Policy-based RL - Assignment 3 - Group 13

Nikolaos-Maximos Bilalis, Christos Tsirogiannis, Luca Barbera

Abstract

This report explores the development of an agent that learns to play the Catch game using a policy-based approach. It is shown that the best-performing model is the Actor-Critic model with baseline subtraction and bootstrapping. We prove that baseline subtraction has a greater impact on the performance of the model than bootstrapping. It is shown that by prioritizing the best-sampled trajectories we greatly reduce the variance of the model and consequently create a more stable and well-performing agent.

1. Introduction

In this report, we try to build an agent that will learn to play the Catch game on a policy-based approach. We gradually enhance our agent starting from a simple Monte Carlo model and then adding various stabilizing and improving components. The most significant additions in terms of impact on performance are the introduction of baseline subtraction and bootstrapping. We show that the full Actor-Critic model with baseline subtraction and bootstrapping gives the best results. Following this, we test the generalization of this best-performing agent into different setups of the environment. We prove that our model is able to learn and can generalize sufficiently under different circumstances.

1.1. Environment

In this report, we will use the Catch environment provided by Thomas Moerland, a variation of the original Catch environment in Behavioural Suite (1).

In this variation, the player has to catch the ball which is dropped randomly from the top of the environment. During one step of the environment, the paddle has to move left, right or stay idle and the ball falls down by one row. The reward is $+1$ when the player catches a ball, -1 when a ball is missed and 0 for other steps (ball falling). The game terminates after a certain amount of misses or after a maximum amount of steps. Those parameters are editable and can change the game accordingly. The state of the environment is represented by the position of the paddle, the position of the ball, and the ball's velocity. The environment returns the new state and a reward after each action.

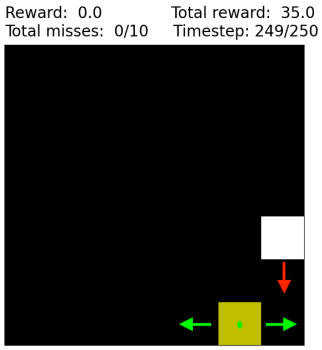

Figure 1. Here the default 7x7 Environment is depicted at the end of the perfect game. The yellow square represents the paddle and the white square represents the ball. The balls are generated randomly from the ceiling (top 7 pixels of the environment). The goal of the game is to catch as many balls as possible by moving the paddle left, right or staying idle.

2. Policy based algorithms

The theoretical groundwork for the following sections is oriented around the fourth chapter of Deep Reinforcement Learning by Aske Plaat (2).

2.1. REINFORCE

In value based methods, a neural network is used to approximate the value of a certain state-action pair on whose basis an external (argmax) policy chooses the best one. Policy based methods on the other hand, directly approximate a probability distribution over the action space. The agent will follow exactly this probability without consulting an external behavioural policy, which means that he also naturally explores the action space.

The first algorithm that will be trained in the Catch environment will be the REINFORCE algorithm, the starting point to policy-based algorithms. The REINFORCE algorithm uses Monte Carlo methods to estimate the expected return of a state-action pair and updates the policy parame-

ters (weights of the neural network) in the direction of the estimated gradient of the expected return with respect to the policy parameters. The algorithm learns optimal weights θ by repeatedly sampling random trajectories (Markov Decision Processes) from the environment, computing the return R_t of each time step and finally updating the policy's parameters according to returns and gradients of the logarithm of probabilities of the actions taken along the trajectory

$$\theta_{t+1} \leftarrow \theta_t + \nabla_\theta J(\theta) \tag{1}$$

$$\nabla_\theta J(\theta) = \sum_t R_t \nabla_\theta \log(\pi(a_t|s_t)), \tag{2}$$

where (in this case) we use the so called Monte Carlo (MC) return R_t. It incorporates the entire future of the respective trace of length T (discounted by γ)

$$R_t = \sum_{k=t}^{T} \gamma^k r_k. \tag{3}$$

Generally, the return R_t can take on other forms, depending on the agent model chosen, as will be shown later.

