Neural Networks and Deep Learning - academic year 2020-21

Homework 3: Deep Reinforcement Learning

Luca Dal Zotto - 1236343
luca.dalzotto.1@studenti.unipd.it

Assignment

The aim of this homework is to learn how to implement and test neural network models for solving reinforcement learning problems. The basic tasks for the homework require to implement some extensions to the code seen in the Lab. More advanced tasks require to train and test a learning agent on a different environment. Given the higher computational complexity of RL, in this homework it is not required to tune learning hyper-parameters using search procedures and cross-validation; however, students are encouraged to play with model hyper-parameters in order to find a satisfactory configuration.

1. Introduction

In this homework I will implement and test some **Deep Reinforcement Learning** (**DRL**) models. In particular, we are asked to expand the notebook used in Lab 07 solving the following tasks:

1. study the impact of the exploration profile, tune some hyper-parameters and tweak the reward function to speed up learning convergence;
2. learn to control the CartPole environment using screen pixels rather than the compact state representation;
3. train a deep RL agent on a different Gym environment.

The Gym environment considered for the first two points is **CartPole-v1**, which has already be used during the lab practise. The main steps of the lab implementations are the definitions of a **replay memory** object, a simple **DQN**, an **exploration policy** and the implementation of a training loop with an update step.

For the first task, I have studied how the **exploration profile** (either using an **epsilon-greedy** or a **soft-max** policy) impacts the learning curve. Moreover, I briefly analyzed the effect of some important hyper-parameters, focusing only on those related to the RL implementation, such as the replay memory capacity, the number of episodes after which updating the target network, the bad state penalty, the discount rate gamma, and also different formulations of the reward function.

With this knowledge, I finally implemented a small grid search on the most promising hyper-parameter combinations, in order to find a configuration able to solve the game in the smallest number of episodes.

The second task was extremely challenging and time consuming: instead of using the compact state representation provided by the gym environment, we were asked to use directly the **screen pixels** to learn to control the CartPole environment. I tried many approaches, such as manipulating the observation space (pixel frames) in different ways, relying on some convolutional architectures to automatically extract features, giving as input to the network also past frames or the difference between consecutive frames, and finally trying a manual feature extraction approach.

The last part of the homework consists in facing a new Gym environment, training a deep RL agent from scratch. I chose the **MountainCar-v0** game. Even though it is simple with respect to other environments, it is more complex than the CartPole one, since it requires to find a long term strategy to win.

In the next section, I will describe in more details the model architectures and hyper-parameters considered, while in section 3, I will present the results obtained along with some considerations and comments. Note that both these two sections are divided into 3 sub-sections reflecting the task division. Then, after some final conclusions, I also included an Appendix with some additional figures and tables.

For what concerns the code part, I provided three different notebook files named LDZ_nndl_2020__homework_3_part$x$.ipynb with x from 1 to 3.

2. Methods

2.1. Task 1 - hyper-parameters exploration

In the first part of the homework, I have considered the implementation proposed by the teacher assistant and then I have modified it step by step, exploring the role of different hyper-parameters. Let's first recall the **CartPole-v1** environment (Figure 1). According to the description provided in the documentation, it consists in "*a pole attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center*".

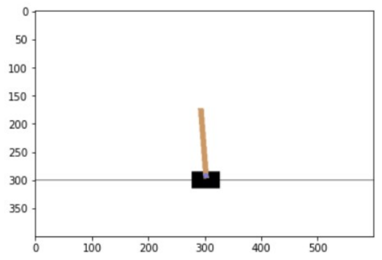


Figure 1: Frame of the CartPole environment.

The **state** of the environment consists in a 4-tuple (cart position, cart velocity, pole angle, pole angular velocity), while the **actions** that the agent can perform are only two: push the cart to the left or to the right. Moreover, the **reward** is simply +1 for every time-step that the pole remains upright. To limit the problems typical of Q-learning (such as highly correlated data and the incrementally created training set), I exploited the same workarounds seen in the Lab: indeed, I have used a replay memory mechanism based on a **deque** of fixed capacity to store the tuple (state, action, next state, reward) and kept a separate network (**target network**) periodically updated. I also used the same network architecture (a fully connected one with 2 hidden layers of 128 hidden neurons each). Recall that the input of this network is a 4-tuple state and the output is the estimated **Q-value** (expected long term reward) associated with each possible action (in this case, we have 2 values).

