward of 0 and gets stuck there. After inspecting the amount of time the evaluated episodes ran for, we see all of them ran until the environment expired (1500 timesteps). This indicates that NEAT consistently evolved avoidance behaviour and got stuck in this policy.

Environments with noise

AirMan In the environment with random initial positions, all algorithms but DQN consistently found a policy that beats AirMan, however, they are less consistent with their score than in the case of static initial positions (higher variance among runs). GA finds a good policy from the start, but does not improve much thereafter. NEAT finds a policy to beat AirMan consistently after approximately 30 generations, but mostly stagnates after this, and does not reliably find a policy with 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 time-frame.

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 evolves a policy that wins from BubbleMan most of the time. This could be explained in two ways. Either GA takes most of its damage from the projectiles shot by, or contact with, BubbleMan, or GA avoids 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 is capable of defeating BubbleMan consistently.

FlashMan In the FlashMan environment, PPO is the one making any progress on learning the environment. NEAT evolves up to 20-40 generations, but this seems it is just learning to shoot in the right direction and not really avoiding and attacking the enemy. GA does not make any progress whatsoever and appears to just be stuck with policies that as effective as random button pressing.

HeatMan PPO finds an almost optimal policy against HeatMan quickly and consistently. Besides that, the results are very similar to that of BubbleMan.

Conclusion

The RLAs presented significantly greater capacity to deal with noise in all of the game environments than the EAs. One possible explanation for this is that: credit assignment is allowed in RLAs, therefore, the usefulness of each action in the face of a particular state is taken into consideration regardless of the success of the policy in the upcoming states; with the EAs on the other hand, if the actions taken at the beginning of the episode are inappropriate, it does not matter if the policy would perform well along most of the rest of the episode because the policy never arrives at the steps where it could show its strength. Furthermore, if a solution has good performance but its offspring is tested with an initial environmental condition that the parent could not cope with, this offspring will have poor performance. This may create strong selection pressure for solutions that "work well" on every starting position, leading the search to get stuck into a mediocre "do it all" local optimum.

It is noteworthy that though all four algorithms have been trained on the same amount of data, both EAs were trained in much less time than the RLAs. The RLAs needed about $10\times$ the amount of time that the EAs needed[6]. This means that for the cases in which efficacy was similar for RLAs and EAs, the efficiency of the EAs was higher.

The current study has experimented with noise only in the initialisation of the environments, and using flawless sen-

[6]This is related to differences in the nature of the two classes of algorithms investigated, e.g., calculating gradients can be more expensive than mutation operations.

