Deep Reinforcement Learning
Final Assignment

Alejandro Sánchez Roncero (s5279402) [*1] **Yorick Juffer (s1993623)** [*1]

Abstract

This study explores the effectiveness of the Deep Q-Network (DQN) and Proximal Policy Optimization (PPO) algorithms in two game environments: the Catch game and Atari Space Invaders. For the Catch game, the DQN agent exhibits a consistent, although somewhat slow, convergence. After training, it manages to capture the ball in 95 out of 100 episodes. In Space Invaders, the PPO agent shows significant learning improvements, considerably outpacing an agent employing random actions, getting 300 more rewards on average. Despite this progress, certain behaviors hint at incomplete learning, which could be attributed to constraints in computational resources and training duration. The observations underscore the potential of DQN and PPO in these contexts and point towards future research directions, including advanced DQN and PPO architectures and optimized training methodologies.

1. Introduction

Deep reinforcement learning (DRL) has emerged as a powerful paradigm for training agents to learn and make decisions in complex environments. In this study, we explore the effectiveness of Deep Q-Network (DQN) and Proximal Policy Optimization (PPO) algorithms in training agents for two distinct environments: Catch and Space Invaders.

This paper is structured as follows; section 2 outlines the Catch and Space Invaders environments and describes their configuration. Then, section 3 explains both DQN and PPO algorithms together with the followed procedure to obtain a final trained model. The performance of the algorithms in both environments is shown in section 4, while in section 5 we discuss the results. Finally, the conclusions from the study are drawn in section 6.

[*]Equal contribution [1]Rijksuniversiteit Groningen, Netherlands. Correspondence to: Alejandro Sánchez Roncero <a.sanchez.roncero@student.rug.nl>, Yorick Juffer <o.y.juffer@student.rug.nl>.

2. Environment

2.1. Part 1 - Catch

The Catch environment is a 21x21 grid in which an agent controls a 5x1 paddle at the bottom of the environment with the aim of catching a 1x1 ball. The ball moves from top to down and can rebound off the left and right environment edges. It has a vertical speed of $v_y = -1$ cell$/s$ and a horizontal speed of $v_x \in \{-2, -1, 0, 1, 2\}$ cell$/s$. The environment was designed following the same design principles as the OpenAI Gym library (Brockman et al., 2016).

At every time step, the agent has three discrete actions available: move the paddle left or right by one pixel, or do nothing at all and remain in position. An episode ends once the agent has either caught the ball or missed it, at which point it receives a reward of 1 or 0 respectively. A still of the environment can be found in Figure 1.

Figure 1. Catch environment. At the bottom, we can see the paddle, capable of moving horizontally. The small yellow square represents the ball. The agent is learned to move so that it catches the ball once it reaches the bottom.

2.2. Part 2 - Space Invaders

In this work, we employed the SpaceInvadersNoFrameskip-v4 environment, a component of the OpenAI Gym library (Brockman et al., 2016). The Space Invaders game, originally an Atari 2600 classic, was digitized and integrated into the OpenAI's reinforcement learning environment suite. The game environment consists of the player commanding a laser cannon, capable of horizontal movements along the screen's bottom edge. The objective entails shooting down a descending wave of alien spacecraft while simultaneously dodging their counterattacks, therefore saving the Earth from the invasion (it can be seen in Figure 2).

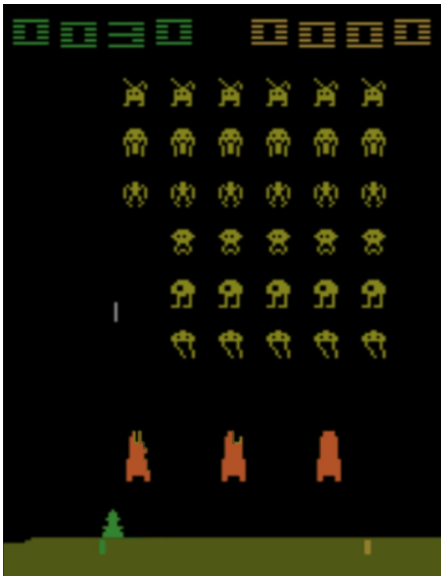

Figure 2. SpaceInvadersNoFrameskip-v4 environment from OpenAI Gym library (Brockman et al., 2016). At the bottom, we can see the player, capable of moving horizontally. The orange blocks are structures part of the environment, which can be destroyed by being shot and do not give any rewards. The space invaders are in yellow and they move progressively to the ground. The total score can be seen at the top.