Due to the fact that full trajectories are randomly sampled (until the agent actually learns), REINFORCE has a very high variance and very low bias. Every next trace is uncorrelated from the previous one and the lessons the agent learns from one do not help with the next. This high variance can lead to significant fluctuations in the gradient estimates and in a very slow, rarely monotonous, convergence. Countering high variance while keeping low bias will be the central challenge of this work.

It is important to notice, that weights will be *increased* towards the gradient, as the *loss function* needs to be maximised and not minimised - we expect to converge towards high results at the end of an episode, not low losses. In other words, the neural network approximating the policy will be updated using gradient *ascent* instead of descent.

We will introduce further policy based agents that, in principle, still try to maximise the cumulative reward, but have a different way of measuring the weight (here the return R_t) with which to go in that direction.

2.2. Actor-Critic with n-step Bootstrapping

To tackle the aforementioned issue of high variance and enhance the performance of this family of models, the Actor-Critic approach was introduced. The Actor-Critic approach combines value-based and policy-based elements in an effort to make the most out of the two strategies' advantages. This approach consists of two separate agents, the policy-based Actor (low bias), and the value-based Critic (low variance). The Actor learns a policy that maps states to the probability of actions, while the Critic learns to evaluate the value function of the current policy. The Actor model uses the Critic network's estimate of the value V_ϕ to bootstrap its form of return (Q-value)

$$R_t = Q_n(s_t, a_t) = \sum_{k=t}^{t+n-1} \gamma^k r_k + \gamma^n V_\phi(s_{t+n}), \tag{4}$$

and then update the actor's values again in direction of the gradient Eq. 2. The Critic on the other hand, uses the mean squared error between its predicted value and the Q value to update its parameters:

$$\phi_t \leftarrow \phi_t - \alpha \nabla \sum_t (Q_n(s_t, a_t) - V_\phi)^2. \tag{5}$$

Introducing an upper bound to the amount of steps that are regarded to estimate the return, helps to further reduce high variance. Based on this method, the agent calculates the rewards received from n subsequent states, rather than the entire tail of the episode to calculate the total return. This allows for more efficient updates, faster learning and less randomness between the actions. When the new ball drops, it is usually irrelevant what the paddle did 3 steps ago. However, as always, due to the bias-variance trade-off, this method introduces more bias to the model. This means that an appropriate balance has to be achieved regarding the number of steps that will be used for bootstrapping. While this might solve the problem of high variance in the cumulative reward estimation, it does not provide a solution to the high variance of the gradient estimate that was introduced from Critic's estimate of the state-value function.

2.3. Actor-Critic with Baseline subtraction

The next method to tackle the problem of the high variance in the gradient estimate of the Critic is Baseline subtraction. A Baseline can be any value that is independent of the action taken. In this work, we use as Baseline the estimate of the expected reward at a specific time step

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

introducing the advantage function A_n as the new measure how much to change the respective weights in Eq. 2.

In this way, we measure how much better or worse an action is than the average action, given the current state. This is called an Advantage function. The baseline can be subtracted from the bootstrapped Q-value in the n-step case (cf. Eq. 4) or from the full return in the Monte Carlo case (cf. Eq. 3). By subtracting the baseline from the actual reward, the variance of the gradient estimate can be reduced, thus letting the model learn the optimal policy more reliably.

2.4. Actor-Critic with Bootstrapping and Baseline subtraction

The two enhancements of the Actor-Critic approach can be combined and result in a powerful model that gives superior results in comparison with the REINFORCE algorithm. All the above variations will be tested in the Catch game and their performance will be evaluated. However, we expect this full Actor-Critic model that combines Bootstrapping and Baseline subtraction to perform better and to converge noticeably faster than the rest. Going further, we will refer to this model as the *full* model.

In the following sections, we will perform a Hyperparameter optimization based on this model to determine the best parameters. After a good performing set of parameters is found we will then proceed to the Experiments section where the different agents will be evaluated and then the best agent will be stressed under different game configurations to further examine how well the agent can generalize.

3. Hyperparameter Tuning

Having looked at the different agent variations there are, we are now interested in tuning certain hyperparameters that are most going to change the performance of the agent. Constants over all our experiments are shown in Tab. 1.

