Evaluating Policy Gradient Methods

Marios Sirtmatsis,Yugantar Prakash

1 Introduction

For this project, we decided to further explore policy gradient methods and test two different ones on multiple environments. Policy gradient methods are very important and responsible for some of the biggest successes in RL. Advanced methods like A2C, A3C or DDPG showed how strong the underlying concepts of policy gradient techniques can be and with these methods, researchers were able to outperform state-of-the-art performance on multiple RL benchmarks [6, 5]. The goal of this work is to evaluate the REINFORCE with baseline algorithm as well as the One-Step Actor Critic on two environments. These two environments have different difficulties of getting solved and therefore, testing our implementation of the algorithms shall be tested on both to show their robustness.

2 Policy Gradient methods and parameterisation

We tried different kinds of parameterisation for the policy and the value function. As already introduced in the book by Sutton and Barto, the main requirement for parameterisation functions to work is being differentiable [7]. Therefore, we tried a non-linear and a linear parameterisation. Overall, the non linear parameterisation, done through neural networks, appeared to be less stable and very sensitive to hyperparameters (learning rates). Using linear function approximation and therefore linear parameterisation worked better and resulted in a more stable performance of the agent.

2.1 Neural Networks [Possible extra points for trying multiple parameterisations]

Similar to a lot of most recent work, one can try and use policy gradient methods with a Neural Network parameterisation. Therefore, instead of approximating the policy and the value function as linear functions, one can rely on non-linear function approximators, being neural networks. We tried using this approach in this work as one possible parameterisation strategy. Therefore, both, policy and value function were parameterised as small fully connected neural networks. The policy therefore consisted of an input layer with the size of the state space of the used environment. It then mapped this input to a 128 neuron fully connected layer, followed by a layer with n neurons, where n is the number of actions in the used environment. Between the layers a ReLU activation was used while the output layer is passed through a Softmax function to produce probabilities, as needed for a policy function. Similarly, for the value function approximator network, the input size also was the same as the number of features in the state space. The hidden layer was the same as for the policy network and the output layer only consisted of one neuron, which outputs the value function approximation for a state input.

2.2 Linear function Approximation

For linear function approximation, we used two different approaches for approximating the value function and the policy.

- **Value Function Approximation:** For approximating the value function, it was first important to decide how to represent state samples that the value function gets as an input and upon which it has to determine which features are more important than others, using weights w. For the state representation, we therefore used a fourier basis, where each state feature was transformed to a vector. These vectors were than stacked together to one. This resulted in a vector of the following form (similar to HW2)

$$\phi(s) = [1, \cos(1\pi x_1), \cos(2\pi x_1), \ldots, \cos(M\pi x_1), \cos(1\pi x_n), \cos(2\pi x_n), \ldots \cos(M\pi x_n)]$$

  where $x_1, ..., x_n$ are the features of a state S_t with $S_t \in \mathbb{R}^n$.. Before passing the features to be transformed to a fourier basis however, they had to be normalized by a simple min max scaling, where

$$S_t = \frac{S_t - \min}{\min - \max}$$

  or

$$x_i = \frac{x_i - \min}{\min - \max}$$

  for all $x_i \in S_t$. For then training a linear function approximator, one simply had to optimize weights w through gradient ascent as in the REINFORCE (with baseline) and One Step Actor

critic algorithms. The calculation of the value function approximation, could then be done by calculating

$$v(s; w) = w^T \phi(s)$$

and the gradient update in the optimization progress was done through the use of

$$\nabla v(s; w) = \phi(s).$$

- **Policy Parameterisation:** For parameterising the policy, we did not rely on a fourier basis as explained before. We first tried doing that but it did not result in any reasonable performance. This could be due to many reasons but we assume that the vector created through using a fourier basis for the state combined with the action space was not able to represent the state action pairs in a way, that could be learned properly. Instead, we used a one hot representation, inspired through resources that we found online [3]. In this representation, the state action representation looks as follows:

  $$\phi(s, a) = [0, 0, 0, ..., x_1, x_2, x_3, ..., x_n, ..., 0]$$

  Every entry in the representation vector is zero except for the one that corresponds with a row a in a $|\mathbf{A}| \times |\mathbf{S}|$ representation where a is the action. To then get probabilities over actions for a particular state, particularly a state-action mapping (policy), we calculate:

  $$\pi(a|s; \theta) = \frac{\exp \theta^T \phi(s, a)}{\sum_b \exp \theta^T \phi(s, b)}$$

  where θ is the weight vector learned for the policy parameterisation. The gradient used for the gradient ascent update in the REINFORCE and One Step Actor critic algorithms then is

  $$\nabla \pi(a|s; \theta) = \phi(s, a) - \sum_b \pi(b|s; \theta)\phi(s, b).$$

3 Environments

The goal of this work is to try and implement two different policy gradient algorithms. One is REINFORCE with baseline and the other one is a One Step Actor Critic. To evaluate these algorithms, it is important to execute them on multiple environments and see how they perform. Therefore, we followed the approach of starting with a simple environment to get the algorithms to work initially and move on to more difficult ones from there. The first environment, and one of the most popular benchmark environments in RL, is CartPole, which was first introduced by Sutton and Barto [7]. The second environment that is used to evaluate our algorithms is the Acrobot environment, introduced by Sutton and Barto [7]. For both environments, we use OpenAI gym as simulator and testing suite [4]. Gym enables an easy use of RL environments for testing algorithms and solving interesting tasks, from classic control up to Atari or even more realistic problems.

3.1 The CartPole Environment

A full documentation and explanation on how the CartPole environment works and what the goal is can either be found in the documentation of OpenAI gym or in the book of Sutton and Barto [2, 7]. However, to briefly explain, the CartPole environment is an environment where the agent can navigate a cart either to the left or to the right. The goal of the agent is to keep the pole on the cart straight or at least somewhat straight for as long as possible. For every time step in an episode that the agent "survives", it gets a reward of +1. If the cart moves more than 2.4 units from the center or the pole is 12 degrees away from vertical, the episode ends. Also, the episode ends after a certain threshold of episodes. In our case, this threshold was 200 but this could be any number. The state that the agent observes at each time step consists of four values being the *Cart Position, Cart Velocity, Pole Angle* and *Pole Angular Velocity*. The CartPole environment is considered solved when having an average accumulated reward per episode of 195 for the last 100 episodes. A sample figure of the environment can be found in Figure 1.

