Reinforcement Learning - Spring Semester 2023
Assignment 3. Policy-based RL

Antonia Christodoulou (s3614794), Kyriakos Aristidou (s3510123), Shreyansh Sharma (s3772241)

Abstract

Our work focuses on the investigation of different *policy-based* methods in reinforcement learning to solve the Catch environment. We implement **REINFORCE** and **Actor-Critic** algorithms as well as some variations of Actor-Critic which include bootstrapping or/and baseline subtraction techniques. After concluding to simple Actor-Critic to be the best agent, we make several experiments using it, changing many different configurations of the environment such as the size, speed, etc so as to examine its performance.

1. Introduction

Reinforcement learning (RL) is a strong machine-learning technique that has received a lot of attention over the last several years. In the previous assignment, we investigated the value-based methodology of RL, using algorithms such as *SARSA, Q-learning, and DQN*, with the objective of learning the values of actions in different states. In this assignment though, we will examine the policy-based approach. The policy-based approach, as opposed to the value-based approach, optimizes the policy directly instead of acquiring it from a representation of the value function.

The goal of this assignment is to build and compare several policy-based RL algorithms. To accomplish this, we will use the Catch environment (Moerland), a task in which the purpose is to operate a paddle to catch falling balls.

Initially, we implement various algorithms from scratch: ***REINFORCE***, ***ActorCritic*** *with bootstrapping, with baseline subtraction, and a combination of the two.* We then tune the hyperparameters of each algorithm in order to make it a fair comparison. After which we compare their performance and select the best configuration, which is later utilized for more experimentation on the environment variations. More specifically, using the best agent, we will examine different configurations of the environment (like the ball's speed, the grid's size, etc.) to see how our agent's performance is affected.

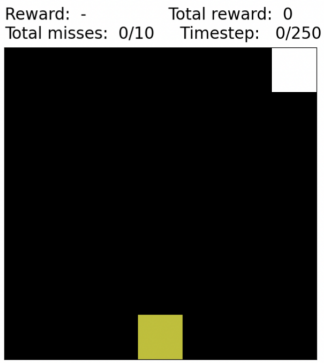


Figure 1. Illustration of the Catch environment provided by the authors of Assignment 3. The yellow block at the bottom represents the paddle and the white block above is the falling ball. The black background presents the space within which the game takes place.

2. Catch Environment

The Catch environment, as illustrated in *Figure 1*, is a game in which you use a paddle to catch falling balls. To get rewards, you have to move the paddle(the yellow block) and catch as many balls(the white block) as possible. The balls fall at various speeds from the top of the screen. At each timestep, you get a +1 reward if the position of the yellow block is the same as the white block's, you get -1 if the white block reaches the bottom but is found in a different position than the yellow one, and, finally, you get +0 reward in any other case. The state space includes the height and width of the grid (number of rows and columns) while the action space includes three different possible moves of the paddle: left, right, or stay idle.

The task finishes either if we reach the maximum steps set by the user, or if the paddle reaches the maximum number of misses set by the user as well.

To adjust the difficulty level, initialization configurations can be used to personalize the environment. The number of rows and columns on the game board, for example, can be changed to broaden or narrow the space that the paddle must travel. The speed rate at which the balls fall can also be adjusted to make the game quicker or slower. There are, also, different observation types, such as raw pixel images

or preprocessed vector representations of the game state.

Because of its ease of use and adaptability, the Catch environment is widely utilized. It is simple to use and may be tailored to a variety of experimental and testing scenarios.

3. Methodology

3.1. REINFORCE

Reinforce is a famous algorithm in reinforcement learning and is used to train agents to execute tasks in situations that are unpredictable like the catch environment in this case. The agent in Reinforce engages with the environment and is rewarded for each action it takes. The aim of the agent is to gradually develop a policy that maximizes the predicted reward. The agent accomplishes this by employing a stochastic policy, which generates a probability distribution over actions at each time step. The agent selects its actions by sampling from this distribution.

The main concept of Reinforce is to update the agent's policy using the policy gradient theorem. The policy gradient theorem assumes that the product of the expected reward and the gradient of the log probability of the action chosen with respect to the policy parameters determines the expected reward's gradient with respect to the policy parameters. In a few words, we can adjust the policy configurations so that the expected reward is higher by using the gradient of the log probability of the taken action.

