Lab 2: Deep Reinforcement Learning for Cartpole

Elisa Bin
940330 4182
KTH Royal Institute of Technology
ebin@kth.se

Pier Luigi Dovesi
940724 4954
KTH Royal Institute of Technology
dovesi@kth.se

1 Formulation of the RL problem

The system we are considering consists of a cart on a frictionless track with a pole attached on a revolute join. We are going to describe the state of the system with 4 components: car position x, car velocity $\dot{x}$, pole angle θ and its derivative $\dot{\theta}$. Therefore $state = [x, \dot{x}, \theta, \dot{\theta}]$. Moreover, the action space is given by $A = \{l, r\}$, where l represent a movement on the left and r one on the right. Rewards are given by:

$$reward(state) = \begin{cases} 1, & \text{if } -12.5° \leq \theta \leq 12.5° \text{ and } -2.4m \leq x \leq 2.4m \\ 0, & \text{all other cases} \end{cases}$$

Defining the Reinforcement Learning problem in this way we obtain an infinite number of possible states and the state distribution is continuous. For this reason standard Reinforcement Learning methods are not applicable in this case and we will apply Deep Learning Reinforcement Learning techniques.

2 Algorithm Implementation

In the main we iterate through all the 1000 episodes and following, we initialize the environment and we iterate thorough the number of steps (DQN algorithm lines 1 and 2). In the function *get_action* we apply the $\epsilon - greedy$ policy (DQN algorithm line 3). We apply the action to the environment updating the next state and the reward (line 4). When we are calling *append_sample* the new experience will be appended to the memory queue that will automatically discard the oldest, once full (line 5). In the function *train_model* we sample from memory creating a *mini_batch* with the same dimension of the batch size (line 7). For the whole batch (line 6) we the compute predictions from the network values, *model* and the target network values *target model* (line 8, and line 9). Then we get the target value adding the reward to the discounted max values of the just computed network target values (line 10). Finally we train our network model using the just updated target values and the updated input for just one epoch (line 11). The network parameters are updated only every *target_update_frequency* iterations and to do that we use the function *update_target_model* (line 14). Once one episode ends we might check the if the game has been solved, for doing this we check the last 100 rewards collected in the episode, if the mean is over 195/200 the game is considered solved and the training interrupted (line 16).

3 Neural network layout

The network is a simple MLP, i.e. a feed-forward network with dense (fully connected) layers. The activation function is ReLU in the hidden layers and linear in the output layer (since it is a regression problem). The weights' initialization is He uniform. We tried both with one hidden layer and with two. The implementation is straightforward since with the Sequential model offered by Keras we just "stack" the layers one after the other. For the first hidden layer is necessary to specify the input

dimension, that in our case is the size of the state. In output layer instead the number of nodes has to mach the action dimension. Finally, the used loss in the MSE and as optimizer we use Adam.

4 Model architecture and hyperparameters search

In this section we are going to investigate the effect of different parameters on the learning performances.

4.1 Smoothing function

Only for plotting purposes we computed a parallel score computation imposing a strong smoothing factor on the score plot. We chose to implement a simple exponential smoothing with a factor $\alpha = 0.01$. This preserve the overall trends of the graph with a general score underestimation effect. In particular it makes hard to understand from the plot if the model is able to solve the game or not (especially for very fast models). We specify that the score we got using for this procedure is not the same used to classify a model in the Check Solving Condition section.

Exponential smoothing:

$$SmoothedScore_t = \alpha Score_t + (1 - \alpha) SmoothedScore_{t-1} \tag{1}$$

Where $SmoothedScore$ is the score with exponential smooting, t is the time-step (greater than 0), $score$ is the real score and $alpha$ is the smoothing factor.

4.2 Network architecture

First we try to modify the number of nodes keeping one layer network. We try 16, 32 and 64 nodes. Comparing the three graphs in fig. 1 we observe that the best performances are obtained with 64 nodes in the hidden layer. This is easily explainable thanks to larger number of available parameters that can better estimate the value function.

