Reinforcement Learning Lab 2

Oleguer Canal	Federico Taschin
oleguer@kth.se	taschin@kth.se
19950727-2871	19960614-4674

December 2019

Question A)

In order to formulate the problem as a Reinforcement Learning problem, we need to define the states, transitions, rewards, and actions.

- **States**: We define a state s as a tuple $s = (x, \dot{x}, \theta, \dot{\theta})$ where x and $\dot{x}$ are respectively the distance of the cart from the center and its derivative, and θ and $\dot{\theta}$ are respectively the pole angle -w.r.t. the vertical line- and its derivative. Note that this state space is a subset of $\mathbb{R}^4$ and therefore continuous.
- **Actions**: We have only two possible actions $a \in \{-1, +1\}$ which apply respectively -1 and +1 Newtons of force to the x axis of the cart.
- **Rewards**: Rewards depend only on the state: $R_s = +1$ if $|x| < 2.4m$ and $|\theta| < 12.5^o$, $R_s = 0$ otherwise.
- **Transitions**: The problem is deterministic, therefore the transitions depend only on the state and the action taken. From a given state $s = (x, \dot{x}, \theta, \dot{\theta})$ we go with probability 1 to state $s' = f(s, a)$ where $f(s, a)$ represents the laws of motion.

As we defined it, this problem can not be solved with traditional Reinforcement Learning techniques. Both the state-value function $V(s)$ and the action-state-value $Q(s, a)$ need to be computed for each state s, but here we have an infinite number of states. One solution could be to discretize our state space, but this would mean losing precision. Moreover, increasing the density of the space would exponentially increase the complexity of the problem. What we need is a way to evaluate $Q(s, a)$ directly from the state. Deep learning turns out to be very useful in this, as it allows to approximate $Q(s, a)$.

Question B)

Algorithms 1 and 2 show the most important steps of the training. Here we briefly analyze the code and helper functions.

__main__() Here we perform the training of the agent. Pseudocode can be seen in Algorithm 1. It runs a certain number of episodes. For each of these episodes, it applies actions given by $get\_action(s)$ to the environment, and collects rewards and new states, saving them in memory. Note that memory is a queue that keeps only the last $memory\_size$ samples. After collecting a sample, it trains the agent's learning model by calling $train\_model()$, whose pseudocode can be seen in Algorithm 2. The total score of each episode is saved and plotted at the end. Moreover, a sample of random state-action values -generated at the beginning by taking random actions- is stored and updated with the values computed in the training, and plotted at the end. This way we can see whether the Q function is converging or not.

DQNAgent() This is the initialization of the DQNAgent with the state and action sizes and its default parameters. Among the most important we have:

- **Discount factor**: determines the weight that future rewards have in the value of the current state.
- ϵ: Probability of taking a random action in an ϵ-greedy policy.
- **Memory size**: Number of samples (s, a, r, s') that will be kept in memory.
- **Train start**: We will not train the agents models until we have this number of samples in memory.
- **Target update frequency**: Determines after how many episodes we update the agent's target model with the learning model.
- **Batch size**: Number of samples to use when fitting the models. Will be discussed in more detail in Question D).
- **Learning rate**: Scale for step taken by optimizer when fitting the models. Will be discussed in Question D).

It calls $build\_model()$ to create the target and learning neural networks of the agent.

train_model() In this function we train the learning network. First, we take $batch\_size$ random samples $x = (s, a, r, s')$ from memory. Then, for each sample x, we predict $Q(s, a)$ with both the learning and the target network. The goal is to make the learning network converge to the optimal Q values. We then fit by giving the batch states as input and the $target$ as supervised output. Initially however, for each sample in the batch, the code just assigns a random value to the sampled actions -not very useful. We will improve this in Question C).

build_model() This function is called in the initialization of the agent and creates the target network and learning network by using Tensorflow and Keras. We will describe this in detail in question D).

update_target_model() The only purpose of this function is to set the agent's learning network weights to the agent's training network.

get_action() This function returns an action for the given state. Initially the code just returns a random aciton, but we will improve this in Question E).

append_sample() Helper function that appends a sample (s, a, r, s') to the memory.

Algorithm 1: __main__

```
Initialize environment
Initialize DQNAgent
for each episode e do
    s = initial state
    while not done do
        Apply a = get_action(s)
        Collect (s, a, r, s') and append it to memory
        Train agent's learning model with train_model
        s = s'
    end
    if e% update frequency == 0 then
        Update agent's target model with learning model
    end
end
```

Algorithm 2: train_model

```
Data: memory, batch_size
batch = batch_size random samples from memory
target = prediction from learning model
target_val = prediction from target model
for each target[i] do
    target[i] = random value
end
Fit learning model with target values
```

Question C)

In DQN we aim to approximate the Q value of a state-action pair by using deep learning. From the definition of Q^π

$$Q^\pi(s, a) = \mathbb{E}\left[r + \lambda Q^\pi(s', a') | s, a\right] \tag{1}$$

and the definition of Q^*

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \lambda \max_{a'} Q^*(s',a')|s,a\right] \tag{2}$$

we can then represent Q^π with a deep Q-network with weights $\mathbf{w}$ such that $Q^\pi(s,a) \approx Q(s,a,\mathbf{w})$ and define a mean-square loss function

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}\left[\left(r + \lambda \max_{a'} Q(s',a',\mathbf{w}) - Q(s,a,\mathbf{w})\right)^2\right] \tag{3}$$

However, some problems arise. First, training samples are not independent since they're sequential. Moreover, since policy can sharply change with small variations of Q, the Q values will oscillate. To solve the first problem, we take a random subset of the memory and use it for training after each action. To solve the second instead, we will keep the first Q term in Eq. 3 fixed. This leads us to the use of two networks: the first is the target network Q_t, the second the learning network Q_l. Eq. 3 becomes:

