Policy-based Reinforcement Learning
Reinforcement Learning, Master CS, Leiden University (2023)

Dinu Catalin-Viorel

1. Introduction

The goal of any RL algorithm is to train an agent to make decisions in dynamic environments. The agent interacts with an environment, taking action and receiving feedback in the form of rewards and penalties, with the goal of maximizing its cumulative reward over time.

In this assignment, we will investigate the policy-based approach to reinforcement learning. In contrast to value-based methods, which compute the optimal value function, policy-based methods learn a direct mapping from states to actions. Instead of optimizing the value function, policy-based methods maximize the expected cumulative reward by applying gradient descent to update the policy parameters. This approach works particularly well in environments with continuous action spaces or high dimensions.

We will explore the policy-based approach using the Catch environment. The Catch environment is a simple game where an agent controls a paddle to catch falling objects. We used Python with Tensorflow to implement and train a policy-based RL algorithm to control the paddle. We will also explore different policy-based algorithms such as the REINFORCE, Actor-Critic and Proximal Policy Optimization(PPO) algorithms and investigate how their hyperparameters affect the learning process and the final performance.

2. Background

Usually, the problems solved by Reinforcement Learning are sequential decision problems. Markov decision processes model those sequential decision problems because they have the Markov property (the next state depends only on the current state and the actions available on it). Similar to the formalisation of the Markov Decision Process (MDP) from the book (Plaat, 2022), we will define:

$$(\mathbb{S}, \mathcal{A}, T(s'|s,a), r(s,a,s'), \gamma) \rightarrow \text{MDP} \tag{1}$$

- $\mathbb{S} \rightarrow$ finite set of legal states,
- $\mathcal{A} \rightarrow$ finite set of legal actions
- $T(s'|s,a) = \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a) \rightarrow$ probability that action a in state s at time t will transition to state s' at time $t+1$
- $r(s,a,s') \rightarrow$ reward received after action a transition from state s to s'
- $\gamma \in [0,1] \rightarrow$ discount factor

An additional important concept of reinforcement learning is policy function π, which describes how an action from state s is chosen.

$$\pi : \mathbb{S} \rightarrow p(\mathcal{A}) \tag{2}$$

Where, $p(\mathcal{A}) \rightarrow$ a probability distribution function over $\mathcal{A}$. $\pi(a|s) \rightarrow$ probability to take action a in state $s (s \in \mathbb{S})$. An agent that operates in the environment samples traces(τ_t^n) (a sequence of actions, their rewards and the state after action). So, at time t, the agent is in the state s_t, it will repeat the following procedure for n times: It will choose an action a_t which moves it in a new state $s_{t+1} \sim T(s_t, a_t, s_{t+1})$ and it will get o reward of r_t after following the action a_t in state s_t.

$$\tau_t^n = \{s_t, a_t, r_t, s_{t+1}, ..., s_n, a_n, r_n, s_{n+1}\} \tag{3}$$

$$\tau_t = \tau_t^{\infty} \tag{4}$$

The sum of the cumulative reward of a trace is called a return(the overall reward got by the agent that follows a certain trace)

$$R(\tau_t) = r_t + \sum_{i=1}^{\infty} \gamma^i * r_{t+i} \tag{5}$$

Informally, the goal is to find the optimal policy $\pi^\star$, such that it maximises the return of τ_0(the trace starting in the initial state s_0). We also introduce two functions:

- State value $V^\pi(s)$ This function returns the expected return value of an agent starting in the state s that follows the policy π;

$$V^\pi : \mathbb{S} \rightarrow \mathbb{R} \tag{6}$$

$$V^\pi(s) = \mathbb{E}_{\tau_t \sim p(\tau_t)}\left[R(\tau_t)|s_t = s\right] \tag{7}$$

$$V^\pi(s) = \mathbb{E}_{\tau_t \sim p(\tau_t)}\left[\sum_{i=0}^{\infty} \gamma^i * r_{t+1}|s_t = s\right] \tag{8}$$

- State-Action value $Q^\pi(s,a)$ This function returns the expected return value of an agent starting in the state s choosing action a and following the policy π;

$$Q^\pi : \mathbb{S} \times \mathcal{A} \to \mathbb{R} \tag{9}$$
$$Q^\pi(s,a) = \mathbb{E}_{\tau_t \sim p(\tau_t)}\left[R(\tau_t) | s_t = s, a_t = a\right] \tag{10}$$
$$Q^\pi(s,a) = \mathbb{E}_{\tau_t \sim p(\tau_t)}\left[\sum_{i=0}^{\infty} \gamma^i * r_{t+1} | s_t = s, a_t = a\right] \tag{11}$$

3. Environment

3.1. Description

The Catch environment is similar to the game breakout, where the agent controls a paddle to catch balls that fall from the top of the screen. The environment is typically displayed as a square grid, with the paddle at the bottom and the balls dropping from the top. The goal of the agent is to catch as many balls as possible while avoiding misses.