Using stochastic gradient ascent, the Reinforce algorithm updates the network parameters based on the gradient of the predicted return. The sum of the rewards obtained for each action in the current episode, weighted by the probability that the action will be taken, is used to compute the expected return, this is what we call the Monte Carlo method.

First, we compute the discounted returns G_t for each time step t using the rewards r_k received at time steps $k \geq t$ and the discount factor γ. This equation computes the total discounted return from each time step t to the end of the episode at time step $T-1$, where $\gamma \in [0, 1)$ is the discount factor that determines the weight of future rewards relative to immediate rewards:

$$G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k \tag{1}$$

Next, we compute the baseline b, which is the expected value of the returns across all time steps:

$$b = \frac{1}{T} \sum_{t=0}^{T-1} G_t \tag{2}$$

Then, we compute the normalized returns $\hat{G}_t$ by subtracting the baseline b from each return G_t and dividing the result by the standard deviation σ of the returns. This equation standardizes the returns, which helps to reduce the variance of the gradients and improve the convergence of the algorithm.:

$$\hat{G}_t = \frac{G_t - b}{\sigma} \tag{3}$$

where σ is the standard deviation of the returns across all time steps

After computing the normalized returns, we update the policy network parameters θ by taking a gradient step in the direction of the policy gradient $\nabla_\theta \log \pi_\theta(a_t|s_t)$ weighted by the normalized return $\hat{G}_t$. This equation updates the policy network parameters using the policy gradient, which is the gradient of the expected return with respect to the policy parameters. The learning rate α determines the step size of the update.:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) \hat{G}_t \tag{4}$$

Then we compute the loss, we first calculate the policy loss and entropy loss for each time step t. First, we loop through the saved log probabilities and the corresponding returns for each time step. For each time step, we calculate the policy loss as the negative log probability of the selected action multiplied by the corresponding return:

$$\mathcal{L}_{policy} = -\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \cdot G_t \tag{5}$$

where $\hat{G}_t$ is the normalized return for time step t.

Next, we calculate the entropy loss by taking the negative sum of the logarithm of the policy probabilities $\pi_\theta(a_t|s_t)$ for each time step t. The entropy loss encourages the policy to be more exploratory by penalizing it for being too certain about its actions.

$$\mathcal{L}_{entropy} = -\sum_{t=1}^{T} \pi_\theta(a_t|s_t) \cdot \log \pi_\theta(a_t|s_t) \tag{6}$$

We then sum up the policy loss and entropy loss across all time steps to get the total loss, where β is the entropy weight that control the strength of the entropy regularization:

$$\mathcal{L} = \sum_{t=0}^{T-1} (\mathcal{L}_{\text{policy}}(t) + \mathcal{L}_{\text{entropy}}(t) \cdot \beta)$$

Finally, we calculate the gradients of the total loss with respect to the policy parameters θ and update the parameters using gradient descent:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}$$

where α is the learning rate.

3.2. Actor-Critic

As explained above, in REINFORCE we use Monte Carlo to estimate the return of a whole episode which leads to high variance. The Actor-Critic algorithm uses a hybrid architecture combining both value-based and policy-based methods aiming to reduce the variance and stabilize the learning process. So, the architecture used for this algorithm consists of two parts:

- the **Actor**: which determines the action that the agent will follow (the policy-based part).
- the **Critic**: which criticizes how good or bad this action is (the value-based part).

The general idea behind Actor-Critic is that the actor executes some actions, the critic provides feedback and the policy is updated based on this feedback. So, in this way, our agent becomes better and better at playing a game (the Catch in our case). However, not only the policy has to be updated but, also, the value function which determines how effective or not the action is. So, value and policy losses are calculated, and, then, using backpropagation the weights of the networks (θ, ϕ) are updated.

Summing up, we have two neural networks to train:

- for the ***Policy function*** controlled by parameters θ
- for the ***Value function*** controlled by parameters ϕ

So, now there are two techniques to reduce variance:
1) *Bootstrapping*, and 2) *Baseline Subtraction*.

