Assignment 3: Policy-based RL
Course: Reinforcement Learning 2023

Yuze Zhang s2819368 - Louis Siebenaler s3211126 - Peter Breslin s3228282

1. Introduction

Previously, the deep Q-learning algorithm was explored in detail to address high-dimensional Reinforcement Learning (RL) problems using value-based algorithms. With this final assignment, we turn our attention to policy-based methods. Unlike value-based algorithms, policy-based methods do not attempt to estimate the optimal value function describing the maximum expected cumulative reward received by an agent following a particular policy. Instead, policy-based algorithms try to directly determine the optimal policy that maximizes the cumulative reward. The choice of whether to use value- or policy-based methods depends on the problem case at hand, which in many circumstances is defined by the environment. In stochastic environments, the outcome of actions made stochastically are not fully described by the current state. Estimating the optimal value function can be challenging in such environments as the expected reward can vary over episodes, hence value-based methods may struggle and require extensive exploration. Policy-based methods however are equipped to handle stochastic environments since they work to directly optimize the policy, meaning that the agent can learn to take the stochasticity of the environment itself into account to better balance exploration and exploitation. In this assignment, we examine and implement from scratch the policy-based REINFORCE and Actor-Critic algorithms using a stochastic environment.

1.1. Environment

The Catch environment adapted from the DeepMind bsuite (Osband et al., 2019) by Thomas Moerland was used throughout this work. In Catch, the agent is tasked with catching falling balls from the top of the screen before they hit the bottom. The agent controls a paddle at the bottom of the screen which can be moved horizontally to catch the balls as they appear randomly from the top. The agent receives a **+1** reward for each ball it catches and a **-1** reward for each ball it fails to catch (i.e. if the ball reaches the bottom of the screen). The game ends after a fixed number of timesteps or if the maximum number of missed balls is reached, after which the agent's final score is given by the sum of its rewards over that episode.

The state space of the environment is described by two discrete parameters given by the positions of the paddle and the balls on the screen (represented as discrete grid coordinates), while the action space consists of three discrete possibilities; the agent can move the paddle right, left, or choose to remain still. There are several parameters that can be varied to change the environment's behaviour which include its size, speed by which the balls drop, and observation type (i.e. the state space can either be a vector with xy-locations of the paddle and the lowest ball or a binary two-channel pixel array). Throughout the primary investigation, the default Catch environment of size 7×7, with speed 1.0, and pixel observation type is used. However, different configurations of the environment are also tested in this work.

2. Methodology

2.1. REINFORCE

REINFORCE is a classic policy-based algorithm which follows a basic framework for policy-based RL (Williams, 1992). The algorithm starts with the initialization of the policy function π_θ, which is effectively a neural network with parameters θ. The initialized policy network, π_θ, was first used to map the states to action probabilities and sample a trajectory one step at a time through one episode with T steps. The sampled trajectory allows us to calculate the discounted cumulative reward at each state at step t along the sampled trajectory as a qualifier of the current policy,

$$\hat{Q}(s,a) = R_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k \,, \tag{1}$$

where r_k is the reward received from the environment at each step following the current step at t. This qualifier can be used to push the parameters of the policy neural network in the direction of the optimal action at each state throughout the sampled trajectory by multiplying the cumulative reward from the trace with the gradient of the probability of the chosen action (action probability from hereon) returned by the network $\nabla \pi_{\theta_t}(a|s)$ at step t. However, by multiplying the cumulative reward with the gradient of action probability, the good actions could be doubly improved (i.e. we are pushing more often and harder on good actions with high values), which may lead to instability during the learning process. This potential instability can be mediated by diving the product with the general action probability at each state, and we arrive at the following update rule:

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha\hat{Q}(s,a)\frac{\nabla\pi_{\theta_t}(a|s)}{\pi_\theta(a|s)} \\ &= \theta_t + \alpha\hat{Q}(s,a)\nabla_\theta \log \pi_\theta(a|s)\,, \end{aligned} \tag{2}$$

where α is the learning rate, which can be viewed as the core of the REINFORCE algorithm. To perform the learning process for the `catch` environment using REINFORCE, we firstly need to build a policy neural network that maps states (either two frames of pixelated images or a vector containing the row and column information of the paddle and the lowest ball) to action probabilities (probabilities of moving right, moving left, or staying still; different than action probability defined earlier), $\pi_\theta(\cdot|s)$. This network was first used to provide the action probabilities, $\pi_{\theta_t}(\cdot|s)$, to trace a trajectory based on the initial policy, π_θ, and the action with the highest probability was chosen for each step, followed by obtaining the next state, s_{t+1}, and the action reward, r_t. After sampling the full trace, the cumulative reward at each step, $R_t = \hat{Q}(s,a)$, was calculated based on Eq. 1. We can then use the cumulative reward and the logarithmic action probability at each step to obtain the loss of the policy network and perform the gradient update. The fact that the loss function, in this case, should be maximized cannot be overlooked since we are pushing the policy to return good actions with high rewards, so we need to consider the loss as the opposite value stated in Eq. 2 out of convenience for the policy network to back-propagate the gradient and minimize the loss. Through back-propagation, the current policy can be updated at all steps simultaneously. This process must be repeated multiple episodes to reach the optimal policy, catching as many balls as possible.