3.2. States and Actions

The state space of the catch environment can be represented in different ways depending on the chosen observation type. The state can either be represented as a vector containing the x and y positions of the paddle and the lowest ball or as a binary pixel array with the paddle position in the first channel and all the balls in the second channel. The action space is discrete and consists of three possible actions: move the paddle left, move the paddle right, or stay idle.

3.3. Rewards

The reward function in the catch environment is designed to provide a positive reward when the ball is caught and a negative reward when a ball is missed. A reward of +1 is given when the agent catches a ball when it reaches the bottom row, a penalty of -1 is given when the agent misses a ball that reaches the bottom row, and 0 for all other situations.

3.4. Objective

The goal is to learn a policy that maps states to actions that will allow the agent to catch as many balls as possible while avoiding misses. The episode terminates when either the maximum number of steps is reached or the maximum number of misses is reached.

4. Methods

Different from the value-based methods, policy-based methods try to map the state space ($\mathbb{S}$) directly to a probability distribution of the action space. This mapping can be achieved for continuous and discrete action spaces. For this report, we will consider discrete action spaces ($\mathcal{A}$) because the environment we are studying has a discrete action space.

$$\mathbb{S} \to p(\mathcal{A})$$

For a discrete action space, we consider:

$$\pi_\theta(a|s) = softmax(f_\theta(s)), s \in \mathbb{S}, a \in \mathcal{A}$$
$$softmax(y_i) = \frac{e^{y_i}}{\sum_k e^{y_k}}$$

To know which policy is best, we need some kind of measure of its quality. We denote the quality of the policy that is defined by the parameters as $J(\theta)$. It is natural to use the value function of the start state as our measure of quality.

$$J(\theta) = V^{\pi_\theta}(s_0)$$
$$\theta^* \leftarrow argmax(J(\theta))$$

In order to compute optimal parameters (θ^*) of the policy ($\pi_\theta(a|s)$) we distinguish 2 different types of optimisation algorithms:

- Gradient-Based Methods (sections 4.1, 4.2, 4.3)
- Free-Gradient Methods (sections 4.4)

4.1. Monte Carlo Policy Gradient - REINFORCE

This method has the objective to compute $argmaxJ(\theta)$ and it can be achieved by gradient ascent with the following update step:

$$\theta' = \theta + \alpha * \nabla_\theta J(\theta)$$

Where α is the learning rate. In order to compute the gradient of $J(\theta)$ we will take the following steps:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\tau_0 \sim p_\theta(\tau_0)}\left[R(\tau_0)\right] \\
&\overset{\text{log-derivative trick}}{=} \mathbb{E}_{\tau_0 \sim p_\theta(\tau_0)}\left[R(\tau_0)\nabla_\theta \log p_\theta(\tau_0)\right] \\
&\overset{\text{gradient of trace}}{=} \mathbb{E}_{\tau_0 \sim p_\theta(\tau_0)}\left[R(\tau_0)\sum_{t=0}^{n} \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \\
&= \mathbb{E}_{\tau_0 \sim p_\theta(\tau_0)}\left[\sum_{t=0}^{n} R(\tau_t) * \nabla_\theta \log \pi_\theta(a_t|s_t)\right]
\end{aligned}$$

We observe that this method has to sample a whole episode in order to compute the quality of the policy, which means that the policy-based method is low bias. However, this method is high-variance since the trajectory are generated randomly. Those two properties lead to 2 consequences:

- Policy evaluation of full trajectories has low sample efficiency and high variance, so policy improvement happens infrequently, leading to slow convergence compared to value-based methods;

- This approach often finds a local optimum since convergence to the global optimum takes too long;

Another problem raised from the training period of such a method is that the probability distribution may collapse due to getting stuck to the local minimum that exploits only one action that at some point got some positive reward. In order to avoid this premature convergence, we can use entropy regularisation:

$$H(\pi_\theta(.|s_t)) = -\sum_{a\in\mathcal{A}} \pi_\theta(a|s_t) * \log \pi_\theta(a|s_t)$$

For a high entropy, we observe a low certainty of which action will lead to a positive return, while for low entropy, we observe a high centrality of which action will lead to a positive return. So adding this to the formula of computing the gradient step will reduce the risk of probability distribution collapsing. We want to find the parameters θ^* that maximise $J(\theta)$ and entropy on each state.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_0\sim p_\theta(\tau_0)}[\sum_{t=0}^{n} R(\tau_t) * \nabla_\theta \log \pi_\theta(a_t|s_t) + \\ + \eta * \nabla_\theta H(\pi_\theta(.|s_t))]$$

Where η is the entropy coefficient. The entropy regularisation will be used for all other gradient methods.