3.2.1. Bootstrapping

Bootstrapping is a technique that refers to the target estimation Q_n. Unlike the Monte Carlo method which computes the full expected reward at a certain timestep t, bootstrapping sums up the rewards of depth *n*. As n gets higher we result in the Monte Carlo method and bootstrapping is gone. In our case, we use n=100 after experimenting with different values of n such as n=1, 40, 100, 200. So, the procedure is sampling a trace, and then iterating over the number of timesteps, we compute the n-step target:

$$Q_n(s_t, a_t) = \sum_{k=0}^{n-1} r_{t+k} + V_\theta(s_{t+n}) \tag{7}$$

Then we update the Critic network as:

$$\theta \leftarrow \theta + \eta \sum_t [Q_n(s_t, a_t) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{8}$$

and the Actor Network is updated as:

$$\phi \leftarrow \phi - \eta \sum_t \nabla_\phi (Q_n(s_t, a_t) - V_\phi(s_t))^2 \tag{9}$$

3.2.2. Baseline Subtraction

The learning process can be further stabilized and the variance reduced using the **Advantage function (A(s,a))**. The idea is that we want to push up the probability of taking actions that result in a Q-value higher than the average value of that state. So, the Advantage function basically calculates the extra reward that an action results in, compared to the average reward of that state:

$$A(s, a) = Q(s, a) - V(s) \tag{10}$$

So, using the baseline subtraction technique, the algorithm first samples a trace and then computes the target which might be calculated using n-step bootstrapping (Q_n) as shown above or the Monte Carlo method (Q_{MC}):

$$Q_n(s_t, a_t) = \sum_{k=0}^{n-1} r_{t+k} + V_\theta(s_{t+n}) \tag{11}$$

or

$$Q_{MC}(s_t, a_t) = \sum_{i=t}^{\infty} \gamma^i \cdot r_i \tag{12}$$

where γ is a discounted factor, r indicates the reward, t indicates the timestep, and V_θ is the value of that state.

The Advantage is calculated as shown in equation (4) and then the Critic network is updated as:

$$\theta \leftarrow \theta + \eta \sum_t [A_n(s_t, a_t) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{13}$$

and the Actor Network is updated as:

$$\phi \leftarrow \phi - \eta \sum_t \nabla_\phi (Q_n(s_t, a_t) - V_\phi(s_t))^2 \tag{14}$$

4. Experiments

Our experiments are divided into two parts. Firstly, we use the default configurations of the Catch environment and check the performance of the algorithms (REINFORCE, Actor-Critic, Actor-Critic with bootstrapping, Actor-Critic with baseline subtraction, and Actor-Critic with both) comparing them and aiming to find the best agent.

The **default parameters** of the environment are:

- **Number of rows:** 7
- **Number of columns:** 7
- **Speed of the falling ball:** 1.0
- **Observation Type:** pixel
- **Maximum number of steps in one episode:** 250
- **Maximum number of misses in one episode:** 10

After determining the best agent, we move on to the second part of our experiments where we use the best algorithm in modified environments to examine if and how the performance of the agent is affected. We make changes in the size of the environment, the speed of the falling ball, and the observation type. All the experiments and plots presented in the following sections are averaged over 10 repetitions.

4.1. Networks and Model Architectures

4.1.1. REINFORCE Architecture

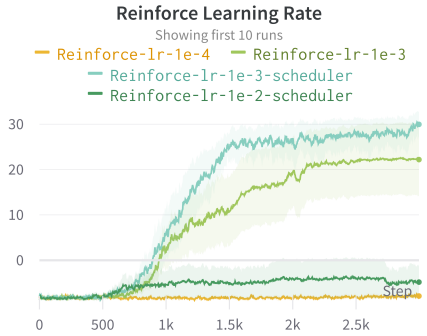
As mentioned before for the catch environment, we have two options for observations: ”vector” and ”pixel”. For ”vector” observations, we use a simple policy network that consists of fully connected layers. The policy network takes in the current state of the environment as an input and outputs a probability distribution over the possible actions. We choose this architecture because the observation space is small and does not contain spatial information.

Alternatively, for ”pixel” observations, we employ a convolutional neural network (CNN) architecture that accepts the environment’s raw pixel values as input. Since they can learn hierarchical representations of the input by convolving filters over the input image, CNNs have particular advantages for image-based observations. Two convolutional layers with 3x3 filters and ReLU activation functions are used in our particular architecture, followed by two fully linked layers. The network’s top layer produces a probability distribution over all potential actions.