2.2. Entropy Regularization

Like many other types of RL, policy-based RL suffers the risk of being stuck in local optima, and small differences in hyper-parameters can potentially result in drastically different training performances. Hence, an exploration strategy is always required for the agent to learn a larger fraction of the state space and achieve the optimal policy. A common approach for explorations in policy-based RL is entropy regularization. The goal of entropy regularization is to provide the incentive to occasionally try a sub-optimal action based on the current policy, which means adding an ad hoc term in the loss function to push towards some of the sub-optimal policies slightly (not as much as the current optimal policy). This requires manipulating the action probabilities, $\pi_\theta(\cdot|s)$, at each step. The entropy of a discrete probability distribution, $p(X)$, in general, can be expressed in terms of

$$H[p] = -\sum_x p(x)\log p(x)\,, \tag{3}$$

which peaks at intermediate probabilities, $p(x)$. In terms of policy-based RL, this results in a new update rule,

$$\theta_{t+1} = \theta_t + \alpha\hat{Q}(s,a)\nabla_\theta \log \pi_\theta(a|s) + \beta\nabla_\theta H[\pi(\cdot|s)] \tag{4}$$

(example for REINFORCE), where the β parameter can be tuned to adjust the amount of exploration for the training process. A high β encourages more exploration, and a low β means less exploration. Entropy regularization ensures the policy stays as wide as possible while moving $\pi_\theta(a|s)$ towards the optimal policy, and a properly tuned exploration strategy with entropy regularization can potentially improve learning speed.

2.3. Actor-critic

The action selection in each episode of REINFORCE algorithm shown above is random with low bias. However, the variance during learning is high because the size and direction between updates vary strongly for different samples. The sources of the high variance are (1) gradient estimates because all action probabilities are positive and (2) estimates of cumulative rewards because chosen actions in a single trajectory are random. Consequently, the Actor-Critic (AC) algorithm was developed to address the issue of high variance by utilizing the advantage of low variance in the value-based approach. Hence, the AC approach is a combination of value-based elements and a policy-based method. That is, the "actor" part of the algorithm follows a policy-based approach, and the "critic" part follows a value-based approach (Sutton & Barto, 2018). Within the AC algorithm, two methods were used to tackle the problem of high variance; bootstrapping for better reward estimates and baseline subtraction for lowering the variance of gradient estimates. Because of the involvement of value-based elements, a learned value function, $V_\phi(s)$, is introduced in the AC algorithm, which requires an independent neural network (with parameters ϕ) from the "actor" neural network adopted from the *bona fide* REINFORCE policy network in Sec. 2.1 or a modified policy network with a second top head, the value-head. Either bootstrapping or baseline subtraction can potentially lower the variance significantly compared to the pure policy-based approach, and we will introduce these in succession in the following sections.

2.3.1. BOOTSTRAPPING

Bootstrapping uses the principle of temporal difference learning, which bootstraps the value function step-wise by computing the intermediate n-step values through an episode, $V_\phi(s_{t+n})$. These n-step values are between the Monte Carlo target, full trajectory, and the temporal difference target, one-step with end value $V_\phi(s_{t+1})$ at step $t+1$, which trades off variance for bias. The n-step target can be calculated from

$$\hat{Q}_n(s_t, a_t) = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_\phi(s_{t+n}) \,, \tag{5}$$

where γ is the discount factor. To update the value function, we need to compute the squared loss between the target and the value at the current state at step t, $V_\phi(s_t)$,

$$\mathcal{L}(\phi|s_t, a_t) = \left[\hat{Q}(s_t, a_t) - V_\phi(s_t)\right]^2 . \tag{6}$$

We also need to update the policy simultaneously for each step. Instead of using the previous qualifier mentioned in Sec. 2.1, we need to replace it with the target, $\hat{Q}_n$, while the rest of Eq. 2 remains the same,

$$\nabla_\theta \mathcal{L}(\theta|s_t, a_t) = \hat{Q}_n(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t) \,. \tag{7}$$