3.2 The Acrobot Environment

The Acrobot environment is a little more complex than the CartPole environment. The Acrobot is a pendulum which consists of two links, where both links point downward and are connected through a joint. The upper link is the only one being actuated. The goal for this environment is to swing the two links to have the lower one reach a certain height with its bottom tip. This can be done through an agent by applying force to either move the connecting joint between the two links to the left or to the right. The state that the agent observes at each time step consists of certain processed physical quantities, being the two rotational joint angles passed through cosine and sine functions and the joint angular velocities. The reward for each time step is -1 and 0 when transitioning into the terminal state as the goal of the environment is to get the lower tip up as soon as possible. Further information about the environment can be found in the documentation of OpenAI Gym [1]. A sample image to better understand what the environment looks like can be found in Figure 2.

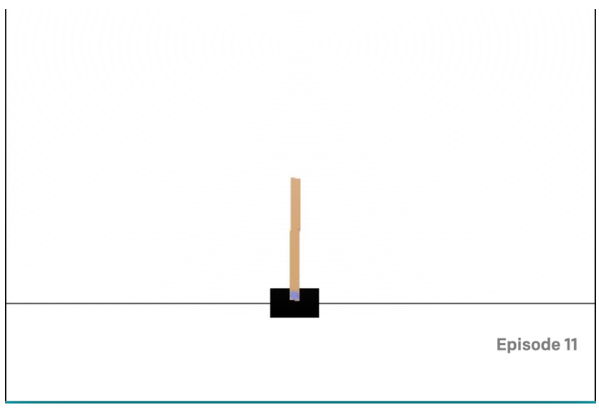


Figure 1: A sample image of the CartPole environment.

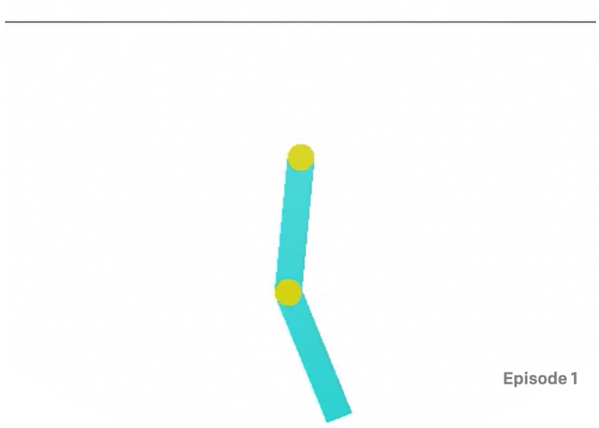


Figure 2: A sample image of the Acrobot environment.

4 REINFORCE with baseline

The theorem that needs to hold any policy gradient method to work is the policy gradient theorem. For the REINFORCE with baseline, one simply modifies the policy gradient theorem by adding an arbitrary baseline function, that is not allowed to vary with a varying action a. Formally, we have:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a (q_\pi(s,a) - b(s)) \nabla \pi(a|s,\theta)$$

This can then be used as described in the book of Sutton and Barto, to form an update rule, which looks as follows[7].

$$\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

It is easy to see how $\frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$ is the same as $\nabla \ln \pi(A_t|S_t,\theta_t)$, as deriving $\ln(x)$ leads to $\frac{1}{x}$. In the REINFORCE algorithm with baseline, this update rule is used to perform gradient ascent to optimize the weights θ of the policy or to gradient descent to minimize with the same update negated. As a baseline, one could use any function that fulfils the requirement described above. However, the book suggests to just use an estimation of the value function that is learned while learning the weights of the policy. We stick with this approach in this work. All of this assembled, results in the following algorithm, algorithm 1, again, coming from the book of Sutton and Barto [7]. The pseudo code shows how the algorithm works. Basically, the most important requirement is to have a differentiable policy and state-value function parameterization. From there one, one calculates a δ, which is then used to update the weights of the parameterised policy and value function for each value inside a trajectory collected throughout multiple episodes. The weights are updated through gradient ascent as described earlier.

Algorithm 1: The REINFORCE algorithm using a baseline $\hat{v}$
Input:
1) A differentiable policy parameterization $\pi(a|s,\theta)$
2) A differentiable state-value function parameterization $\hat{v}(s,w)$
Algorithm parameters:
1) step sizes $\alpha^\theta > 0, \alpha^w > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$;
for *episode in episodes* **do**
  Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\theta)$;
  for *step in episodesteps* **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$;
    $\delta \leftarrow G - \hat{v}(S_t, w)$;
    $w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S_t, w)$;
    $\theta \leftarrow \theta + \alpha^\theta \gamma^t \delta \nabla \ln \pi(A_t|S_t,\theta)$;

5 One Step Actor Critic

While REINFORCE also learns value function, it is distinctly different from an Actor Critic Method because the latter uses the value function learnt for bootstrapping. As Sutton and Barto [7] point, this bootstrapping is what introduces a bias and a dependence on the quality of the function approximation, in exchange for reduced variance and accelerated learning. One Step Actor Critic method, in particular (OS-AC) replace the full return with one-step return:

$$\theta_{t+1} = \theta_t + \alpha(G_{t:t+1} - \hat{v}(S_t,w)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

$$= \theta_t + \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1},w) - \hat{v}(S_t,w)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

$$= \theta_t + \alpha \delta_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

where w are the parameters of the value function model $\hat{v}$, leaving us with the following psuedo code:

Algorithm 2: The One Step Actor Critic algorithm using a parameterized $\hat{v}$
Input:
1) A differentiable policy parameterization $\pi(a|s,\theta)$
2) A differentiable state-value function parameterization $\hat{v}(s,w)$
Algorithm parameters:
1) step sizes $\alpha^\theta > 0, \alpha^w > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$;
for *episode in episodes* **do**
  Generate S first state in an episode;
  $I \leftarrow 1$;
  while S *is not terminal* **do**
    $A \leftarrow \pi(\cdot|S,\theta)$;
    Take Action A, observe R, S';
    $\delta \leftarrow R + \gamma\hat{v}(S',w) - \hat{v}(S,w)$;
    $w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S_t,w)$;
    $\theta \leftarrow \theta + \alpha^\theta I \delta \nabla \ln \pi(A_t|S_t,\theta)$;
    $I \leftarrow \gamma I$;
    $S \leftarrow S'$;

6 Experimental Results

For every environment and method combination, certain experiments will be repeated. However, not all experiments are conducted on all environments as they are sometimes used to show something for a specific setup and investigating them multiple times would not be reasonable. We especially decided to play around with different parameterisations for the different algorithm to be able to try multiple things instead of just repeating the same methods multiple times.