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}\left[\left(r + \lambda \max_{a'} Q_t(s',a',\mathbf{w}) - Q_l(s,a,\mathbf{w})\right)^2\right] \tag{4}$$

After each action, we train the learning network to match the term $r+\lambda \max_{a'} Q_t(s',a',\mathbf{w})$. Then, we will periodically update Q_t with the (trained) Q_l, and go on with the new episodes. In this implementation, we use two networks that take the state as input and return the values for each action as output. The algorithm is therefore:

Algorithm 3: DQN

Initialize memory M
Initialize networks Q_l and Q_t
for *episode* $e = 1$ *to* N **do**
  $s =$ initial state
  while *episode not finished* **do**
    $a =$ get_action(s)
    Perform action a and get reward r and new state s'
    store (s, a, r, s') in M
    Sample minibatch (s_i, a_i, r_i, s'_i), $i = 1$ *to* n from M
    Initialize targets t_i, $i = 1$ *to* n
    $t_i = \begin{cases} r_i & if\ s_i\ is\ last\ state \\ r_i + \lambda \max_a Q_t(s_i, a) & otherwise \end{cases}$ Train Q_l with the minibatch and loss $(t - Q_l(s, a))^2$ for all samples
  end
  Every k episodes, set $Q_t = Q_l$
end

Question D)

The model we instantiate is Sequential, which means that each layer added to the model will be sequentially appended to the network. The first layer is a Dense layer with input size of 4, 16 units, ReLU activation function and uniform initialization of weights. A Dense layer performs the operation $\sigma(\mathbf{W}\mathbf{x})$ where σ is the activation function, $\mathbf{W}$ the weight matrix, and $\mathbf{x}$ the input. In our case, σ is the ReLU activation function, defined as

$$ReLU(x) = \begin{cases} x & if\ x > 0 \\ 0 & otherwise \end{cases} \tag{5}$$

The weight matrix $\mathbf{W}$ is a 16 x 4 matrix initialized randomly from an uniform distribution. We then add an output Dense layer of size 2 that uses a linear activation function. We then compile the model by specifying some training parameters: the loss and the optimizer. The loss function is the Mean Square Error, consistent to our results of Eq. 4. The optimizer is Adam (Adaptive Moment) with learning rate initially set to 0.005. This network therefore takes a state $s = (x, \dot{x}, \theta, \dot{\theta})$ as input and outputs two values corresponding to $Q(s, +1)$ and $Q(s, -1)$, the Q values for the two possible actions.

Question E)

An ϵ-greedy policy is a policy that greedily chooses the best action with probability $(1 - \epsilon)$ and a random action with probability ϵ. Algorithm 4 shows the

pseudocode of the get_action() function. The greedy choice corresponds to taking the prediction of the Q_l network for the state s and chosing the action that has the maximum predicted value.

Algorithm 4: get_action

Data: Network Q_l, State s
if $U(0,1) > \epsilon$ **then**
  return $\arg\max_a Q_l(s)$ // Return action that maximizes prediction of $Q_l(s)$
else
  return random{-1, +1}// Return random action
end

Question F)

Algorithm 5 shows the pseudocode of the *train_model*() function. In line 1 we draw N samples (s_i, a_i, r_i, s'_i) from the memory. Then, in lines 2 and 3 we prepare the input data structures for the two networks. *states* contains each state s_i of the samples, while *new_states* contains each s'_i of the samples. In lines 4 and 5 we use the two networks Q_l and Q_t to predict the values for each action for *states* and $new_s tates$ respectively. Lines 6, 7, and 8 were not provided in the code. Here we set, for each state s_i that we observe in the samples, its correct value, which is $r_i + \lambda \max_a Q_t(s'_i, a)$. We take care of checking whether a sampled state s_i was the last one in its episode, and in that case we do not add $\lambda \max_a Q_t(s'_i, a)$. Finally, in line 9, we fit the Q_l network by giving the *states* as input and the correct *learning_values* as target.

Algorithm 5: train_model

Data: Memory M, Batch size N, Target net Q_t, Learning net Q_l
1 Randomly draw N samples (s_i, a_i, r_i, s'_i) from M
2 $states = [s_1, s_2 \ \dots \ s_N]$
3 $new\_states = [s'_1, s'_2 \ \dots \ s'_N]$
4 $learning\_values = Q_l(states)$
5 $target\_values = Q_t(new\_states)$
6 **for** $i = 1$ *to* N **do**
7 $learning\_values[i][a_i] = r_i$
8 **if** s_i *was not the last state in the episode* **then**
9 $learning\_values[i][a_i] += \lambda \max_a target\_values[i]$
10 **end**
11 **end**
12 Q_l.fit(*states*,*learning_values*)

Question G)

In order to assess the effect of different model architectures, we trained the agent using different layouts. The single hidden layer models tested follow a structure as depicted by figure 1. To simplify notation, we will encode these models by the number of hidden units. For example d16d2: meaning the architecture is composed by a dense layer of 16 units, followed by an output layer of 2 units (the two possible actions).

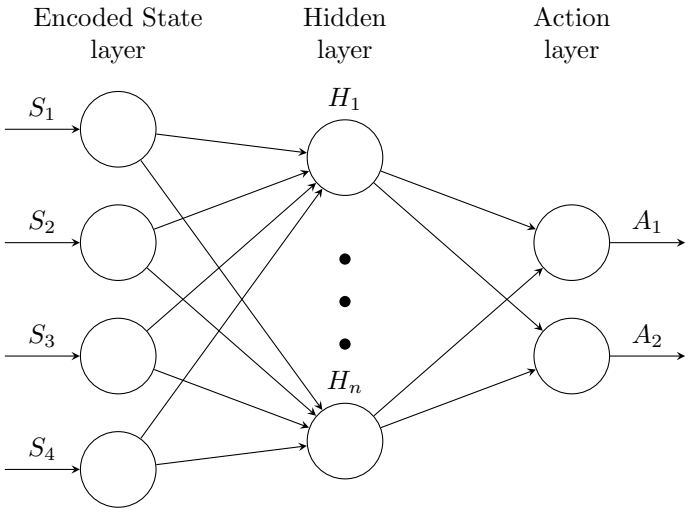

Figure 1: Architecture of the single hidden layer models implemented. S_1 to S_4 represent the 4 parameters that encode a state (input) and A_1, A_2 in the action (output) layer represent the 2 possible actions to take.