Architecture	
Input Shape	(columns, rows, 2) or 3
Hidden layer 1	64
Batch Normalisation	default
Hidden layer 2	32
Batch Normalisation	default
Dropout	20%
Output	3 or 1
Parameters	
Initializer	GlorotNormal
Optimizer	Adam
Discount γ	0.99
Learning rate Critic	0.05

Table 1. Structure of the neural network that was used for both the Actor and the Critic network and other parameters that stay the same for all experiments. The difference in both lies only in the output shape and in the learning rate.

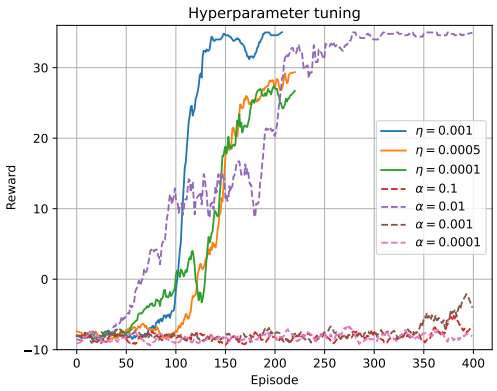

Figure 2. Hyperparameter tuning for network learning rate α and loss regularization η. The underlying agent is the full network, Actor-Critic with 5-step bootstrapping and baseline subtraction. Runs are averaged over 5 repetitions and smoothed with a savgol filter of window length 10. The learning rate was tuned with fixed $\eta = 0.01$ and the exploration with fixed $\alpha = 0.01$.

3.1. Method and setup

In general, one could test different learning rates, exploration strengths, model architecture like number of layers and neurons, gradient descent algorithms (optimizers), kernel initialization methods, discount factor, and so on. We are going to focus on the first two. The learning rate α is an obvious hyperparameter to tune and the strength of regularization η will determine the degree of added exploration. There is already some exploration due to the probabilistic nature of the policy, but adding to the loss an extra regularization term $H = \eta \sum_t \pi(a_t|s_t) \log(\pi(a_t|s_t))$ (see Entropy and Regularization in (2)) punishes going too often in the same direction, or in other words, favors exploration. We know that the loss function already includes the normalization of probability (via the logarithm in Eq. 2) to ensure that an action of high probability is not favored doubly, but this term introduces the possibility to actually tune the trade-off between exploitation and exploration.

3.2. Plot and Discussion

After an initial extensive grid search that is too messy to report, we could narrow down the range of learning rate and regularization to: $\alpha \in \{0.1, 0.01, 0.001, 0.0001\}$ and $\eta \in \{0.001, 0.0005, 0.0001\}$. The corresponding performances are shown in Fig. 2.

It is clear to see, that the learning rate has the biggest impact on the agent's performance and $\alpha = 0.01$ clearly yields the best outcome. It is interesting to notice that a learning rate of 0.01 is rather large compared to more complex problems. Still, the agent needs a while to converge, as the sampled trajectories are subject to high variance and the chance to

actually learn from most of them is small.

More interesting is the tuning of the exploration rate. We already discussed the high susceptibility to variance in policy-based agents and a larger exploration rate would increase it even further. Without exploration the agent is forced to only focus on two directions (never going right for example), but high exploration introduces too much variance. When we already know the supermarket to be east of us, we will not try to explore in a northwest direction. Another hyperparameter is the number of steps n to bootstrap. It seemed logical to keep n under or close to the number of rows (here 7). The value of 20 steps further, for example, does not present important information to the agent, as already two more balls will have dropped. A 1-step bootstrap, on the other hand, prevents the agent to move quickly enough to catch a ball that is further away.

4. Experiments

Having tuned the most important hyperparameters, we can continue on to comparing the performance of different agents and also in different configurations of the Catch environment. The first runs with the full model (5-step bootstrap and baseline subtraction) showed that it would eventually converge to score 35 on many consecutive runs, but that the way to get there was very unstable and long. To counter this high variance, we introduced (mini) batched updates of the parameters and selection of high-performing traces (an idea inspired by Prioritized Experience Replay, see (2)); this will be explained in more detail in the bonus section 5.2.

In the experiments that were conducted in the frame of this section, the setup shown in Tab. 2 was used.