6.1 CartPole - REINFORCE

In the following subsections different experiments will be shown and interpretations to collected results will be given.

6.1.1 Hyperparameter Search

For hyperparameter search, we tried different values for the two step sizes α^θ and α^w. When using neural networks for parameterising policy and value function, the agent's learning process was very noisy. We found that the step sizes had to be very low so that the training process does not diverge and result in very bad performance. In order to establish a better readability and interpretability of the hyperparameter search plot, we used a filtering technique to further reduce the noisyness of the accumulated rewards over time. Figure 3 shows this plot with less noise. The plot depicts the accumulated reward per episode for different hyperparameter settings. These settings are indicated through different line colors and annotated in the legend of the plot. For completeness, Figure 4 depicts the figure without any filtering. One can see how noisy the collected rewards are, because of a very unstable or in some cases even really underperforming training process. Overall, we found that too high step sizes brought a good start of the training process but resulted in a performance drop after some time. Step sizes that were too low were not even able to learn properly in just 300 episodes, which indicates a very slow learning process.

For the linear function approximation case, as mentioned above, we used a Fourier basis to transform states S_t into a feature representation. Therefore, we were able to tune the two αs as before as well as M, which is the number of fourier features per state value that should be used. As we wanted to explore a relatively big parameter space, we chose to not visualise the results as a plot but to rather use a table which includes the mean of the sum of reward per episode and the standard deviation just this, calculated over 300 episodes of training. One hyperparameter search run, which consists of 300 episodes, can then be evaluated based on these values of mean and standard deviation. The best possible run would have a very high mean of reward sums (in CartPole close to 200) and a very low standard deviation. This would indicated that the agent was able to perform very well and that the deviation of collected accumulated rewards per episode was not to high. Table 1 shows exactly this table. One can see that the best hyperparameter run was achieved by using $M = 15, \alpha^\theta = 0.01$ and $\alpha^w = 0.001$. In this run, the agent must have learned very fast because the standard deviation is low, compared to other high performing runs. Also the agent must have behaved quite close to optimal behaviour as the mean of all accumulated rewards per episode is somewhat close to 200, which is the optimal score for the CartPole environment. Overall, one can argue that for selecting hyperparameters with a fourier basis as feature representation, one has to put the relationship between step sizes α^w and α^w into context with the feature size M. Some combinations perform very well while others don't. We did not exactly see a clear pattern on when these hyperparameters match well together and when they don't. However, one can clearly see that having a too low feature size M combined with very low learning rates typically does not work too well. Also, when having a too high feature size with a too

low learning rate, the performance drops significantly. Still, one can not simply translate this into a pattern, as finding the right step sizes for feature sizes is a very task dependent and tedious task. It is important to find step sizes that are somewhere in between, so that the agent does not learn too fast and converge to sub-optimal behaviour as well as not too slow without ever converging in a reasonable time. Still, one pattern that we could extract is that for REINFORCE with baseline the performance of the algorithm is better when having $\alpha^{\theta} \geq \alpha^{w}$.

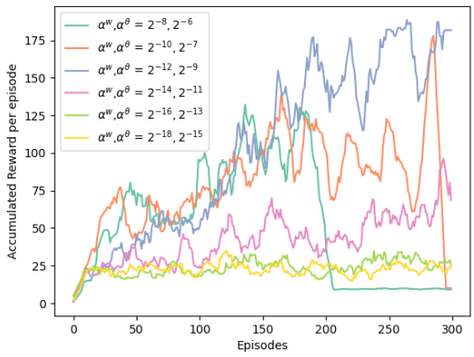


Figure 3: A plot showing the accumulated reward per episode for different learning step sizes. The lines were smoothed for better readability.

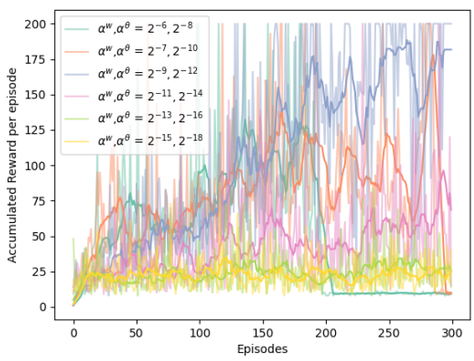


Figure 4: A plot showing the accumulated reward per episode for different learning step sizes without smoothing.

6.1.2 Performance of Linear Function Approximation vs Neural Networks

We also found it interesting to compare the performance of REINFORCE with baseline on CartPole using linear function approximation vs using neural networks as parameterisation. Figure 5 shows the performance of both parameterisation methods, where the performance measure simply is the accumulated reward per episode over time. One can see that both accumulated reward plots are kind of noisy and that the neural network parameterisation does a better job in stabilising in a higher level of reward at some point. However, both methods are able to solve the CartPole task in a reasonable amount of episodes, by having an average over 195 for the last 100 episodes. Still, one can see that the trend of the accumulated reward is not one that goes straight up. This can be explained by the nature of learning itself and the combination of exploration and exploitation. Even though, the agent learns a good behaviour over time, it still explores to some extend. Therefore, it cannot always perform best but rather has to explore states that it had not reached before to possibly result in a better outcome in the long run.

Especially in the CartPole environment, learning can get very noisy. In CartPole, as the goal is to balance a pole on a cart, individual actions can have a high influence on the overall performance in an episode. For example, if the agent pushes the cart to far left through some exploratory actions, it may not be possible to get the cart into a position where the pole gets straight again. This would lead to a very bad performing episode, but still, this could be a scenario that is very important to training, as the small accumulated reward of the episode gives a high penalty on this kind of behaviour.

6.1.3 Exploration Experiment [Possible extra points for this experiment]

In Policy gradient methods as used in our experiments, exploration is not handled explicitly. Rather, the policy is modeled as a probability distribution where each action has a certain probability of being selected. Therefore, as soon as not all probabilities except for one are zero, the agent is exploring.