Similarly, we also tested models with different number of hidden layers and units per layer. Such models follow the same structure described in figure 1, but adding multiple hidden layers. To refer to these kind of models we will encode them as, for example d8d16d8d2: Meaning a that the model presents 3 hidden dense layers of, respectively, 8, 16, and 8 neurons. Followed by the 2 units output layer.

In particular, in this experiment we tested the models (sorted from simpler to more complex): **d4d2**, **d16d2**, **d64d2**, **d8d16d2** and **d8d16d8d2**. For this test, we used the same meta-parameters for each of the models. That is: setting a discount factor (df) of 0.95, a learning rate (lr) of 0,001, a memory size (ms) of 10000 and 10000 training episodes. Additionally, to make the interpretation of scores easier, we smoothed the obtained scores using an exponential moving average with a factor of 0.995. Figure 2 shows the results of these experiment. As it can be inferred, the higher the complexity, the better the results. For

single hidden layer models (**d4d2**, **d16d2** and **d64d2**) we can clearly see how augmenting the number of hidden units helps achieve higher scores at the cost of more training time. It is important to notice, though, that training time is not usually considered a constrain. As for adding more hidden layers we can say that, in this environment, the performance achieved is similar to the best models of a single layer. Therefore, on next sections we are going to stick with a simpler architecture and only consider single layer models. Figure 3 shows the smoothed Q function value convergence. While there are not substantial differences among the different architectures it can be inferred that simpler models tend to give higher average maximum Q values. In particular we decided to move forward using a **d32d2** model. As we expect it, with the right parameters, to be able to solve the given problem.

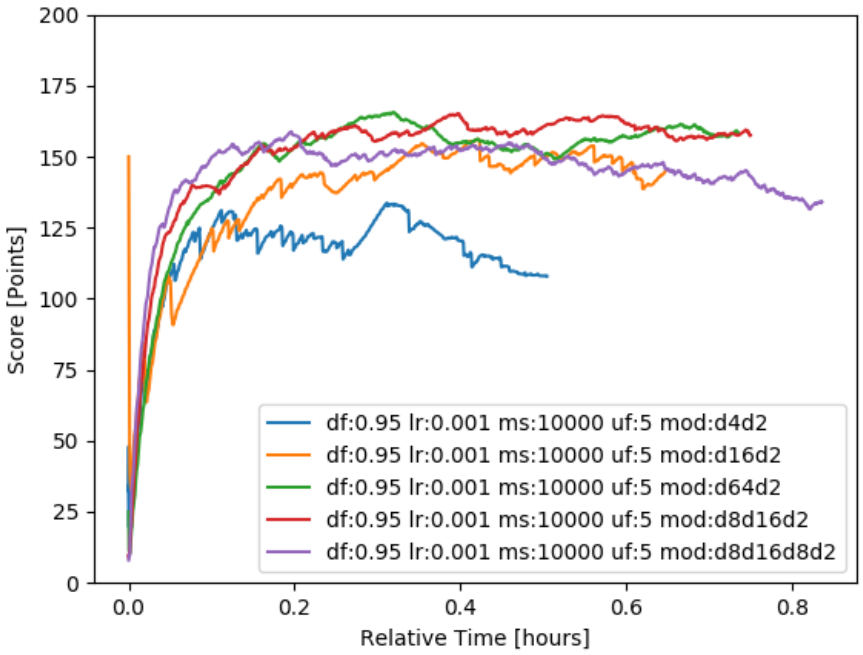

Figure 2: Smoothed score values obtained in the test of different model architectures. Notice how the more complex the model is, the faster it learns but the more time it needs. After 1000 iterations we could say that the difference between the four most complex is not significant.

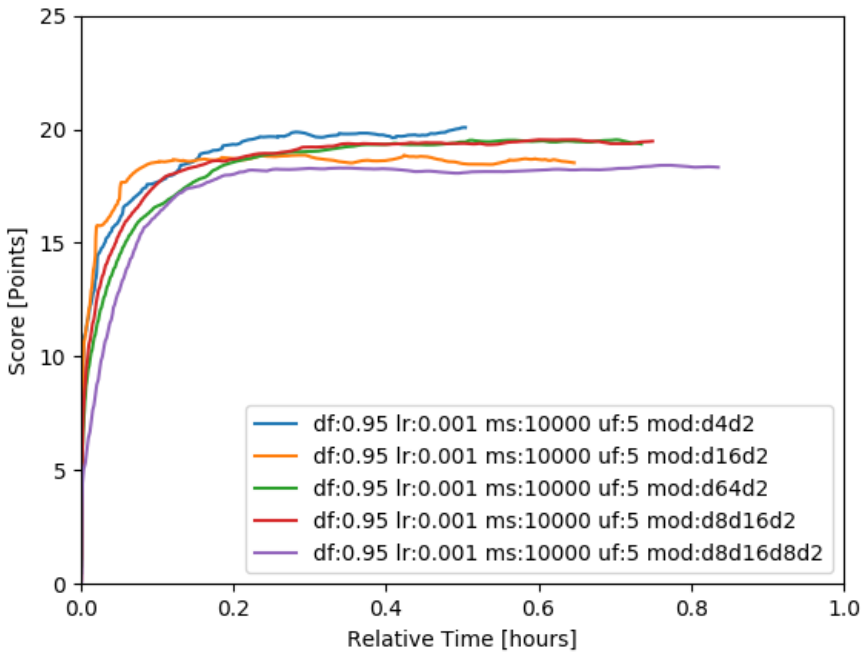

Figure 3: Smoothed maximum q mean values obtained in the test of different model architectures.

