# Policy-based RL algorithms: REINFORCE, Actor-Critic and Clip PPO Reinforcement Learning - Assignment 3

**Ioannis Koutalios (s3365530)**[1] **Eduard R. Munne (s3687988)**[1] **Maria F. Pinar (s3289850)**[1]

## 1. Introduction

In this assignment, we will study the policy-based approach in Reinforcement Learning (RL). Both algorithms, REINFORCE and Actor-Critic, use this approach to learn to perform a task. In this case, the algorithms will learn to play paddle using the Catch environment. The main objective of the assignment is to study different policy gradient techniques.

We will be using the adapted Catch environment (Osband et al., 2020). It is composed of columns and rows (set as 0) and the paddle and the balls (set as 1). The agent moves the paddle to catch the falling balls. There are three different discrete actions for the paddle: stay, move left, and move right. The environment is adjustable, so the parameters can be changed to increase the difficulty for the agent to catch the balls.

The principal difference between value-based and policy-based algorithms is the way the policy is learned. In the policy-based approach, the policy is learned directly, without learning the value function. This method has some advantages like the allowance of continuous actions or the learning of stochastic policies. One of the algorithms we will be using is the REINFORCE algorithm. The policy is represented as the parameters (weights) of a neural network. These parameters are updated using gradient ascent in the direction of the better action.

The other algorithm we will use is the Actor-Critic algorithm, which combines value-based and policy-based techniques. This combination is done by the use of two neural networks, the actor which predicts the actions, and the critic which predicts the Q-values. Compared to REINFORCE, this method has the advantage of presenting low variance and low bias (Plaat, 2022), improving the performance and stability of the model.

In Section 2 we explain in detail the different algorithms used. Then, in Section 3, we present the results of the experiments. Finally, in Section 4 we interpret the results.

[1]Leiden University, P.O. Box 9513, 2300 RA Leiden, The Netherlands.

3rd Assignment for the course of Reinforcement Learning

## 2. Methods

### 2.1. REINFORCE

The Monte Carlo Policy Gradient (REINFORCE) is one of the most widely used policy-based algorithms. This approach works by updating the weights of the policy in the direction of higher reward, without explicitly calculating the value for each state-action pair. This results in an increase in the probability of those actions that have returned a higher reward.

In REINFORCE we define the performance objective $J(\theta)$ that we aim to maximize with a gradient ascend approach. Our objective function can be defined as: $J(\theta) = \mathbb{E}_{\pi_\theta}[R]$.

Since the $\mathbb{E}_{\pi_\theta}[R]$ is the expectation for the reward by using our policy $\pi_\theta$ and selecting. Since $R$ is a constant that does not depend on the parameters $\theta$ of the model, we use the trick of the logarithm of a derivative:

$$\nabla_\theta \log f(\theta) = \frac{\nabla_\theta f(\theta)}{f(\theta)} \tag{1}$$

As a result, after performing some mathematics (Moerland, 2021) on $J(\theta)$ we end up having:

$$\nabla_\theta \, \mathbb{E}_{\pi_\theta} \left[R(S,A)\right] = \mathbb{E}_{\pi_\theta} \left[R(S,A) \cdot \nabla_\theta \log \pi_\theta(A|S)\right] \tag{2}$$

Where we introduce a parameter that depends on the weights inside the gradient. As a result, we push the derivative inside the expectation and therefore, the effect is the push-up of the density function with a magnitude $R$. When higher is $R$ harder is the push-up of the estimated probabilities. Considering that in our problem $R$ is the discounted reward, the gradient is pushed in the direction of higher reward. Since we aim to maximize this value, we implement a gradient ascend method.

In this algorithm, we use a discounted reward since we are working with an episodic task. This is because rewards at first timesteps have more influence than future rewards and by learning these actions we can make our model learn faster (Dewanto & Gallagher, 2021). As a result, rewards obtained in further actions have a lower impact on the policy's update.

**Algorithm 1** REINFORCE algorithm (Plaat, 2022)

**Input:** A differentiable policy $\pi_\theta(a|s)$, parametrized by $\theta \in \mathbb{R}^d$, learning rate $\alpha$, number of epochs $E$ and the batch size $M$, discount factor $\gamma$.
**Initialization:** A policy network with randomly initialized $\theta \in \mathbb{R}^d$
**for** $e \in 1, \dots, E$ **do**
  grad $\leftarrow 0$
  **for** $m \in 1, \dots, M$ **do**
    Generate a trace $h_0 = \{s_0, a_0, r_0, s_1, \dots, s_T\}$ with policy $\pi_\theta(a|s)$
    $R \leftarrow 0$
    **for** $t \in T-1, \dots, 1, 0$ **do**
      $R \leftarrow r_t + \gamma \cdot R$
      grad $+= R \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)$
    **end for**
  **end for**
  $\theta \leftarrow \theta + \alpha \cdot$ grad
**end for**
**return** $\pi_\theta(a|s)$