Now, the policy gradient ascent is updated using the n-step $\hat{Q}_n$ instead of the return from a full trajectory, R. We can also perform batch updates for both the policy gradient, $\nabla_\theta \mathcal{L}(\theta|s_t, a_t)$, and the value loss, $\mathcal{L}(\phi|s_t, a_t)$, via summation over the entire episode,

$$\phi = \phi - \alpha \nabla_\phi \sum_t \mathcal{L}(\phi|s_t, a_t) \tag{8}$$

$$\theta = \theta + \alpha \sum_t \nabla_\theta \mathcal{L}(\theta|s_t, a_t) \,. \tag{9}$$

It should be pointed out that we are maximizing the policy loss while minimizing the value loss, so the value loss does not take the opposite value as the policy loss. The two gradients, shown in Eq. 8 & 9, are back-propagated through the actor and the critic network separately to update the policy towards the optimal policy while updating the value function for each step. This process should be repeated over multiple episodes to achieve convergence.

2.3.2. Baseline Subtraction

The principle of baseline subtraction is based on the fact that subtracting a baseline from a set of probabilities does not affect the expectation value of the set. Meanwhile, baseline subtraction gives the algorithm an upper hand in reducing the variance by only pushing the policy based on better-than-average actions rather than all the possible actions (worse-than-average actions are pushed down instead). Under the context of the AC algorithm, the standard choice for the baseline is the expected value function at the current state, $V_\phi(s_t)$. We can subtract $V_\phi(s_t)$ from the state-action value estimate. Without bootstrapping, the state-action value estimate is simply the cumulative reward (qualifier), $\hat{Q}(s,a) = R_t$, through the Monte Carlo trajectory mentioned in Sec. 2.1, and with bootstrapping, state-action estimate is the target, $\hat{Q}_n(s_t, a_t)$, mentioned in Sec. 2.3.1. The quantity after the baseline subtraction, the advantage function, evaluates how much better each action is compared to the expectation value of its state, as shown in Eq. 10 & 11 for only the baseline subtraction and the combined baseline subtraction with bootstrapping, respectively.

$$\hat{A}_{MC}(s_t, a_t) = \left[\sum_{i=t}^{\infty} \gamma^{i-t} r_i\right] - V_\theta(s_t) \tag{10}$$

$$\hat{A}_n(s_t, a_t) = \left[\sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n V_\theta(s_n)\right] - V_\theta(s_t) \tag{11}$$

Now, we can use the advantage function to update the policy, the actor neural network, as shown in Eq. 12, while the squared advantage function *per se* can be used to update the critic network.

$$\nabla_\theta \mathcal{L}(\theta|s_t, a_t) = \hat{A}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{12}$$

The update is performed at each step over one episode, and the policy should be able to converge after a certain number of episodes of learning.

2.4. Clip-PPO

To further explore policy-based methods, the Proximal Policy Optimization (PPO) algorithm was examined, specifically to use as a new loss function to replace the standard loss in our AC model. PPO refers to a family of policy gradient methods that involve a cyclical process of gathering data by interacting with the environment and optimizing a “surrogate” objective function (Schulman et al., 2017). The surrogate objective function is simply an approximation of the “true” objective function while being easier to optimize. Depending on the problem case, the complexity of the objective function can make it difficult to represent or estimate accurately, and directly optimizing it can therefore prove challenging. PPO addresses this by incorporating a more tractable surrogate objective function which enables the agent to focus on significant aspects of the objective function that are easier to optimize. This can help improve the stability and robustness of the model since only the most relevant aspects of the objective function are captured by the surrogate, whereas directly optimizing the objective function can result in high variance (particularly in stochastic environments). Furthermore, this can lead to more efficient and effective updates to the policy parameters, as well as to help the agent better generalize to unseen situations.

To update the policy network, the AC model uses the standard method based on the advantage function. As explained by Eq. 12, the loss function in the AC model for updating the policy uses the log probability of the action selected by the current policy, given the current state of the environment i.e. $\log \pi_\theta(a_t|s_t)$. This provides a way of measuring the likelihood that the policy will select a particular action in the current state. By then using the advantage function to estimate how much better or worse the action was compared to the average action, the algorithm can update the policy in a direction estimated to be most optimal. With PPO, the loss function for the policy network replaces $\log \pi_\theta(a_t|s_t)$ with the ratio $r_t(\theta)$ of the current and old policies:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta,\,\text{old}}(a_t|s_t)} \tag{13}$$

where $\pi_{\theta,\,\text{old}}(a_t|s_t)$ denotes the same action being taken under the old policy. The idea behind this is to ensure that the new, updated policy does not deviate too much from the old policy, so as to prevent the agent from "forgetting" any previously learned behaviours as it explores new states. It can be noted that, if $r_t(\theta)$ is greater than 1, the current policy is more likely to select the given action in the current state compared to the old policy, hence the policy is said to be improving since it is selecting better actions. If $r_t(\theta)$ is between 0 and 1, the action is less probable for the current policy, and it is then performing worse than the old policy in the given state. Hence, $r_t(\theta)$ is a likelihood ratio.