Question H)

To estimate the effect of each parameter and the correlation among them (if any) we used a grid search algorithm to test all possible combinations of parameters. To do so, we picked 3 values for each parameters to test and used them to train the **d32d2** for 10000 episodes, obtaining a total of 27 experiments. The chosen values for each parameter are:

- Discount factor: $\mathbf{df} \in \{0.8, 0.9, 0.99\}$
- Learning rate: $\mathbf{lr} \in \{0.001, 0.005, 0.05\}$
- Memory size: $\mathbf{ms} \in \{1000, 10000, 100000\}$

Figures 4, 5 and 6 show the obtained scores for every parameter combination with respect to, respectively, the discount factor, the learning rate, and the memory size. Again, to ease interpretation, we smoothed the results using the exponential moving average algorithm with a smoothing factor of 0.995.
We can now analyze the effect and correlations of all the meta-parameters:

- **Discount factor**: Figure 4 clearly shows how a discount factor of 0.99 outperforms lower values of this variable. We can affirm, thus, the higher the discount factor, the higher scores will be reached. This means that, in this particular problem and structure, we need to give a high weight to future predictions. It is also important to mention that this parameter, seems to be the main cause of different training times. As we can see in the figure, the larger the discount factor, the slower the training will be. Interestingly, we can see that the larger the learning rate, the poorer the model performs, specially for low discount factors. We can see the correlation between these two parameters by comparing horizontally the plots. This is because a low discount factor implies the value of future states has less weight and high learning rate introduces larger steps in the stochastic gradient descent, meaning that it is harder to achieve model convergence.

- **Learning Rate**: Figure 5 depicts the effect the learning rate has on the model training. In this case, the lower the leaning rate, the higher scores will be achieved and the more stable they will be. Nevertheless, the training will be slower and usually, will take more time to get to stationary performance. Here we can see again the relationship mentioned between discount factor and learning rate. It is important to notice that if discount factors are low, selecting low learning rates improves the model performance much more significantly compared to high discount factors, where all the models already perform similarly.

- **Memory size**: Figure 6 shows the effect of using different memory sizes to train the models. It is easy to see how this parameter, while generally better for a value of 10000, does not seem to make a big difference on the training. This can be explained by the fact that at each time-step we are only sampling 32 states from the 1000 to 100000 available. Remembering older states does not add valuable information.

We can finally conclude that to achieve the highest scores as fast as possible we need to pick a high discount factor, a low learning rate and the memory size role is not as critical, but a value of around 10000 will generally work better.

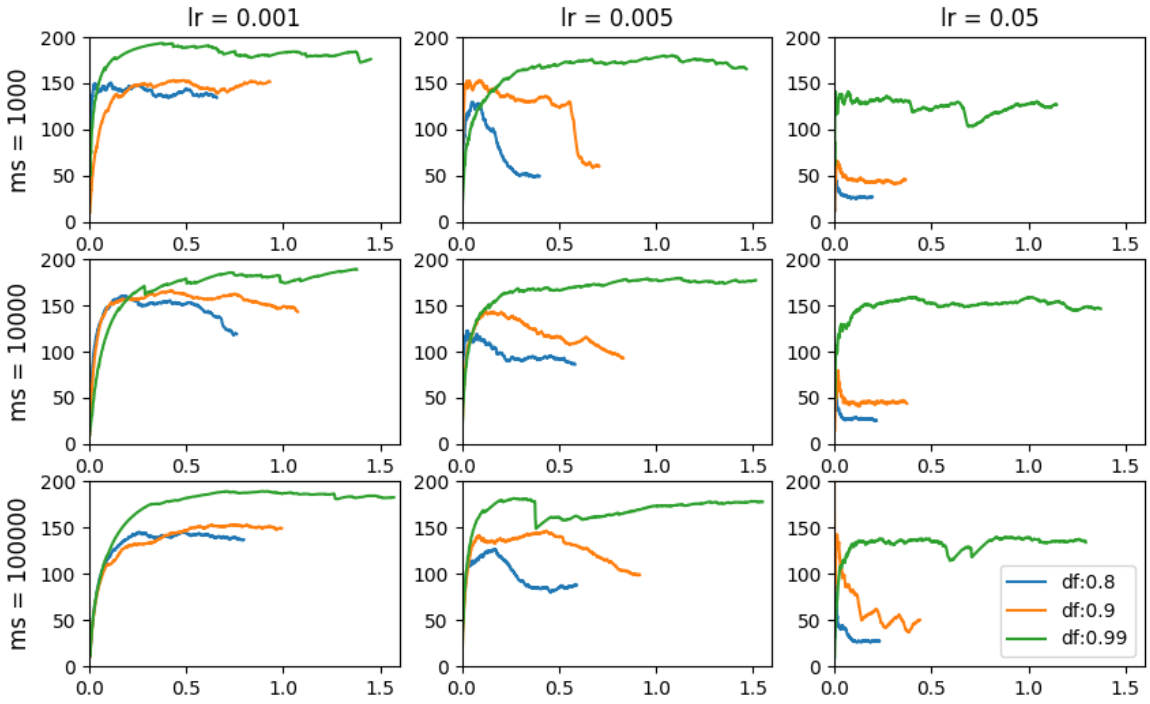

Figure 4: Test on the effect of the different discount factors (df) against each of the other parameters. Notice how, the higher the discount factor, the slower the learning, but the higher scores achieved.