This "NoFrameskip" variant of the environment enables the execution of every action for precisely one step. This distinction contrasts with other versions of the environment where certain frames might be skipped. The state observation the environment returns is an RGB image representing the game screen. This image comprises three channels (red, green, and blue, respectively) and has a resolution of 210 by 160 pixels.

The action space within this environment is classified as discrete, containing six possible actions: No operation, fire, move right, move left, move right while firing, and move left while firing. The reward structure of the game awards varying points based on the type of alien ship hit, ranging from 5 to 30 points per ship.

Algorithm 1 Deep Q-Network

for episode=1,2,... **do**
  s_t = reset environment
  terminal = false
  while not terminal **do**
    if evaluate **then**
      $a_t = argmax_{a \in A}(Q(s_t, a))$
    else
      a_t = action(s_t, ϵ)
    end if
    s_{t+1}, r_t, terminal = step(a_t)
    add (s_t, a_t, r_t, s_{t+1}) replay buffer D
    $s_t = s_{t+1}$
  end while
  if train **then**
    train agent on D
    if update **then**
      $\theta^- \leftarrow \theta$
    end if
  end if
  decay ϵ
end for

3. Method

3.1. Part 1 - Deep Q-Network

For the Catch environment, a Deep Q-Network (DQN) algorithm (Mnih et al., 2015) was used to train the agent. This involves training a convolutional neural network (CNN) as the Q-Network to output the expected cumulative rewards of taking an action for a given state. An outline is given in Algorithm 1. The network of the CNN consisted of three convolutional layers (followed by the ReLU activation), a single flattening layer and two fully connected layers. The last layer has as many outputs as there are actions in the action space of the environment (i.e., three outputs). The full architecture can be seen in Figure 3.

To increase the stability of the learning process, a fixed target network was used. The target network θ^- is a copy of the original network θ (i.e., the same architecture but a different instantiation) that gets updated with the weights of the original network every **n** iterations. By decoupling the target network from the online learning process, we prevent the algorithm from chasing a moving target during training, achieving more consistent updates. Also, with the aim of giving the agent a sense of motion, we input to the CNN a stack of 4 consecutive frames.

To additionally improve the stability and data efficiency of the DQN algorithm, an experience replay buffer D was used.

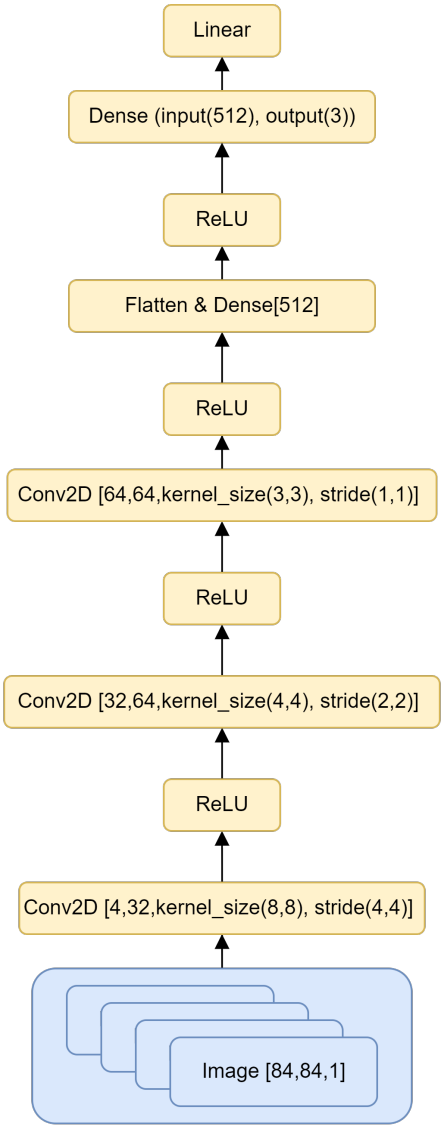

Figure 3. Architecture of the CNN used for learning the state-action value function $Q(s, a)$ in the Catch environment. Conv2D[4, 32, kernel_size(4,4), stride(2,2)] indicates a convolutional layer of 4 and 32 input/output channels respectively, with a kernel size of width 4 and height 4 and a stride of 2/2 in the x/y directions.