The algorithm (see Algorithm 1) works as follows. We generate $E$ epochs which will allow our model to converge. Each epoch consists of $M$ traces (i.e. batch size). This allows us to reduce the variance of the updated weights by taking the average of $M$ episodes. For each trace $m \in 0, \dots, M$ we start sampling a trace by following the policy $\pi_\theta(a|s)$. The discounted reward at each timestep $t$ is measured, and we push the weights of the policy to the direction of the gradient. When higher is the discounted reward, harder we will push the gradient. This emphasizes actions with high rewards to have a higher probability of being taken, allowing the model to learn. At each time step, we compute the gradient for each of the $M$ traces, where each trace consists of $T-1$ steps. We then accumulate the gradients across all traces and update the total gradient with a learning rate of $\alpha$. We do not normalize (i.e. divide by the batch size $M$) since we want to avoid higher batch sizes requiring more episodes to learn. Finally, we update the weights of the policy model $\theta$ by taking a step in the direction of the gradient.

### 2.2. Actor-Critic

The Actor-Critic algorithm, as suggested by its name, consists of two parts: the actor and the critic. The first is responsible for choosing actions based on the current state of the environment, while the latter evaluates the quality of the actor's actions. The policy function outputs a probability distribution over possible actions, and the actor selects an action according to this distribution. Since we are working with a discrete set of actions we will have a probability mass function. On the other hand, the critic tries to learn the value function for a given state. The critic's output is used to update the policy function in order to guide the actor towards actions that result in a higher expected reward. This approach essentially combines the advantages of a policy-based (actor) with value-based learning (critic).

Policy-based methods can suffer from high variance that can arise from two sources that we need to combat. The first source of variance comes from the cumulative reward estimate. We implement bootstrapping to mitigate it. We only take into consideration the rewards from a fixed number of consecutive steps (denoted as $n$) to calculate the target:

$$Q_n(s_t, a_t) = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_\phi(s_{t+n}) \tag{3}$$

We then update the value function using a squared loss:

$$L(\phi|s_t, a_t) = (Q_n(s_t, a_t) - V_\phi(s_t))^2 \tag{4}$$

while the policy is updated by gradient ascend:

$$\boldsymbol{\nabla}_\theta = Q_n(s_t, a_t) \cdot \boldsymbol{\nabla}_\theta \log \pi_\theta(a_t|s_t) \tag{5}$$

The variance from gradient estimates can be countered using the technique known as baseline subtraction. We calculate the advantage function by subtracting the value function from the value estimate $V_\phi$ (output from the critic model) as follows:

$$A_n(s_t, a_t) = Q_n(s_t, a_t) - V_\phi(s_t) \tag{6}$$

we then use the advantage function in the same way we would use $Q_n$ to calculate the squared loss and the policy gradient when updating our networks.

The full algorithm with bootstrapping and baseline subtraction can be seen in Algorithm 2. Our implementation was done in a way that we can independently switch on and off both the bootstrapping and the baseline subtraction, so we can analyze the impact of each one individually and combined (Section 3.2.1). The algorithm works as follows. We generate $E$ epochs, which consist of $M$ traces (i.e. episodes). For each trace, we start sampling the trace following the policy $\pi_\theta(a|s)$. Then, we calculate the target and the advantage using Equations (3) and (6) respectively. After each episode, we calculate the loss and update the weights of both networks using Equations (4) and (5) after the $M$ traces, with the substitution of $A_n$ instead of $Q_n$ as it is written in Algorithm 2.

**Algorithm 2** Actor-Critic algorithm (Plaat, 2022)

**Input:** A policy $\pi_\theta(a|s)$, parametrized by $\theta \in \mathbb{R}^d$, value function $V_\phi(s)$, learning rate $\alpha$, number of epochs $E$ and the number of traces $M$, depth $n$, discount factor $\gamma$.
**Initialization:** An actor network with randomly initialized $\theta \in \mathbb{R}^d$ and a critic network with randomly initialized $\phi \in \mathbb{R}^d$
**for** $e \in 1, \dots, E$ **do**
  **for** $m \in 1, \dots, M$ **do**
    Generate a trace $h_0 = \{s_0, a_0, r_0, s_1, \dots, s_T\}$ with policy $\pi_\theta(a|s)$
    **for** $t \in 0, \dots, T-1$ **do**
      $Q_n(s_t, a_t) = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_\phi(s_{t+n})$
      $A_n(s_t, a_t) = Q_n(s_t, a_t) - V_\phi(s_t)$
    **end for**
  **end for**
  $\phi \leftarrow \phi - \alpha \cdot \nabla_\phi \Sigma_m \Sigma_t [A_n(s_t, a_t)]$
  $\theta \leftarrow \theta + \alpha \cdot \Sigma_m \Sigma_t [A_n(s_t, a_t) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)]$
**end for**
**return** $\pi_\theta(a|s)$