There exists the possibility that, for the current policy, the action is much more probable compared to the old policy. In this case, $r_t(\theta)$ will become very large which will lead to the algorithm taking substantially big gradient steps which can cause instability and poor performance. To avoid this, the rate of policy updates must be balanced by limiting the amount by which the policy can change. This is achieved by adding a regularization term to the loss function to clip the difference between the current and old policy parameters. Hence, Clip-PPO is a variation of PPO that uses a "clipping loss" to prevent the policy from changing too much in each iteration. This is practiced by including a clipping threshold for $r_t(\theta)$ which acts to constrain the magnitude of the policy update to be within the threshold bounds of $[1-\epsilon,\, 1+\epsilon]$. This helps prevent excessively large gradient steps to be taken and hence promotes policy updates that are more stable. Combining Eqs. 12 & 13, and including a clipped likelihood ratio $r'_t(\theta)$, we arrive to the PPO loss for the policy network:

$$\nabla_\theta \mathcal{L}(\theta|s_t, a_t) = \min\left[r_t(\theta)\hat{A}_t,\; r'_t(\theta)\hat{A}_t\right] \nabla_\theta \tag{14}$$

It can be noted that the minimum value between $r_t(\theta)\hat{A}_t$ and $r'_t(\theta)\hat{A}_t$ is chosen so as to ensure that the policy update is never larger than the advantage estimate. Following the original work of Schulman et al. (2017), a value of $\epsilon = 0.2$ was used for the clipping threshold in our implementation.

Finally, rather than simply performing batch updates for the policy and value networks, a common technique with PPO is to perform multiple updates to each network using the same batch of data. The multiple updates are generally known as epochs, and the number of epochs to perform is often referred to as "K epochs". During each epoch, the networks are updated using the same batch, but with different mini-batches sampled randomly from the batch for each update. By allowing the networks to learn from the same data multiple times, the stability and performance of the training process can by improved. Furthermore, this can help avoid overfitting since the same data is learned in different ways since the mini-batches are sampled in a random way. The number of epochs to use depends on various factors such as the training process, the environment, and the available computational resources. Depending on the architecture of the networks used in the work by Schulman et al. (2017), the values chosen for K ranged between 3 and 15. For the work presented here, we choose a value of $K = 5$.

3. Experimental Procedure

3.1. Network Architecture and Parameter Exploration

We implemented the REINFORCE, different variations of the AC, and the Clip-PPO algorithm for the `Catch` environment. For each, we trained a network, using the `adam` optimizer, modelling the policy function consisting of 3 fully connected hidden dense layers of dimensions 256, 126, and 256 neurons. The choice of activation function between hidden layers is `ReLU` and the `Softmax` is the output activation function in order to ensure that the output is consistent with a discrete normalized probability distribution of size 3 for the action selection. When using the 'pixel' observation type for the state space, a flattening layer is added to the beginning of the architecture to ensure that it is compatible with a dense neural network, however for the 'vector' observation type this is not needed. We also experimented with convolutional neural networks (CNN) as an alternative architecture when using the 'pixel' observation type. Although CNN were able to determine the optimal policy for the standard 7×7 environment, training was significantly slower compared to the dense architecture. Hence, we deem CNN's overkill for a simple environment like `Catch`.

For the AC and Clip-PPO algorithm an additional neural network for modelling the V-value was implemented. This is needed in order to perform bootstrapping and thus reduce the high variance existing in the cumulative reward estimate for the REINFORCE algorithm. The architecture of the V-

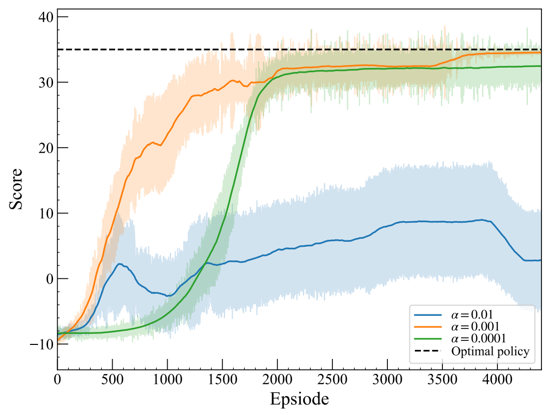

Figure 1. Learning curves for values of learning rate α using the full AC algorithm, with $n = 2$, and $\eta = 0.05$. Each configuration was repeated $N = 5$ times to account for the randomness of the action selection in the algorithm. The shaded regions correspond to the standard deviations among the repeated training runs divided by $\sqrt{N}$. We identify that $\alpha = 0.001$ works best.

value network is identical to that of the policy network with exception of the output layer, which has a linear activation function and returns a single value.