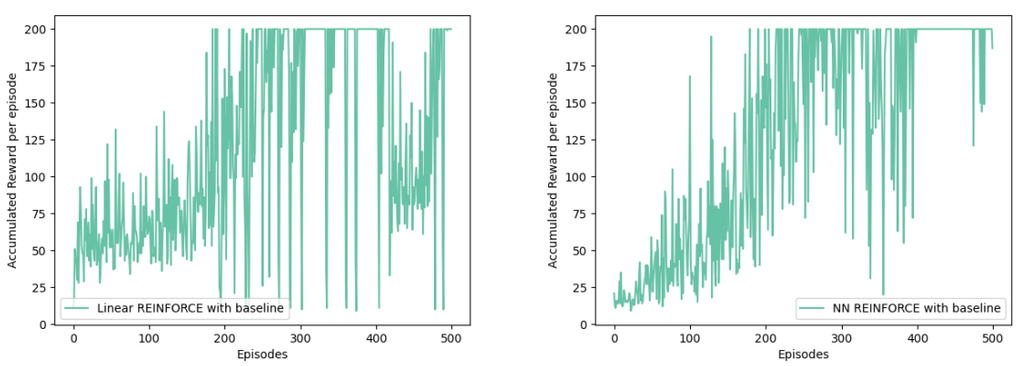
Mean Sum Rewards	Std Sum Rewards	M	α^{θ}	α^{w}
5.806667	14.445851	5	0.0001	0.001
2.520000	11.774956	5	0.0001	0.010
2.383333	10.753126	5	0.0001	0.020
9.303333	36.748759	5	0.0010	0.001
8.960000	31.980490	5	0.0010	0.010
8.033333	26.802094	5	0.0010	0.020
09.920000	63.539334	5	0.0100	0.001
53.466667	61.716142	5	0.0100	0.010
53.543333	60.932707	5	0.0100	0.020
5.160000	12.333467	15	0.0001	0.001
23.696667	12.825677	15	0.0001	0.010
22.606667	12.169852	15	0.0001	0.020
50.523333	36.052316	15	0.0010	0.001
31.046667	17.621894	15	0.0010	0.010
23.530000	12.131602	15	0.0010	0.020
132.263333	64.276543	15	0.0100	0.001
129.250000	68.582851	15	0.0100	0.010
95.743333	71.639636	15	0.0100	0.020
25.366667	14.230913	30	0.0001	0.001
23.470000	13.106071	30	0.0001	0.010
21.170000	12.530540	30	0.0001	0.020
51.916667	32.836307	30	0.0010	0.001
24.840000	14.365505	30	0.0010	0.010
23.196667	12.073580	30	0.0010	0.020
114.110000	59.513510	30	0.0100	0.001
105.453333	58.081849	30	0.0100	0.010
14.586667	5.296145	30	0.0100	0.020

Table 1: Hyperparameter Search results for REINFORCE with baseline on the CartPole environment over 300 episodes. Different values for M, α_θ and α_w were used.

We therefore found it interesting to investigate how much the agent actually explores and how this correlates with the performance that it is able to achieve over time. To do so, we trained an agent using the REINFORCE with baseline algorithm with linear parameterisation having the hyperparameters $\alpha^{w} = 0.001$, $\alpha^{\theta} = 0.01$ and $M = 15$ for 1500 episodes. For each of the episodes we calculated the fraction of randomness as the number of actions that the agent selects that do not have the highest probability in the policy distribution divided by all actions in the episode. Formally, we computed

$$FractionOfRandomness = \frac{|B|}{|A|}$$

with $B = \{b_1, b_2, ..., b_n | b_i \notin \arg\max \pi(A_t|S_t;\theta)\}$ and A being all actions in an episode. In Figure 6 one can see a plot of the just described experiment. On the x-axis, we have the episode number. On the y-axis, the left hand side depicts the accumulated reward per episode while the right hand side shows the fraction of randomness per episode. One can clearly see that in the beginning, the randomness is quite high. This means that the agent is exploring a lot and does not have a high confidence in selecting certain actions in certain states yet. This, obviously, affects the accumulated episode reward negatively, as the agent is not able to steer into the right directions when needed. However, one can



(a) (b)

Figure 5: The performance of the REINFORCE with baseline algorithm using linear (a) and neural network (b) parameterisation as described above.

see that over time the agent is able to learn which actions are desirable in which states and therefore gets more and more confident, reducing the number of actions that are selected randomly. One can also see that less exploration at some point results in a higher performance. This does not say that exploration itself is bad for performance but rather indicates that exploration is a burden that has to be gone through to ever be able to achieve reasonably good results.

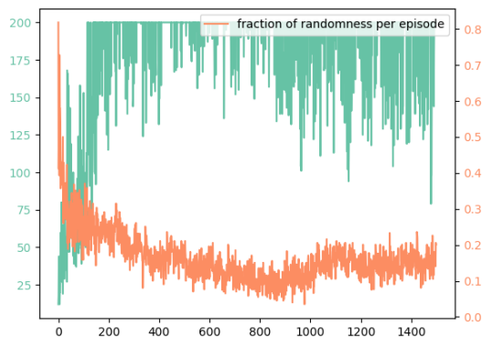


Figure 6: The plot of the exploration experiment without confidence threshold.

However, Figure 6 shows that even after 1500 episodes, the agent does not reach a state, where it has a lower exploration than around 0.1. One could ask, if it would be reasonable to modify the action selection process of the agent to be more into exploitation, to induce the agent to select the action that it is most confident about (having highest probability in the policy distribution) at some point. We were interested in finding out, whether this approach would result in a better or worse performing agent and how the agent behaviour would change. Therefore, we implemented an experiment, where we let the agent stochastically select actions if the probability of the policy for an action is lower than a threshold. However, if the policy has an action where the probability for selecting it is higher than a certain threshold, we deterministically had the agent select exactly this action. Figure 7 shows the figure where we ran the same exploration experiment as in the paragraph before but this time with the modified agent that exploits when confident enough. One can see that this did not end in a better but rather in a noisier and worse performance. By that, one can imply that deterministically manipulating exploration in such a simple way is not necessarily the best way to go. Throughout the learning process, the agent may have assumptions about certain actions that are not true. For instance, the agent might have a high probability for a certain action even though it is the completely wrong action for the state that it is in. Having stochasticity avoids having the agent select actions like these too often.

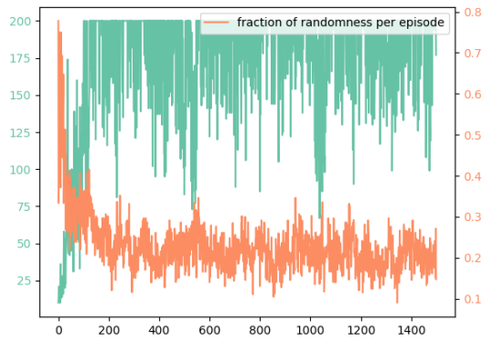


Figure 7: The plot of the exploration experiment with confidence threshold over 1500 episodes. The x-axis shows the episodes, the y-axis shows the accumulated reward per episode on the left and the fraction of randomness on the right.

