University of California Berkeley

Computer Science Department

CS 294-112

HW3: Q-Learning and Actor-Critic methods

Submitted by
Cristobal Pais
CS 294-112
Oct 10th
Fall 2018

HW3: Q-Learning and Actor-Critic methods

Cristobal Pais

Oct 10th 2018

1 Problem 1: Q-Learning

1.1 Q0: Warm-up with Lunar Lander and Pong from RAM

Following the recommendations from the problem statement, we test our Vanilla and Double Q-Learning implementations using the Lunar Landing and Pong RAM versions. In addition, we compare the performance of our agent when using MSE or Huber loss functions for modeling the total error when updating our estimates of the Q_ϕ function.

From the results in Figures 1 and 2, we can see that our implementation is able to achieve the theoretical reward level of 150 during the first 500k time-steps as indicated in the problem statement. Significant differences can be observed in the improving patters of both implementations, showing us the sensitivity of our agent to different loss functions when updating the Q estimates. However, both versions are able to reach the desired value, allowing us to conclude that our Vanilla Q-Learning implementation is working properly.

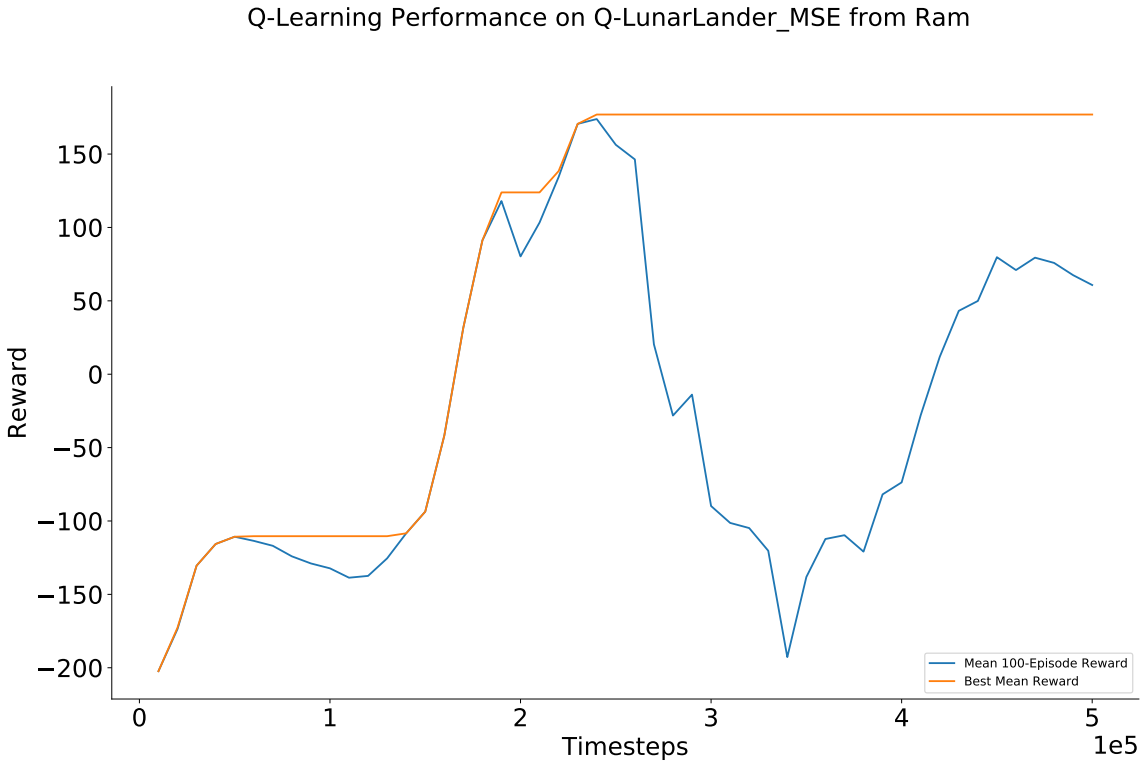

Figure 1: Mean and best reward average evolution for Lunar Lander using Vanilla Q-Learning with MSE as the error loss function. Best performance is able to achieve above the theoretical goal of 150.

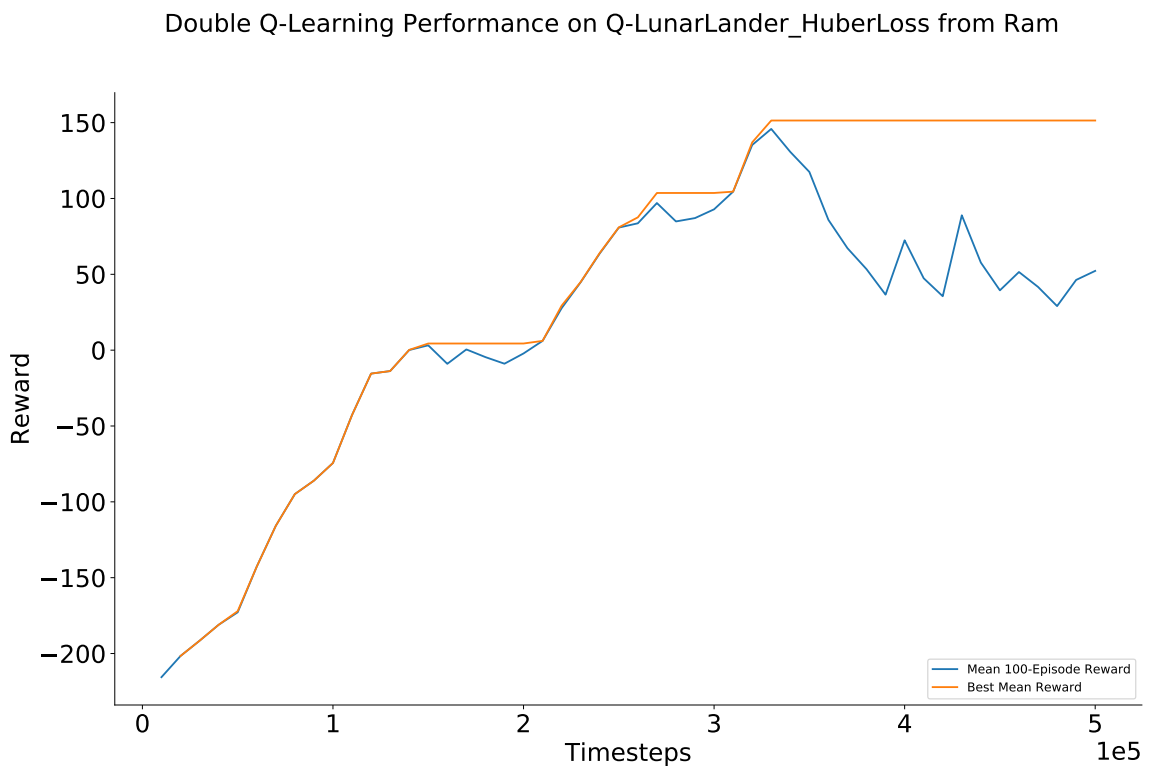

Figure 2: Mean and best reward average evolution for Lunar Lander using Vanilla Q-Learning implementing Huber loss as the error function. Best performance is able to achieve above the theoretical goal of 150.

Following the previous discussion, the same performance is obtained when training our agent to play the classic Atari game "Pong" from RAM observations. In Figure 3 and 10, we can observe how the suggested reward evolution levels are achieved when training our agent with the MSE or Huber loss function.

When testing our Double Q-Learning implementation, we can observe — as expected — an improved performance with respect to the Vanilla Q-Learning approach when using the Huber loss function, as seen in Figure 10. In this case, we are able to reach better reward levels (almost 200) in about 200k time-steps and the overall performance tends to be more stable than in the simplest approach. On the other hand, worse performance is achieved when implementing the Double Q-Learning algorithm with the MSE as the loss function (Figure 3). This shows us how sensitive the implementation can be to certain parameters/components of the algorithm, as well as certain random seeds: after testing other random seeds, we were able to obtain competitive results again, however, we include this graph for comparison purposes.