As the agent interacts with the environment, it stores a tuple (state, action, reward, next state) in the replay buffer, to later be sampled from to create mini-batches for training. During training, these mini-batches are sampled from D with a uniform distribution. This random sampling breaks the temporal correlation between consecutive samples, enabling more efficient learning and better utilisation of past experiences.

To balance exploration and exploitation, an epsilon-greedy strategy was incorporated into the DQN algorithm. During action selection, the agent has a probability of ϵ to select a random action over one selected by the network. The value of ϵ was decayed at a rate of ϵ_{decay} per episode, clipping its value at ϵ_{end}.

The CNN is trained based on the loss function given in Equation 1.

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)}[(r_t + \gamma \max_{a \in A} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta))^2], \quad (1)$$

where the weights are updated via gradient descent optimisation. $\mathcal{L}(\theta)$ is the mean square error of the difference between the Temporal-Difference target (i.e., the immediate r_t plus the discounted maximum estimate of the Q-value at the next state given the target network θ^-) and the estimated Q-value of immediate state given the network θ.

A grid search was performed to find the most suitable hyperparameters: batch size $\{32, 64\}$, discount factor $\gamma\{0.95, 0.99\}$, number of episodes $\{2000, 3000, 4000\}$, number of not training episodes after which an update is performed $\{50, 100, 500\}$ and learning rate $\{0.00025, 0.0005\}$ using RMSProp optimiser. On the contrary, the following ones were kept constant during the optimization: $\epsilon(1)$, $\epsilon_{decay}(0.001)$, $\epsilon_{end}(0.1)$ and number of episodes after which we alternate between training and evaluation (10). There are in total 71 different combinations. The metric used to grade each configuration is the average reward obtained in the last 50 evaluation episodes, having the objective of maximising it.

The best-performing configuration achieves an average reward of 1, that is, it caught the ball in the last 50 evaluation episodes. Its configuration is as follows: batch size (32), γ (0.99), 4000 episodes, number of episodes before the model is updated (50) and learning rate (0.00025). With this final configuration, we now perform 5 different optimisations during 4000 episodes and average both the reward and losses on each one (for both training and evaluation episodes). This yields a better estimate of the model's performance than just training it once. The obtained results are shown in section 4.

3.2. Part 2 - Proximal Policy Optimization

The chosen algorithm for this task was Proximal Policy Optimization (PPO) from the Stable Baselines3 library (Raffin et al., 2021) (algorithm 2). In the inner loop, an agent runs the learned policy $\pi_{\theta_{old}}$ for T timesteps, collecting a batch of trajectories. Afterwards, they are used to optimize a surrogate loss (Equation 2) with respect to θ. Therefore, PPO is categorized as an online algorithm, since the same policy being learned and optimised is used to collect more samples.

It is of particular interest the surrogate optimization loss

Algorithm 2 PPO, Actor-Critic Style

for iteration=1,2,... **do**
  for actor=1,2,...,N **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1$,...,$\hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$ $\theta_{old} \leftarrow \theta$
end for

(Equation 2). The operator $\hat{\mathbb{E}}_t$ is an empirical estimate of the expectation calculated over a batch of samples at that timestep. The loss $L_t^{CLIP}(\theta)$ limits the extent of policy updates, mitigating the risk of destructive updates [1] and ensuring the newly derived policy remains in close proximity to the previous one. The algorithm also optimizes the value function, which estimates the expected return of a state, by minimizing the squared error against the returns from the trajectory data (as in a supervised learning fashion). Finally, the entropy bonus ensures sufficient exploration.

$$L_t^{Final}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)], \tag{2}$$

where $L_t^{CLIP}(\theta)$ is given in Equation 3, $L_t^{VF}(\theta)$ is a squared-error loss $(V_\theta(s_t) - V_t^{targ})^2$, c_1, c_2 are coefficients and S denotes an entropy bonus.

$$L_t^{CLIP}(\theta = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \tag{3}$$

where ϵ is a hyperparameter.