The performance of REINFORCE and AC strongly depends on the learning rate α and β controlling the entropy regularization. We varied these hyperparameters for the full AC algorithm to find the optimal value. We find that the algorithm learns fastest for $\alpha = 0.001$ and $\beta = 0.075$. Figure 1 shows the learning curves of different values of α for the AC algorithm. An additional hyperparameter of importance for AC is n, which determined the n-step target in bootstrapping. We also tuned this parameter and found that $n = 2$ gives good performance, although the impact of n was significantly less important than α. Hence, for all the algorithms in this work we used $\alpha = 0.001$, $\beta = 0.075$, $n = 2$ (if applicable)[1], and a discount factor $\gamma = 0.99$.

3.2. Environment Experiments

The default setting for the environment so far is a 7×7 grid *catch the ball* game with one ball drop every seven steps (only one ball in the frame at all times) and a "pixel" observation type. To explore how different `Catch` environment setups can influence the agent's behaviour as well as its performance during the training process, we made various changes, including the size of the environment, the frequency (speed) of the drop, and the observation type. During these individual experiments, when only one feature of the environment was changed, irrelevant attributes were set to default values, and to speed up the experimental processes, we decided to terminate the learning process based on the performance of the agent under different environment setups using different stopping conditions.

We first experimented with the size of the environment. Two other grid sizes with equal numbers of rows and columns, 9×9 and 14×14, were trailed during the experiment. Five learning curves were sampled for each environment size, and the average learning curve was calculated along with its variance. Since different learning curves have different restrictions due to the stopping condition, shorter learning curves for each environment size were extended using their final average score over the last 50 episodes.

The experimentation of the size of the environment is followed by observation of how speed changes the learning process. Different speeds of ball drops were tested, including 1, 1.4, 1.5, 1.75, 2, 3, and 3.5, and a column size of 7 is evenly divisible by the speed of 1, 1.4, 1.75, and 3.5 in the list. Same as the experiment for environment size, for each speed, various learning curves were averaged and smoothed along with calculating their uncertainties. Shorter learning curves for each speed were also extended in the same fashion. Additionally, the difference between the agent's learning processes for an environment with a "vector" observation type and a "pixel" observation type was explored. For each observation type, different speed conditions were applied to test the agent's learning efficiency under conditions of one ball and multiple balls appearing in the game. For each observation type/speed combination, all learning curves were averaged and smoothed, and the corresponding variance during training were generated afterwards. Shorter learning curves for each combination were extended.

In addition to the tests conducted above, we explored agent's learning for a non-squared environment, 7×14 with a speed of 0.5. To fully observe the behaviour of the agent, we did not set a stopping condition on the learning curve. Five different learning curves were produced, and an averaged and smoothed curve was generated, followed by the uncertainty calculation. The result was also compared with the agent's learning process for a 7×7 environment. Besides, an animation was visualized using `env.render()` in the end for each step over a 250 steps game, showing the agent playing the game after finding the optimal policy.

To establish a more interesting experiment for observing the learning process and the optimal policy of the agent, we created a modified, more stochastic environment. The increased level of stochasticity was introduced by assigning random motion of the ball in the horizontal direction during a drop $\sim 27\%$ of the time on average, and the ball could not pass the left and right boundary of the game. This simulates additional stochastic "wind" that can blow from either the left or the right with the same probability, which

[1]We actually used $n = 10$ for Clip-PPO as this gives a significantly better performance than $n = 2$.

is only applied above the second row of the environment (catching a ball with a 100% accuracy is impossible if the ball changes its horizontal location between the first and the second row). After the training, different learning curves were smoothed and averaged to produce the final curve and the variance during the learning process. An animation was also produced to observe the behaviour of the agent.