Learning rate	0.01
# of Episodes	350
Max steps	250
n step	5
Baseline	True
Minibatch	4
Exploration η	0.001

Table 2. Default hyperparameters used.

Following this, we will apply the best-performing agent to different variations of the game to examine how well the model can generalize. These variations include changes in the size of the environment, in the input type, and in the speed that the balls drop (speed 1 means that a ball drops once every *row*-th step).

4.1. Comparing various policy-based agents

In the scope of this experiment, the models discussed in sections 2.1-2.4 are set to learn catching the balls of the Catch environment. Their respective performances are shown in Fig. 3, where the environment setup is the default 7x7 pixels with a speed equal to 1. The full Actor-Critic model presented in purple significantly outperforms the other models, while only one other (Actor critic with baseline but without bootstrap) seems to learn at all during the first 400 episodes. The most tenacious obstacle encountered when training was the high variance due to random trace selection. The Actor-Critic approach certainly introduces lower variance as the second, value based V_ϕ, network counters the high variance of the π_θ network. The extent to which the gradient variance is reduced by introducing both a critic and

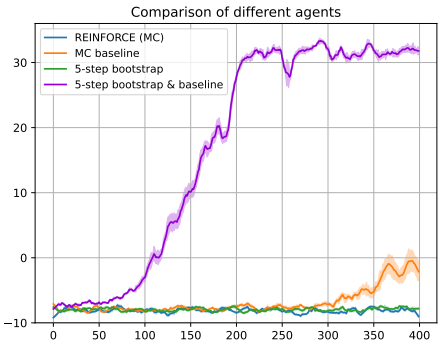

Figure 3. This diagram shows the performance aggregated over five runs including confidence intervals for each the four policy-based models. The purple full (5-step bootstrap and baseline subtraction) Actor-Critic model significantly outperforms the other models and converges after 200 epochs. MC baseline refers to the Actor Critic model with baseline but without bootstrapping. The other models perform poorly and more specifically the REINFORCE model seems to not learn at all in the run of 400 episodes. Implementing baseline subtraction via the advantage function seems to have the biggest impact on performance.

4.2. Environment size

The following experiment will vary the number of rows and columns of the Catch environment grid. This way, we can examine the performance and stability of the best-performing model (full Actor-Critic) when subjected to variations in the environment.

As can be seen in Figure 4, the environment in which the agent performs best is the 7x7 grid. The 9 (rows) by 7 (columns) rectangular grid, where the height of the environment is changed, also seems to be performing well, even though the hyperparameters were not optimized for this specific setup. This makes sense, because as the height increases, the agent has more time to reach the ball and is thus allowed to make a few errors. On the other hand, the 7 (rows) by 9 (columns) rectangular grid, where the width of the grid is increased, does not seem to be learning at all. This

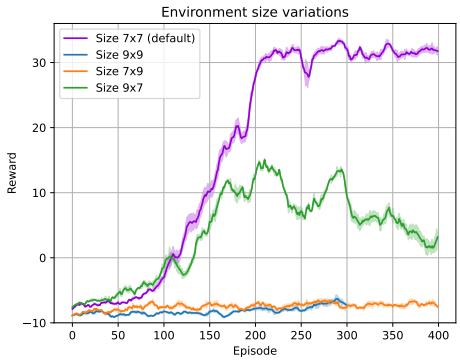

Figure 4. Variation of environment grid, where the sizes are rows x columns. The plots show quadratic (purple 7x7, blue 9x9) and rectangular (green 9x7, orange 7x9) grid variations. The default 7x7 grid on which the hyperparameters were tuned obviously outperforms the others and it takes the agent more episodes to learn in that larger environments.