The first thing I analyzed in depth was the **exploration policy**. Starting from the estimated Q-values, we have to choose the proper action. In order to better **explore** the environment, the action is chosen in a probabilistic way. Two common approaches are the **epsilon-greedy** and the **soft-max** policy. In both cases, the exploration profile is determined by one parameter: for the former, it is ϵ, the probability of taking a non-optimal action, for the latter it is the *temperature*. The higher these two parameters the better we explore the environment, while if they are low, we **exploit** more our current knowledge. For this reason, we start from high values during the first iterations (episodes) and than we decrease them. Some hyper-parameter I considered are the **exponential decay rate**, and, only in the case of the soft-max temperature, the **initial value**.

The mathematical formulation of the exploration profile used is the following:

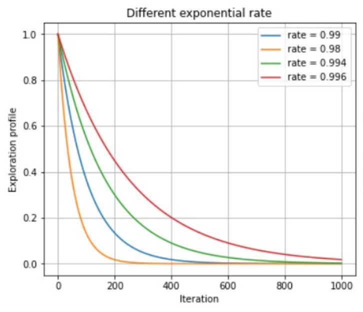
$$f(x) = t_0 \cdot rate^x,$$

where t_0 is the initial value and rate is the decay rate.

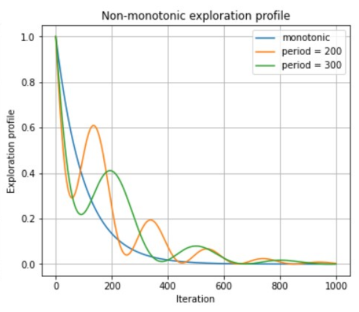
Furthermore, I also considered **non-monotonic** exploration profiles, adding a periodic component given by a trigonometric function:

$$f(x) = \frac{t_0 \cdot rate^x + t_0 \left[-\sin\left(\frac{\pi \cdot x}{T}\right) + 1\right] \cdot rate_2^x}{2},$$

where T is the **period** hyper-parameter, and $rate_2$ is the decay rate of this periodic component. This choice is motivated by the fact that occasionally increasing ϵ or the *temperature* could help to avoid being stuck into local-minima, or at least, to increase the exploration tendency during certain training phases.



(a) *Different decay.*



(b) *Non-monotonic exploration profiles.*

Figure 2: Examples of exploration profiles.

Secondly, I analyzed the marginal effect of the other hyper-parameters. In more details, I considered:

- γ: the discount rate;
- *bad_state_penalty*: penalty for having lost the game;
- *replay_memory_capacity*: how many elements to store in the replay memory;
- *target_net_update_step*: the number of episodes after which updating the target network, by copying the weights of policy net;
- *lr*: learning rate used by the optimizer.

In addition to them, I also played a bit with the **reward** function: besides the penalization seen in class related to the cart position, I tried one based on the **pole angle**, and a linear combination (possibly, with different coefficients) of the two.

According to these results, I finally run a small grid search over the most promising combinations of hyper-parameters, with the aim of reducing as much as possible the number of training episodes required to solve the game.

2.2. Task 2 - Screen pixel control

In the second part of the homework, we are asked to learn to control the CartPole environment directly using the **screen pixels**. As anticipated before, this was a very challenging task, in which many different strategies could be tried. The first modification I made was adding to the network architecture some **convolutional layers**, using them as feature extractor from the different frames, before some fully connected layers. To reduce the tensors' dimension, I added some **pooling** layers: I tried both max pooling and average pooling, because, having a white background, the maximum operator may not be very effective (in principle, when max pooling on the boundary of the pole, it will return a white pixel, since it has maximum values). Moreover, I tried to increase the stride and also added some **batch normalization** layers (instead of pooling layers to avoid interference) to increase learning stability.

In my first attempts, I used as state the entire frame rendered by the environment (which is a 400x600x3 pixel image). Very soon, I realized that this approach was unfeasible for the high computational cost. Even adapting my code to run with a **GPU** was not enough.

For this reason, I modified the image by **cropping** it, **reducing the definition**, considering **only one channel** and also **increasing the contrast** (each pixel is forced to be or black or white). In the most extreme case, the resulting object was a 24x40 black and white image.

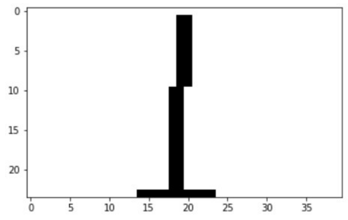


Figure 3: Example of pre-processed frame.

After having tried different pre-processing steps, I changed approach by feeding to the network **more than one image**: I used as state the current and the previous three frames. They were concatenated before applying the first convolutional layer, forming a 4 channel object. The idea behind this choice is that combining the present state with the past ones could provide information regarding the speed and the general evolution of the system.