Model training and figures are generated using the following commands (modifying **double_q = True / False** inside dqn.py, line 35):

```
python run_dqn_lander.py
```

```
python Plotter.py --fileName=Q-LunarLander_HuberLoss
                  --plotName=Q-LunarLander_HuberLoss
python Plotter.py --fileName=Q-LunarLander_MSE
                  --plotName=Q-LunarLander_MSE
python Plotter.py --fileName=DoubleQ-LunarLanding-HuberLoss
                  --plotName=DoubleQ-LunarLanding-HuberLoss
python Plotter.py --fileName=DoubleQ-LunarLanding-MSE
                  --plotName=DoubleQ-LunarLanding-MSE
```

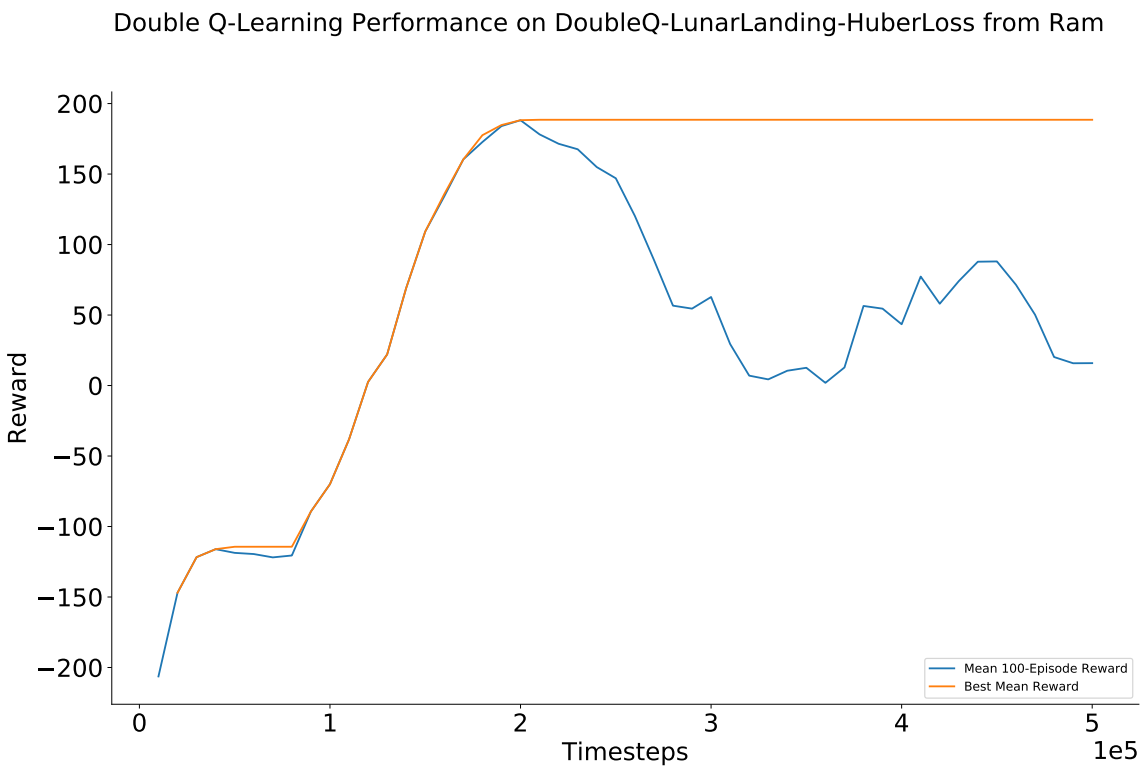

Figure 3: Mean and best reward average evolution for Lunar Lander using Double Q-Learning implementing Huber loss as the error function. Best performance is able to achieve above the theoretical goal of 150, improving the results of the Vanilla implementation.

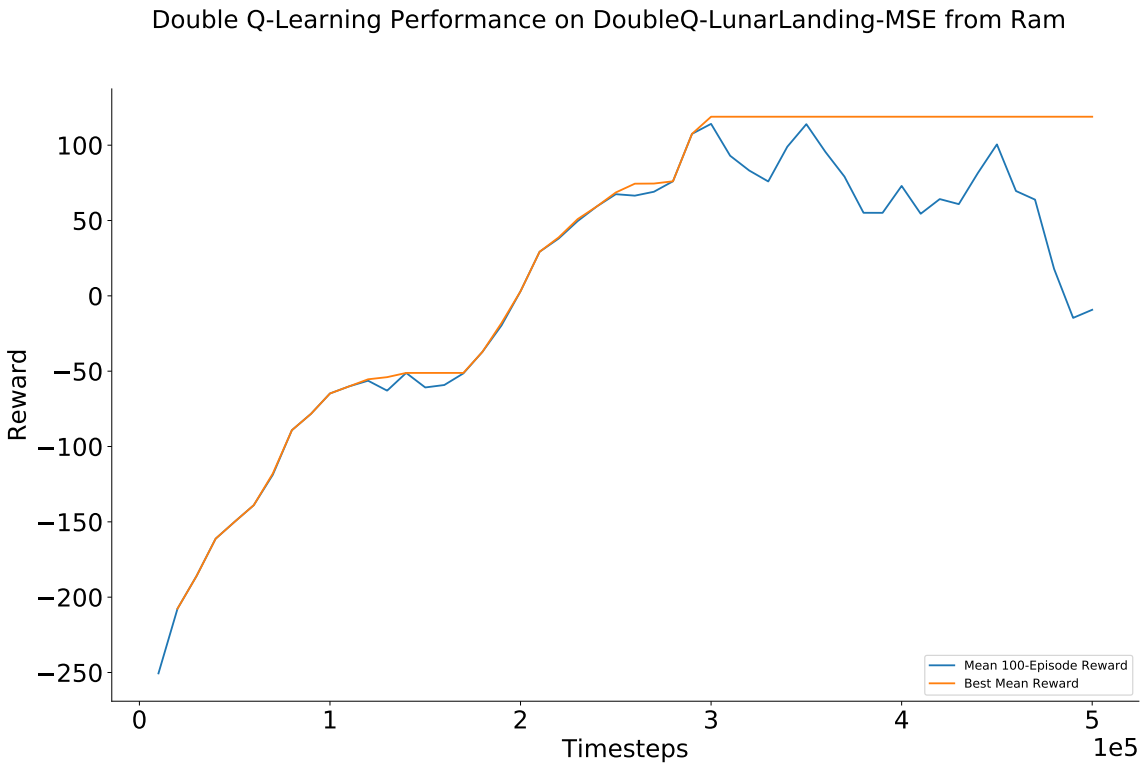

Figure 4: Mean and best reward average evolution for Lunar Lander using Double Q-Learning with MSE as the error loss function. This approach is not able to achieve the theoretical performance of 150.

Similar results are obtained with Pong from RAM, where both the MSE and Huber loss functions are tested for with our Vanilla and Double-Q implementation. In both cases, the evolution of the average returns follows the expected pattern, reaching the different levels mentioned in the statement of the problem. However, the Double-Q implementation with MSE as loss function seems to be slower — and more unstable — than the Vanilla version, reaching lower performance levels. However, performance varies significantly with different random seeds, not being significant to assess the performance of the method.

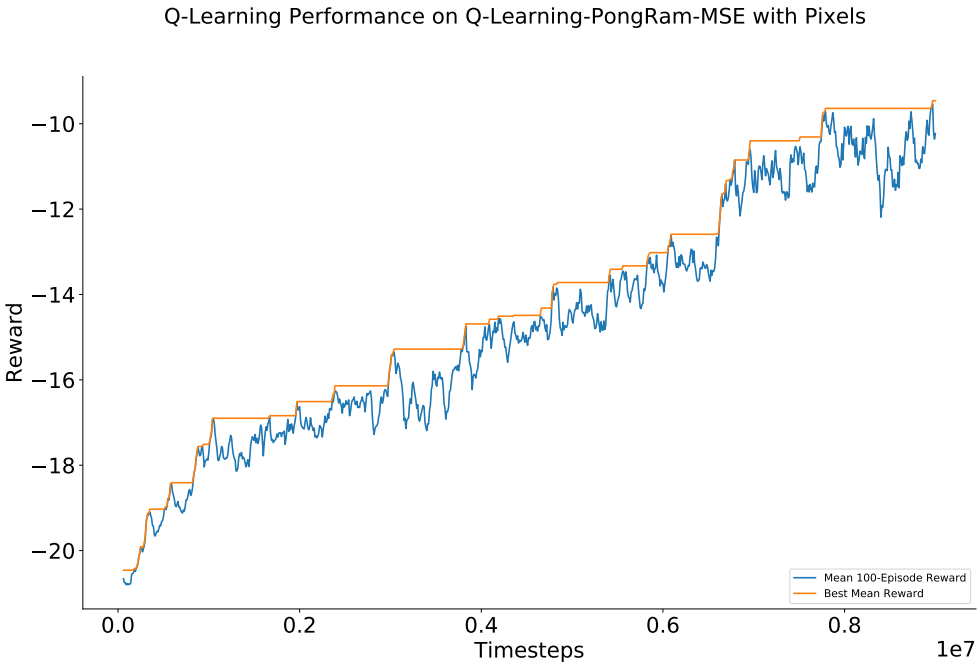

Figure 5: Mean and best reward evolution for Pong-RAM using Vanilla Q-Learning with MSE as the error loss function. Expected performance levels are achieved.