### 2.3. Entropy exploration

Both methods can easily get stuck in a local minimum if they exploit certain actions (high probability for a certain action). To deal with this problem, a certain degree of exploration is required.

We implement the entropy regularization as an exploration method for both the REINFORCE and Actor-Critic algorithms. This method favors exploration by applying the entropy regularization approach. An additional term is added to the loss function. This ensures that the entropy stays greater (i.e. avoid a peaked distribution on one action, widening the distribution) (Plaat, 2022). As a result, the model becomes more stable and avoids the policy to collapse. The policy update equation for the REINFORCE and the actor model in the Actor-Critic then becomes:

$$\theta_{t+1} = \theta_t + R \cdot \nabla_\theta \log \pi_\theta(a_t|s_t) + \eta \nabla_\theta H[\pi_\theta(.|s_t)] \quad (7)$$

where $\eta \in \mathbb{R}$ is a regularization parameter that defines the degree of exploration and $H[\pi_\theta(a|s)]$ is the entropy of the policy, calculated as:

$$H[\pi_\theta(.|s_t)] = -\sum_i p_i \cdot log(p_i) \quad (8)$$

Where $p_i$ is the probability of selecting action $i$ at state $s_t$. In the Actor-Critic method, the entropy is only added to the actor loss, since it is the model that learns the policy and computes the selected action.

## 3. Results

We manually selected the optimal hyperparameters based on the best model performance. We realized that both methods were very sensitive to hyperparameter tuning, especially for the learning rate. After running different experiments and tuning all hyperparameters, we found out that the ones that worked best for the two approaches are the ones displayed at Table 1.

| Hyperparameter | REINFORCE | A/C |
|---|---|---|
| Batch size | 10 | 4 |
| $n$ steps | - | 1 |
| Learning rate | 0.005 | 0.005 / 0.05 |
| Entropy regularization $\eta$ | 0.1 | 0.01 |

*Table 1.* Manually selected hyperparameters for REINFORCE and Actor-Critic algorithms that improve the model's performance and enable evaluation of the best action selection method using fixed values. The learning rate of both the actor (first) and the critic (second) are separated by a slash line (as stated in the header A/C).

To account for randomness, all plots represent the average of 5 runs with a smoothing of window size of 51 applied. The shaded area indicates the standard deviation at each step across the five runs.

### 3.1. REINFORCE

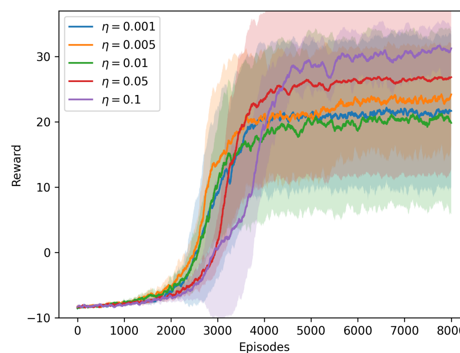


*Figure 1.* Comparison of the reward over episodes for the REINFORCE with different values for the entropy regularization parameter $\eta$. Better performance for higher values of $\eta$, reaching the best output with $\eta = 0.1$.

Our initial evaluation was done on the REINFORCE algorithm. After an extensive study of the impact of different hyperparameters on the algorithm's performance, we settled on the values listed in Table 1.

However, we also recognized the importance of exploration in achieving optimal performance. To address this, we tuned the entropy parameter $\eta$ to adjust the level of exploration. Specifically, increasing $\eta$ resulted in higher exploration,

165 while decreasing $\eta$ reduced exploration. Figure 1 illustrates that the optimal performance for REINFORCE was achieved with an entropy parameter value of $\eta = 0.1$. This value allowed the algorithm to avoid collapsing into a local minimum and enabled it to learn by widening the probability distribution. However, despite achieving good results, the algorithm still suffered from high variance, which is a known limitation of REINFORCE. The results also showed that increasing the exploration rate by raising $\eta$ leads to better performance. This has the cost of taking longer to reach a stable plateau due to the low exploitation of the optimal policy.