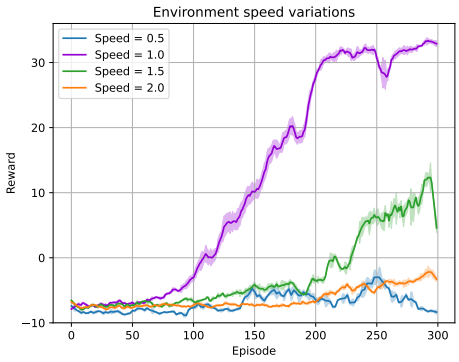

Figure 5. In this diagram the experiments of different runs with different speeds of balls averaged over five runs each are depicted. As it is shown the default 1.0-speed performance presented with purple is the highest while the 0.5-speed runs performed purely.

could be justified by the fact, that the agent's environment is bigger, so that it needs to make more movements in order to reach the dropping box, thus being more sensitive into taking wrong actions, while exploring the state space. We must also recognize, that the maximum achievable reward must be lower than the usual 35, as some boxes can not be caught, even if the agent makes no mistakes. It cannot possibly move 9 columns in 7 steps. The last configuration was a square grid of 9 by 9, which seems to be performing the worst. Even though we would expect it to be performing at least better than the other rectangular experiments, because the agent needs in the worst case (being at the far right side and the box dropping at the far left size), to make the same number of steps with the box until it reaches the bottom. In general, the bigger the size of the grid the longer the agent needs, in order to start learning. We assume the degree of exploration is a very important parameters, especially in different environment grids where the agent needs higher exploration, at least for the first number of episodes.

4.3. Environment speed

Now, the environment is set to its default size (7x7) but the speed at which the balls drop is now changed. It is interesting to see how the best-performing agent will behave and also very insightful for the generalization and the adaptiveness of the model. It is important to note that the speed is not the actual *falling* speed, but the ratio at what intervals balls drop relative to the environment steps. So in a grid that is 7 rows high, speed 0.5 means that a ball drops every 14 steps, speed 1.0 means every 7 steps and so on.

Logically, we expect that the slower the balls drop the better the performance will be. This is assumed as the model will have more time to compensate for some bad choices. For example, if the paddle has to move from one edge of the environment, in the default 7x7 setup with speed 1.0, to the other to catch the next ball then if it takes at least one move to the opposite direction of the dropping ball it will not have the time to eventually reach the correct position. Giving more time to the paddle, or more rows to the environment will lead to increased elasticity towards the mistakes of the model and more significantly give more time to the paddle to re-centered after a catch which will be in the edges. This problem is better reflected as the number of columns increases while the rows stay the same.

However, there is an interesting trade-off here in our experiment shown in Fig 5. We see that for the runs where the balls drop faster (green and orange), the model performs better which contradicts the above assumption. To reason this, we have to take into account that in the experiments we measure the reward, and the reward is achieved only when a ball is caught. When the balls drop faster, the chances to get a reward and learn are higher. For example, in the case of 0.5 (blue) and 2.0 (orange) speeds, by the time one ball falls in the 0.5-speed case, 4 balls will have fallen in the 2.0-speed case. The game might be more difficult for the orange case but it also means that in a set amount of steps, the 2.0-agent receives more non-zero rewards than the 0.5-agent and has thus more to learn from and more opportunities to do so.

4.4. Input as a vector

In this section, the input shape was changed from a 7x7 pixel representation to a vector of three elements. The three elements of the vector represent from left to right, the column of the paddle, column/s, and finally row/s of the lowest ball.

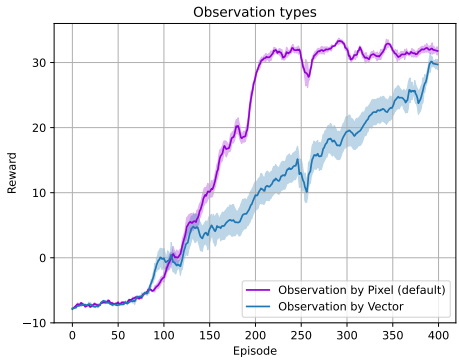

Figure 6. In this plot the difference between the performance of the best model, when the input shape changed from a 7x7x2 representation to a vector of three elements {paddle X, ball X, ball Y} representation, is depicted. We observe that the pixel representation not only converges faster but also is more stable.