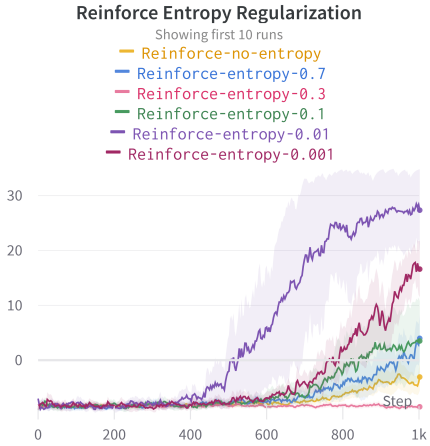
We employ ReLU activation functions in both architectures to add non-linearity and enable the network to learn more complex representations of the input. We employ two fully connected layers, each with 128 units, in the policy network. The first fully connected layer in the CNN architecture consists of 448 units, followed by the first convolutional layer with 16 filters and the second convolutional layer with 32 filters.

4.1.2. Actor-Critic Architecture

As aforementioned, for Actor-Critic we need to train two networks, the Actor and the Critic. In our experiments though, we use one network with two heads. So, the hidden



(a)



(b)

Figure 2. Comparing various hyperparameters for tuning the Reinforce baseline. (a)Reinforce Learning Rate: alter learning rates and add scheduler (b)Reinforce Entropy Regularization: modifying the entropy weight

layers are the same for both parts but they differ in the output which for the actor has a size of three(3), giving the probabilities for each of the three possible moves: right, left, and stay idle, and for the critic has a size of one(1) which is the state’s value after following those actions.

So the network is built as follows: When the observation type is set to ”pixel”, we use two convolutional layers and two linear layers. If we work with ”vector”, then we just use four linear layers. For any case, the activation function is ”relu” and the optimizer is ”Adam”. However, a flattening is needed after the convolutional layers and before the linear layers in the case of the ”pixel” type. Also, the actor’s head exports the probabilities of the three actions after using the softmax function. The exact sizes of the hidden layers as well as other parameters like the learning rate will be shown and discussed in the following subsection.

4.1.3. HYPERPARAMETER TUNING

After running several experiments with different values of the learning rate, gamma, entropy strength, depth of bootstrapping and the use or not of a decay parameter that decreases the learning rate steadily, we concluded the following values for our parameters as presented in *Table 1* and *Table 2*. In *Figure 2* and *Figure 3*, you can see some of the experiments for tuning the main parameters of the algorithms. Specifically, you can see the tuning of the learning rate and entropy regularization for Actor-Critic and Reinforce, and Bootstrapping depth for Actor-Critic. For the learning rate, we also examine if the lr_scheduler helps increase performance, which essentially decreases the learning rate over time.

Actor-Critic Hyperparameters	
lr	1e-2
gamma	0.85
n_steps	100
entropy_weight	0.01
lr_decay	0.99
lr_scheduler	True
num_episodes	1000

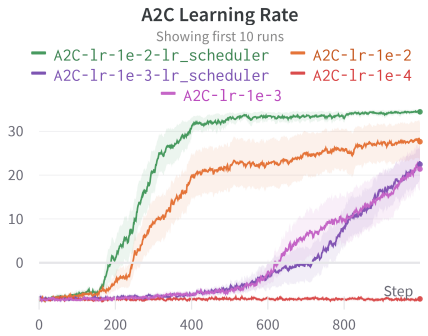
Table 1. Hyperparameters for Actor-Critic algorithm.

REINFORCE Hyperparameters	
lr	1e-3
gamma	0.85
beta	0.01
lr_decay	0.99
lr_scheduler	True
num_episodes	1000

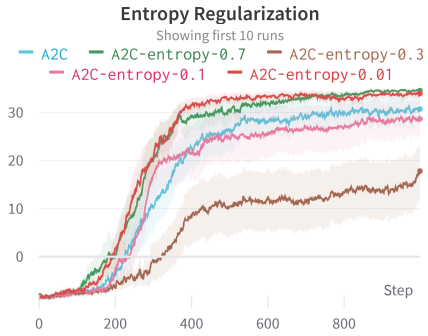
Table 2. Hyperparameters for REINFORCE algorithm.

For ***Actor-Critic***, the learning rate (lr) is set to 1e-2 and we found that decreasing it over time (scheduler) with a standard decay parameter helps its performance. Also, the depth of bootstrapping (n_steps) is found to give better results when set to 100 while lower depths were giving a low performance. However, as shown by our experiments for when the depth is set to 200 it starts performing relatively bad.