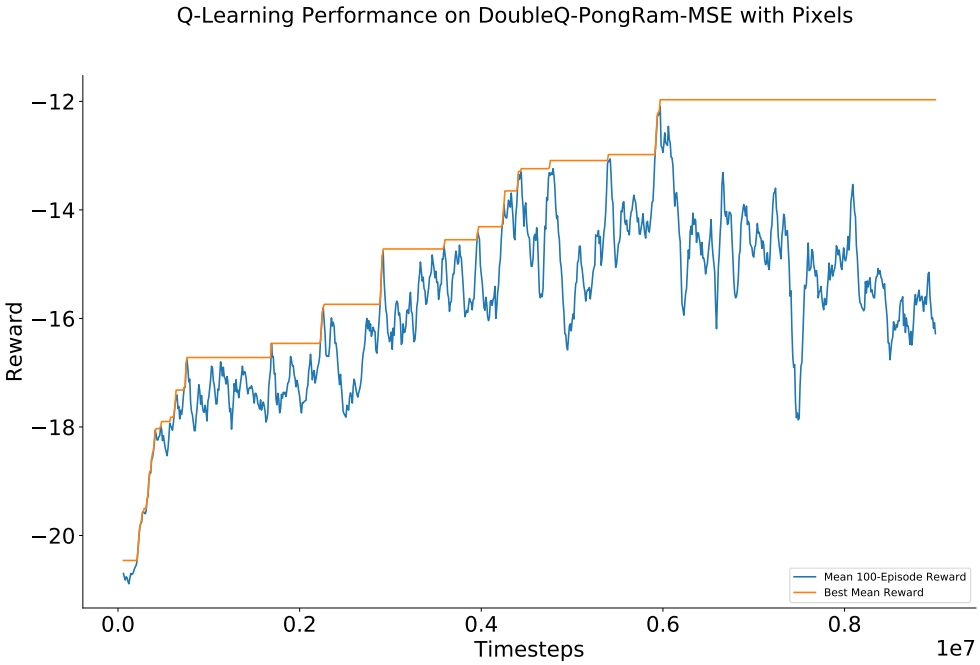

Figure 6: Mean and best reward evolution for Pong-RAM using Double Q-Learning with MSE as the error loss function. Slower convergence than the Vanilla Q-Learning is experienced.

When using the Huber loss function, we obtain better results with our Double-Q implementation reaching higher average return levels in fewer time-steps compared to the Vanilla implementation, as expected. Therefore, after checking the correctness of our code, we proceed with the real experiments based on pixels instead of RAM.

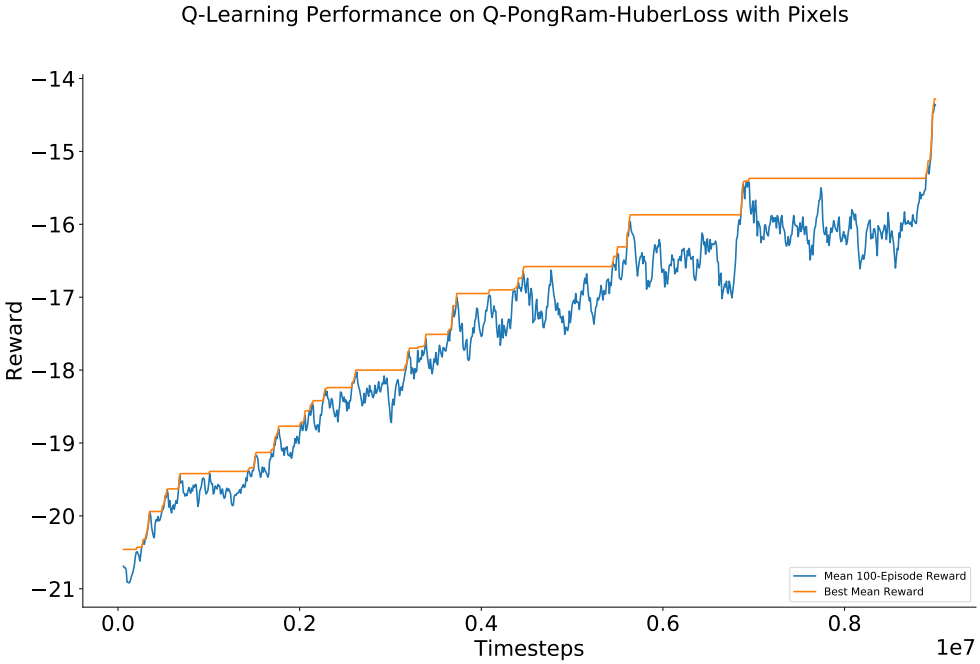

Figure 7: Mean and best reward evolution for Pong-RAM using Vanilla Q-Learning implementing Huber loss as the error function. More stable improvement is achieved in comparison to the MSE approach.

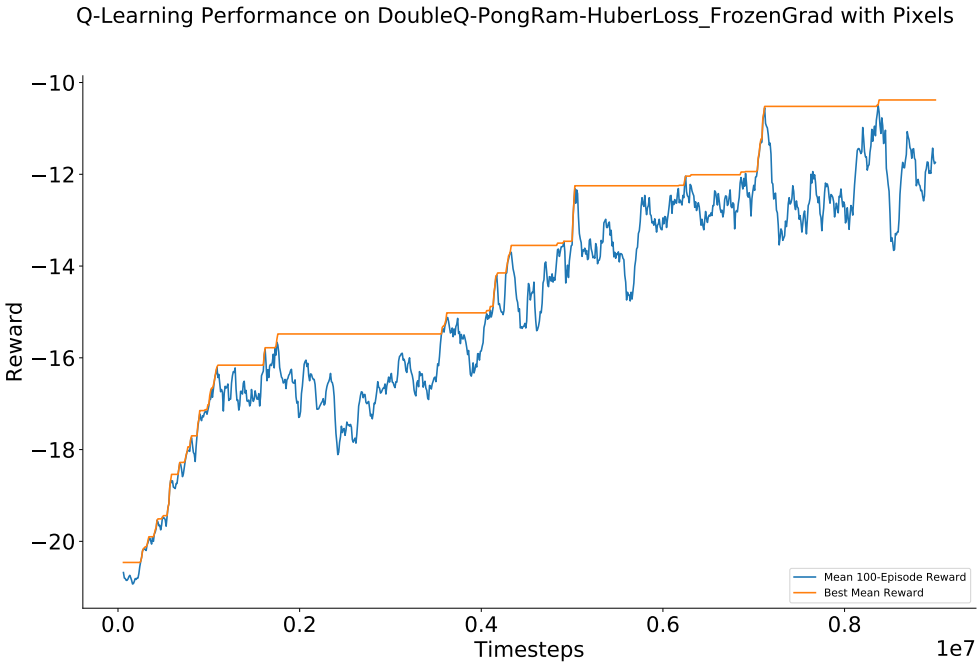

Figure 8: Mean and best reward evolution for Pong-RAM using Double Q-Learning implementing Huber loss as the error function. Faster and better convergence than the Vanilla Q-Learning is observed, reaching a -10 (Double Q) vs -14 (Vanilla) best mean reward after 1e7 time-steps.

Model training and figures are generated using the following commands (modifying **double_q = True / False** inside dqn.py, line 35):

```
python run_dqn_ram.py
```

```
python Plotter.py --fileName=Q-PongRAM_HuberLoss
                  --plotName=Q-PongRAM_HuberLoss
python Plotter.py --fileName=Q-PongRAM_MSE
                  --plotName=Q-PongRAM_MSE
python Plotter.py --fileName=DoubleQ-PongRAM-HuberLoss
                  --plotName=DoubleQ-PongRAM-HuberLoss
python Plotter.py --fileName=DoubleQ-PongRAM-MSE
                  --plotName=DoubleQ-PongRAM-MSE
```

1.2 Q1: Q-Learning and Pong

In this questions, we train our agent to play the classic Atari game "Pong" using raw pixel information from a buffer containing a series of episodes from the game. Results shown are obtained using the TensorFlow-GPU version (NVIDIA GTX770M) after 9 million time-steps.