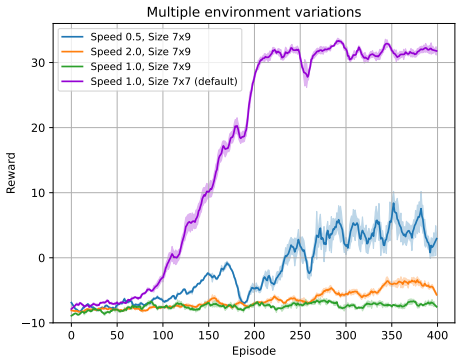

Figure 7. The graphs show the combinations of multiple environment variations at the same time. Clearly, the purple default run outperforms the rectangular runs at different speeds. It is interesting to notice that increasing the number of columns, and the width of the environment, can be countered by decreasing the speed, such that the paddle has time to reach the center in the meantime.

As we notice from Fig 6, the best model with input a vector seems to learn faster at the beginning, which is expected because the feature space is smaller when data are represented that way. However, later it seems that this is not a good representation and perhaps not as informing, as the model with the pixel input learns more efficiently and with higher stability. As we can understand from the confidence intervals the vector representation does not only lead to slower convergence but also to a more unstable learning process.

Moreover, based on the mechanics of this game, the vector representation also leads to some implications if the default parameters of the environment change. For example, if the speed is greater than 1 this means that there will be multiple balls falling at the same time. However, if we represent the input as a vector we will always have information about the lower ball and this would lead to pure performance and inadequate representation of the environment blocking the model from learning meaningful patterns.

4.5. Non-square environment with speed variation

The next experiment introduced two variations: size 7 (rows) by 9 (columns) grid and different speeds $\in \{0.5, 1.0, 2.0\}$. Naively, it is impossible at a speed of 1 or 2 to catch all the balls, as the paddle might be on the far right while the next ball already drops at the left and the agent would have to move 9 spaces in only 7 steps. If the speed is 0.5, however the agent is able to cover 9 steps in the double amount of time and performs better.

Figure 7 demonstrates that for the Catch game with a size of 7x9, a speed of 0.5 produces the best results for this setup. Contrary to previous assumptions derived from Fig. 5, our model is able to learn in a 7x9 environment, but only when subject to speed 0.5, not 1.0. This is probably due to the fact that the model has enough time to re-center the paddle after every catch, even when the ball reaches the edge of the environment. In contrast, speeds of 1.0 and 2.0 do not perform well, indicating that the model struggles to generalize when the speed is too high.

5. Bonus

5.1. Proximal Policy Optimization

Proximal Policy Optimization (PPO) aims to improve policy training stability by limiting the change made to the policy during each epoch, thus avoiding large policy updates (3). Empirical evidence shows that smaller policy updates during training are more likely to converge to an optimal solution. Additionally, large policy updates may cause the policy to fall into a local optima, which can lead to prolonged recovery time or even an inability to recover. The basic idea is that we learn a surrogate function, to ensure that the updated policy does not deviate too much from the old policy during training, in order to maintain stability and avoid catastrophic forgetting. The use of the clipped surrogate objective ensures that the policy update remains within a safe region, while still allowing for some exploration of new policies. By balancing the old and new policy, PPO with clipping can effectively improve the stability and convergence speed of the policy learning process.

$$L^C(\theta) = \hat{\mathbb{E}}_t[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)], \tag{7}$$

where $r_t(\theta) = \pi_\theta(t)/\pi_{\theta\text{old}}(t)$.

In Eq. 7, $L^C(\theta)$ represents the loss of the smallest surrogate

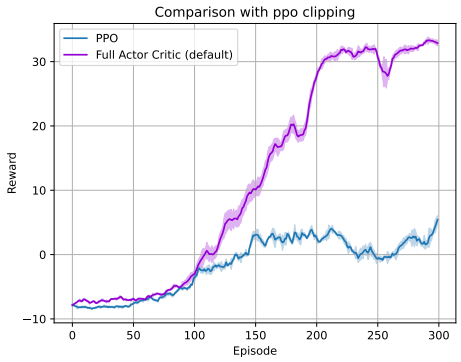

Figure 8. Comparison between PPO clipping and full actor-critic agent. The main hyperparameters applied are described in Table 2. For the loss calculation values of 0.5 and 1 are used for the hyperparameters c1 and c2, respectively. The rewards for the PPO and actor-critic experiments are averaged over 5 runs.