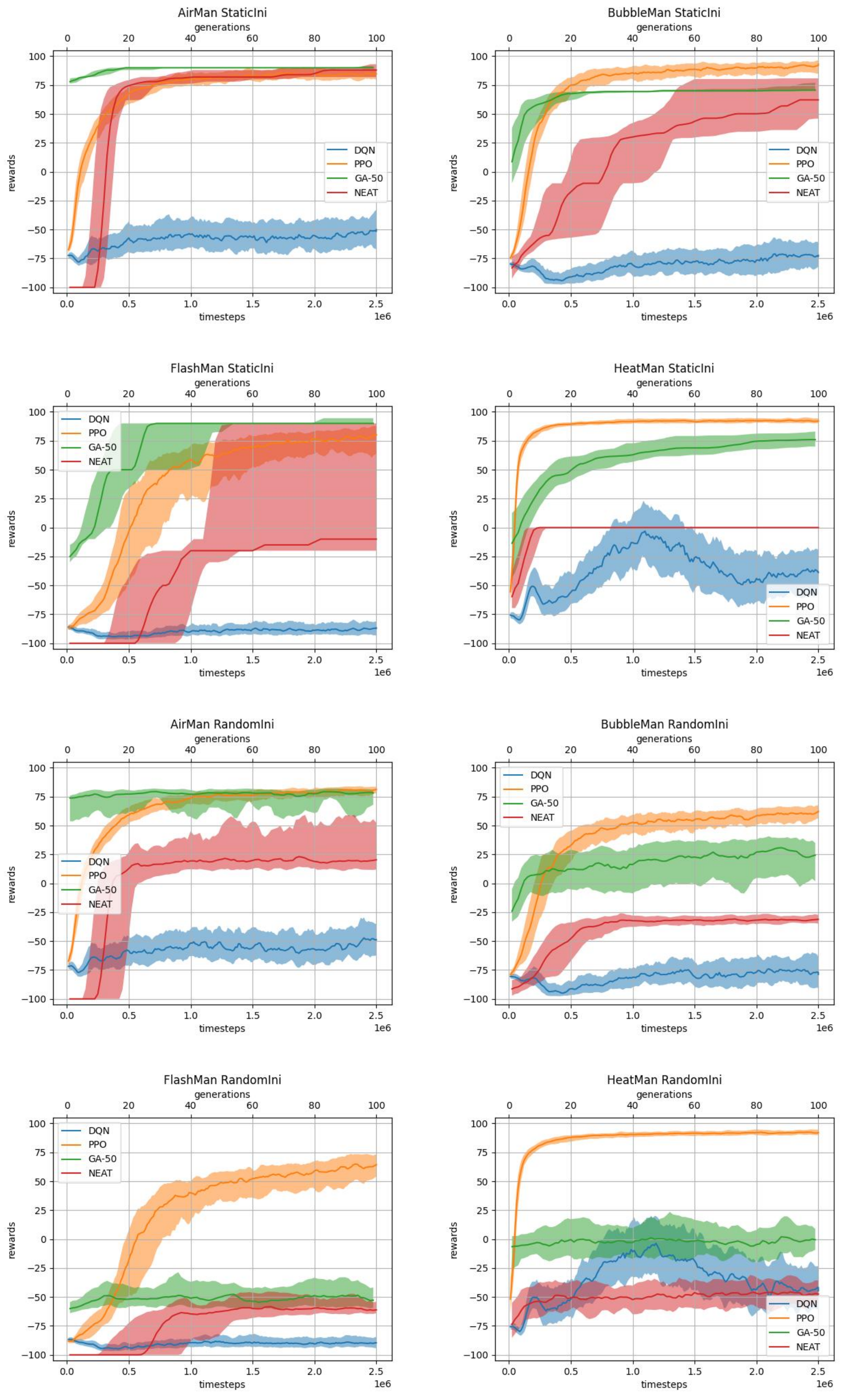

Figure 4: Rewards of policies across the stages/generations averaged among the independent runs for the environments *without noise* (StaticIni) and *with noise* (RandomIni). Lines are medians and shades are first and third quartiles. Definition of 'rewards' from Figure 2 applies.

sors. For future work, it would be interesting to investigate what happens with environments that suffer from noise all along the episode, with the use of noisy sensors, and with high-dimensional problems.

Acknowledgements

This research was funded by the Hybrid Intelligence Center, a 10-year programme funded by the Dutch Ministry of Education, Culture and Science through the Netherlands Organisation for Scientific Research[7], grant number 024.004.022.

References

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

Bellemare, M. G., Naddaf, Y., Veness, J., and 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., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Crespo, J. and Wichert, A. (2020). Reinforcement learning applied to games. *SN Applied Sciences*, 2(5):824.

da Silva Miras de Araújo, K. and de França, F. O. (2016). An electronic-game framework for evaluating coevolutionary algorithms. *CoRR*, abs/1604.00644.

da Silva Miras de Araujo, K. and de Franca, F. O. (2016). Evolving a generalized strategy for an action-platformer video game framework. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 1303–1310.

Drugan, M. M. (2019). Reinforcement learning versus evolutionary computation: A survey on hybrid algorithms. *Swarm and evolutionary computation*, 44:228–246.

Givigi, S. N., Schwartz, H. M., and Lu, X. (2010). A reinforcement learning adaptive fuzzy controller for differential games. *Journal of Intelligent and Robotic Systems*, 59(1):3–30.

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

Ishikawa, F., Trovões, L. Z., Carmo, L., França, F. O. d., and Fantinato, D. G. (2020). Playing mega man ii with neuroevolution. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 2359–2364.

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

LeBlanc, D. G. and 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., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and 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., and 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. and Kotrajaras, V. (2017). Enemy evaluation ai for 2d action-platform game. In *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–6.

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

Schulman, J. (2017). *Deep Reinforcement Learning Bootcamp Lecture 6: Nuts and Bolts of Deep RL Experimentation*. AI Prism.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

Sutton, R. S. (1992). *Introduction: The Challenge of Reinforcement Learning*, pages 1–3. Springer US, Boston, MA.

Taylor, M. E., Whiteson, S., and 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*, pages 1321–1328.

[7]https://www.hybrid-intelligence-centre.nl