6.2 CartPole - One Step Actor Critic

For OS-AC algorithm on CartPole, we decided to parameterize the policy and value function as neural networks with 1 hidden layer of size 64. The input would be the raw state parameterization provided by the Gym library's implementation of the environment: (x, $\tilde{x}$, o,$\tilde{o}$) where they represent cart position, cart velocity, stick angle, stick angular velocity respectively. The output of the policy model is a vector whose size is the cardinality of the action space (2).

6.2.1 Hyperparameter Search

Hyperparameter search for One Step Actor Critic (OS-AC) was done in the same fashion as for CartPole REINFORCE: α^θ and α^w. Noisy learning process of agent required us to have very low step sizes. Figure 8 shows the plot for accumulated reward per episode for different hyperparameter settings. Not surprisingly, we found similar results as previously seen: too high step sizes would train fast early on but fair poorly after some time. However, we do not see that drastic difference in learning performance for smaller step sizes (yellow, green and pink lines have similar performance). Note that the average model for the first 300 episodes significantly underperforms compared to REINFORCE with baseline's performance. Similar to REINFORCE as well, we found that the algorithm is better behaved when $\alpha^\theta \geq \alpha^w$. This does not necessarily mean that the algorithm itself is flawed. We believe this difference in performance here, and later in Acrobot, is primarily caused by our parameterization of the two networks. Unlike what we have done in class so far when facing state spaces with continuous values, in OS-AC we did not split it into bins, nor did we use a Fourier basis to transform states into a feature representation.

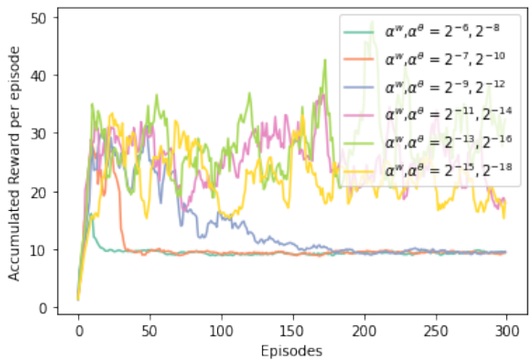


Figure 8: A plot showing the accumulated reward per episode for different learning step sizes. The lines were smoothed for better readability.

6.2.2 Performance in Long-Term Training

We found it interesting to compare the long term training performance of OS-AC since in our hyperparameter search we saw significantly reduced performance compared to REINFORCE with baseline. Unlike Supervised Learning where the dataset is already generated and getting any further data may be a costly and lengthy process, RL with CartPole environment does not have an upper limit on how much data you can run the algorithm for. An interesting correlation that can be made is that we consider number of episodes equivalent to epochs in Supervised Learning. This assumption is reasonably grounded if episodes are thought of as sampling trajectories from the distribution of all trajectories that are possible in the environment.

To test this, we trained the models over 5000 episodes over two hyperparameter values. Figure 9 shows the result for $\alpha^\theta = 2^{-12}$ and $\alpha^w = 2^{-12}$. We see that the model learns very fast (within 500 episode), which seems standard. However, curiously, we see that the model quickly un-learns (within 200 episode) and struggles to learn back to the same level of performance for a very long time. And once it achieves the best performance it can get (around 4500 episodes), it repeats the process of un-learning. In the meanwhile, we see that the loss for both policy and value function networks is constantly decreasing, as we require but the variance in the loss increases significantly. Our understanding for the same is that, as mentioned above, CartPole environment can get very noisy and with large enough hyperparameters, the algorithm can diverge very fast.

For the case of Figure 10, which shows the results for $\alpha^\theta = 2^{-16}$ and $\alpha^w = 2^{-13}$, we see something more akin to the asymptotic nature of learning in supervised learning: after learning over 1000 episodes, the learning slows down and effectively stops. Notice that while the algorithm is stable over the entire learning period, it takes much longer to reach the same performance level as for the case of larger learning rates. We also observe the same fall in performance after some time, although it is less drastic and less common. Similarly, the noise in loss for the policy network is significantly lower but loss for value function network shows the same results. As expected from the hyperparameter search, we also see that the final performance is much higher than that for previous case.

6.3 Acrobot - REINFORCE with baseline

6.3.1 Hyperparameter Search

In Figure 11 and Figure 12, one can see the performance of the REINFORCE with baseline algorithm using neural network parameterisation on the acrobot environment. Same as for the CartPole environments, the plots show the smoothed and unsmoothed performance due to somewhat noisy results. One can clearly see that most step sizes (learning rates in NN language) were not able to bring good results. However, with $\alpha^w = 2^{-11}$ and $\alpha^\theta = 2^{-14}$ the agent was able to perform well on the Acrobot environment.

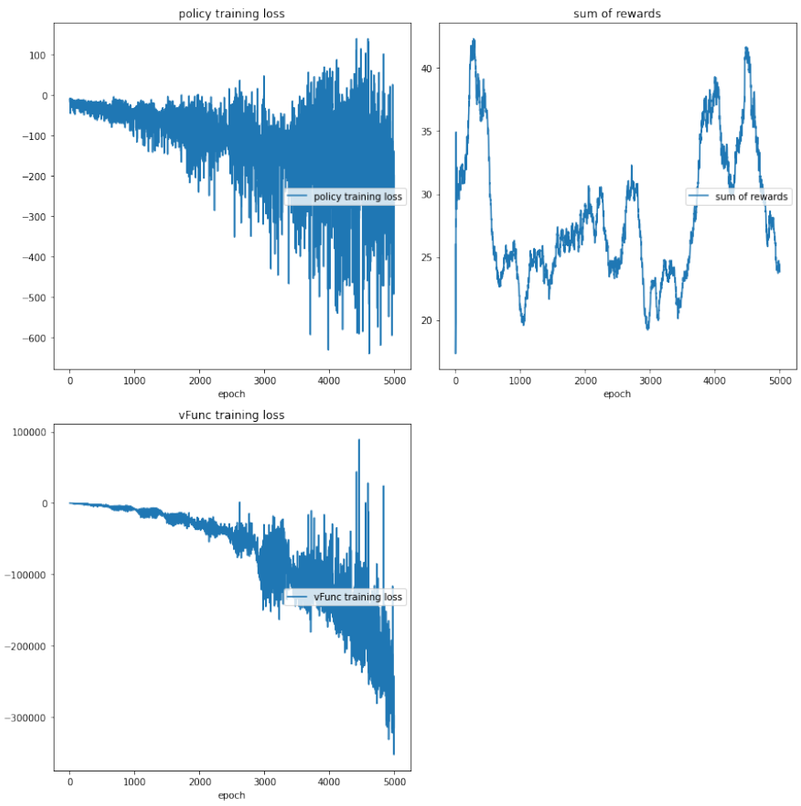


Figure 9: A plot showing the accumulated reward per episode for CartPole environment on Actor-Critic over 5000 episodes, with $\alpha^{\theta} = 2^{-12}$ and $\alpha^{w} = 2^{-12}$