4.2. Actor Critic

The actor-critic approach combines value-based elements with the policy-based method. The actor stands for the action or policy-based approach; the critic stands for the value-based approach. We observe that a low bias characterises Monte Carlo Policy Gradient, action selection being random, but it suffers from a high variance due to sampling a whole episode in order to compute the quality of the policy. Actor-Critic methods appeared in order to combine the advantages of value-based methods (low variance) and policy-based methods (low bias). The high variance of policy methods can originate from 2 sources:

- High variance in the cumulative reward estimate
- High variance in the gradient estimate

For the following methods, we consider the parameterized value function $V_\phi(s)$, which estimates the cumulative reward of the state s.

4.2.1. Bootstrap

To compute a better cumulative reward estimate, we can use intermediate m-step value function estimation to compute the return of a state (s), trading variance for bias.

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

Parameterized value function $V_\phi(s)$ can be computed using gradient decent and the following loss function:

$$\mathcal{L}(\phi) = \sum_{t=0}^{n} (\hat{Q}_m(s_t, a_t) - V_\phi(s_t))^2$$

The new gradient used for updating the parameters of the policy function (θ) becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_0\sim p_\theta(\tau_0)}\left[\sum_{t=0}^{n} \hat{Q}_m(s_t, a_t) * \nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

4.2.2. Baseline Subtraction

Baseline subtraction is a method of reducing the variance of the policy gradient. Subtracting a baseline from a list of numbers will reduce the variance but let the expectation unaffected. Given a state (s) and we sample for each possible action a positive reward, this behaviour will increase the probability of each action, even though the action that scored the best should have its probability increased and the action that scored the lowest should have its probability decreased (even though it has a positive reward)

$$\hat{A}_{MC}(s_t, a_t) = \sum_{k=0}^{\infty} \gamma^k r_{t+k} - V_\phi(s_t)$$
$$\hat{A}_{MC}(s_t, a_t) = R(\tau_t) - V_\phi(s_t)$$

Parameterized value function $V_\phi(s)$ can be computed using gradient decent and the following loss function:

$$\mathcal{L}(\phi) = \sum_{t=0}^{n} \hat{A}_{MC}(s_t, a_t)^2$$

The new gradient used for updating the parameters of the policy function (θ) becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_0\sim p_\theta(\tau_0)}\left[\sum_{t=0}^{n} \hat{A}_{MC}(s_t, a_t) * \nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

4.2.3. Both

We can combine baseline subtraction with any bootstrapping method to estimate the cumulative reward:

$$\hat{A}_m(s_t, a_t) = \sum_{k=0}^{m} \gamma^k r_{t+k} + V_\phi(s_{t+m}) - V_\phi(s_t)$$

So the new gradient for the policy function is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_0\sim p_\theta(\tau_0)}\left[\sum_{t=0}^{n} \hat{A}_m(s_t, a_t) * \nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

4.3. Proximal Policy Optimization(PPO) - Clip

Proximal Policy Optimization ((Schulman et al., 2017)) tries to improve the training stability of the policy by limiting the change you make to the policy at each training epoch. In order to achieve this, two solutions were combined:

- Compare the current policy to the previous policy by using the ratio between those
- Clip the ratio between $1-\epsilon$ and $1+\epsilon$, so it removes the incentive for the policy network to go far from the old one.

$\rho_t(\theta)$	$\hat{A}_m$	Min Value	Clipped	Grad
$[1-\epsilon, 1+\epsilon]$	+	$\rho_t(\theta)\hat{A}_m$	no	YES
$[1-\epsilon, 1+\epsilon]$	-	$\rho_t(\theta)\hat{A}_m$	no	YES
$< 1-\epsilon$	+	$\rho_t(\theta)\hat{A}_m$	no	YES
$< 1-\epsilon$	-	$(1-\epsilon)\hat{A}_m$	yes	0
$> 1-\epsilon$	+	$(1+\epsilon)\hat{A}_m$	yes	0
$> 1-\epsilon$	-	$\rho_t(\theta)\hat{A}_m$	no	YES

Table 1. How minimum function behave ($\hat{A}_m = \hat{A}_m(s_t, a_t)$)

So we define:

$$\rho_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \text{ratio of probabilities}$$

$$\hat{A}_m(s_t, a_t) = \sum_{k=0}^{m} \gamma^k r_{t+k} + \gamma^m V_\phi(s_{t+m}) - V_\phi(s_t)$$