function, θ represents the parameters of the policy network being updated, $\hat{\mathbb{E}}_t$ represents the expected reward over a batch of training data at time t, $\hat{A}_t$ represents an estimate of the advantage function at time t. $r_t(\theta)$ represents the ratio between the probabilities of the action taken at time t under the policy network with parameters θ and the old policy network with parameters θ_{old} (9), clip is a clipping function that limits the range of $r_t(\theta)$ to between $1-\epsilon$ and $1+\epsilon$. (4)

$$L^V(\theta) = \frac{1}{2}\hat{\mathbb{E}}_t\left[(V_\theta(s_t)V_t^{\text{target}})^2\right] \tag{8}$$

In Eq. 8, $L^V(\theta)$ represents the loss of the critic. More specifically, $V\theta(s_t)$ represents the predicted value of the state at time t under the policy network with parameters θ, V_t^{target} represents a target value for the state at time t.

$$L^S(\theta) = \hat{\mathbb{E}}_t\left[\eta \cdot \text{KL}(\pi_{\theta\text{old}}(s_t), \pi_\theta(s_t))\right] \tag{9}$$

In Eq. 9, $L^S(\theta)$ defines the loss (or entropy gain) of the actor and η is the coefficient that controls the strength of the entropy regularization term.

$$L(\theta) = L^C(\theta) - c_1 L^V(\theta) + c_2 L^S(\theta) \tag{10}$$

In Eq. 10, $L(\theta)$ defines the linear combination of the Equations 7, 8 and 9. c_1 and c_2 are hyperparameters that control the relative importance of the value function and the entropy regularization terms.

In general we expect the PPO implementation to overperform the full actor critic agent and provide better results at least in terms of variance, because of the advantage of the policy restriction update, to not be more than the clipping parameter ϵ. However, as seen in Figure 8 the full actor critic agent has achieved far more better results than the PPO experiment. This can be explained, due to the fact that the PPO agent was not fine-tuned (compared to the full actor critic agent) and also the c1, c2 hyperparameters, which have a crucial role in the agent's policy update have not been optimized for the used configuration.

5.2. Prioritizing Memory Buffer

In the scope of applying different methods in order to optimize the actor-critic agent, we decided to implement a prioritized selection of the best runs from the memory buffer. Specifically, during the collection of the traces, we decided to select only the two traces that provide the highest average reward over a batch of four sampled traces. Finally, we average the gradients from those adjacent well-performing traces before updating the weights. This further increases stability of learning. Although the computation complexity rises significantly, we are able to steer the agent in the right direction and to make it follow the optimal policy without suffering from updates that would get stuck in a local extremum.

In Fig.9 we observe the behaviour of the average gradients in the two Actor-Critic agents and also the gradients of the REINFORCE agent over 100 episodes. With a quick look we can observe very clearly that the blue line that depicts the gradients of the REINFORCE method, shows very high variance with a value of $\sigma = 0.293$. Furthermore, we examine the performance of the gradients for the full Actor-Critic implementation (depicted with the orange line) and also the performance of the gradients for the actor-critic implementation with the maximum average reward selection as described above. Between the two Actor-Critic methods, we see a clear difference in the variance of their gradients. The Actor-Critic agent without selection has a variance of $\sigma = 0.129$, while the Actor-Critic agent with selection has a variance of $\sigma = 0.0764$. Empirically, we show that the agent performed significantly better with the additional selection of the best traces and converged much faster.

6. Discussion

In this final section, we will try to give deeper insights into core aspects of this report that will help the reader get a better understanding of the concepts presented above. To achieve this, we will try to give answers to the following research-formulated questions.

Do the policy gradients used by REINFORCE indeed suffer from high variance? In the REINFORCE approach, the model updates the gradients based on the whole trace.

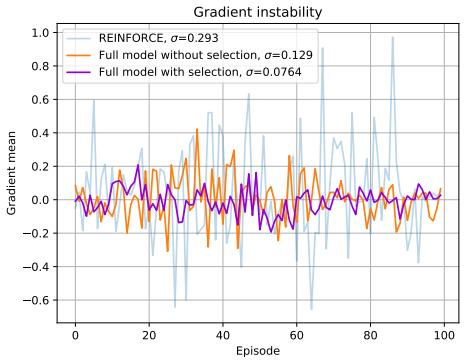

Figure 9. Average gradient of network for different models. It is clear that experience selection and batching (implemented in the purple graph) yields the lowest variance in gradients and thus highest stability in learning. Metric for comparison is for each episode the average of all the layer's gradients.