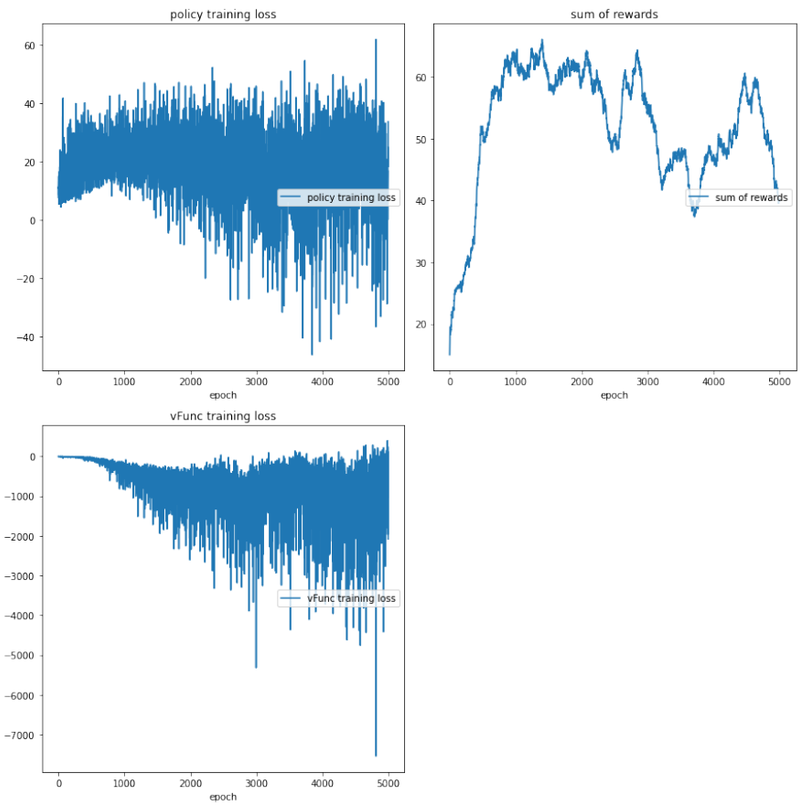


Figure 10: A plot showing the accumulated reward per episode for CartPole environment on Actor-Critic over 5000 episodes, with $\alpha^{\theta} = 2^{-16}$ and $\alpha^{w} = 2^{-13}$

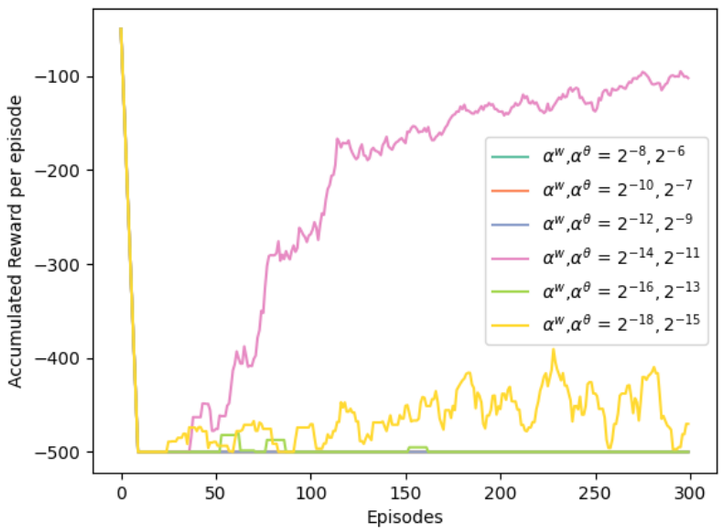


Figure 11: A plot showing the accumulated reward per episode for different learning step sizes. The lines were smoothed for better readability.

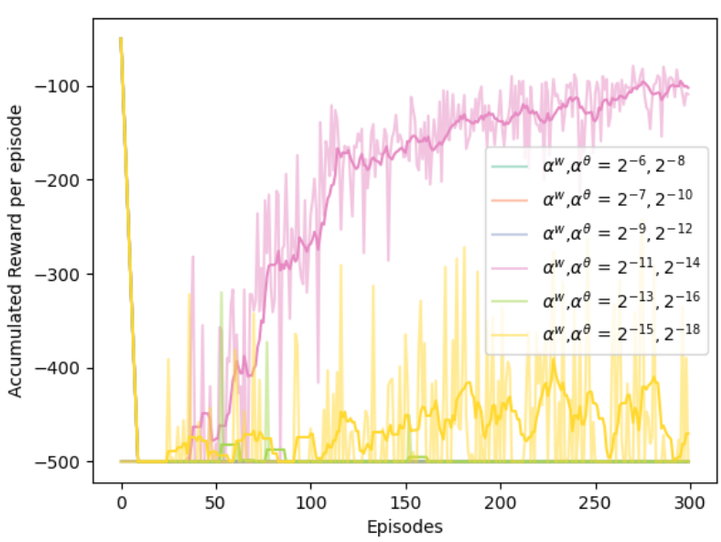


Figure 12: A plot showing the accumulated reward per episode for different learning step sizes without filtering.