The learning setup is similar to that described in subsection 3.1 with some minor modifications. A frame skip of 4 is included in the environment, which empirically shows to improve training stability and model performance. This implies that the agent will perform the same action during 4 frames. Each image is resized to 84 x 84 and converted to greyscale, which improves training efficiency. We also stack 4 frames together to give the agent a sense of motion and transpose the input image to fit the PPO implementation.

Figure 4 shows the architecture of the CNN used for learning. 4 input images are stacked (blue) and then forwarded to a shared CNN-based feature extractor (yellow). It consists of three convolutional layers, each of them followed by the ReLU activation. Finally, the output units of the feature extractor are fully connected with the output units associated to the policy (red) and V-value function (green). Having a common feature extractor reduces the total number of parameters, making the model less prone to overfitting. Also, it is more computationally efficient, makes transfer learning easier and often leads to better generalisation.

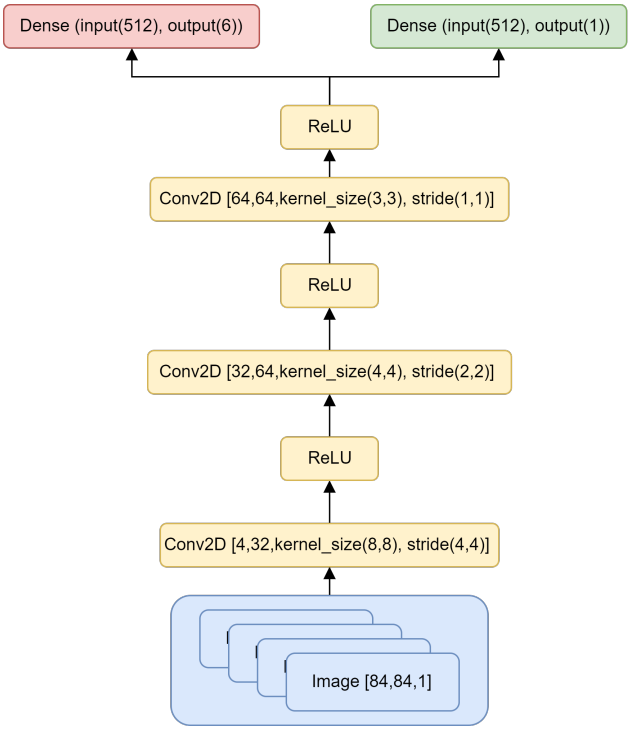

Figure 4. Architecture of the CNN used for learning in the PPO algorithm. Blue: stacked input images, Yellow: shared feature extractor CNN, red: fully connected layer for the policy, green: fully connected layer for the V-value function.

We then perform a grid search to find the following most optimal hyperparameters (alongside their ranges): the number of steps to run for each environment per update {2048, 4096}, the discount factor γ {0.99, 0.995}, the value function coefficient for the loss calculation c_1 {0.5, 0.2} and the number of frame skip during the agent interaction with the environment {1, 4}.

The following hyperparameters are kept constant during the optimization [2]: learning rate (0.0003), batch size (64) and number of epochs (10). The combination of these parameters yields a total of 16 cases. Due to limited hardware resources, each case is run for only 250,000 timesteps. Every 5,000 timesteps we perform an evaluation of the current policy on 10 episodes, storing the current model if the performance is better than the last best-performing one. The

[1] Since PPO is an online algorithm, if the policy update is too large, then the agent may potentially explore a completely different region of the state space. It will then start collecting low-quality trajectories, which are later used to update again the policy, making the algorithm unstable.

[2] Those hyperparameters not explicitly mentioned here are left by default according to (Raffin et al., 2021) implementation.

metric utilized to ascertain the optimal hyperparameters was the Mean Episode Reward. This reward represents an average of the total rewards achieved per episode, providing a measure of the policy's effectiveness.

The two best-performing configurations achieve an average reward per episode of 606.5 ± 160.5 (488.5 ± 107.57), with frame skips, number of steps, c_1 and γ of 1 (4), 4096 (2048), 0.2 (0.5) and 0.995 (0.995) respectively. We now train each of them for 1 million timesteps, discarding the former configuration due to exhibiting a decreasing reward performance and a higher instability over time. The final model is trained 3 different times during 2 million steps, averaging the obtained rewards on each episode. We then evaluate each model for 100 episodes and average the results to obtain a final performance measure of the model.