Using those, we can define our loss function:

$$g_t(\theta) = min(\rho_t(\theta)\hat{A}_m(s_t, a_t), clip(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_m(s_t, a_t))$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_0 \sim p_\theta(\tau_0)} \left[\sum_{t=0}^{n} \nabla_\theta g_t(\theta) \right]$$

4.4. CMA-ES (Covariance matrix adaptation evolution strategy)

The covariance matrix adaptation evolution strategy ((Hansen, 2016)) is a gradient-free optimisation method. Given a black box fitness function $f : \mathbb{R}^n \to R$, this method will try to find the input value that maximises/minimise the function. So for our problem, we consider the policy function $\pi_\theta(a|s)$ and we have to define a fitness function that computes the quality of the policy:

$$J(\theta) = V^{\pi_\theta}(s_0)$$

So we consider the expected cumulative return of a trace starting in the initial state as our fitness function. The algorithm consists of the following recursive steps:

1. Sampling new population of size λ

   $\theta_i^g = m^g + \sigma^g y_i^g, i \in 1...\lambda$
   $y_i^g \sim \mathcal{N}_i(0, C^g) \to$ multivariate normal distribution with zero mean and covariance matrix C^g

2. Compute the new mean of the population using weights w_i ($\sum_{i=1}^{\mu} w_i = 1$ and $w_1 > ... > w_\mu$)

$$m^{g+1} \leftarrow \sum_{i=1}^{\mu} w_i * \theta_{i:\lambda}^g = m^g + \sigma^g y_w^g$$

$$y_w^g = \sum_{i=1}^{\mu} w_i * y_{i:\lambda}^g$$

$$\theta_{i:\lambda}^g, y_{i:\lambda}^g \to i^{th} \text{ element from population according to the fitness function}$$

3. Update the covariance matrix(C) by using evolution path (p_c)

$$p_c^{g+1} \leftarrow (1-c_c)p_c^g + \mathbb{I}_{\{||p_\sigma|| < 1.5\sqrt{n}\}} \sqrt{1-(1-c_c)^2} \sqrt{\mu_w} * y_w^g$$

$$C^{g+1} \leftarrow (1-c_1-c_\mu)C^g + c_1 p_c^{g+1}(p_c^{g+1})^T + c_\mu \sum_{i=1}^{\mu} w_i y_{i:\lambda}^g (y_{i:\lambda}^g)^T$$

   c_c - decay rate for cumulation path for the rank-one update of the covariance matrix; c_1 - learning rate for the rank-one update of the covariance matrix; c_μ - learning rate for the rank-μ update of the covariance matrix;

4. Update the step size (σ) by using evolution path (p_σ)

$$p_\sigma^{g+1} \leftarrow (1-c_\sigma)p_\sigma^g + \sqrt{1-(1-c_\sigma)^2} \sqrt{\mu_w} * (C^g)^{-\frac{1}{2}} * y_w^g$$

$$\sigma^{g+1} \leftarrow \sigma^g * exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{||p_c^{g+1}||}{E[\mathcal{N}(0,I)]} - 1\right)\right)$$

   c_σ - decay rate for the cumulation path for the step-size control; d_σ - damping parameter for step-size update;

5. Experiments

5.1. HyperParameter Tuning

Hyperparameter tuning is a critical step in optimizing the performance of policy-based RL algorithms. We have conducted experiments by varying the values of these parameters which include the learning rate, and the entropy coefficient. Using the grid search approach, we trained the algorithm for variations of these values and measured its performance by evaluating the average reward obtained over 1000 episodes for 5 repetitions.

5.1.1. Learning Rate (α)

One of the most important hyperparameters for RL algorithms is the learning rate. It determines how much the weights of the neural network are updated in response to the gradient which influences the algorithm's stability and convergence rate. A suitable learning rate will allow the model to converge to an optimal solution efficiently. In our experiments, we explored 3 different learning rates: [0.1, 0.01, 0.001].

5.1.2. Entropy coefficient

The entropy coefficient is another important hyperparameter as we are using entropy regularisation. The entropy coefficient is crucial in controlling the trade-off between exploration and exploitation in the learning process. A suitable entropy coefficient will allow the model to improve its stability and prevents premature convergence. In our experiments, we explored three different entropy coefficient values: [0.01, 0.1, 0.5].