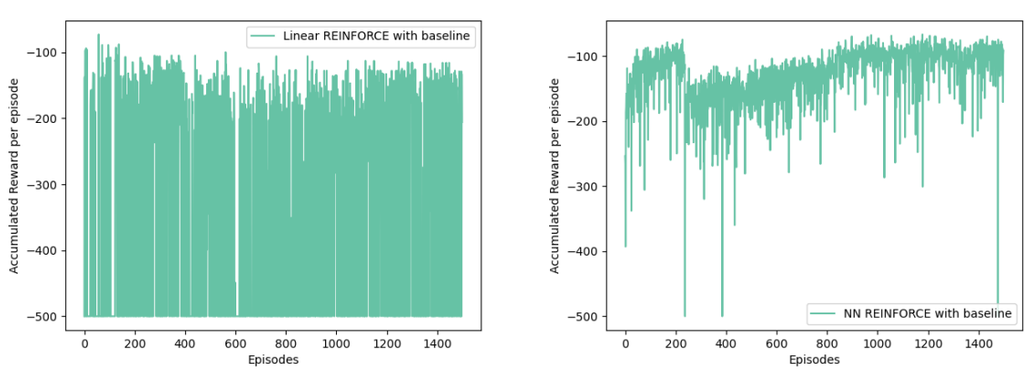
For the linear function approximation parameterisation, one can find the results of hyperparameter search in Table 2. One can see that the results are quite bad. This means that the algorithm is not able to really perform well or learn to solve the task in 300 episodes. However, this does not necessarily mean that the algorithm is flawed. First, it may be that the searched parameter space is too small and that the step sizes and M are not the ones that would be needed to solve the Acrobot environment. Second, it may be the case that using hyperparameter search over only 300 episodes is not enough for this task, to see if the algorithm would work in the long run. Third, it could be that linear parameterisation is not complex enough to capture the complexity of the environment and that one either has to increase M highly or use another approach to get the algorithm perform well.

Mean Sum Rewards	Std Sum Rewards	M	α^{θ}	α^{w}
-357.840000	178.322071	10	0.1	0.01
-357.780000	175.438873	10	0.1	0.02
-366.840000	163.627405	10	0.3	0.01
-359.423333	176.990482	10	0.3	0.02
-344.666667	175.533498	15	0.1	0.01
-390.713333	153.841383	15	0.1	0.02
-443.283333	133.488213	15	0.3	0.01
-379.883333	168.749251	15	0.3	0.02

Table 2: Hyperparametersearch for the Acrobot environment using linear function approximation as parameterization. One hyperparameter run was performed on 300 episodes

6.3.2 Performance of Linear Function Approximation vs Neural Networks

To evaluate the performance of the agent on the Acrobot environment, similar to the evaluation before on the CartPole environment, we ran the REINFORCE algorithm with both parameterisations and the best hyperparameters found through the hyperparameter search for a certain number of episodes. However, this time, we ran the algorithm for 1500 episodes, to evaluate agent performance over a longer run. In Figure 13, one can see plots of the accumulated reward per episode over these 1500 episodes. One can clearly see that the parameterisation using Neural Networks works way better than the applying Linear function approximation with a fourier basis. Still, both methods are very sensitive to hyperparameters, which could be seen in the hyperparameter search section. Overall, it is not clear to say why exactly Neural Networks worked better than the linear function approximation with fourier basis. It could be due to the fact that Neural Networks are able to capture more complex state spaces. However, the Acrobot environment is not that complex, so linear function approximation should also be able to perform well. Therefore, we argue that investing more time in hyperparameter search for the linear parameterisation would be worth considering as future work, because it could be that other hyperparameters let the algorithm perform better than it did in this experiment.



(a) (b)

Figure 13: The REINFORCE with baseline performance in the Acrobot Environment comparing linear function approximation (a) with Neural Network parameterisation (b).

6.4 Acrobot - One Step Actor Critic

For OS-AC algorithm on Acrobot, we decided to parameterize the policy and value function as neural networks with 1 hidden layer of size 100. The input would be the raw state parameterization provided by the Gym library's implementation of the environment where the state consists of: $[cos(\theta_1)\ sin(\theta_1)\ cos(\theta_2)\ sin(\theta_2),\ \theta_{1,v},\ \theta_{2,v}]$ where θ_1, θ_2, $\theta_{1,v}$,$\theta_{2,v}$ represent the two rotational joint angles and the joint angular velocities respectively. The output of the policy model is a vector whose size is the cardinality of the action space (3).

6.4.1 Hyperparameter Search

Hyperparameter search for OS-AC on Acrobot was performed by trying out different learning rate settings: α^{θ} and α^{w}. With Acrobot, a non trivial trajectory (any trajectory with reward more than the minimum of -500) are more uncommon to get than for CartPole. In particular, if policy parameters are initialized incorrectly, then training can take significantly longer to even begin. Because of this stochastic nature of initialization, we can get incorrect understanding of how a certain setting of hyperparameter performs. To make sure this does not affect our experiments, we made sure to use Xavier Uniform initialization and we conducted the experiments several times and compared the separate graphs thus generated.

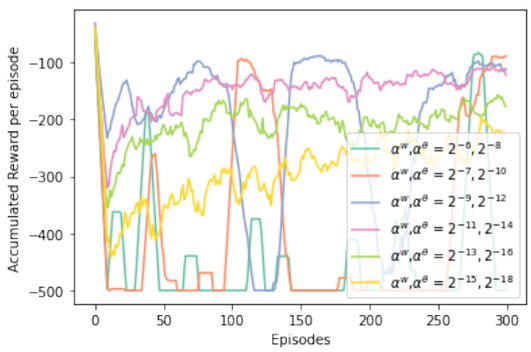


Figure 14: A plot showing the accumulated reward per episode for different learning step sizes for Acrobot over OS-AC

Figure 14 shows the plot for accumulated reward per episode for different hyperparameter settings for one instance. The aforementioned difficulty in reliability of this method can be seen here with the blue, green and red lines. They have steeper slopes (in fact, around 150 episodes blue has the highest performance), but the training is unreliable. This unreliability was more commonly seen for higher values of learning rates. Not surprisingly, we found similar results as previously seen: smaller step sizes would be more reliable but train slower.

6.4.2 Performance in Long-Term Training

We performed the same experiment on Acrobat environment using OS-AC as we did for CartPole to observe the performance of the algorithm in the long-term. We trained two models for 1000 episodes each. To account for the aforementioned stochastic problem with Acrobat environment, we made sure that initial episodes would not have the absolute minimum reward possible (-500) for both cases. Figure 15 contains curves for $\alpha^{\theta} = 2^{-12}$ and $\alpha^{w} = 2^{-12}$. Similar to CartPole, we observe that too large learning rates leads to unreliable learning with very steep slopes, indicating that the agent learns and un-learns very quickly. In addition, this model's best performance was higher than the model with smaller learning rates. This, we understand, is because we did not take large enough episode size and at the end of the episode 1000, second model started to learn again as indicated by the rising slope in Figure 16.