The same motivations determined the next attempt, where I computed the **difference between the past and the present frame**. In this way, the temporal evolution is represented in one single image. This strategy led to the best results so far, but still it was not able to solve the game.

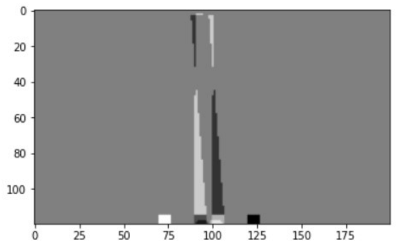


Figure 4: Example of difference between two frames.

Then, I tried another different manipulation of the input by selecting only **two rows of pixel**: one at the bottom and the other at the top of the pole. In principle, they contain information about the position and also about the inclination of the pole. Since the input has now height 2, I flattened it and used only fully connected layers.

As last approach, I tried a **manual feature extraction**: from the images, I computed something similar to the state information provided by Gym. To be more clear, let's consider the following figure.

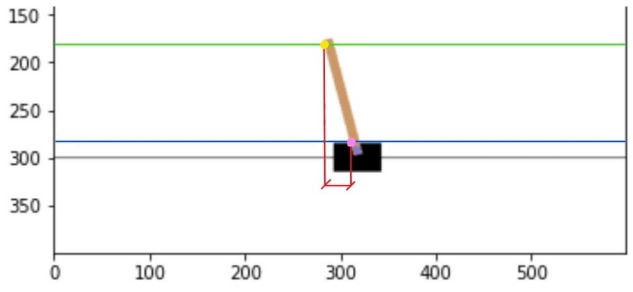


Figure 5: Manual feature extraction illustrated.

I found the first pixels (yellow and pink points in the figure) from the left that were not white at two different height (green and blue lines). The position information is simply represented by the pink point, while, instead of the angle, I computed the difference in the x-axis between those two points. These two manually extracted features have also been used to modify the reward function. Even in this case, to measure the variation in time, I computed the difference of these two features between consecutive frames.

2.3. Task 3 - MountainCar-v0

For the last part of this homework, we were free to choose a new Gym environment where to train deep RL agent. I chose **MountainCar-v0**, where "*the agent (a car) is started at the bottom of a valley. For any given state the agent may choose to accelerate to the left, right or cease any acceleration. The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum*".

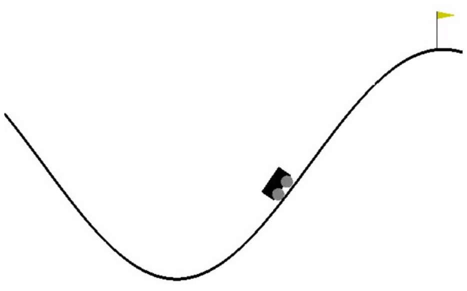


Figure 6: Frame of the MountainCar-v0 environment.

This last observation is extremely crucial: the agent has to build a winning **strategy** where, in some phases, it has to get away from the goal position, making a non-optimal short-term movement. Therefore, I tried different modifications of the reward function, in order to give some a priori knowledge to the agent. For the reasons just explained, adding a reward proportional to the simple position of the car was not enough. Therefore, I tried to give higher rewards when reaching higher elevations (obtained far from the center), also on the left-hand side of the screen. Then I tried a reward proportional to the absolute value of the speed of the car, to encourage building a momentum, and finally a combination of these strategies. I also tested a formulation resembling the mechanical energy, but it was not as effective as expected.

For what concerns the other hyper-parameters, I tried different values for those that seemed to be relevant also in the first part of the homework.

3. Results

3.1. Task 1 - hyper-parameters exploration

In order to **evaluate** the models, every fixed number of training episodes (50, 20 or 10, depending on the cases), I run 5 test episodes where the action was selected deterministically ($\epsilon = 0$ or $temperature = 0$). I stored the average score of these 5 tests, which can be then plotted to appreciate the learning curve. Since the final aim is to reach a score equal to 500 as fast as possible, I implemented an early stopping mechanism which terminates the training if all 5 test episodes reach the maximum. Please, keep in mind that only in this case I considered the game solved (the training score may be misleading, since there is stochastic component, and a single test episode may be unreliable). Moreover, I set 1000 as maximum number of training episodes, and in the tables with the results, I will write 1000+ if the game has not been solved.