This obviously introduces low bias (each generated trace is independent from the last) at the cost of high variance. In order to better reflect this difference we compared directly the policy gradients of the REINFORCE model with our full and improved Actor-Critic model to get a better understanding of the impact of bootstrapping, baseline subtraction and the other stabilizing additions discussed in section 5.2. As we can see in Fig. 9 the full Actor-Critic model is significantly more stable and indeed the REINFORCE model gradients suffer from high variance. This is lastly also represented by the fact that the REINFORCE agent does not learn or at least not until a high number of episodes sampled.

What is the effect of bootstrapping and baseline subtraction, and their combination, within the actor-critic framework on the variance of the policy gradients? As it is already discussed in the previous sections, bootstrapping incorporates information about the current policy into the estimate of the value function. Based on the number of steps the volume of the influence is determined. The baseline subtraction contributes to reducing the variance of policy gradients by penalizing well but less-than-average performing state-action pairs which finally had the biggest impact on our model's performance. Those arguments are also verified by our experiments regarding the different policy-based algorithms where the increase in performance is shown when we gradually add bootstrapping and baseline subtraction to our Actor-Critic model. As we proved, when those two approaches are combined the impact on the performance is even higher and thus our full Actor-Critic model is the best-performing.

How do these techniques compare in terms of performance? As we show in Fig. 3 of the report, the bootstrap over five steps did not significantly increase the performance of the model in the first 400 episode. In contrast, baseline subtraction had undoubtedly more impact on the rewards. However, while we can claim that the impact of baseline subtraction influences positively the performance, the performance as a whole still remains very low without bootstrapping. We really see a difference only when we combine those two techniques together.

What is the effect of entropy regularization on performance? During our experiments, we noticed the importance of the entropy regularization factor η in the performance and stability of the models. In the models we examined, entropy played the role of balancing between exploration and exploitation. By disencouraging the policy to favour actions of high probability, entropy regularization can help the agent to explore more of the state-action space and discover new, potentially better behaviour. Entropy regularization can also help the policy converge to a better optimum by preventing premature convergence to suboptimal policies (local extrema). These benefits are achieved by penalizing the model for being too certain about the next actions. Before introducing entropy in our model we were frequently seeing in our experiments that the model never learned to do specific actions. For example, in some runs it never learned to go right and the reward converged to less than 30. These issues were caused due to too much exploitation of the model. Once the hyperparameter optimization was finished, the impact of the entropy became clear - as is presented in Fig 2.

How does the agent perform on tasks with different configurations and why? The agent performs relatively well in different configurations of the environment, namely speed, size, and input type. However, the default setup always seems to bring the best results. This might be due to the fact that the hyperparameter tuning ran for this specific setup but this does not necessarily mean that we can not recognize patterns and see that our model generalizes sufficiently even for tasks that it is not specifically optimized for. In Fig 7 for example, we saw that for the Size 7x9 the best speed is 0.5. This happens because the model now has the time needed to re-center the paddle after every catch at the edge of the environment. As expected, the other two speeds, 1.0 and 2.0 do not perform at all. This is a clear indication that the model is able to generalize and with hyperparameter tuning and sufficient amount of epochs it can converge.

All group members participated equally in the experiments and report.

References

[1] Title: Behaviour Suite for Reinforcement Learning
URL: https://arxiv.org/pdf/1908.03568.pdf

[2] Author: Aske Plaat
Title: Deep Reinforcement Learning
Publisher: Springer Nature Singapore

[3] *OpenAI Docs* PPO
Accessed: 01-05-2023
https://spinningup.openai.com/en/latest/algorithms/ppo.html

[4] *Medium* Proximal Policy Optimization
Accessed: 02-05-2023
https://towardsdatascience.com/proximal-policy-optimization-ppo-explained-abed1952457b