We observe similar curve for Figure 16 contains the graphs for $\alpha^{\theta} = 2^{-16}$ and $\alpha^{w} = 2^{-13}$ and we observe what we expected: The model learns over 300 episodes (aka epochs in the graph), and then the training stabilizes over the next 700 episodes. Policy and value function network losses slowly get more noisy as the model learns a policy that more frequently leads to higher rewards.

7 Conclusions [Possible extra points for discussing approaches and future possibilities for experiments]

Throughout this work, one could see many different experiments on the two policy gradient algorithms REINFORCE with baseline and the One-Step Actor Critic. Also, results were interpreted to the best of the authors' knowledge. However, there stay open questions about certain things. While working with different parameterisations, one could easily see how important it is to select the parameterisation that fits to the environment that the algorithm is working with. Using a fourier-basis as feature representation may, for instance, not be suitable when having data where locality or other data characteristics matter. This problem can be solved through investing a lot of time in designing differentiable functions that capture the state space well. For example, in a Gridworld environment, one could encode the state positions and give them some kind of context to better represent the state space in an embedded feature space. However, doing this is not really suitable for environments where the state space is huge. This is why it is important to rely on function approximators that can capture highly complex data representations. There, Neural Networks come into play, as they are universal function approximators and could therefore parameterise and policy or value function of any environment. As we have seen above though, they can be quite unstable or not work very well on too simple environments. Then it is important to better rely on simpler and more stable function approximation techniques.

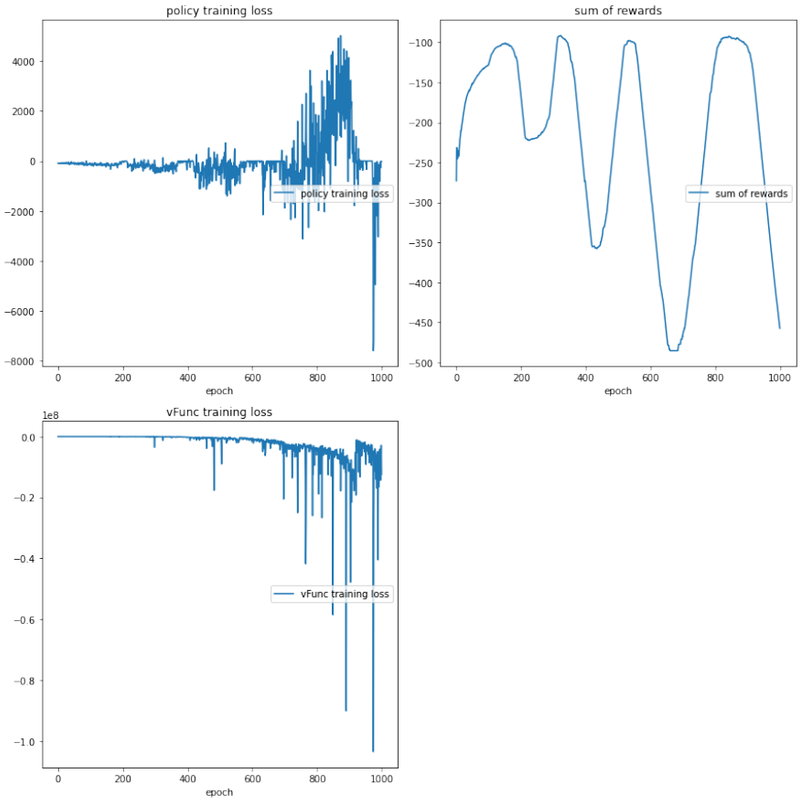


Figure 15: A plot showing the accumulated reward per episode for Acrobot environment on Actor-Critic over 1000 episodes, with $\alpha^{\theta} = 2^{-12}$ and $\alpha^{w} = 2^{-12}$

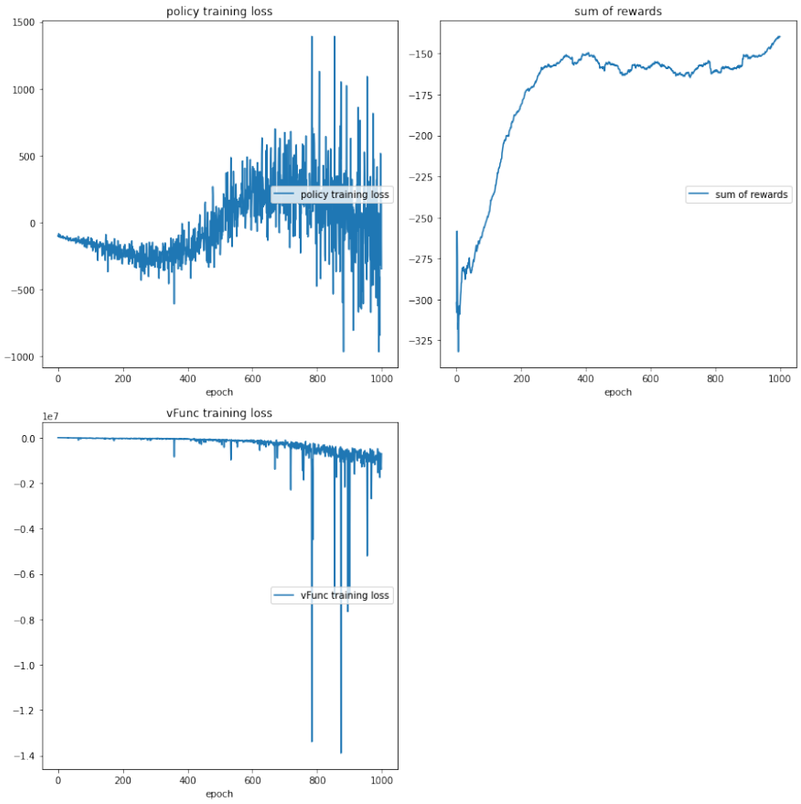


Figure 16: A plot showing the accumulated reward per episode for Acrobot environment on Actor-Critic over 1000 episodes, with $\alpha^{\theta} = 2^{-16}$ and $\alpha^{w} = 2^{-13}$

References

[1] "Openai gym's acrobot," https://gym.openai.com/envs/Acrobot-v1/, Accessed: 2021-12-08.

[2] "Openai gym's cartpole," https://gym.openai.com/envs/CartPole-v1/, Accessed: 2021-12-08.

[3] "Policy gradients from scratch with python," http://quant.am/cs/2017/08/07/policy-gradients/, Accessed: 2021-12-08.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[5] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[6] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

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