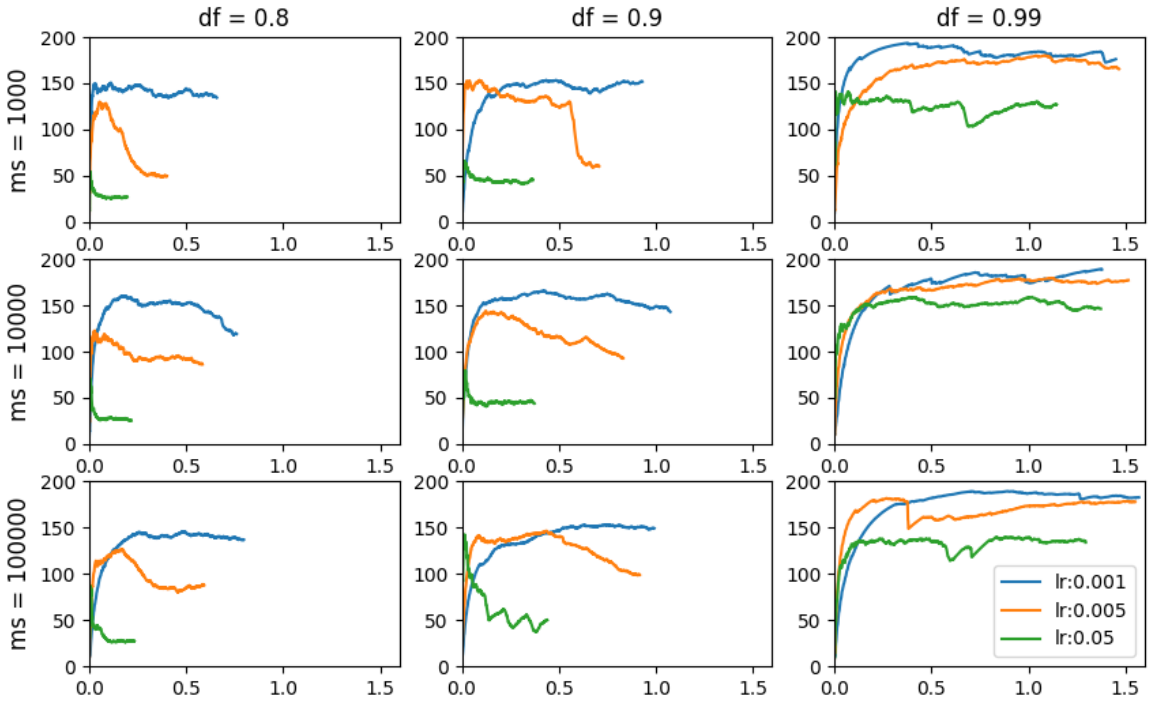

Figure 5: Test on the effect of the learning rates (lr) against each of the other parameters. Notice how, the lower the learning rate, the slower the learning, but the higher scores achieved.

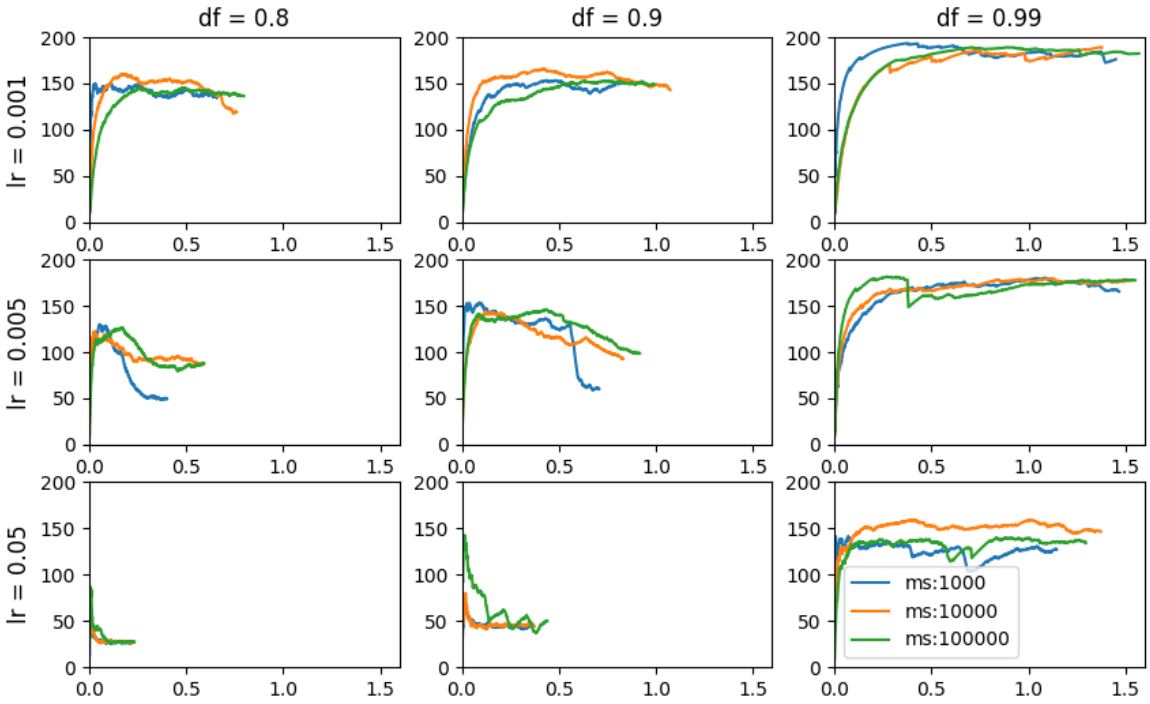

Figure 6: Test on the effect of the different memory sizes (ms) against each of the other parameters. Although no clear difference can be observed between 10000 or 100000, we can observe how having a memory size an order of magnitude bigger than 1000 helps the learning. Additionally, we can see how the memory size has very little effect on the training time.