5.2. HyperParameter tuning for MCPG (REINFORCE)

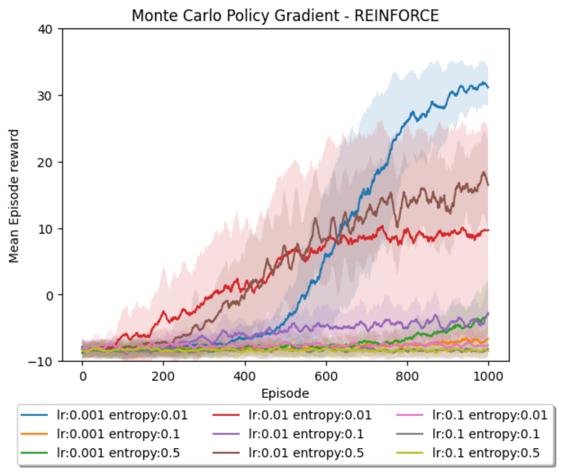

Figure 1. Comparison of rewards we got for 1000 episodes on the MCPG (REINFORCE) model with different combinations of α and entropy coefficient averaged over 5 repetitions

From the plot 1, we can see that the graph lines of a learning rate of 0.1 with all combinations of entropy coeff. not performed well indicating that the agent is not learning, likely because the learning rate is too high, causing the agent to overshoot the optimal policy and never converge. Similarly, we can see that the graph line for an entropy coeff. of 0.1 with all combinations of learning rates not performed well indicating that the agent is not learning, likely because the entropy coefficient is too high, causing the agent to do excessive exploration leading to difficulty in converging to the optimal policy. The rewards for the model with a learning rate of 0.01 and entropy coeff. of 0.01 increase initially up to a certain point and after that stayed the same, this is because both the learning rate and the entropy coeff. are low, causing the agent to take too long to converge, or the agent is getting stuck in local optima and is unable to find a better policy. Also, the deviation of the rewards is high, indicating a high degree of variance in the learning process.

From the plot 1, we can see that the MCPG model uses the learning rate of 0.001 and entropy coeff. of 0.01 performs well, and the rewards are increasing steadily indicating that the agent has converged to a near-optimal policy and then increases again, potentially indicating that the agent found a way to improve its policy. The entropy coeff. of 0.01 gives a balance between the exploration and exploitation which helps in reducing the variance by preventing the agent from being trapped in local optima.

5.3. HyperParameter tuning for Actor-Critic Bootstrap

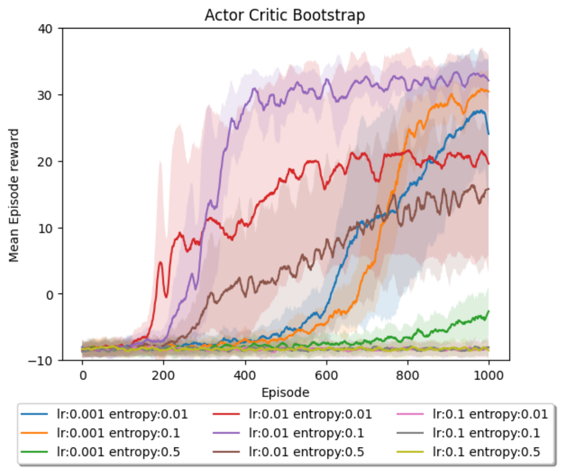

Figure 2. Comparison of rewards we got for 1000 episodes on the Actor-Critic Bootstrap model with different combinations of α and entropy coefficient averaged over 5 repetitions

From the plot 2, we can see that the graph lines of a learning rate of 0.1 with all combinations of entropy coeff. not performed well indicating that the agent is not learning, likely because the learning rate is too high, causing the agent to overshoot the optimal policy and never converge. The rewards for the model with a learning rate of 0.01 and entropy coeff. of 0.01 increase steadily, but the deviation of the rewards is very high, this is because of both the learning rate and the entropy coeff. are low, causing the agent to take too long to converge, or the agent is getting stuck in local optima and is unable to find a better policy. Also, the deviation of the rewards is high, indicating a high degree of variance in the learning process.

From the plot 2, we can see that the Actor-Critic Bootstrap model uses a learning rate of 0.01 and entropy coeff. of 0.1 performs well, and the rewards are increasing steadily indicating that the agent has converged to a near-optimal policy and then performs well, indicating that the agent found a way to improve its policy. Also, the model with a learning rate of 0.001 and entropy coeff. of 0.1 performed reasonably, but the model with a learning rate of 0.01 and entropy coeff. of 0.1 converged faster than the others.