Nevertheless, the high variance in the results means that the performance of REINFORCE is significantly dependent on the run, and requires many episodes to reach the maximum score. This limitation emphasizes the need for a more stable and efficient algorithm, such as the Actor-Critic method.

### 3.2. Actor-Critic

The Actor-Critic algorithm addresses the limitations of REINFORCE by combining a value-based approach with policy-based learning and additionally incorporates techniques such as bootstrapping and baseline subtraction to enhance its performance. As a result, this method overcame REINFORCE in terms of efficiency.

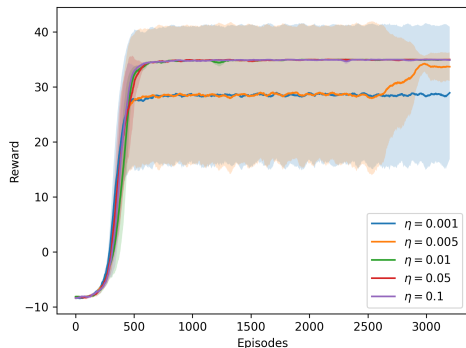


*Figure 2.* Comparing the reward over episodes for the Actor-Critic with different values for the entropy regularization parameter $\eta$. The performance is worse for values $\eta < 0.01$. For higher values the performance does not vary.

Similar to our approach with REINFORCE (as described in Section 3.1), we manually tuned the hyperparameters of the Actor-Critic algorithm to optimize its performance. Since the actor's learning relies on the accuracy of the value prediction provided by the critic, the two models are dependent on each other. Accordingly, we set a higher learning rate for the critic ($\alpha_c = 0.05$) than for the actor ($\alpha_a = 0.005$). This strategy enabled the critic to learn more quickly, thereby improving the accuracy of the actor's policy learning.

In order to determine the optimal level of exploration, we performed regularization tuning by setting different values for the entropy parameter $\eta$. As shown in Figure 2, low values of $\eta$ (0.001 and 0.005) caused the agent to suffer overfitting, leading to results where the agent got trapped in a local minimum with high variance (as indicated by the high standard deviation of results). However, values $\eta \geq 0.01$ enabled the agent to achieve the maximum reward in all cases.

#### 3.2.1. Ablation Study

Finally, we studied the performance of the Actor-Critic by deactivating some parts of the model (bootstrapping and baseline subtraction) and we also compared it with the REINFORCE method. The obtained results are displayed in Figure 3.

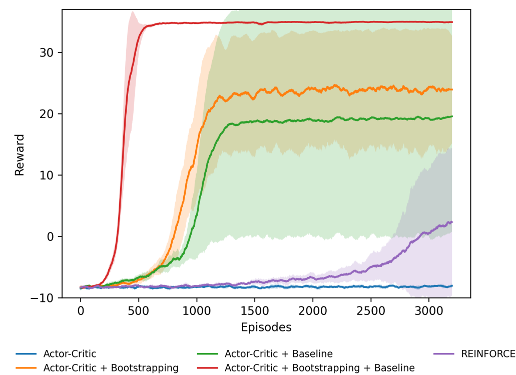


*Figure 3.* Ablation study comparing the performance of the Actor-Critic with four different configurations, adding and removing bootstrapping and baseline subtraction and comparing with REINFORCE. The best performance is observed when bootstrapping and baseline subtraction are included. Adversely, when both techniques are removed, the algorithm do not learn.

The performance of the Actor-Critic model with the best hyperparameter configuration was evaluated by removing certain components, such as bootstrapping or baseline subtraction, and compared with the best-performing REINFORCE algorithm. As shown in Figure 3, the performance of the different models was significantly distinct. The Actor-Critic model with both bootstrapping and baseline subtraction significantly outperformed the rest of the models, exhibiting high stability and reaching the maximum reward after only 500 episodes. Removing either of these features introduced high variance into the model, which is a common problem for policy-based methods. Both bootstrapping and baseline subtraction are designed to address this issue, and their combination proved to be much more effective than either of them alone. While the Actor-Critic model with bootstrapping outperformed the one with baseline subtraction and was slightly more stable, both models performed well

overall and outperformed the other models in the ablation study.

The absence of both components in the Actor-Critic makes the learning process ineffective. There could be various reasons behind this, such as overfitting of the hyperparameters to the complete Actor-Critic (the one including both components) during tuning or the insufficient amount of training episodes. However, with correct hyperparameter tuning and additional training episodes, it is expected that the model can learn effectively. In comparison, REINFORCE performs significantly worse than Actor-Critic. Although it can reach the maximum reward with $\eta = 0.1$ (Figure 1), it requires more than 4,000 episodes to achieve this and exhibits higher variance than the complete Actor-Critic. In Section 4, we will discuss potential reasons for this outcome.