4. Results

4.1. Algorithm variation

We now present the results of our implementations of the REINFORCE, different variations of the AC, and Clip-PPO algorithm. Figure 2 shows the learning curves of the algorithms in a 7×7 `Catch` environment with speed = 1, averaged over 20 training runs each time. For the chosen game setting, the best achievable score is 35, hence under the optimal policy the agent is expected to achieve this target score. The algorithms are implemented with entropy regularization as an exploration method, which ensures that the policy $\pi_\theta(a|s)$ moves to the optimal policy, while also forcing $\pi_\theta(a|s)$ to stay as wide as possible. The consequence of this is an improved learning speed and that $\pi_\theta(a|s)$ is encouraged to explore more widely, while at the same time not focus on unpromising strategies. Furthermore, since entropy is explicitly added to the optimization goal, the results of the algorithm are more stable for different random seeds and less sensitive to hyperparameters.

We discern that for each algorithm, the agent experiences learning, although for the REINFORCE (blue) and AC with bootstrap (red), the agent converges on average to a suboptimal policy after 20000 episodes [2], while for the AC with baseline and the full AC, the optimal policy has always been determined within 20000 episodes. The main reason for the poorer performance of the REINFORCE algorithm is the high variance of the policy gradients, which arises because an entire episode is sampled to compute the expected Q-value $\hat{Q}(s_t, a_t)$ as shown in Eq. 1. As a consequence, policy updates occur infrequently, causing slow convergence and often local optima are found instead. Implementing the concept of bootstrapping helps overcome the high variance problem, where the value function is used to compute an n-step target $\hat{Q}(s_t, a_t)$, thus trading-off variance for bias. Although bootstrapping should help improve the results of REINFORCE, we find that for $n = 2$, the performance of the AC with bootstrap is almost identical to that of REINFORCE. It is possible that the policy gradient suffers from high bias in this case, and a larger value of n would yield an improved performance for the AC with bootstrapping only.

[2]REINFORCE and Actor-Critic with bootstrap are able to determine the optimal policy, however they are not always able to achieve this within 20000 episodes.

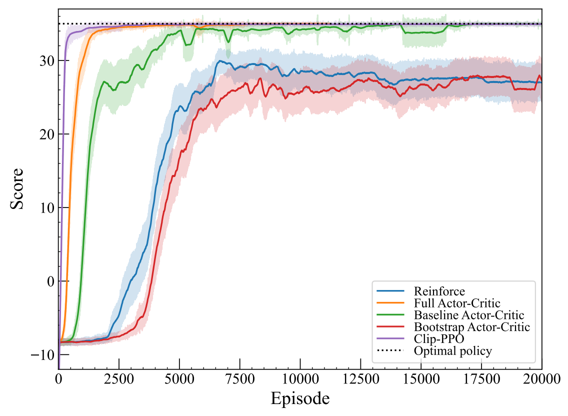

Figure 2. Learning curves for different variations of policy-based algorithms for a 7×7 environment with speed = 1. Each algorithm was trained $N = 20$ times and the shaded colored regions correspond to the standard deviations among the repeated training runs divided by $\sqrt{N}$. We identify that Clip-PPO reaches the optimal policy (black dotted line) the fastest.

An additional method to reduce the high variance is baseline subtraction, which reduces the variance in the expected Q-value $\hat{Q}(s_t, a_t)$, but leaves the expectation unaffected. This has the advantage over bootstrapping that there is no trade-off variance for bias. As a result, we find that the AC with baseline subtraction alone (green) performs very well and was able to find the optimal policy in less than 20000 episodes, while also exhibiting a significant faster learning early on compared to REINFORCE and AC with bootstrapping. Combining both bootstrapping and baseline subtraction forms the full AC algorithm, and we find that the performance of the agent significantly improves. Learning takes place rapidly, and on average the agent converges to the optimal policy within 2500 episodes, which is a huge improvement to the REINFORCE algorithm and also the weaker variants of the AC. This reflects the ability of the interplay of bootstrapping and baseline subtraction to overcome the problem of high-variance in the policy gradient, thus explaining why AC algorithms are among the most popular model-free RL algorithms in practice.

Lastly, we see that the implementation of AC with a Clip-PPO loss function (purple) outperforms all other models. This proved to consistently converge to the optimal policy more rapidly, even more so than our full AC algorithm (note that, while the AC model with Clip-PPO includes bootstrapping, it does not include baseline subtraction as this pertains to the loss function which is the very component adjusted through the Clip-PPO algorithm). When comparing to the other learning curves, the benefits outlined in Section 2.4 regarding the use of Clip-PPO can be clearly seen. From the fast pace in which the model succeeds in determining the optimal policy paired with the low standard deviation exhibited, it can be inferred that the inclusion of Clip-PPO led to more efficient and effective updates to the policy parameters. In the case of stochastic environments like `Catch`, directly optimizing the objective function can prove challenging and result in high variance. Moreover, if the function is of a high complexity, the efficiency by which the agent can update the policy is reduced as it will require more extensive exploration. The inclusion of the surrogate objective function addresses these concerns by enabling the agent to focus on only the most important aspects of the environment so as to better guide it in its search. This acts to reduce the variance and improve the stability of the model, effects that are evident in the learning curve. Furthermore, it can be inferred that the inclusion of update epochs to perform multiple updates using the same batch of data improved the generalizability of the model. The possibility of overfitting is reduced from this process since each mini-batch of data is randomly sampled, further acting to improve the stability and performance of the training.