5.4. HyperParameter tuning for Actor-Critic Baseline Subtraction

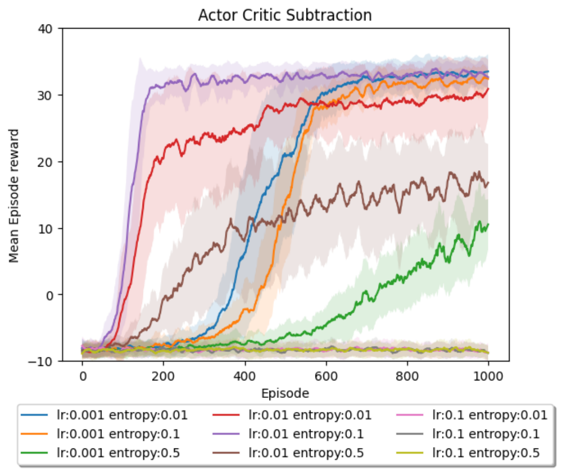

Figure 3. Comparison of rewards we got for 1000 episodes on the Actor-Critic Baseline Subtraction model with different combinations of α and entropy coefficient averaged over 5 repetitions.

From the plot 3, we can see that the graph lines of a learning rate of 0.1 with all combinations of entropy coeff. not performed well indicating that the agent is not learning, likely because the learning rate is too high, causing the agent to overshoot the optimal policy and never converge. Also, the combination of different learning rates with the entropy coeff. of 0.5 is not performing as well, likely because of the entropy coeff. is too high, causing the agent to do excessive exploration leading to difficulty in converging to the optimal policy. Also, the deviation of the rewards is high, indicating a high degree of variance in the learning process.

From the plot 3, the best hyperparameters for the actor-critic baseline subtraction model are a learning rate of 0.01 and an entropy coeff. of 0.1. we can see that the models with a learning rate of 0.001 and entropy coefficients of 0.01 and 0.1 also performed well, but the combination of a learning rate of 0.01 and the entropy coeff. of 0.1 learnt faster. This is because of the entropy coeff. of 0.01 and the learning rate of 0.1 helps in a steady learning process and a moderate level of exploration respectively.

5.5. HyperParameter tuning for Actor-Critic Both

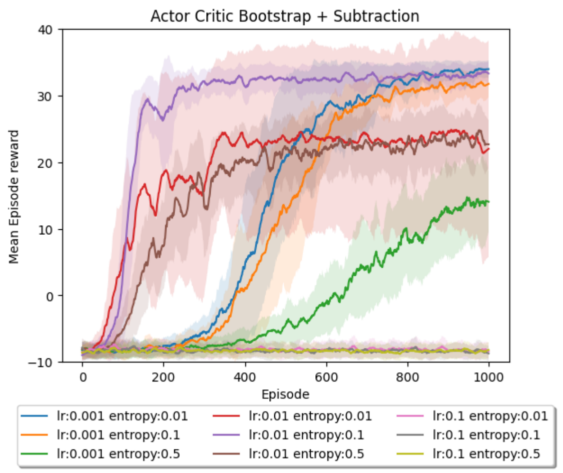

Figure 4. Comparison of rewards we got for 1000 episodes on the Actor-Critic: Bootstrap + Baseline Subtraction model with different combinations of α and entropy coefficient averaged over 5 repetitions

From the plot 4, we can see that the graph lines of a learning rate of 0.1 with all combinations of entropy coefficient don't perform well indicating that the agent is not learning, likely because the learning rate is too high, causing the agent to overshoot the optimal policy and never converge. Also, the best hyperparameters for the Actor-Critic Bootstrap + Baseline Subtraction model are a learning rate of 0.01 and an entropy coefficient of 0.1. we can see that the models with a learning rate of 0.001 and entropy coefficients of 0.01 and 0.1 also performed well, but the combination of a learning rate of 0.01 and the entropy coefficient of 0.1 learnt faster. This is because the entropy coefficient of 0.01 and the learning rate of 0.1 helps in a steady learning process and a moderate level of exploration respectively.

5.6. Policy-Based RL Algorithms comparison

The plot 5, shows the rewards we got across the Policy Gradient algorithms we mentioned earlier. The rewards are obtained using the best hyperparameters for each algorithm after tuning. The comparison shows the differences in learning efficiency, stability, and convergence of these algorithms in the given catch environment. From the plot 5, we can see that all 4 algorithms converged, but the MCPG took a lot of time to find the optimal policy compared to the Actor-Critic models. Also, the MCPG has high deviations in the rewards showing high variability in the learning process compared to the actor-critic models. The Actor-Critic Bootstrap model took less time than MCPG, but the other actor-critic models are faster. From the plot 5, we can see that both the Actor-Critic: Baseline Subtraction and Actor-Critic: Bootstrap +

Baseline Subtraction models performed almost equally. We picked the Actor-Critic: Bootstrap + Baseline Subtraction as the best model because, by using both bootstrapping and baseline subtraction, the algorithm strikes a better balance between bias and variance in the gradient estimates. By taking this model as our best model, we can use the benefits of both Bootstrapping and Baseline subtraction and also the variance is less. The table 2, shows the best hyperparameters we got for each algorithm after tuning.