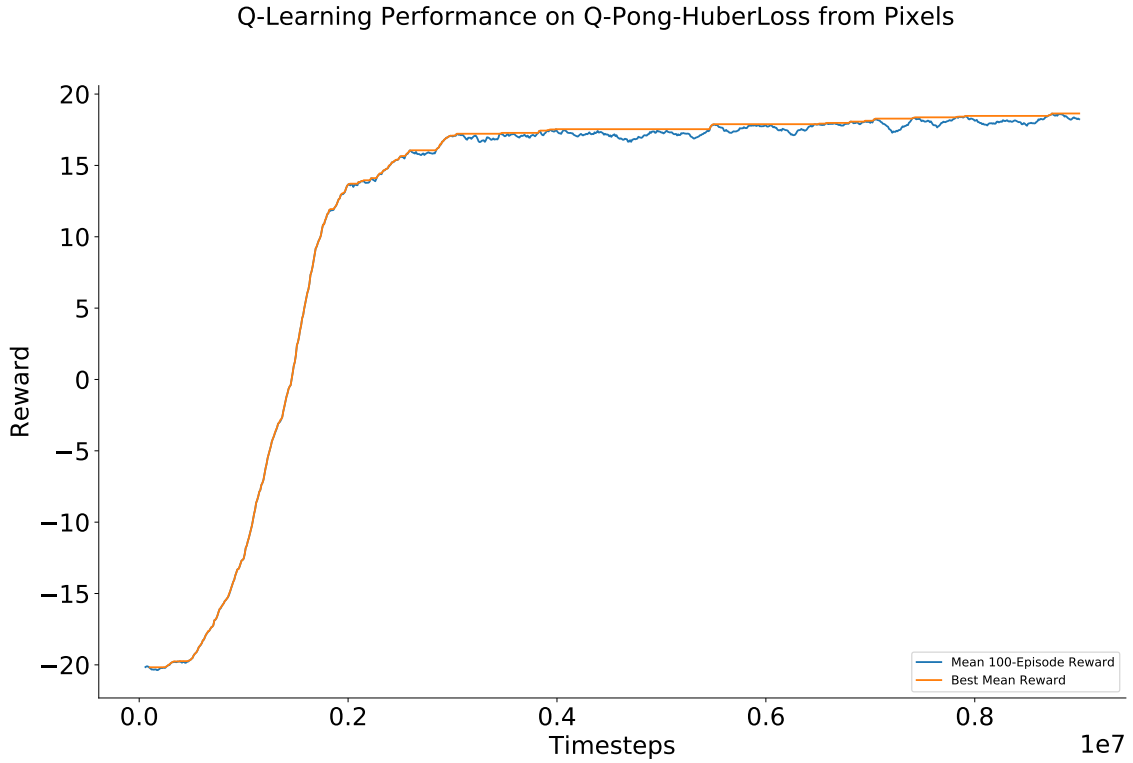

Figure 9: Mean and best reward average evolution for Atari game "Pong" using Vanilla Q-Learning with Huber loss as the error function during 9 million time-steps. This approach is able to achieve the theoretical performances indicated in the statement of the problem, reaching a final average return close to 20 units (winning almost all the time).

As can be seen, the implementation — even with MSE as a loss function — tend to be very stable in terms of best mean reward per episode and the current mean reward. In addition, an almost monotonic increasing pattern can be seen, improving the performance of our agent with every time-step. A significant improvement is achieved after the first 2 million time-steps, allowing our agent of being able to continuously win the game.

Model training and figures are generated using the following commands (changing the loss function in lines **205,206** inside the dqn.py file):

```
python run_dqn_atari.py
```

```
python Plotter.py --fileName=Q-PongPixel_HuberLoss
                  --plotName=Q-PongPixel_HuberLoss
python Plotter.py --fileName=Q-PongPixel_MSE
                  --plotName=Q-PongPixel_MSE
```

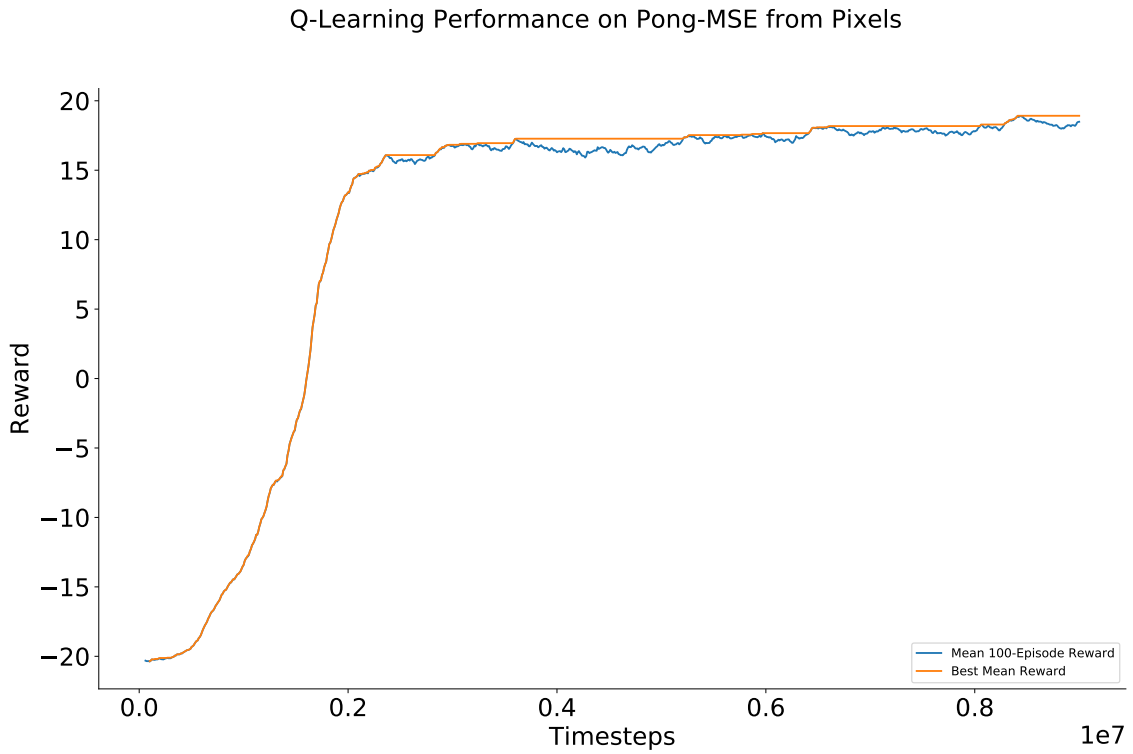

Figure 10: Mean and best reward average evolution for Atari game "Pong" using Vanilla Q-Learning with MSE as the error loss function during 9 million time-steps. This approach is able to achieve the theoretical performances indicated in the statement of the problem, reaching a final average return close to 20 units (winning almost all the time).

1.3 Q2: Double Q-Learning and Pong

Following the lecture slides and the discussion developed in Hado Van Hasselt et al. (1), we implement a Double Q-Learning approach in our original code, by taking the optimal actions from the current action-value function $Q_\phi(s_t, a)$ instead of the target network $Q_{\phi'}(s_t, a)$ allowing us to avoid the inherent overestimation of the Vanilla approach. From the results in Figure 11, we can see that in this case, both implementations tend to be very similar in terms of performance, converging to the same optimal value after a sufficient amount of iterations. However, interesting is to observe how the Double Q-Learning approach can be slower than the Vanilla implementation for this particular problem and random seed selected. Different random seeds were tested obtaining similar results but in the opposite direction: Double Q-Learning was able to obtain better performance than the Vanilla implementation.

However, this is not a characteristic pattern for this specific instance and thus, shows us the sensitivity of the method.

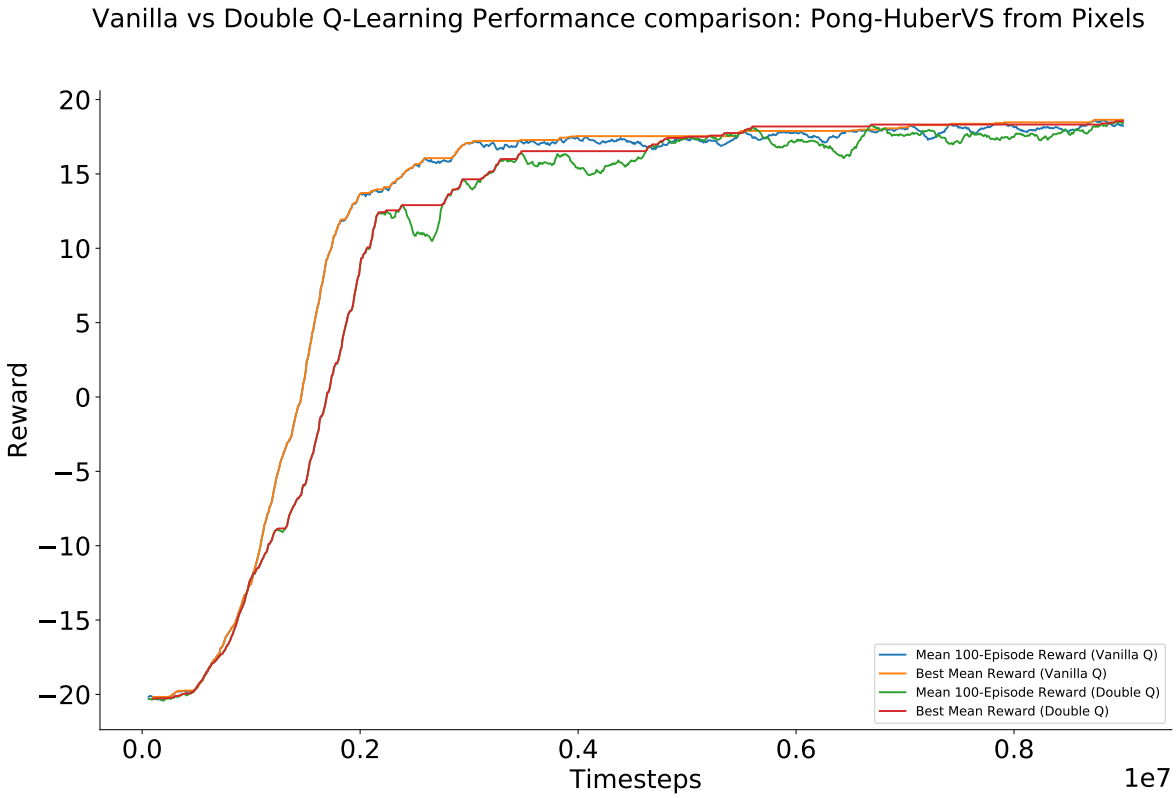

Figure 11: Double Q-Learning versus Vanilla Q-Learning implementations using Huber loss function performance comparison for the classic Pong game using Pixels within 9 million time-steps. From the results, we can observe that both implementations converge to the same solution but Double Q-Learning tends to be slower than the Vanilla implementation but reaches higher average returns by the end of the training.