4. Results

4.1. Catch

The agent in the catch environment was evaluated for ten episodes after every ten training episodes. The average reward of these evaluation steps was used as an indication of performance. The average of the five training runs can be seen in Figure 5, which displays the average reward of the evaluation steps.

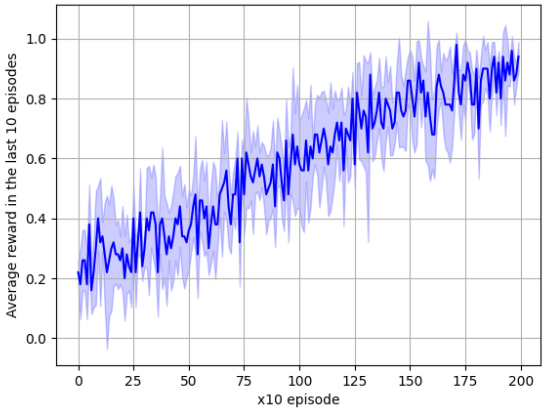

Figure 5. Average of the rewards obtained by the agent during the last 10 evaluation episodes in the Catch environment.

4.2. Space Invaders

The final trained model was evaluated for 100 episodes and the mean and standard deviation of the game length and score were tracked. The average game length was 3900 ± 550 and the agent achieved an average score of 485.50 ± 142.98. For sake of a comparison a second agent was run through the environment, but this one selected random actions (sampled from the action space with a uniform distribution). The average game length of this agent was 2162 ± 708 and it achieved a score of 149.69 ± 84.22. The result of game length are found in Figure 6 and the average score is found in Figure 7.

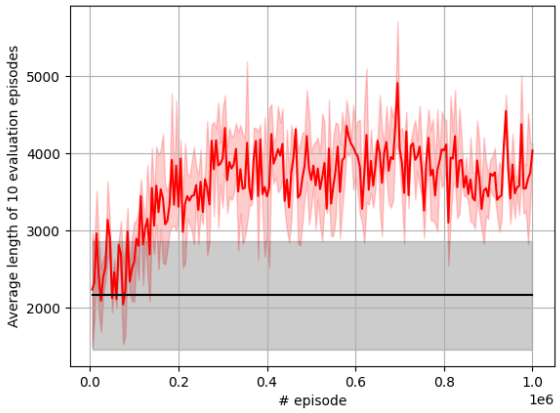

Figure 6. Average and standard deviation of an episode's length calculated from 10 evaluation episodes. Red: our agent, Black: an agent taking random actions (i.e., sampling from the action space with a uniform distribution).

5. Discussion

5.1. Catch

From Figure 5 its clear to see that the agent is able to converge to a solution, where towards the end of its training it is able to catch most balls. The DQN algorithm in that sense, is well suited for this task. However, it also becomes apparent that the algorithm is slow in its convergence. Something like a double deep Q-network or even more extreme such as the rainbow architecture could be much faster to find a solution (Hessel et al., 2018).

Additionally, the experience replay buffer implemented sampled trajectories uniformly when constructing batches meaning that all trajectories are given equal weight. This is a relatively inefficient use of the reply buffer as not all trajectories carry the same quality of information. An alternative would be to implement a prioritised experience replay buffer, which samples trajectories proportional to the temporal difference errors (Schaul et al., 2015).

Finally, it is not clear from the data what the final solution is, just that it's approaching it. In the present implementation, only 4000 episodes are considered, however from Figure 5 it can be seen that the model is still improving at the point of termination.

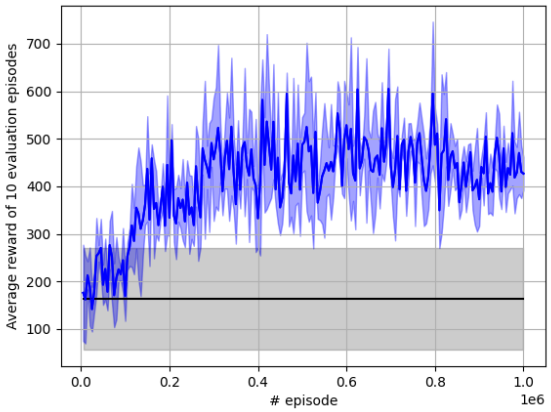

Figure 7. Average and standard deviation of an episode's reward calculated from 10 evaluation episodes. Red: our agent, Black: an agent taking random actions (i.e., sampling from the action space with a uniform distribution).