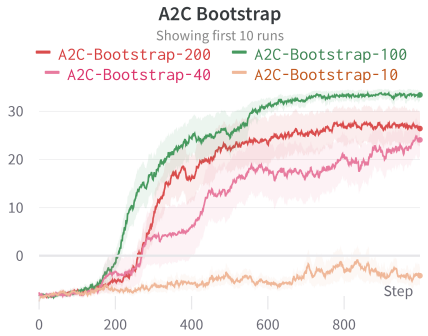
For ***REINFORCE***, the learning rate (lr) is set to 1e-3 and we found that decreasing it over time (scheduler) with a standard decay parameter helps the performance of the algorithm same as the actor-critic. We set beta to 0.01 which represents the entropy weight to control the strength of the entropy regularization as mentioned in the previous section. The entropy regularization was vital in order to find the best agent for the Reinforce algorithm. As *Figure 2(b)* shows, specifically the orange line, reinforce algorithm with no



(a)



(b)



(c)

Figure 3. Comparing various hyperparameters for tuning the baseline. (a)A2C Learning Rate: alter learning rates and add scheduler (b)Entropy Regularization: modifying the entropy weight (c)A2C Bootstrap: Experimenting with different number of steps for bootstrapping

entropy regularization has really bad results.

5. Results & Discussion

5.1. Part I - Find the best agent

5.1.1. REINFORCE vs Actor-Critic

As explained before, we run experiments using the networks for REINFORCE and Actor-Critic (A2C) and from *Figure 4*, we can come to the conclusion that Actor-Critic without bootstrapping or baseline subtraction performs better than the other algorithms. Actor-Critic with bootstrapping and baseline subtraction performs almost as well as the vanilla A2C, although it shows some instabilities and fluctuations. You can, also, observe that A2C with only bootstrapping is performing much worse than all the other variations of A2C. The reason behind this could be the depth of bootstrapping which in our case was set as 100, which although it gives better results, has a higher standard error. On the other hand, REINFORCE appears to be the worst agent as it does not even reach the maximum number of possible rewards for this environment which is 35, in 1000 epochs. It learns slower and does not converging to the optimum. It also suffers from high variance as shown in *Figure 4*. Therefore, the next section of our experiments will be performed using A2C without bootstrapping or baseline subtraction.

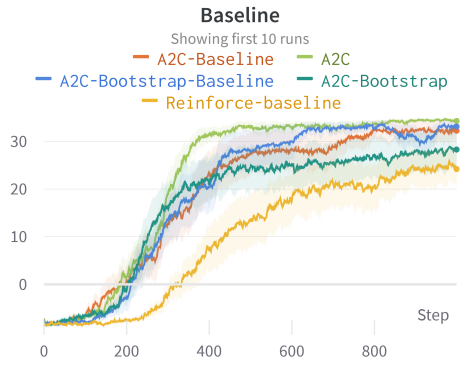


Figure 4. Comparisons between five different algorithms. The main algorithms are **REINFORCE** and **Actor-Critic (A2C)** but also some variations: *A2C-Bootstrap* which stands for Actor-Critic using bootstrapping, *A2C-Baseline* for A2C using baseline subtraction and *A2C-Bootstrap-Baseline* which uses both techniques. A2C performs the best and REINFORCE the worst.

5.2. Part II - Modify the environment

Experiments with varying environment factors can have different maximum and minimum rewards. So in order to understand them visually on a graph we decided all experiments that have changes in the environment variables, have their rewards normalized between a range of -1 to +1.

For our Actor-Critic architecture, we utilize a convolutional layer as the input layer, when we have the observation type as pixel. So the architecture works with square environments and does not support rectangular environment size. To overcome this issue we decided to add padding to rectangular-shaped environments, with a padding value of zero.

All experiments are performed using Wandb (Biewald, 2020) to log and make experimentation easier and Pytorch is utilized for building all the neural networks (Paszke et al., 2019).

5.2.1. Changing the size of the environment

The default experiments were run on an environment with rows and columns equal to 7. For this experiment, we see how our baseline performs when encountering changes in the size of the environment.

From *Figure 5* we can see as expected for the default environment, we continue to perform well and converge to the optimal reward. But when we increase the size of the environment to 10X10 we see a drop in performance and also gives an average maximum reward of 0. Similar behavior can be noticed for 25X25 environment size as we never achieve a positive reward. We believe this to be due to the current model architecture designed for default environment size and does not scale well. Further experiments can be done to increase the depth of the network as well as the hidden nodes in order to find the best parameters.