Question I)

In order to assess the impact the target update frequency has on the training we tested a **d32d2** model with different update frequencies **uf** $\in \{1, 10, 50, 100\}$ setting the other meta-parameters to be: $df = 0.99$, $lr = 0.003$ and $ms = 10000$. Figure 7 shows how lower values of target update frequencies tend to reach stationary performance faster and tend to get higher scores. While it is important to de-correlate the network used for evaluating future steps than the one we are actually learning, it is better to match them frequently to have as much updated information available as possible.

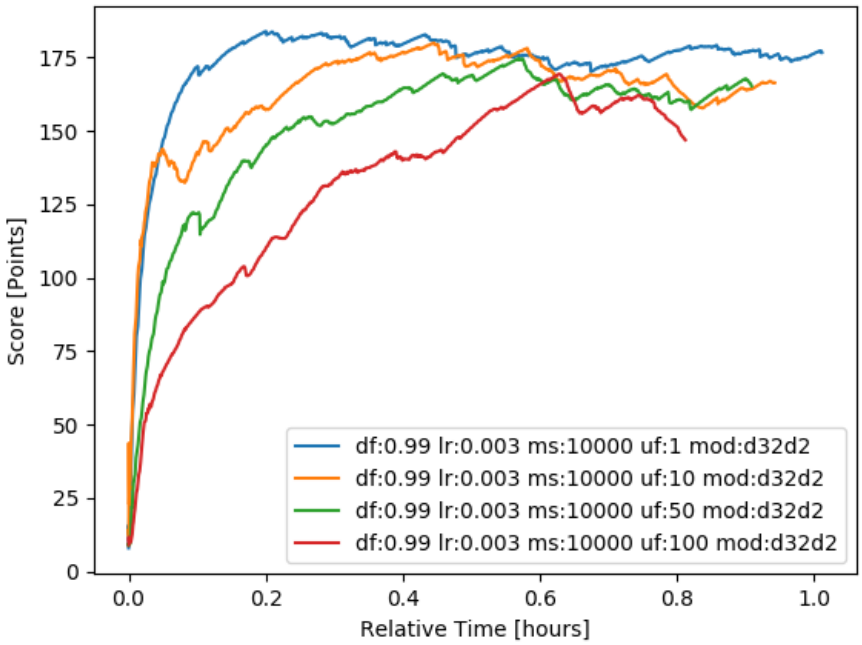

Figure 7: Smoothed score values obtained in the test of different target update frequencies: 1, 10, 50, 100. Notice how the lower the frequency, the sower is the learning as more model updates are needed.

Question J)

Based on the previous experiment conclusions, we chose to train a **d32d2** model using: $df = 0.99$, $lr = 0.001$, $ms = 10000$ and $uf = 1$. We then set the training to stop when 100 consecutive episodes score's mean was greater than 195. Figure

8 shows the convergence of the score values using the mentioned parameters. After 147 episodes it has achieved the stopping training condition.

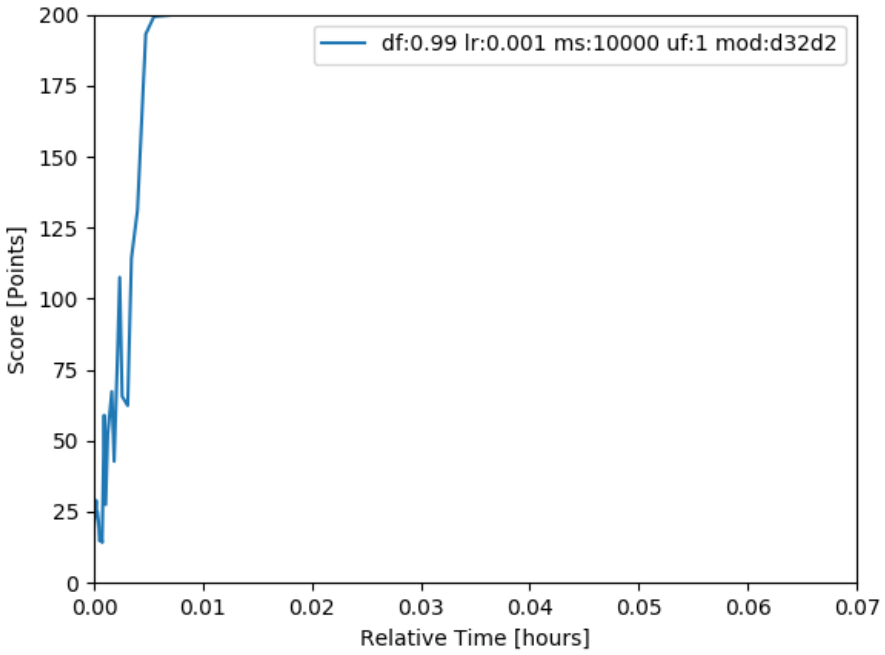

Figure 8: Smoothed score values obtained using the optimal estimated parameters: $df = 0.99$, $lr = 0.001$, $ms = 10000$ and $uf = 1$ and a **d32d2** model. Notice how fast it converged compared to other models, it only took 174 episodes and 1:23 minutes.

Hyperparameter Optimization using Gaussian Process

As we experienced, testing different combinations of hyperparameters -discount factor, learning rate, number of neurons, etc.- can become very confusionary and time consuming. In fact, each evaluation for a given set of hyperparameters requires several minutes, and the possible combinations are too many. Therefore, we decided to implement a Gaussian Process regression to estimate the best parameters automatically. We decided to opt for it instead of a grid search or a random search because it should allow us to reach the best parameters faster.