4.2. Environment Experiments

As previously mentioned, different experiments were conducted based on different types of environments for the actor critic agent with $n = 2$ bootstrapping and baseline subtraction, such as the size of the environment, the speed of the ball drop, and the observation type ("vector" or "pixel"). In addition to the change of attributes, the agent was also tested under two other alternative environments, one with a 7×14 size and the other with stochastic ball trajectories. The description of these environments is elaborated in Sec. 3.2. In the following, we compare the performances of the agent in the aforementioned various environment settings. Note that, although the model with Clip-PPO loss performed best, it was not completed in time for the following analysis, hence why we chose to use the full AC model.

The upper-left panel of Fig. 3 shows the results of the learning processes for the three different environment sizes. The 7×7 environment results in an agent with the fastest learning process and the highest final score. Both the learning speed and the final score decrease with increasing environment size. Since the speed of the ball drop is set to 1 (one ball is dropped after the previous ball reaches the bottom), a larger environment indicates a slower game. Hence, for a larger environment, the agent requires a longer time to learn the game and would not have as many chances to score during one episode as an agent playing a smaller game.

The upper-right panel shows how different speeds of the ball drop influence the agent's learning curve. Generally, a more frequent ball drop corresponds to a more complex game. This is because the agent needs to keep track of multiple balls at once and quickly react to the next ball after catching the current ball. Consequently, when the frequency of the ball drop is high (e.g. speed=3.5), the agent might be struggling with tracking each ball in the game and reacting to each ball's position. However, if the frequency of the ball drop is slightly higher than the default setting (one ball at one time), the agent can still keep track of all the balls, while the environment provides the agent more chances to score. In this scenario, the agent is able to achieve a higher final score than an agent playing under the default settings. One thing to note is that the agent's learning process when speed=1.75 is faster than when speed=1.5. One possible explanation is that unlike speed=1.5, when speed=1.75, there are a consistent number of balls in the game at all times, so it might be easier for the agent to recognize the pattern and learn the optimal policy quickly.

The lower left panel shows the agent's learning curves for different combinations of ball drop frequencies and observation types. When a single ball is in the game at all times (speed=1), an agent under the "vector" observation type learns the game faster but has a lower final score compared to an agent under the "pixel" observation type. The same behaviour can be observed for two different observation types when the speed=1.75. Nonetheless, if the speed is increased to 3.5, an agent under the "vector" observation type learns at approximately the same rate as an agent under the "pixel" observation type, and unlike the agent under the "pixel" observation type, the agent with a "vector" observation type struggles to achieve higher scores after 4000 episodes. The major flaw of using "vector" to observe the environment is that when there are multiple balls in the game at the same time, based on the nature of the observation, the agent would only be able to keep track of the lowest ball before moving on to tracking other balls in the game. Effectively, the agent is playing a game with a smaller number of rows and the same number of columns compared to the agent observing using pixelated images, making the game more difficult for the agent to learn.

The lower-right panel indicates the learning curves for three different environment setups, a default square 7×7 environment with a speed of 1, a non-square 7×14 environment with a speed of 0.5, and a 7×7 environment with stochastic wind and a speed of 1. The agent achieves similar scores and learning performances under the default environment and the stochastic environment. However, for the 7×14 environment, the agent learned much slower and was not able to achieve a final score similar to the others due to a slower game provided by the environment (speed=0.5). After observing the animations generated after training for the two environment variants, it is clear that two different strategies were applied for these two variants. For the non-square environment, the agent chooses to return to the central column after each catch, so it can reach the next ball in time (the agent cannot catch the ball if they are on opposite sides of the columns). In comparison, for the stochastic environment