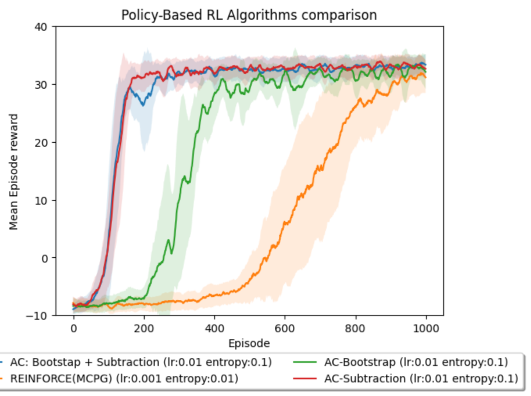

Figure 5. Comparison of rewards we got for 1000 episodes across Policy-Based RL Algorithms including REINFORCE (MCPG), Actor-Critic: Bootstrap, Actor-Critic: Baseline Subtraction, and Actor-Critic: Bootstrap + Baseline Subtraction averaged over five repetitions for this plot. The rewards are obtained using the best hyperparameters for each algorithm after tuning.

Algorithms	Learning Rate	Entropy Coeff.	Rewards
Reinforce (MCPG)	0.001	0.01	29.40 ± 4.63
Actor-Critic : Bootstrap	0.01	0.1	32.60 ± 1.50
Actor-Critic : Baseline Subtraction	0.01	0.1	33.00 ± 1.79
Actor-Critic : Bootstrap + Subtraction *	**0.01**	**0.1**	$\mathbf{33.80 \pm 0.98}$

Table 2. Best Hyperparameters we got for different Policy-Based RL Algorithms along with the rewards we got for 1000 episodes averaged over 5 repetitions. (* the best model)

5.7. Environment Variations

For evaluating the performance of the model, we used two metrics

- **Mean reward with standard deviation :** The reward we got for 1000 episodes on our best model (Actor-Critic: Bootstrap + Baseline Subtraction model) averaged over 5 repetitions.
- **Theoretical max reward (TMR)** is the maximum reward a perfect agent can achieve without ever missing a ball in 250 steps.

The best model referred to in the below sections denotes the Actor-Critic: Bootstrap + Baseline Subtraction model.

5.7.1. Varying the size of the environment

We tried 2 more environment sizes (5x5 and 11x11) on our best model and compared its performance with the default size of 7x7. From the table 3, we can clearly see that our best model not performed well on the 11x11 environment, this may be because of the larger size of the environment which makes it harder for the model to learn the optimal policy. Also, our network architecture may be small for the larger environment. Our model performed well on environment 5x5 than the default environment (7x7). This may be because there are fewer possible states for the agent to encounter, which makes it easier for the model to learn an optimal policy effectively and efficiently. Also, in a smaller environment, random exploration is more likely to lead to beneficial state transitions, as there is a higher probability of facing relevant states and actions.

Env. Size	Rewards	TMR
5x5	$\mathbf{49.20 \pm 0.98}$	**50**
7x7	33.80 ± 0.98	35
11x11	13.60 ± 12.03	22

Table 3. Comparison of rewards we got for 1000 episodes on our best model with different env sizes averaged over 5 repetitions.

5.7.2. Varying the speed of the ball drop

The table 4 shows the mean rewards we got from our best model on different speed values (0.5, 1.0, 1.5).

Size	Speed	Rewards	TMR
7x7	**0.5**	$\mathbf{17.20 \pm 0.98}$	**18**
7x7	1.0	33.80 ± 0.98	35
7x7	1.5	49.20 ± 2.40	63

Table 4. Comparison of rewards we got for 1000 episodes on our best model with different speeds averaged over five repetitions.

From the table 4, we can see that the mean reward the best model got for the environment of size 7x7 with speed 0.5 is less than the mean reward we got for the environment of size 7x7 and speed 1.0, but the theoretical max reward for each environment is different, so based on that our best model performed well on the environment with slower speed (0.5) than the environment with normal speed (1.0). This may be because slower the speed of the ball gives the agent more time to react to the ball's movement, this makes the learning process easier, the agent has to find the optimum from a smaller number of future states with more time to react. on the other hand, our model didn't perform well when the speed is 1.5 this may be because the faster the ball drop speed, the agent has to react faster means it has to consider a lot of future steps. Also, with a speed of 1.5, there may be multiple balls on the screen at the same time.

5.7.3. Varying the observation type and Speed

We also changed the observation type to 'vector' and tested the performance along with changing the speed, the results can be seen from the table 5. For the observation type as 'vector' we used a bigger network architecture as the agent has only a vector of length 3: [x_paddle,x_lowest_ball,y_lowest_ball] to work with.