Model training and figures are generated using the following commands (modifying **double_q** = **True** / **False** inside dqn.py, lines **35** or **188**):

```
python run_dqn_atari.py
```

```
python Plotter.py --fileName=DoubleQ-PongPixel_HuberLoss
                  --plotName=DoubleQ-PongPixel_HuberLoss
```

1.4 Q3: Hyper-parameters comparison

In order to study the sensitivity of our implementation to certain hyper-parameters, we test it with different — and fixed — learning rates $lr \in \{0.00001, 0.001, 0.005, 0.05, 0.1, 100\}$. We selected this hyper-parameter due to its critical impact in the performance of the algorithms: as we have already seen in previous assignments, different learning rates can lead to results varying from optimal/practical to unstable and intractable in terms of quality and solving time. Therefore, significant research efforts are currently focused on determining the best values/schedules/updating rules for this hyper-parameter, such that no hand-tuning or brute force approaches are needed to extract the best from the algorithms.

From the results in Figure 12 we can see how different results can be with different learning rates. When testing our fixed values, an explicit pattern can be observed: lr values greater or equal than 0.005 lead to significantly poor performance, not being able to improve our agent's returns at all during 4 million time-steps. On the other hand, a fixed $lr = 0.001$ (green line) obtains very similar performance to the Vanilla version (with the original lr optimal schedule), even beaten its performance by the end of the training period by a small margin while $lr = 0.00001$ seems to have a very slow convergence — as expected since it is very small — but with some improvement during the first 4 million time-steps (orange line).

For completeness, we include the best mean rewards curves in Figure 13, being able to observe the evolution of the best results for each learning rate.

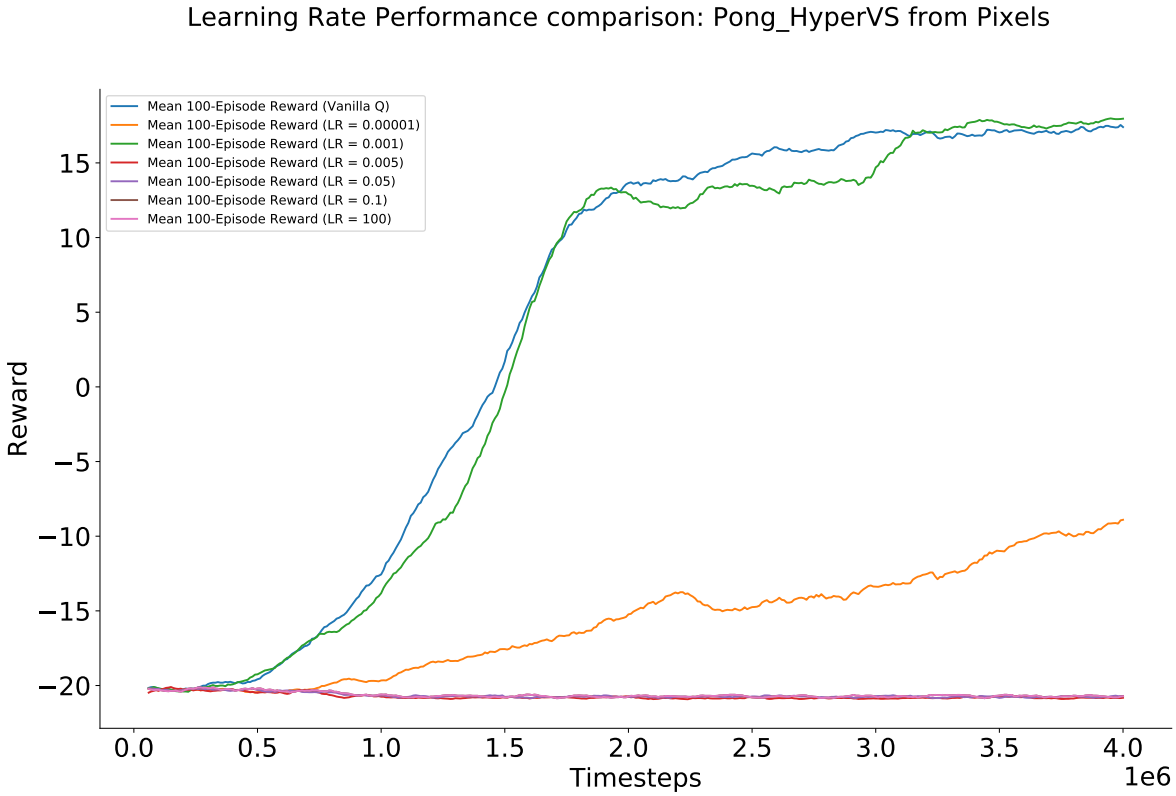

Figure 12: Different learning rates $lr \in \{0.000001, 0.001, 0.005, 0.05, 0.1, 100\}$ performance comparison for Pixels Pong. As can be seen, significant poorly performance is obtained without the originally optimized scheduled provided in the *run_atari_dqn.py* file when larger learning rates are applied. Good but less stable performance is achieved with a fixed $LR = 0.001$, obtaining similar average returns to the ones achieved with the Vanilla implementation and even slightly surpassing it after 3 million iterations. The impact of the different hyper-parameters value shows us how sensitive is our Q-learning implementation, requiring a significant amount of effort in order to tune the performance of the model.

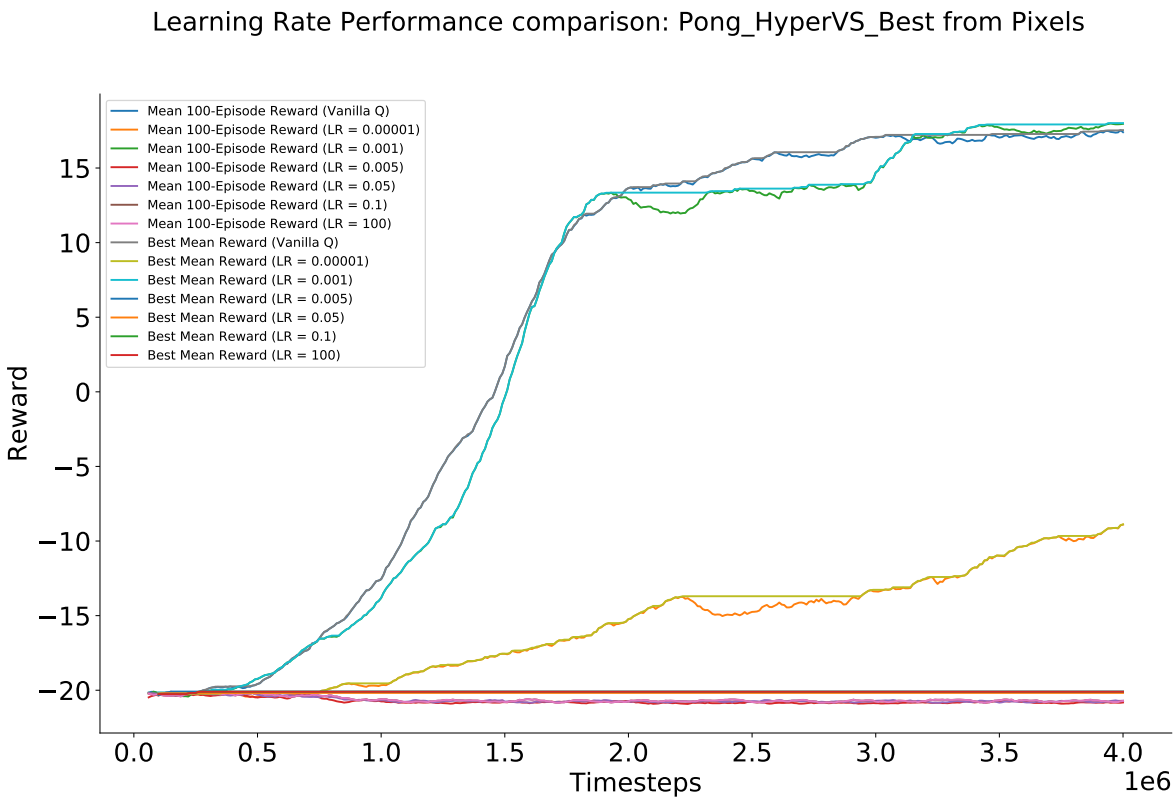

Figure 13: Different learning rates $lr \in \{0.000001, 0.001, 0.005, 0.05, 0.1, 100\}$ performance comparison for Pixels Pong including the best mean reward values.