Now looking at rectangular environments, we see that 14X7 barely achieves any positive rewards and 7X14 never learns anything due to the width being bigger than the height. Whereas on the other hand for an environment size of 14X7 we see decent performance as it gets a maximum average reward of 0.5 at the end of 1000 epochs. Again we believe this to be due to our choice of architecture and additional padding added for rectangular environments.

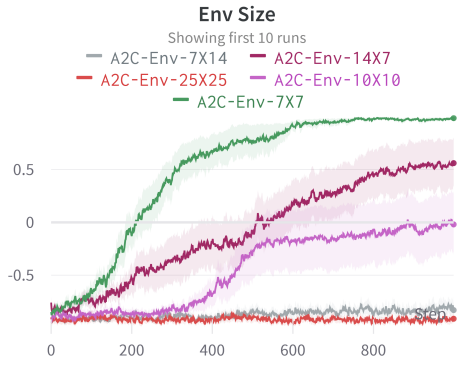


Figure 5. Changing Environment Size for A2C baseline model to see its effect on the total reward obtained.

5.2.2. Changing the speed of the falling ball

The default speed of the ball for all experiments so far was 1.0. For this experiment, we alter the speed of the ball to see its impact on the performance of *A2C*.

From *Figure 6*, we can see that for a speed of 5 *A2C* does not get any rewards and this is because it cannot intercept the ball at high speeds. For a speed value of 2.0, it performs relatively well but does not converge to the optimal reward in 1000 epochs. We believe this to be due to higher speeds and a lesser probability of interception of the ball. Finally looking at a speed of 0.5 we see a lower performance than at speed 1. We believe this happens due to lesser rewards obtained over the same number of maximum steps which reduces the agent's capability to learn. If we were to increase the number of epochs, we expect to see an increase in the overall reward obtained, but cannot guarantee the optimal reward.

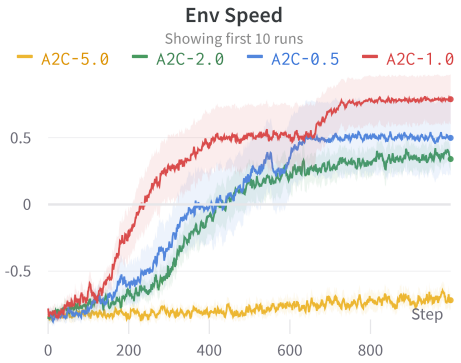


Figure 6. Comparison between different speeds at which the ball drops in the environment. All experiments are run on *A2C* for the default environment parameters apart from the speed.

5.2.3. Changing the observation type: pixel vs vector

As mentioned in the section about the Environment, apart from size and speed, there are other configurations that we can modify and examine the performance of our agent. One of the main configurations is the observation type. We have two options:

- **Pixel:** Pixel is set as the default observation type and it is a binary two-channel array that "observes" and saves the locations of the paddle in the first channel and the ball's locations in the second one.
- **Vector:** Vector keeps the xy-coordinates of the paddle's locations and the ball's location when reaching the lowest level.

As shown in *Figure 7*, the agent appears to learn faster and, eventually, to achieve a slightly higher reward when using vectors as the observation type while with "pixel", it seems to need more steps to converge. This can be caused by the fact that Catch is a relatively easier environment and numerical representations of the observation space are a simple way that helps the agent to capture the relationship between its actions and the resulting state transitions faster.

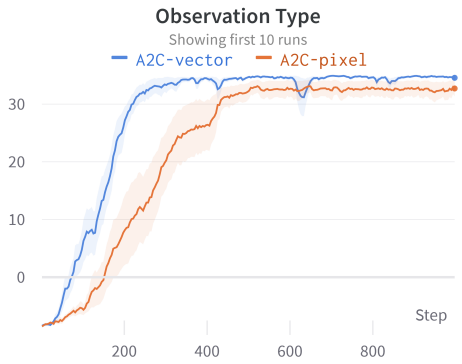


Figure 7. Comparing how A2C performs when modifying the observation type of the environment. When using "vector", the agent learns faster and achieves higher rewards.

5.2.4. Environment variations

For the final experiment, we chose to combine at least 2 variations in environment configurations and test its outcome. From *Figure 8* we can see the different variations that we implemented.