Gaussian Process Regression is a non-parametric, Bayesian approach to regression. Given some input data $\mathbf{X} = \{\mathbf{x}^{(i)}\}_{i=1}^{N}$ and the corresponding targets

$\mathbf{y} = \{y^{(i)}\}_{i=1}^{N}$, we assume that the targets are jointly Gaussian with distribution $\mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K})$ where the covariance matrix $\mathbf{K}$ is a kernel such that

$$\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij}\beta^2 \tag{6}$$

where the kernel function k computes the dot product of the input vectors in an higher-dimensional space, and is intended to represent a measure of similarity between those points. Using such a covariance function implies the assumption that similar points will be strongly correlated. We can then predict the distribution of a previously unseen point $\mathbf{x}^*$ by observing that

$$p(y^*, \mathbf{y}|\mathbf{x}^*, \mathbf{X}, \beta) = \mathcal{N}\left((y^*, \mathbf{y})|\mathbf{0}, \mathbf{C}\right) \tag{7}$$

where $\mathbf{C}$ is the covariance function of the data and the point to be predicted

$$\mathbf{C} = \begin{bmatrix} \mathbf{K}, & \mathbf{k} \\ \mathbf{k}^T, & c \end{bmatrix}, \tag{8}$$

with

$$\mathbf{k} = k(\mathbf{x}^*, \mathbf{x}_n) \quad \forall i = 1 \,..\, N$$

and

$$c = k(\mathbf{x}^*, \mathbf{x}^*) + \beta^2$$

The parameter β is the noise, and allows our prediction to not pass exactly through all the known points. We can then marginalize the known values $\mathbf{y}$ and obtain $p(y^*|\mathbf{x}^*, \mathbf{X}, \mathbf{y}, \beta)$ obtaining

$$p(y^*|\mathbf{x}^*, \mathbf{X}, \mathbf{y}, \beta) = \mathcal{N}(y^*|\mu, \sigma) \tag{9}$$

with

$$\mu = \mathbf{k}^T\mathbf{K}^{-1}\mathbf{t} \tag{10}$$

$$\sigma = c - \mathbf{k}^T\mathbf{K}^{-1}\mathbf{k} \tag{11}$$

Thus, the prediction we obtain is actually a distribution for the predicted value. This is the key feature that allows us to find faster the better hyperparameters.

Optimizing Hyperparameters As we said above, GP Regression allows us to predict the value of a scalar function with multidimensional inputs. Moreover, the result of this prediction is a distribution probability for the predicted value. GP Regression is therefore suited for those problems in which evaluating a data point is costly and one wants to reduce the number of evaluations as much as possible. By exploiting the predicted distribution, we know the uncertainty in the areas of the parameter space and we can therefore evaluate the points that have the higher probability of being a maximum, rather than directly evaluating the predicted maximum. This fits very well our problem: in fact, we can define an evaluation function $eval(\mathbf{x})$ of our agent that takes as input the hyperparameters vector $\mathbf{x}$ and returns a scalar value representing

the score. We will discuss later in more detail how do we compute this score. Therefore, our evaluation function is a function $\mathbb{R}^D \to \mathbb{R}$ where D is the number of hyperparameters. We use GP Regression to predict where the maximum of this function is, reducing the number of evaluations. Algorithm 6 shows the steps of the process. Note that GP keeps in memory all evaluated points and thus it is easy to stop it, restart it from a saved state, and get all the evaluated points. The hyperparameter search space S must be a discretized subset of $\mathbb{R}^D$.

Algorithm 6: GP Hyperparameters Optimization

```
1 Initialize GP
2 Initialize hyperparameter search space S
3 x = initial point
4 while True do
5 |   y = eval(x)
6 |   GP.add(x, y)
7 |   x = GP.most_likely_max(S)
8 end
```

Most Likely Maximum A key step in the process is estimating which point $\mathbf{x}$ -set of parameters- has the highest probability of being a maximum. In fact, we do not want to only evaluate to the unknown point that has the highest predicted mean, since if its variance is small enough, it could have very small chances of being higher of the current maximum and, on the other hand, a point which has a lower value could have an higher variance and therefore higher chances of actually being higher than the current maximum. Algorithm 7 shows the pseudocode of the most_likely_max() function. In line 1 we get the current maximum among the known evaluations of the function. Then, in line 4, we iterate trough all points in the parameter space. For each point we predict the evaluation mean μ and variance σ^2 and compute the probability of that point being higher than the current maximum. This is given, in line 6, by 1 minus the cumulative distriution of a Gaussian with mean μ and variance σ^2 evaluated at the current maximum. At the end, we return the point -a vector with the hyperparameters- that has the highest probability of being a maximum.

Evaluating the agent In the first trials, our $eval(\mathbf{x})$ function returned the average of the score over 100 episodes after having trained the agent with the parameters $\mathbf{x}$ for 1000 episodes. However, subsequent checks shown that the result was too variable: training again the agent with the same parameters sometimes led to very different results. This happened in particular for simpler models -e.g. networks of only one layer with 4 or 8 neurons. Therefore, we went for a more extreme -but also more consistent- approach: each evaluation consists of training the agent 4 times and evaluating it over 500 episodes each.

Algorithm 7: Most Likely Max

Data: known points P, known values V, gaussian process GP, parameter space S

$known\_max = \max_v V$
$candidate\_max$ = None
$max\_probability = -\infty$
for *point in S* **do**
  μ, σ^2 = GP.predict(*point*)
  p_over_max = 1 - cdf($known\_max|\mu$, σ^2)
  if $p\_over\_max > max\_probability$ **then**
    $max\_probability = p\_over\_max > candidate\_max = point$
  end
end
return *candidate_max*

The result is then the average of the average scores over the 500 episodes for each of the 4 training. After running the GP Regression for a while and testing again the obtained agents we found out that the evaluation is a bit exaggerated, and maybe 2-3 training with only 100 test episodes each would have been sufficient. Nevertheless, this approach yelded stable evaluation results.