### 3.3. Environment Variation

We wanted to explore if our agent is able to learn different variations of the default environment. We chose to work with the Actor-Critic model, with both bootstrapping and baseline subtraction as it was the best-performing model as we discussed in Section 3.2.1.

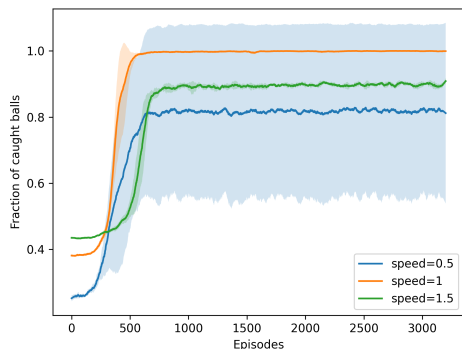


*Figure 4.* Comparing Actor-Critic's performance when varying the speed of the falling balls on the environment. The agent suffered from instability during training when a small speed was used. For high speed although the agent learned the optimal policy, it was impossible to catch all the balls because of their relative distance.

In our first experiment, we varied the speed of dropping new balls. The default value of 1 means that each new ball drops when the previous one reaches the bottom. By changing the value of the speed we can manipulate the environment to have an interval between the two balls with a value $\mathrm{rows} \,//\, \mathrm{speed}$ where the double $/$ is the notation for the floor division. One issue arises from the fact that by varying the speed, we have more or less total number of balls dropped for each episode, which means that the maximum reward also changes. One way of solving this issue and making the results comparable is to normalize the rewards after the training by dividing them by the total number of balls. We can, then, safely compare the values of the fraction of caught balls that we calculated for each different variation.

We trained our agent in the environment with speeds $[0.5,\ 1.0,\ 1.5]$, the total number of balls for each one was at $[18,\ 35,\ 62]$. The fraction of caught balls at each training stage can be seen at Figure 4. As we can see our agent had the best performance in the default environment. Smaller speed led to instability during training. In some instances, the agent was able to learn the optimal policy, while in some cases plateaued at lower rewards. One possible explanation for this is the smaller number of events per episode did not allow our agent to effectively train. The hyperparameters also were tuned for the default environment, which might also lead to underperformance in different variations. For the increased speed we had great stability, and our agent was able to learn the optimal policy. The reason the ratio of caught balls was smaller than 1 doesn't have to do with any underperformance from the agent. Sometimes balls dropped one after the other were too far away to get caught even while following the optimal policy.

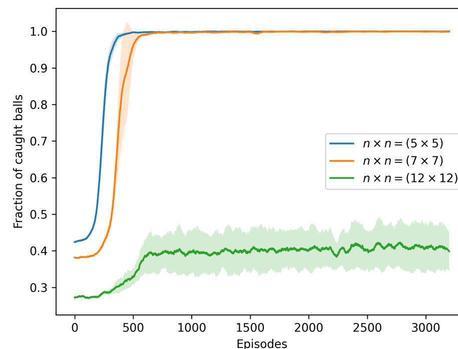


*Figure 5.* Comparing the performance of the Actor-Critic algorithm when varying the size of the environment. Grids with smaller sizes yielded better results. The agent was unable to learn the optimal policy for the biggest grid size.

We also experimented with different sizes of the environment. As in the previous experiment, this also affected the total number of balls dropped in each episode, so we decided to compare the ratio of balls caught to the total number. The default size for our environment is $7 \times 7$, in which 35 balls are being dropped. We varied it to $5 \times 5$ (50 balls) and $12 \times 12$ (20 balls). The results are shown in Figure 5. As we can see for the biggest grid, our agent was unable to learn the optimal policy. Our network was designed for smaller grids and more events per episode, which lead to this underperformance. For the other two grid sizes, our agent learned the optimal policy. The only improvement that came from the smallest grid size, was the fewer episodes which were needed for the peak performance.

We then switched to a non-square grid. We chose a grid of size $14 \times 7$ and also varied the speed of dropping new balls. For consistency, we once again compare the ratio of caught balls to the total number of balls dropped. The results are shown in Figure 6. In the same plot, we also have the results for the default grid size ($7 \times 7$), for the different values of speed. As we can see our agent was able to learn in the non-symmetric grid and only had issues when the value of speed was low. Compared to the default grid size, the small value of speed did not affect our agent that much. For the big value of speed, our agent was once again able to learn the optimal policy. This time it reached the fraction of caught balls of 1, because, unlike the default grid size, it was able to catch consecutive balls dropped on opposite sides of the grid, because their horizontal distance was bigger this time.