First looking at an environment with maximum misses 20 and max steps 300, it performs the best among all variations. This was an expected result as the steps and maximum misses increase the agent now sees more rewards and has a longer time to learn from its mistakes.

Next, we ran two experiments to compare environment size 14X7 with a speed of 0.5 on both vector and pixel observation types. The vector environment reaches the optimal reward well under 1000 epochs, whereas the pixel environment struggles to obtain positive rewards till the end. we believe this could be due to the padding that we chose to add to rectangular environments and due to the sparse rewards obtained in an environment with a speed of 0.5. Additionally, we can see that rectangular environment perform better with vector observations as seen in *Figure 5*.

Then we ran experiments on the environment size 7X7 for vector observations modifying the speed to 0.5 and 2.0. For an environment with a speed of 0.5, it performs really well and converges to the optimal reward. Whereas, when speed is 2.0 the agent barely learns to obtain positive rewards. This was an expected result as when we represent the environ-

ment in vector observation type for speed 2.0, we have two balls that the paddle can catch one after the other. This can be easily represented in pixel observations, but in vector observation due to the fixed size of the state, the agent only sees the coordinate of the lowest ball, and cannot anticipate the second ball's position. This leads to a surprise negative rewards for the agent and it cannot learn any further.

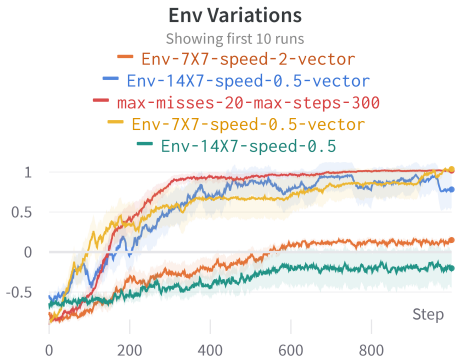


Figure 8. Comparing how A2C performs when modifying different environment variables like speed, size, max steps and max misses.

6. Conclusion

To sum up, we have worked with the Catch environment in order to evaluate REINFORCE and Actor-Critic algorithms as well as some variations of Actor-Critic. We compared their performance and came to a conclusion that REINFORCE indeed suffers from high variance which can be overcome by utilizing Actor-Critic instead. Bootstrapping and Baseline subtraction are two techniques used in Actor-Critic aiming to stabilize the learning process and reduce the variance but from our experiments, we understand that this does not necessarily hold in our case. Some reasons behind this could be the specific environment that we are working with which is relatively simple so the simpler version of the A2C algorithm performs well without any more complexity needed or maybe the bootstrapping depth should be lower to reduce variance. Additionally, we believe our hyperparameter tuning to have played huge role in getting high performance from the vanilla Actor-Critic algorithm.

Comparing the two techniques: bootstrapping and baseline subtraction, we can say that the learning process using bootstrapping is more stable however, using baseline convergence to the optimal reward is faster. Moreover, we have examined the effect of the entropy regularization factor which appears to play a major role in the performance as its presence increases the reward and makes the learning process faster if its strength is tuned optimally.

Finally we have run multiple experiments using A2C on different configurations of the environment coming to the main conclusion that the agent performs poorly in very big environments due to our current architecture tuned for the default environment. Rectangular environments, pose a harder challenge as the agent learns slower. Finally, other properties like speed or observation type can also affect the agent's performance, and depending on the combinations we might get different results.

7. Contribution of the team members

All the team members have worked together for the best results in this Assignment. For the first part of the experiments, Kyriakos Aristidou worked on REINFORCE algorithm, and Antonia Christodoulou and Shreyansh Sharma worked together on Actor-Critic. Shreyansh Sharma worked on the "experiments.py" file while all three of us added different types of experiments on the "experiments_config.py" file. We, all, ran experiments and discussed together the results so that we could contribute to writing the final report.

References

Biewald, L. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

Face, H. Deep reinforcement learning: Advantage actor-critic, February 2018. URL https://huggingface.co/blog/deep-rl-a2c.

Moerland, T. Adapted from deepmind bsuite.

Moerland, T. Continuous markov decision processes. *arXiv preprint arXiv:2104.05309*, 2021.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance.pdf.

Plaat, A. Deep reinforcement learning, a textbook, Jun 2022.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. MIT press, 2nd edition edition, 2018. URL https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.