Observation	Speed	Rewards	TMR
pixel	1.0	33.80 ± 0.98	35
vector	1.0	33.80 ± 1.60	35

Table 5. Comparison of rewards we got for 1000 episodes on our best model with different observation types and speeds averaged over five repetitions with theoretical max rewards(TMR).

Performance of Observation Type 'vector': From the table 5, we can see that changing the observation type from 'pixel' to 'vector' didn't give us any improvement. On the other hand, 'pixel' observations provide a more detailed representation of the environment, capturing the spatial relationships between the paddle and the balls which gives enough information for the agent to learn.

Performance of Observation as 'vector' + Speed > 1.0: We can't use observation type as 'vector' with a speed greater than '1.0'. A speed > 1.0 for vector space is impossible to train properly because two or more balls can be seen at the same time on the board, but the representation will fail to show the position of all balls. It will only show the position of the lowest ball.

5.7.4. Combination of variations

We tried a non-square environment of size 7x14 with a ball drop speed of 0.5 on our best model, the table 6, shows the rewards we got for that variation along with the default settings and some smaller non-square environments. From the table 6, we can clearly see that our best model performed well on the default setting (size: 7x7, speed: 1.0) than the non-square environments. This may be because of the larger size of the environment which makes it harder for the model to learn the optimal policy. Also, the slower speed of the ball time gives more time for the agent to react which in turn makes the agent consider more future steps to find the optimal decision.

Env. Size	Speed	Rewards	TMR
7x14	0.5	4.80 ± 5.88	18
7x14	1.5	7.40 ± 4.27	63
7x7	1.0	33.80 ± 0.98	35
5x7	1.0	33.40 ± 2.33	50
5x7	0.5	16.80 ± 0.98	25

Table 6. Comparison of rewards we got for 1000 episodes on our best model with different combinations of environment variations averaged over five repetitions with theoretical max rewards(TMR).

We increased the speed of the ball drop to 1.5 to see if that affects it, also we tried our best model on different non-square environments of size 5x7 with values of speed as 1.0 and 0.5. From the results 6 we can see that the larger size of the environment needs more time to learn and also we haven't used any pooling layers in the architecture which may also be a problem. Based on the theoretical max rewards the non-square environment models didn't perform well as compared to the square environments.

5.8. Our Best Model Vs. Clip-PPO Vs. CMA-ES

We ran the Clip-PPO and CMA-ES on our catch environment. The plot 6, shows the rewards we got for 200 episodes from our best model, Clip-PPO and CMA-ES averaged over 5 repetitions. We used only 200 episodes as CMA-ES needs more time to run. We can see that both CMA-ES and Clip-PPO learn faster than our best model. Both CMA-ES and Clip-PPO learnt very quickly which makes them suitable for bigger environments.

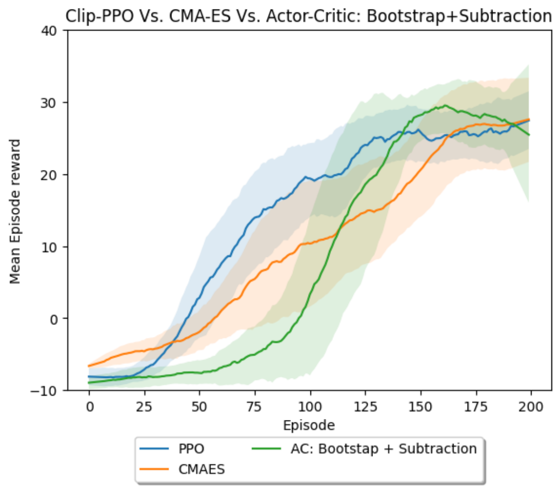

Figure 6. Comparison of rewards we got for 200 episodes on our best model, Clip-PPO and CMA-ES averaged over five repetitions.

6. Discussion

In conclusion, the policy-based RL algorithm has been successfully applied in a catch environment, the implementation results demonstrate that the agent was able to learn an optimal policy that catches the ball dropping from the system, leading to significantly higher rewards. Our experimentation on the hyperparameters and the comparison of MCPG and different variations of Actor-Critic algorithms show the importance of hyperparameter tuning and the advantage of combining value-based and policy-based methods for better performance. Our study demonstrates the possibility of the policy-based RL algorithm to solve various challenging problems in robotics and gaming. Even though our experimentation is based on a particular environment, the understanding acquired can be applied to other environments. Additionally, exploring other advanced algorithms like Clipped PPO, and CMA-ES may lead to even better performance and wider application in solving real-world problems making it an exciting area for future research.

References

Hansen, N. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

Plaat, A. *Deep Reinforcement Learning*. Springer, 2022.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.