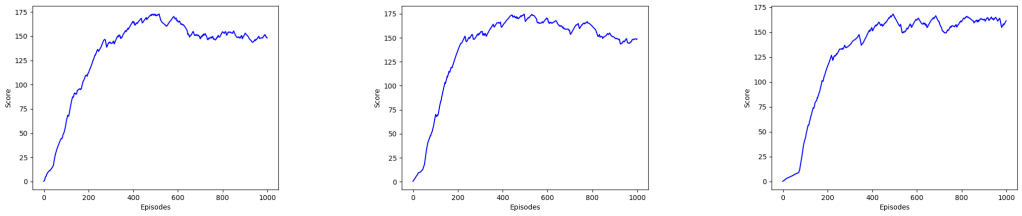

Figure 1: Scores with the following set of parameters: discount factor = 0.95, learning rate = 0.005, memory size = 1000 and target update frequency = 1 and different number of nodes for layer 1, in particular: 16, 32 and 64

Furthermore we investigate the possibility of increasing the number of hidden layers from one to two. This might allow the network to understand higher level features.Comparing the three plots in fig. 2 we observe how increasing the number of nodes in the second hidden layer results in worst performances therefore we select as better configuration so far the following: nodes for layer 1 = 64, nodes for layer 2 = 0, discount factor = 0.95, learning rate = 0.005, memory size = 1000 and target update frequency = 1. This might be due to an under-fitting problem since the learning might take too long before exploiting the advantage of the additional depth. Another hypothesis instead is that the deeper net could actually perform better, but a larger research of hyperparameters has to be adopted. In the end we will report results from an extensive grid search for the optimal parameters.

4.3 Discount factor

We compared the effect of performances of different values of discount factor in fig. 3. In particular, we tested $\lambda = 0.8$, $\lambda = 0.95$ and $\lambda = 1$ and we noticed that for bigger λ the score is increasing. In fact we got best performances for $\lambda = 1$ where we obtain a score value of almost 200. Discount factor equal to one means we are not providing any discount for longer term rewards, even if we were

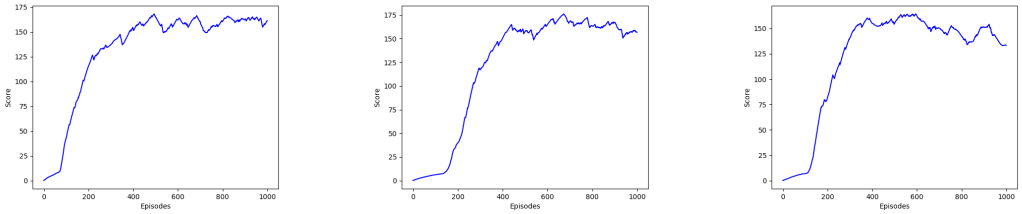

Figure 2: Scores with the following set of parameters: nodes for layer 1 = 64, discount factor = 0.95, learning rate = 0.005, memory size = 1000 and target update frequency = 1 and different number of nodes for layer 2, from left to right: 0, 8 and 16

expecting a performance degradation, it seems that providing high values also to long term rewards has an overall positive effect on the learning. In the last section we will show other possible discount factor alternatives.

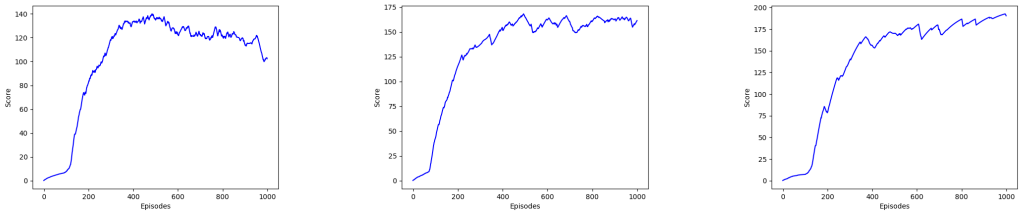

Figure 3: Scores with the following set of parameters: nodes for layer 1 = 64, nodes for layer 2 = 0, learning rate = 0.005, memory size = 1000 and target update frequency = 1 and different discount factor, in particular: 0.8, 0.95 and 1

4.4 Learning Rate

For the learning rate η,in fig. 4 we compared the initial value of $\eta = 0.005$ with $\eta = 0.01$ and $\eta = 0.001$. From the plot we observe a significant increase in performances with smaller learning rate. This is probably due to to the fact that with a smaller learning rate it is less likely to immediately find sub optimal local minima. Moreover, for small batch sizes (like 32), the learning rate need to be small enough to avoid noisy behavior.