5.2. Space Invaders

The results show that the agent is able to improve its score as compared to an agent that performs random actions. Both average game time and average reward increase substantially up until around the 400,000 step mark, after which it tapers off significantly. On average, our model achieves 300 more rewards than the random one, indicating the learnt policy performs better. Upon observing the agent's performance it became evident that it tended to stay close to the left side of the screen (where it starts the game), despite there not being any aliens to shoot as seen in Figure 8. It also shows that the agent is able to take cover behind the barriers, though again, this tended to be behind the left one most of the time. Unfortunately, this behaviour would continue even after the barrier was destroyed, indicating that the utility of barriers was not clearly constructed in the model. One possible explanation for this behavior is that the training time was insufficient, potentially preventing a better solution from emerging later in the process.

Due to the limited hardware capabilities, it was not feasible to explore more training opportunities. Also, from Figure 7 we observe that there is a high variance during the whole training process. We have tried to tackle it by tuning two more hyperparameters, namely the ϵ used in the $\mathcal{L}^{CLIP}$ and the Generalized Advantage Estimator. By decreasing the former we restrict the policy updates, while increasing the latter favours variance against bias. However, the obtained results exhibit variance in the same order of magnitude. We also compare our results to those obtained by Tianshou benchmark, where they train a PPO model on the same environment for 10 million timesteps. We observe that they face the same variance problem. After 1 million timesteps, their average reward is close to 396.

Finally, a demonstration of the agent playing in the environment can be seen in Demo.

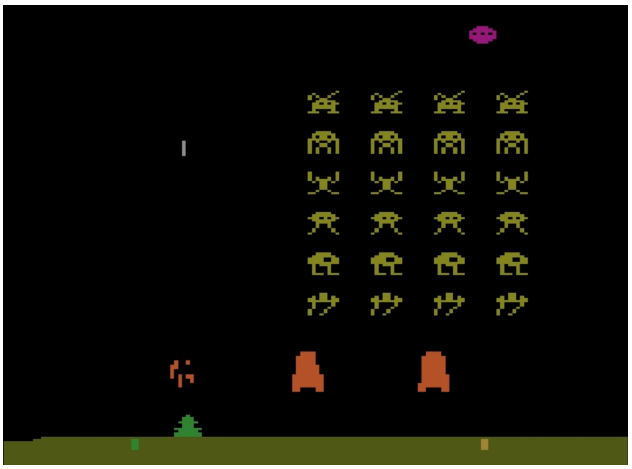

Figure 8. The agent taking cover behind the remains of the left barrier, whilst no aliens are near.

6. Conclusions

Our experiments with DQN in the Catch and Space Invaders environments reveal several key findings. In the Catch environment, the DQN algorithm performed commendably, albeit at a relatively slow rate of convergence. Several improvements, such as implementing double deep Q-network or Rainbow DQN architecture, can be considered for faster convergence (Hessel et al., 2018). Prioritised experience replay (Schaul et al., 2015) can be used for more efficient utilization of the replay buffer.

In the more complex Space Invaders environment, our PPO model displayed learning progress by consistently outperforming a random policy. Although it showed improvement in terms of game length and rewards, there were signs of incomplete learning. The agent exhibited a bias towards the left side of the screen, a behaviour that might be attributed to insufficient training. Given the limitations in computational resources, we were unable to conduct more extensive training, which could potentially address this issue.

Comparing our results with the Tianshou benchmark in the same environment, we found a similar high variance in the reward metric. This shows that our results are not an isolated case, and challenges with variance are a prevalent issue in this field of research.

Furthermore, a more extensive analysis of learned behaviours, such as the agent's preference for certain regions

of the screen, may shed light on potential improvements to the training process or reward function design.

References

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

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015. doi: 10.1038/nature14236.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. Stable-baselines3: Reliable reinforcement learning implementations. *The Journal of Machine Learning Research*, 22(1):12348–12355, 2021.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.