Data: parameters $\mathbf{x}$

$scores = \varnothing$
for $i = 1$ *to* 4 **do**
  Train agent with parameters $\mathbf{x}$ for 1000 episodes
  res = test agent over 500 episodes
  $scores$.append(res)
end
return mean($scores$)

GP Regression settings and results We set the covariance matrix $\mathbf{K}$ with a Squared Exponential Kernel,

$$k(\mathbf{x_i}, \mathbf{x}_j) = \sigma_k^2 \exp\left\{-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2l^2}\right\} + \delta_{ij}\beta^2 \tag{12}$$

where l is the length scale parameter and β the noise. The parameter l tunes the correlation between different points: the higher l the more distant points will be correlated. After all the considerations made in the previous questions, we have an idea of what the optimal parameter space could be. We therefore set the parameter space as follows:

- **Discount factors** [0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.99]

- **Learning rates** [0.001, 0.005, 0.01, 0.05, 0.1, 0.15, 0.2]
- **Memory sizes** [1000, 2000, 5000, 10000]
- **Update frequencies** [1, 10, 25, 50]
- **Number of layers** [1, 2, 3]
- **Neurons per layer** [4, 8, 16, 32, 64]

Note that although we do not have many values per parameter, the search space size is 11760, which is impracticable since a single evaluation requires several minutes. Since the parameters have very different scales, the GP Regression scales them in the interval [0, 1]. After just 4 evaluations, the GP Regression found a very good model, and after 6 it found one that scored a perfect average of 200 points. We stress again that these values are the result of the average score over 500 episodes, repeated for 4 different training and averaged again. Therefore, a score of 200 means that the agent did 500 perfect episodes, 4 times consecutively, being trained again from zero each time. We felt that this model was, however, too complex -having 2 hidden layers of 64 neurons each- and we let the GPR continue to see if it could find less complex models that still perform well. Table 1 shows some of the best parameters found, ranked in order of complexity rather than score, since they all solve the problem. The best model in our opinion is model 1: it scores an almost perfect 198.44 while being simpler than the others -just one layer of 32 neurons. Our intuitions in the sections above were correct: the best discount factor is 0.99, the best learning rate is 0.001, memory size 10^4, and update frequency of 1. More complex models, such as 6 and 7, score a perfect 200, but the model is more complex -two layers of 64 neurons each. Looking at the results we see, however, that some variability in the score remains despite our aggressive evaluation: one can see that models 5,6, and 7 are equal in everything but the learning rate, with no aapparent relationship between the score and learning rates in 0.001, 0.005, and 0.01. This was surprising for us since 0.01 is ten times higher than 0.001, but the score of models 6 and 7 does not change. Model 5 has learning rate of 0.005 -in the middle- and score almost the same, $199.91 \approx 200$. Figure 7 shows, evaluation after evaluation, the score of the agent. We see that the GP Regression is able to find many maximums. Visiting low-score points is also a necessary steps to make it learn the shape of the function. However, we believe that these can be reduced with the improvement described in the paragraph below.

GP Regression improvement We found this approach really useful, and we decided that it will turn out useful again, many times outside the scope of this project. Therefore we will continue the improvement of our implementation, in particular in the followings:

- **Learning Kernel parameters** At the moment one has to set the kernel parameters manually. The idea is to automatically learn them by maximizing the probability of the data with respect to them.

Id	DF	LR	MS	UF	HL	N	Score
1	0.99	0.001	10^4	1	1	32	198.44
2	0.99	0.001	5000	1	1	64	195.32
3	0.99	0.001	1000	1	1	64	198.37
4	0.99	0.001	2000	1	1	64	199.87
5	0.99	0.005	10^4	1	2	64	199.91
6	0.99	0.001	10^4	1	2	64	200.0
7	0.99	0.01	10^4	1	2	64	200.0

Table 1: From left to right: identifier, discount factor, learning rate, memory size, update frequency, hidden layers, neurons per layer, score

Figure 9: Evaluated score of the agent per evaluation step of Algorithm 6. We can observe that after 4 evaluations we start to get almost-maximum values, and from 6 evaluations we get the first global maximum.

- **Numerical issues in cdf** Computing the cumulative distribution for small probabilities leads to severe approximation errors that makes the process sub-optimal. This happens in particular when the current maximum is well above the prediction of the rest of the points. Other approaches instead of the cdf must be tried.

Appendix

Here we include the missing code from the skeleton that we completed. The rest of the code used to conduct the experiments, perform hyper-parameter optimization with Gaussian Processes, and plot the results can be found in this repo.

As for the get action function, we implemented the ϵ-greedy policy as follows:

```
def get_action(self, state):
    state = np.reshape(state, [1, self.__state_size])
    if random.uniform(0, 1) > self.epsilon:
        actions = self.model.predict(state)
        return np.argmax(actions)
    else:
        return random.randrange(self.__action_size)
```

To train the agent's model, we implemented Bellman's equation as:

```
def train_model(self):
        if len(self.memory) < self.train_start:
            return  # Do not train if not enough memory
        # Train on at most as many samples as you have in memory
        batch_size = min(self.batch_size, len(self.memory))
        # Uniformly sample the memory buffer
        mini_batch = random.sample(self.memory, batch_size)

        update_input = np.zeros((batch_size, self.state_size))
        update_target = np.zeros((batch_size, self.state_size))
        action, reward, done = [], [], []
        for i in range(self.batch_size):
            update_input[i] = mini_batch[i][0]  # Store s(i)
            action.append(mini_batch[i][1])  # Store a(i)
            reward.append(mini_batch[i][2])  # Store r(i)
            update_target[i] = mini_batch[i][3]  # Store s'(i)
            done.append(mini_batch[i][4])  # Store done(i)

        # Learning network predictions
        target = self.model.predict(update_input)
        # Target network predictions
        target_val = self.target_model.predict(update_target)

        #Q Learning: get maximum Q value at s' from target network
        for i in range(self.batch_size):
            target[i][action[i]] = reward[i]
            if not done[i]:
                target[i][action[i]] = reward[i]
                                       + self.discount_factor
                                       *np.max(target_val[i])

        # Train the inner loop network
        self.model.fit(update_input,
                       target,
                       batch_size=self.batch_size,
                       epochs=1,
                       verbose=0)
        return
```