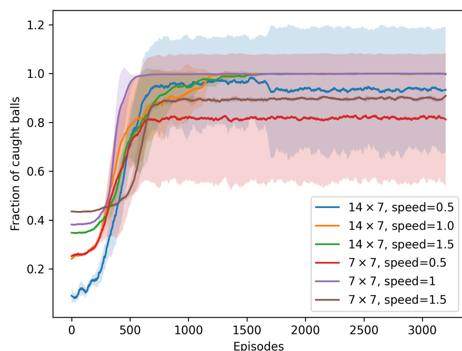


*Figure 6.* The performance of the Actor-Critic algorithm in the non-squared grid size for different speeds, compared with default squared grid. The agent was able to perform at the same levels as in the default environment.

Finally, we wanted to see if our network is able to train while using a different input representation. The observation array in the default environment is twice the size of the grid, with one hot indicator for each ball that is present in the second channel and one hot indicator for the paddle location in the first channel. This type of observation type is called “pixel”. We switched to the “vector” representation that uses an array of length 3 and stores the x-location of the paddle, and the x and y location of the ball. Due to this length limit, this observation type was unable to handle multiple balls being present at the same time. Therefore, we couldn’t use speed $> 1$, because it would lead to multiple balls on the grid. In Figure 7 we see the results of training using the “vector” observation type, compared to the results we got for the default “pixel” type. As it becomes clear the new representation leads to higher instability. Our agent was able to learn the optimal policy in some of the runs, while in some of them, it got stuck in worse performance. This instability has to do with the observation type being inept compared to the default one. Our network was, also, developed for the default environment, meaning some hyperparameters and network architecture, are not so well-tuned for this change.

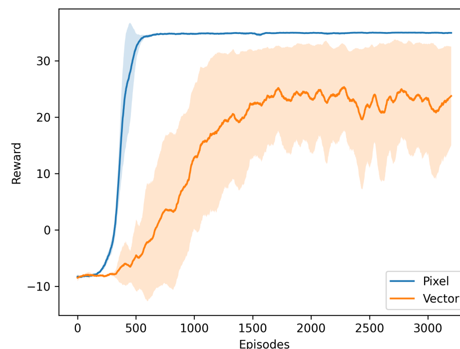


*Figure 7.* The performance of the Actor-Critic for the “vector” observation type compared to the default “pixel” type. The new representation has more instability while training.

### 3.4. Model evaluation

We studied the resulting models by doing a model evaluation after the training process. First, we trained each of the algorithms with the best hyperparameters (Table 1). Then, we saved each model. In the case of the Actor-Critic, the actor was used as the model. Finally, the agent was tested by running without learning. Due to the stochastic nature of the algorithms, selection action policy can not be completely exploitative. To minimize the exploration we used $\eta = 0$. We performed 1,000 evaluations for both algorithms and measured the returned rewards. For REINFORCE, the mean reward was 32.22 and the standard deviation was 2.31. For the Actor-critic the mean reward was 34.96 and the standard deviation was 0.29. These results can slightly change due to the stochastic nature of both algorithms.

## 4. Discussion

### 4.1. REINFORCE issues

As shown in Section 3, the REINFORCE algorithm suffers from high variance in its results. This variance can be attributed to two main factors. The first factor is that positive rewards will always push the weights in the direction of those actions. To counteract this, it is necessary to normalize different traces so that worse traces push down the weights, even if the discounted reward is positive. This is achieved through baseline subtraction, which normalizes the results by measuring the advantage of an action with respect to the mean. The second factor is related to the high variance in the estimate of the cumulative reward. This is because, in the initial epochs, many actions are chosen randomly, which can lead to a wide range of outcomes. The Actor-Critic algorithm solves both of the aforementioned problems through the use of a second value-based model.

330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384

The predicted value is then used to address the two main sources of variance in REINFORCE.

### 4.2. Actor-Critic vs REINFORCE

As shown in the ablation study (Section 3.2.1) and in the model evaluation (Section 3.4), the Actor-Critic algorithm outperforms REINFORCE in terms of speed and stability of learning. Specifically, Actor-Critic is able to reach the maximum reward after $\sim$ 500 episodes, while REINFORCE requires over 4,000 episodes (Figure 1). It should be noted that one reason for REINFORCE's slower learning speed is the use of a larger batch size, $M = 10$, compared to the batch size of $M = 4$ used in Actor-Critic. This means that the weights are updated after more episodes. The absence of a normalization step pushes changes harder while reducing the variance. As a result, the learning progress is slowed down. We also tested lower batch sizes, but this resulted in an increased variance that made the model even more unstable. Therefore, we decided to stick with $M = 10$ and balance the trade-off between speed and stability.