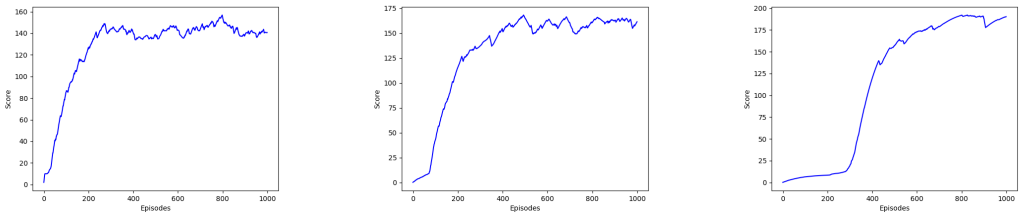

Figure 4: Scores with the following set of parameters: nodes for layer 1 = 64, nodes for layer 2 = 0, discount factor = 0.95, memory size = 1000 and target update frequency = 1 and different learning rate, in particular: $\eta = 0.01$, $\eta = 0.005$ and $\eta = 0.001$

4.5 Memory Size

Furthermore, in fig. 5 we compared the performances of the network with different memory buffer sizes. In particular, we tried memory size 1000, 2000 and 5000. We observe that the obtained score is comparable, but increasing the memory size, generally, has a positive effect on the performances. The slight beneficial effect might be due to the larger amount of data available for the model during the sampling in the training.

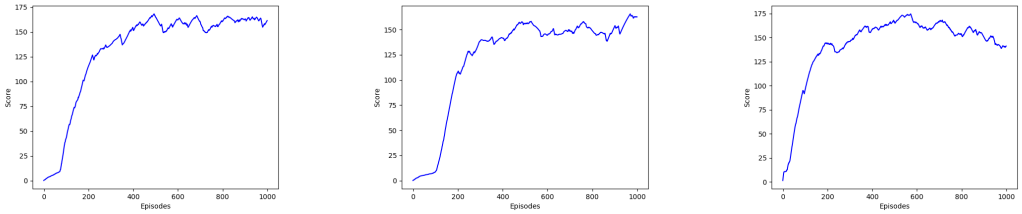

Figure 5: Scores with the following set of parameters: nodes for layer 1 = 64, nodes for layer 2 = 0, discount factor = 0.95, learning rate = 0.005, target update frequency = 1 and different memory size, in particular: 1000, 2000 and 5000.

4.6 Target Update Frequency

We investigated the impact on the performances of the variable $target\_update\_frequency$ (inversely proportional to how often the target network is updated) in fig. 6. In particular, we tried to update the target network each episode, every 50 and every 100 episodes. As expected, we observe a drastically decrease in performances with the decreasing of the update-frequency, indeed we obtain the best result by updating the target at each episode.

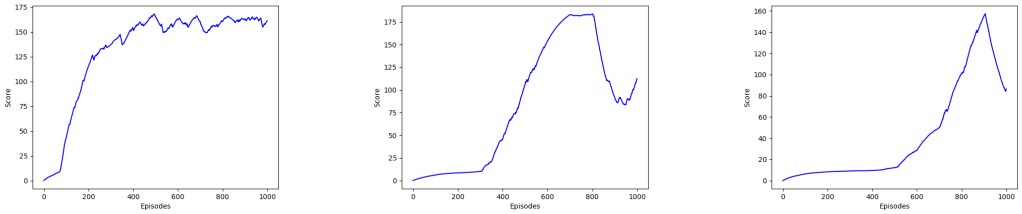

Figure 6: Scores with the following set of parameters: nodes for layer 1 = 64, nodes for layer 2 = 0, discount factor = 0.95, learning rate = 0.005, memory size = 1000 and different target update frequency, in particular: 1, 50 and 100.