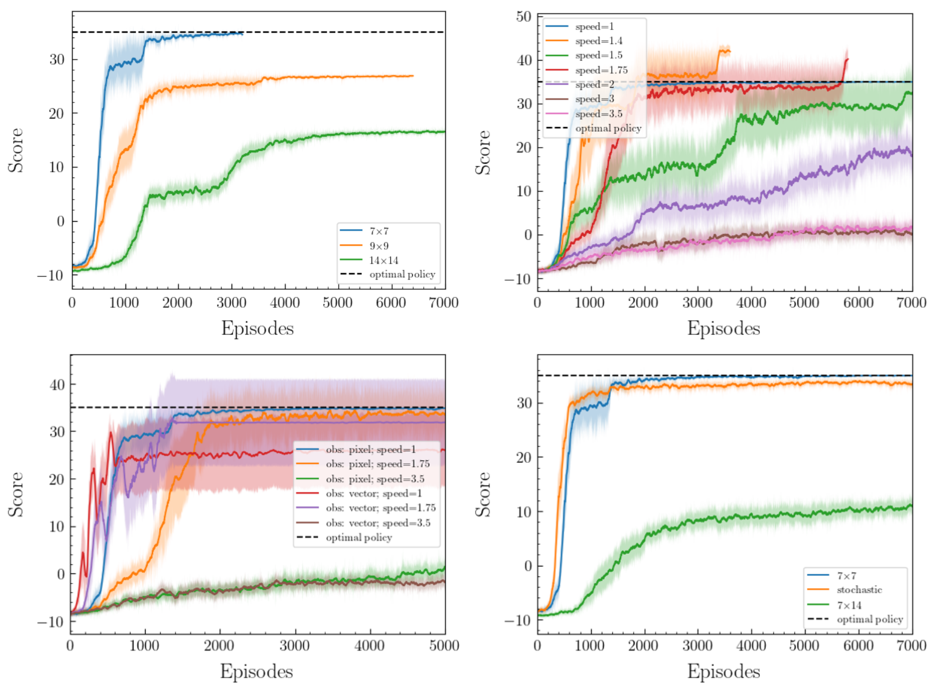

Figure 3. Agent learning performances under different environment conditions. The black dashed line indicates the optimal policy for the default environment with a 7×7 grid and speed of 1. Optimal policies for other environments vary depending on environment properties.

where balls shift columns occasionally, the agent is adapted to follow the ball in the horizontal direction to increase its chance of catching the ball.

5. Discussion & Conclusions

In this work, policy-based RL algorithms were examined by implementing and testing multiple variants of the REINFORCE and Actor-Critic (AC) algorithms on the `Catch` environment. In total, 5 models were developed and extensively explored. While a CNN was initially used to create the policy networks (and for the value network in the case of the AC models), a better and more efficient performance was achieved using a dense neural network. Relevant hyperparameters for each model were tuned to define the most optimal set-up, and experiments were repeated with results averaged to account for randomness of the action selection.

While each model exhibited learning (albeit to varying degrees), the models with REINFORCE and bootstrapped AC showed sub-optimal performance. The REINFORCE model suffered from high variance in the policy gradients, leading to slow convergence and poorer performance. While bootstrapping helped the bootstrapped AC model overcome this variance problem, the performance was still poor when compared to the other AC variants. The AC model with baseline subtraction further reduced the variance, better stabilizing the learning process and leading to faster convergence. While considerable improvements were seen with the full AC model which was able to converge to the optimal policy in significantly fewer episodes on average, the variant using a Clip-PPO loss function outperformed all other models. By defining a surrogate objective function, Clip-PPO provided a more efficient and effective policy update with reduced variance and improved stability.

To further investigate these methods, the performance of the full AC agent under various environment settings was examined. Results proved that the agent's final score and ability to learn are significantly affected by the configuration of the environment, with smaller environment size leading to better performance and faster learning. A slightly higher ball drop speed improved the final score whereas a much higher speed negatively impacted the performance. Moreover, the use of pixelated images gave better scores, especially when multiple balls were simultaneously in the game. Finally, the agent was found to have different strategies for different set-ups, as observed in the rectangular 7×14 environment and the environment with increased stochasticity.

The work presented here explored policy gradient techniques and their application among different algorithms. The construction and performance of different variants of the REINFORCE and AC algorithms were examined, and important insights about their operation across a range of environmental configurations were provided.

References

Osband, I., Doron, Y., Hessel, M., Aslanides, J., Sezener, E., Saraiva, A., McKinney, K., Lattimore, T., Szepesvari, C., Singh, S., Van Roy, B., Sutton, R., Silver, D., and Van Hasselt, H. Behaviour Suite for Reinforcement Learning. *arXiv e-prints*, art. arXiv:1908.03568, August 2019. doi: 10.48550/arXiv.1908.03568.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal Policy Optimization Algorithms. *arXiv e-prints*, art. arXiv:1707.06347, July 2017. doi: 10.48550/arXiv.1707.06347.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pp. 5–32, 1992.