Now we can see the results, starting from the effect of different exploration profiles, keeping all other parameters equal to the configuration seen in the Lab (for the complete list, consult the notebook). In Table 1, the different values considered for the hyper-parameters related to the exploration profile are listed. Note that default values (the one used when analyzing another hyper-parameter) are written in bold.

hyper-parameter	**values**			
rate	0.98	**0.99**	0.994	0.996
initial value t_0	**1**	2	5	10
period T	**0**	300	200	100
exploration policy	ϵ-greedy	**soft-max**		

Table 1: Hyper-parameters of the exploration profile.

In the Appendix, I reported also the tables with the results, where we can see the number of episodes needed to solve (with the convention explained before) the game. From these first results, I made some initial observations. The first one is that, on average, the epsilon-greedy policy has better results. Secondly, the non-monotonic exploration profile with period 100 is the best configuration for the soft-max policy. Thirdly, I have to be extremely careful when tuning these hyper-parameters, since small variations imply large changes in the results. The plots of the learning curves can be found in the notebook.

Then I analyzed the marginal effect of the other hyper-parameters. Clearly, if we put together the best values for each of them, we will not reach the best performance, since very often, the optimal value of a parameter depends on the others. For this reason, this part should be seen as a curios exploration of the main hyper-parameters of a Deep Reinforcement Learning model. The values tested for each hyper-parameter are reported in Table 2.

All the results and some additional plots, can be found in the notebook. In the Appendix, I reported only the most relevant ones, while here, for brevity, I will discuss some important aspects:

- with small discount rate (γ), the performances decrease, while values such as 0.97 and 0.99 lead to similar results;

hyper-parameter	**values**				
γ	0.90	0.94	**0.97**	0.99	
bad state penalty	**0**	1	2	5	
replay memory capacity	5000	**10000**	20000	50000	100000
target net update step	1	2	5	**10**	20
learning rate	0.001	**0.01**	0.1	1	
reward penalization	none	**position**	angle	both	

Table 2: Other hyper-parameters tested.

- bad state penalty does not influence the results;
- it seems that with an higher replay memory capacity, the game is solved in less episodes. However, I also noticed that the time required is higher, probably due to higher computational requirements. For this reason, I kept the value 10,000.
- apparently, updating the target network more frequently lead to better results, with some exceptions;
- learning speed is highly influenced by the learning rate, whose optimal value seems to be between 0.1 and 1;
- adding a penalization to the reward function based on the angle of the pole leads to better results, as we could have expected. Note that the best combination I found is the one where the angle is penalized 10 times more than the position.

To conclude this task, I also run a small grid search with some promising hyper-parameter combinations. Since I was able to find some very fast models, I reduced the minimum number of training samples to collect before starting updating the network from 1000 to 256. In this way, I found a model that solved the task after just 30 training episodes.

Then, I further checked the performances of the best models by performing additional test episodes, and I found out that the fastest models did not guarantee 100% probability of success: for example, the model trained with just 30 episodes had a success rate around 70%, while a model trained with 40 episodes, had a success rate of 90%. One of the fastest model I found, which was also able to reach a probability of success of exactly 100% (I run 100 test episodes), required 80 training episodes, which are still very few! I also saved the video of some episodes showing the pole staying perfectly upright and in the middle of the screen.

3.2. Task 2 - Screen pixel control

In the second notebook, it is possible to find a section for each possible strategy I considered for solving the task, and since they were different under many aspects, in each of them I re-defined the network, the update step, the training loop and also the exploration policy. Another observation about the code is that I manually interrupted the execution of some training loops once I realized they were too slow or that they were not leading to any good result. I decided to keep them to show at least one example for each possible strategy that I tried.

For example, this is the case of the first attempts where I tried to use as input the entire frame, using the CPU but also with an adaptation of the code working with the GPU.

To obtain a result in a reasonable amount of time, I tried different pre-processing steps, as described in the previous section. In this way, the learning speed increased. However, the scores reached after 1000 training episodes were only slightly higher than 100. Similar disappointing results were reached when considering as input the past 3 frames along with the current one and also when using only two rows of pixels at different heights with a fully connected architecture, as previously described.

The strategy relying on the convolutional layers as feature extractor which was able to reach the best performances is the one using the difference between consecutive frames. However, even in this case, the model was not able to overcome 200 points in less than 1000 training episodes.

What I noticed is that the results were extremely sensible to the hyper-parameter choices. With higher computational resources, one could have tried to fine tune the model. However, in my opinion, to solve the problem, one should focus on more advanced RL methods such as DDQN or Dueling DDQN.