Plots of this section are generated using the following commands (not adding the best_reward flag for Figure 12):

```
python PlotterVS_Hyper.py --file_name1=Q-Pong-HuberLoss
                          --file_name2=Q-Pong-LR000001
                          --file_name3=Q-Pong-LR0001
                          --file_name4=Q-Pong-LR0005
                          --file_name5=Q-Pong-LR005
                          --file_name6=Q-Pong-LR01
                          --file_name7=Q-Pong-LR100
                          --plot_name=Pong_HyperVS_Best
                          --best_reward
```

In addition, all experiments were performed by modifying the learning rate scheduling proposed inside the run_dqn_atary.py file (lines **38-43**) and simply running the script.

2 Problem 2: Actor-Critic algorithm

2.1 Q1: CartPole sanity check

In order to perform a sanity check of our Actor-Critic implementation, we run four different combinations of numbers of target updates and number of gradient steps for our critic network, using the following commands:

```
python train_ac_f18.py CartPole-v0 -n 100 -b 1000 -e 3
                       --exp_name CartPole1_1 -ntu 1 -ngsptu 1
```

```
python train_ac_f18.py CartPole-v0 -n 100 -b 1000 -e 3
                       --exp_name CartPole100_1 -ntu 100 -ngsptu 1
```

```
python train_ac_f18.py CartPole-v0 -n 100 -b 1000 -e 3
                       --exp_name CartPole1_100 -ntu 1 -ngsptu 100
```

```
python train_ac_f18.py CartPole-v0 -n 100 -b 1000 -e 3
                       --exp_name CartPole10_10 -ntu 10 -ngsptu 10
```

In addition, Figure 14 is generated via the following command:

```
python MultiPlot.py -re="CartPole" -o=Cartpole_All
```

In Figure 14 we can clearly see how the Vanilla implementation with only one target update and one gradient iteration is clearly the one with the worst performance: it is not able to improve the average return among all iterations, obtaining a poor average below 10. This pattern was expected since we are only updating the critic target values once and only performing one gradient step, not being able to capture the real (or close) value of the state function $V(s')$, leading to very poor results (lack of training of the critic NN).

On the other hand, best performance is achieved by the balanced configuration (10,10). This was expected since a balanced combination will allow the critic network to obtain good target values — representative among all time-steps — while training the critic NN for a decent amount of gradient descent steps, being able to learn the new values of ϕ in a reasonable way. This can be reflected in the performance of the other combinations 1,100 and 100,1. Looking at the results, we can see that this unbalance between updating the target values and the training of the critic NN leads to more erratic convergence patterns: we are giving too much priority to one step over the other, using very "old" (non representative) target values or a non-convergent neural network (worse estimations for ϕ parameters).

Therefore, we can conclude that in the current setting, a combination of 10 and 10 target updates and gradient steps obtains the best performance among our experiments.

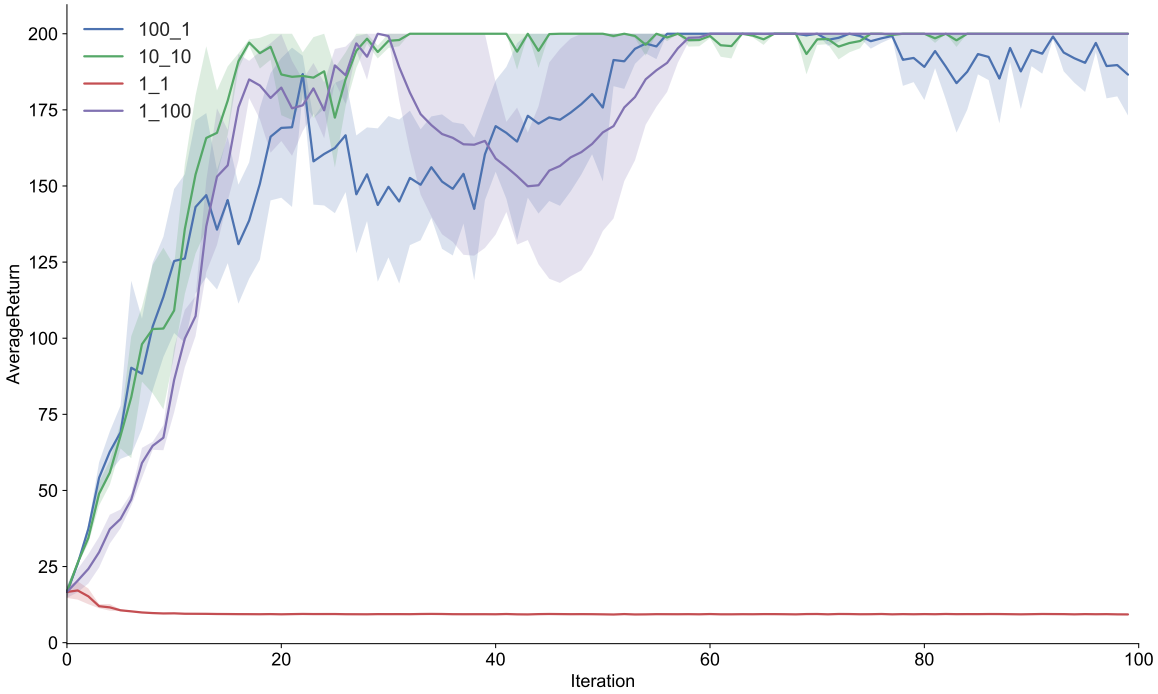

Figure 14: Performance comparison for different combinations of the number of updates of the targets and gradient step iterations for the CartPole instance. Best performance is obtained by the balanced configuration (10 and 10) while worst performance — as expected — is achieved by the 1 and 1 configuration. The explanation is clear: we need to find a balance between the number of training steps and the updates of the target values for the critic network in order to be able to capture the underlying values of the state function V. Otherwise, we have erratic patterns like with 1,100 and 100,1 combinations, where we are giving priority to only one of the relevant training steps, losing valuable information during some steps (as can be seen from the convergence pattern of these two combinations).

2.2 Q2: Inverted pendulum and Half-Cheetah

In order to test our Actor-Critic implementation with more difficult tasks, we proceed to run the InvertedPendulum and HalfCheetah environments.

Following the best combination found in the previous section (CartPole experiments), we perform the following experiments consisting of the classic InvertedPendulum and HalfCheetah instances, with 10 target updates and 10 gradient descent steps in a cycle (100 iterations in total when calling the critic's update function):

```
python train_ac_f18.py InvertedPendulum-v2 -ep 1000 --discount 0.95
                       -n 100 -e 3 -l 2 -s 64 -b 5000 -lr 0.01
                       --exp_name ip_10_10 -ntu 10 -ngsptu 10
```

```
python train_ac_f18.py HalfCheetah-v2 -ep 150 --discount 0.90
                       -n 100 -e 3 -l 2 -s 32 -b 30000 -lr 0.02
                       --exp_name hc_10_10 -ntu 10 -ngsptu 10
```

In order to include an interesting performance pattern, we provide results when we modify the initialization values of the logstd variables: with a vector of ones and a vector of zeros. From the results in Figures 15, 16, we can see that a similar performance to our policy gradient implementation is achieved, showing the usefulness and potential of the Actor-Critic approach allowing us to reduce the variance of the average rewards. In addition, better performance is obtained when initializing the logstd variables with zeros, reaching a more stable and faster convergence towards the theoretical optimum value.

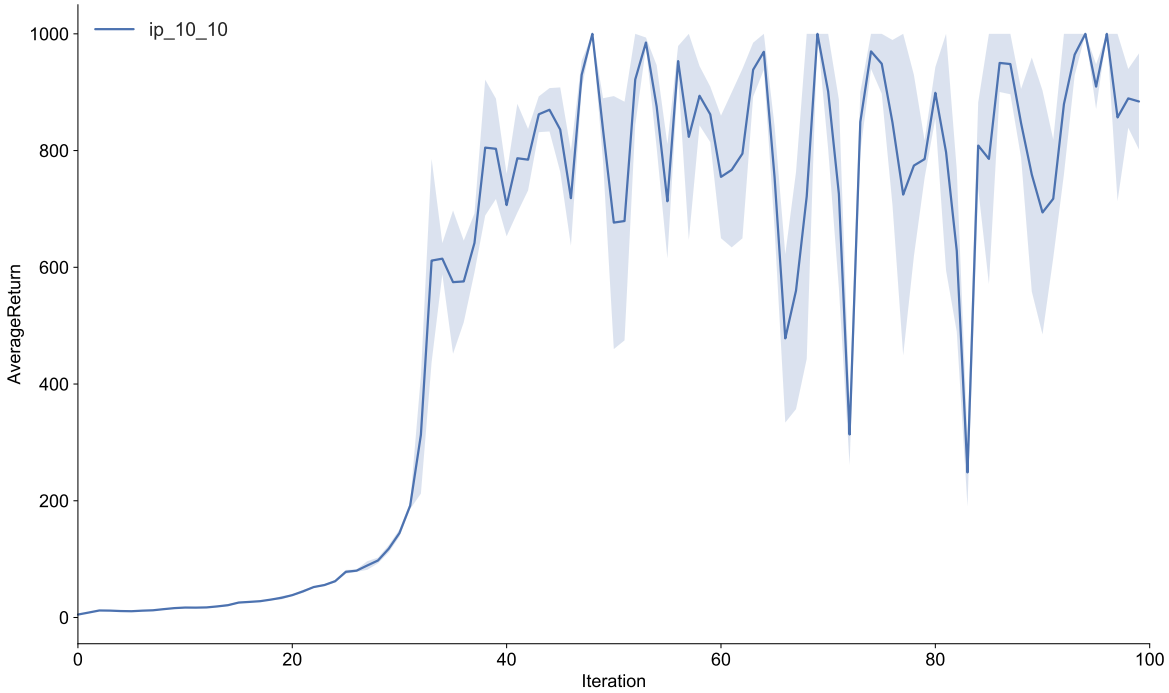

Figure 15: Inverted Pendulum average returns using the balanced combination of 10 target updates and 10 gradient descent steps and initializing the logstd values to ones. Similar performance to our previous policy gradient implementation is achieved while decreasing the variance of the returns.