5 Check Solving Condition

Taking into account the previous analysis and also performing an extensive grid search to discover other combinations, we choose the best configurations of hyperparameters and we checked whether they actually solve the problem. In fig. 7, fig. 8, fig. 9 and fig.10 the 4 fastest configuration we found. As we can see, the grid search highlight that many different possible configuration are able to effectively "solve the game" even if with very different trends. In particular we notice that the combination with two hidden layers: 64, 32 seems very effective.

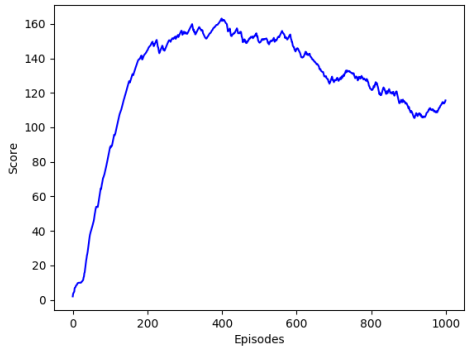

Figure 7: Scores with the following set of parameters: nodes for layer 1 = 64, nodes for layer 2 = 32, discount factor = 0.95 learning rate = 0.005, memory size = 1000 and target update frequency = 1. Problem solved after -100 episodes.

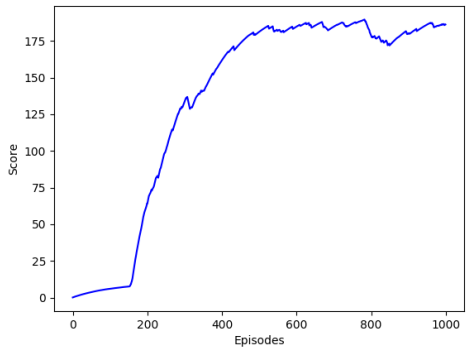

Figure 8: Scores with the following set of parameters: nodes for layer 1 = 64, nodes for layer 2 = 32, discount factor = 0.99 learning rate = 0.005, memory size = 4000 and target update frequency = 1. Problem solved after 376 episodes.

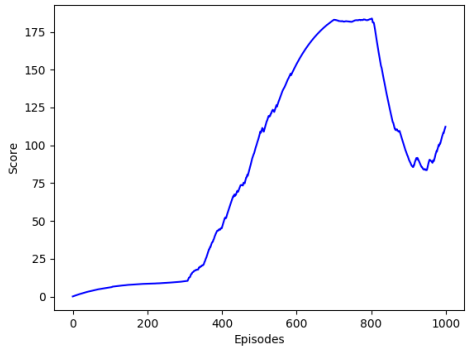

Figure 9: Scores with the following set of parameters: nodes for layer 1 = 64, nodes for layer 2 = 0, discount factor = 0.95 learning rate = 0.005, memory size = 1000 and target update frequency = 50. Problem solved after 540 episodes.

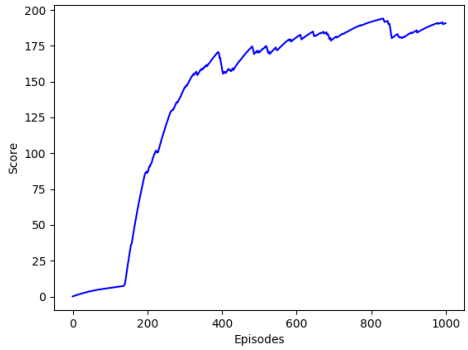

Figure 10: Scores with the following set of parameters: nodes for layer 1 = 64, nodes for layer 2 = 32, discount factor = 0.99 learning rate = 0.005, memory size = 2000 and target update frequency = 1. Problem solved after 568 episodes.

6 GitHub Code

Link to GitHub repository: reinforcement_lab_2