### 4.3. Entropy regularization

As shown in Figure 1 and 2, the entropy regularization parameter influences the performance of both algorithms. In the REINFORCE algorithm, a better performance and a slower learning is obtained with higher values of $\eta$, namely $\eta = 0.1$. A higher exploration results in slower but better learning of the environment. For the Actor-Critic, the difference between different parameter values is more subtle. For entropy regularization values larger than 0.005, the speed at which the algorithm learns is very similar, and the highest performance is always achieved. A minimum amount of exploration is necessary for the algorithm to learn the environment, but after that point, it behaves more stable when increasing $\eta$.

### 4.4. Environment variation

In Section 3.3 we modified the environment to test if the actor-critic is able to perform and learn the optimal policy. What we observed is that in the general case that is possible, with some limitations. Our agent performed well when we increased the speed of dropping new balls, and also in smaller grid sizes. The performance was also great in non-squared grids. We had some issues with smaller speeds and bigger grid sizes, partly due to fewer events happening in each episode. When we switched to the "vector" observation type, our agent suffered from instability during training.

### 4.5. Conclusions

We can conclude that both REINFORCE and Actor-Critic learn the environment, but differences in performance are observed for both algorithms. REINFORCE is slower and a higher exploration is needed to get the optimal performance. The Actor-Critic algorithm using bootstrapping and baseline is able to learn faster and achieve the optimal performance. The Actor-Critic was also able to perform relatively well in different variations of the default environment.

## A. Bonus

### A.1. Linear annealing

We studied the performance of both algorithms while applying the anneal function. This function varies linearly the entropy regularization parameter from higher to lower, i.e. applying more exploration at the beginning and then more exploitation. This function was applied with two different percentages 20% and 60%, and from an initial value equal to the optimal value (see Table 1) to a final value of $\eta = 0.001$. The resulting performance is shown in Figure 8 and Figure 9. A better performance was obtained for the REINFORCE algorithm when using the anneal function with a 60%. The next best performance was obtained with the entropy regularization parameter $\eta = 0.1$. The Actor-critic's performance was optimum for both methods. However, the optimal performance was obtained faster without using the annealing function.

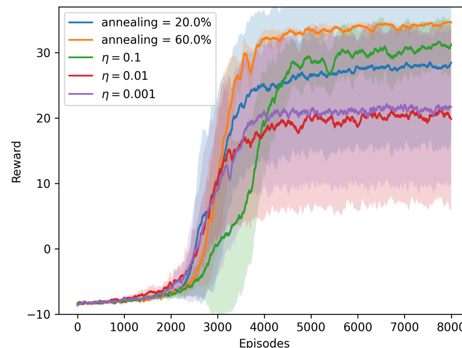


*Figure 8.* Comparing the performance of the REINFORCE algorithm when applying the anneal function with different percentages and different values of the entropy regularization parameter.

### A.2. Clip PPO

Another policy-based method that exhibits good performance is PPO, which includes many versions such as Clip PPO or PPO-Penalty. It consists of a simplified derivation of TRPO with better run time complexity (Plaat, 2022) by doing first-order derivatives instead of second-order.

In this assignment we will focus on Clip PPO, which generates a different loss function to constraint the weights update of the model and reduce the high variability of the

385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439

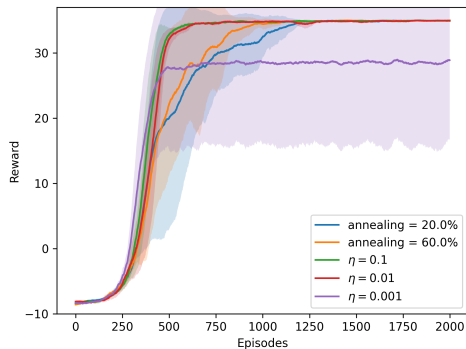


*Figure 9.* Comparing the performance of the Actor-Critic algorithm when applying the anneal function with different percentages and different values of the entropy regularization parameter.

parameter values $\theta$ (Schulman et al., 2017). This algorithm, as the Actor-Critic approach, has two models (actor $\pi_\theta$ and critic $\pi_\phi$). As a result, we define the loss for the actor as:

$$\mathbb{L}(s, a, \theta_{old}, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}A, g(\epsilon, A)\right) \quad (9)$$

where $A$ is the advantage, $\epsilon$ is an hyperparameter that indicates how far the new policy $\pi_\theta$ can go with respect to the old $\pi_{\theta_{old}}$ and $g(\epsilon, A)$ depends on the advantage like:

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & \text{if } A \geq 0 \\ (1-\epsilon)A & \text{if } A < 0 \end{cases} \quad (10)$$

We update the rewards of the actor by computing the loss measured as Equation (9). The critic model learns in the same way as Actor-Critic. We evaluated this model for different parameters of $\epsilon$ which defined how we clipped the parameters. Lower values of $\epsilon$ restricted the update, while high values allowed bigger updates of the parameters $\theta$. Figure 10 shows the performance of the model and the comparison with the Actor-Critic. While the performance of Clip PPO is comparable to the Actor-Critic, it tends to converge to local minima (high variance in the results). Lower values of the clipping parameter $\epsilon$ result in less aggressive updates and thus reduce the likelihood of overfitting. However, this comes at the cost of requiring more episodes to learn the optimal policy.

### A.3. Cartpole environment

For some extra experimentation, we were interested to see how our agent will perform in a different environment. We chose the "Cartpole" environment from the `gym` module of OpenAI (Brockman et al., 2016; openAI, n.d.) and trained using the Actor-Critic model with bootstrapping and baseline subtraction, which was our best-performing model in

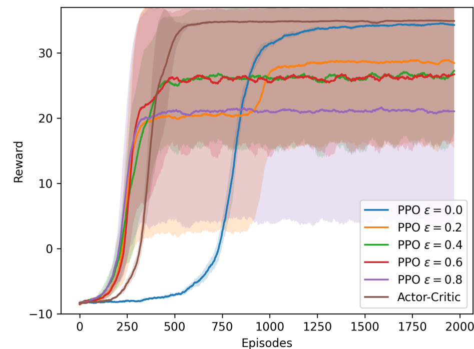


*Figure 10.* Comparing the performance of the Clip PPO algorithm with different values of $\epsilon$ with the same hyperparameters than Actor-Critic (1). Low values of $\epsilon$ lead to better performance.

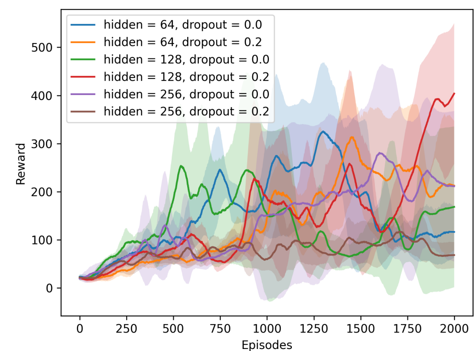


*Figure 11.* The average rewards per episode for the Actor-Critic agent in the cartpole environment. The different lines represent nodes in the hidden layer of both the actor and the critic and a different value for the dropout layer. The performance was not optimal and the agent had instability during training.

the "Catch" environment. While training for this environment we noticed some well-known patterns of overfitting. The rewards per episode could rise to the maximum value (500) and suddenly drop after a few episodes. To prevent overfitting we tried different network architectures, with no significant improvement in the results. We varied the number of nodes in the hidden layer of both the actor and the critic network and also added a "Dropout" layer. The results can be found in Figure 11. As we can see our agents suffered from great instability during training. Of the different architectures that were tried, the most promising one was with 128 nodes in the hidden layer and a dropout value of 0.2. Although we were not able to train our agent to reach the optimal policy, the experiments have shown that it is possible to adapt the same Actor-Critic agent in different environments, with minimal changes. More careful tuning of the different hyperparameters would yield better results.

440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494

## References

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

Dewanto, V. and Gallagher, M. Examining average and discounted reward optimality criteria in reinforcement learning. *CoRR*, abs/2107.01348, 2021. URL https://arxiv.org/abs/2107.01348.

Moerland, T. Continuous markov decision process and policy search (lecture notes), 2021. URL https://thomasmoerland.nl/wp-content/uploads/2021/04/continuous_mdp.pdf.

openAI. Cart pole#, n.d. URL https://www.gymlibrary.dev/environments/classic_control/cart_pole/.

Osband, I., Doron, Y., Hessel, M., Aslanides, J., Sezener, E., Saraiva, A., McKinney, K., Lattimore, T., Szepesvari, C., Singh, S., Roy, B. V., Sutton, R., Silver, D., and Hasselt, H. V. Behaviour suite for reinforcement learning, 2020.

Plaat, A. *Deep Reinforcement Learning*. Springer Nature Singapore, 2022. doi: 10.1007/978-981-19-0638-1.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017.