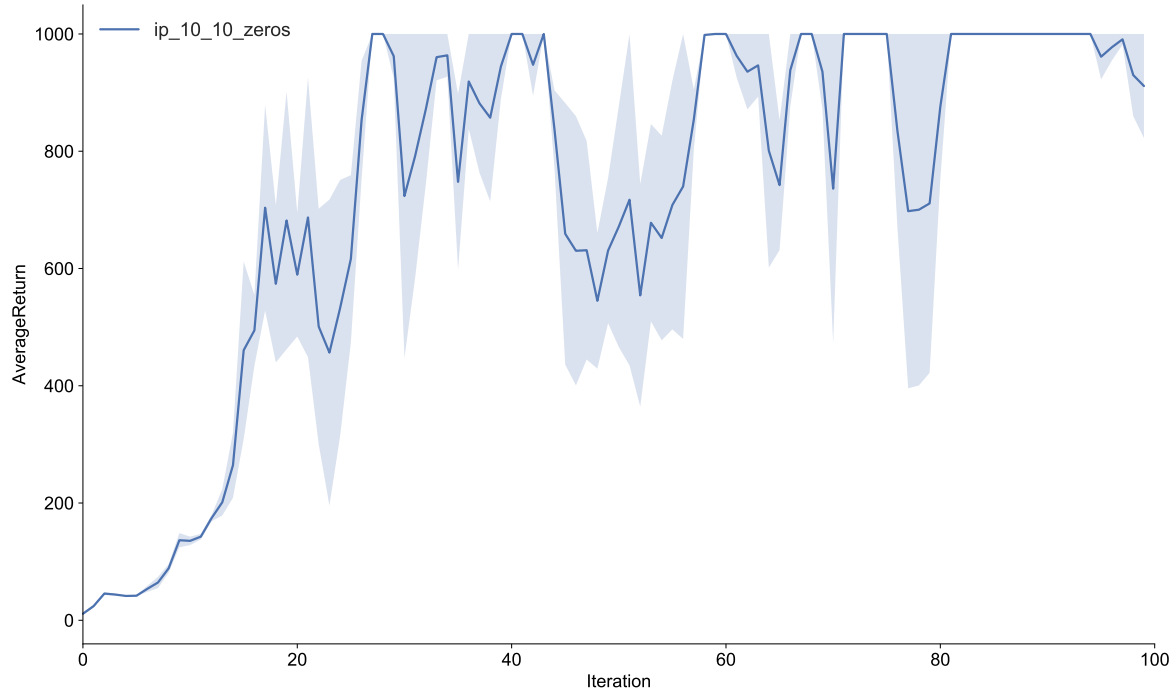

Figure 16: Inverted Pendulum average returns using the balanced combination of 10 target updates and 10 gradient descent steps and initializing the logstd values to zeros. Similar performance to our previous policy gradient implementation is achieved while decreasing the variance of the returns. Interesting is to observe a better performance than in the case where we initialize the logstd values with a vector of ones.

Looking at the results from Figures 17, 18, we can observe a similar pattern to the one present in the InvertedPendulum instance: performance tends to be very similar to the one obtained with our previous policy gradient implementation while at the same time, less variance is achieved for the returns. In addition, initializing the logstd variables with a null tensor achieves better results as with the InvertedPendulum instance.

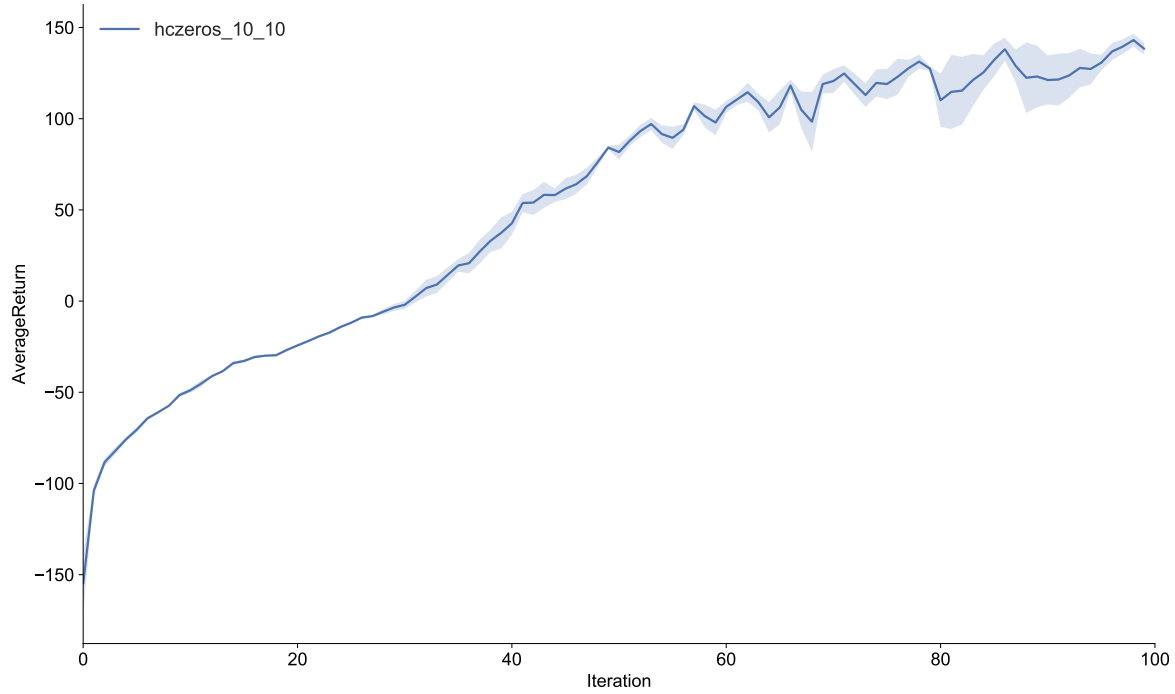

Figure 17: HalfCheetah average returns using the balanced combination of 10 target updates and 10 gradient descent steps. Similar performance to our previous policy gradient implementation is achieved while decreasing the variance of the returns (using zeros for initializing logstd variables).

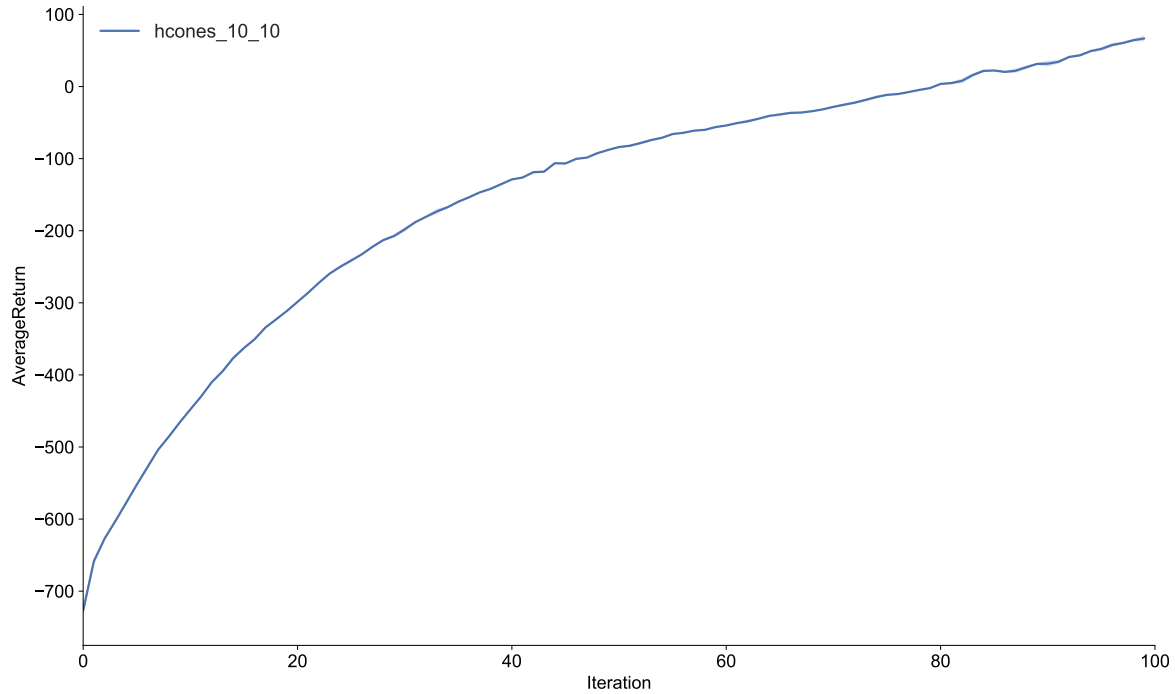

Figure 18: HalfCheetah average returns using the balanced combination of 10 target updates and 10 gradient descent steps. Worse performance to our previous policy gradient implementation is achieved while decreasing the variance of the returns (using ones for initializing logstd variables).

All Figures 15, 16, 17, and 18 were generated with the following code:

```
python MultiPlot.py -re="ip_10_10_ones" -o=IP_10_10
```

```
python MultiPlot.py -re="ip_10_10__zeros" -o=IPzeros_10_10
```

```
python MultiPlot.py -re="hc_10_10_ones" -o=HC_10_10
```

```
python MultiPlot.py -re="hc_10_10_zeros" -o=HCzeros_10_10
```

2.3 Bonus: value function $V(s)$ NN-Architecture and Hyper-parameters

In order to test different structures for the critic network as well as hyper-parameters such as the learning rate for its training, we test the following configuration with the InvertedPendulum and HalfCheetah instances by adding some extra lines inside our main code (see README.txt file):

- Learning rate is set to 0.001
- Number of hidden layers is set to twice the number of hidden layers in the actor-network.
- Number of neurons per hidden layer is set to twice the number of neurons per hidden layer in the actor-network.
- 20 target update iterations and 20 gradient descent steps are performed per cycle.

Thus, after modifying the internal values directly inside the original script and generating a copy with Bonus at the end, the following commands are executed for performing the experiments:

```
python train_ac_f18_Bonus.py InvertedPendulum-v2 -ep 1000
                             --discount 0.95
                             -n 100 -e 3 -l 2 -s 64 -b 5000 -lr 0.01
                             --exp_name ip_20_20_NN -ntu 20 -ngsptu 20
```

```
python train_ac_f18_Bonus.py HalfCheetah-v2 -ep 150 --discount 0.90
                             -n 100 -e 3 -l 2 -s 32 -b 30000 -lr 0.02
                             --exp_name hc_20_20_NN -ntu 20 -ngsptu 20
```

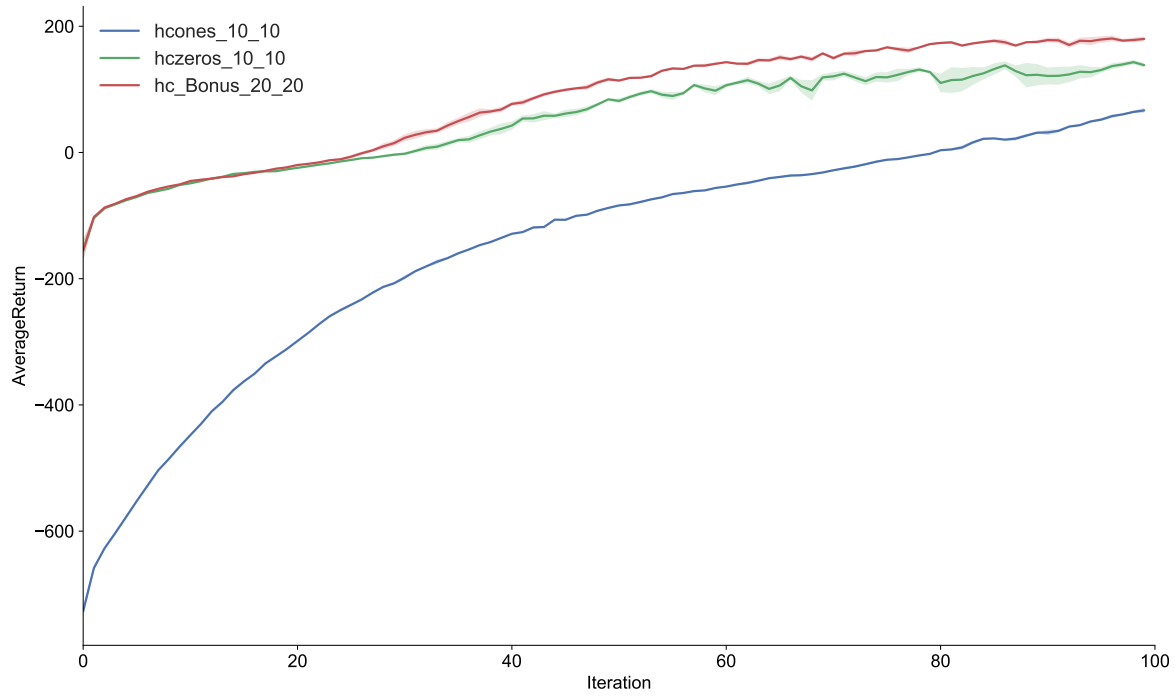

Figure 19: Performance comparison for the Half-Cheetah instance including the modified NN-architecture and hyper-parameters (red line). From the results, better performance is obtained by the modified net including 20 target update iterations and 20 gradient descent steps per cycle.

Note: Plots of this section are generated using the following commands.

```
python MultiPlot.py -re="hc_20_20_NN" -o=HC_20_20_bonus
```

```
python MultiPlot.py -re="ip_20_20_NN" -o=IP_20_20_bonus
```

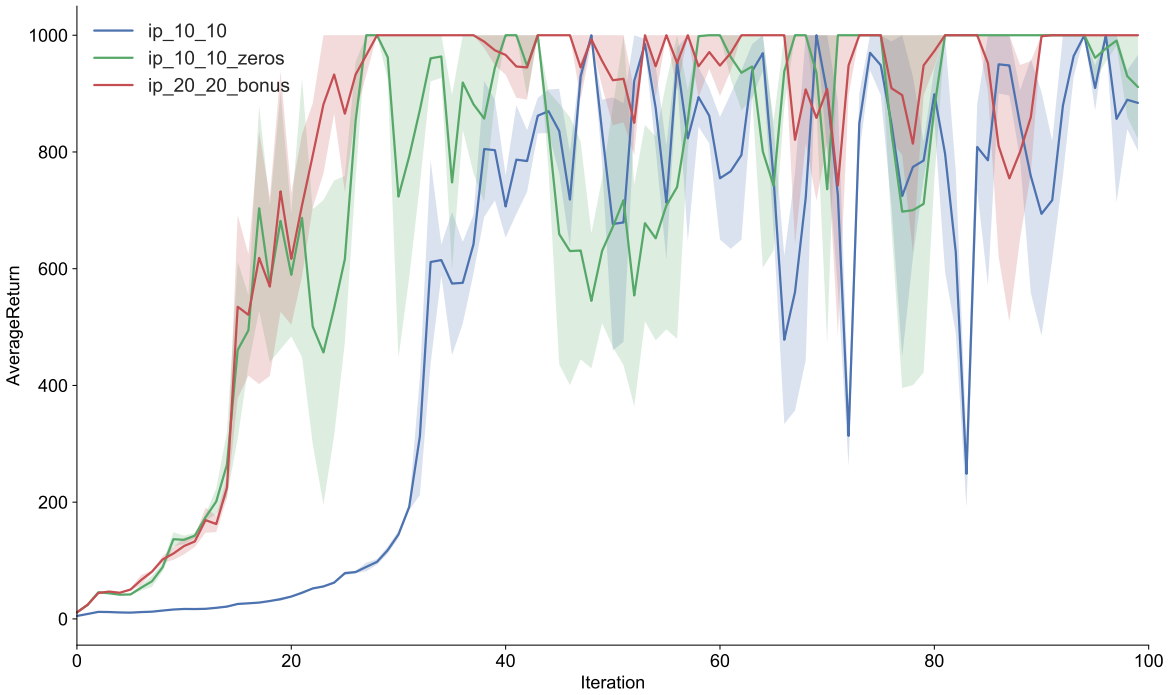

Figure 20: Performance comparison for the Inverted Pendulum instance including the modified NN-architecture and hyper-parameters (red line). From the results, better performance is obtained by the modified net including 20 target update iterations and 20 gradient descent steps per cycle, reaching the theoretical optimal value of 1000 early while experiencing less variance during the rest of the trajectory when comparing it with the previous best setting (green line).

As can be seen from the results, significant differences in performance can be obtained with different configurations of the Actor-Critic algorithm, being a real challenge for the researcher to explore, select, and justify his/her implementation.

References

[1] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In AAAI, volume 2, page 5. Phoenix, AZ, 2016.