On the other hand, I tried a third option, based on a **manual feature extraction**. As described in the previous section, I extracted from the raw pixels four simple features that somehow resemble the compact 4-dimensional state provided by the Gym environment. After having tuned the most crucial model hyper-parameters, I was able to find many configurations able to solve the game quite quickly. For example, a configuration was able to obtain the maximum score in 10 consecutive test episodes after only 230 training episodes. I also found faster (in terms of required training episodes) solutions, but their success rate was not that high. Of course, the results of the previous task are better, since here we are approximating the state of the pole based on the screen observation, while before we used the exact state provided by the Gym environment.

3.3. Task 3 - MountainCar-v0

The evaluation of the models trained in this part follows a procedure similar to the one seen in the first part of the homework: I run some consecutive test episodes every 50 training episodes, and counted how many times the game was solved within the maximum time, the average time needed to solve the game, and also the best position reached during the episodes.

After having played with different configurations, I found that the best results were achieved with the reward function encouraging higher elevations and, most importantly, higher speeds. Regarding the other hyper-parameters, I noticed that in this case it was better to:

- update the target network less frequently with respect to the CartPole environment;
- use smaller learning rates;
- use the soft-max policy with initial temperature 1 and a monotonic exploration profile.

In the Appendix, it is possible to see the learning curves (both in terms of score and best position achieved) obtained with the best configurations.

To conclude, I further analyzed the solutions obtained with some configurations. In particular, I found that in one case the game was solved in approximately half the available time, with some exceptions, due to particular starting positions of the car. Since the state was just 2-dimensional, I visualized the behavior policy of this particular agent, representing the action chosen depending on the position and speed of the car.

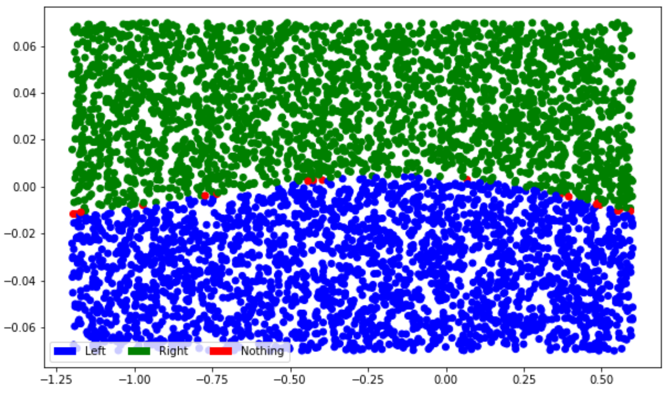


Figure 7: Behavior policy.

As we can see, this configuration is highly determined by the current speed of the car (y-axis), while the position (x-axis) has a marginal role in selecting the action. Moreover, the decision boundary between the actions “left” and “right” is well defined, while the action “do nothing” is almost never chosen.

Finally, watching some video episodes, it can be noticed that for some starting positions, the agent solves the game with a single running start to the left, while in other cases, it needs to drive back and forth more than once.

4. Conclusions

To conclude, I would like to highlight some aspects that I have learnt doing this homework:

- RL models are very sensible to the hyper-parameters used, but, at the same time, it is difficult to perform an accurate optimization of them, due to the computational requirements;
- the reward formulation plays a central role in the learning process;
- RL from the pixels is very challenging. When possible, one could try to make a manual feature extraction exploiting some a-priori knowledge.

Appendix

Task 1

In the following tables, I report the number of training episodes needed to solve the game, depending on the different hyper-parameters chosen.

rate	**0.98**	**0.99**	**0.994**	**0.996**
epsilon-greedy	300	180	840	380
soft-max	600	740	920	780

Table 3: Different decay rate.

initial value t_0	**1**	**2**	**5**	**10**
soft-max	740	940	800	380

Table 4: Different initial value t_0.

period T	**0**	**300**	**200**	**100**
epsilon-greedy	180	260	1000+	340
soft-max	740	840	820	260

Table 5: Different period T.

γ	0.9	0.94	0.97	0.99
training episodes needed	1000+	1000+	740	720

bad state penalty	0	1	2	5
training episodes needed	740	840	580	580

replay memory capacity	5000	10000	20000	50000
training episodes needed	620	740	260	260

target net update steps	1	2	5	10
training episodes needed	280	420	680	740

learning rate	0.001	0.01	0.1	1
training episodes needed	1000+	740	220	640

reward penalizatoin	position	angle	both	none
training episodes needed	740	300	380	1000+

Table 6: Results obtained with different hyper-parameters.

The following figures show the learning curves of the best models with the epsilon-greedy and the soft-max policy. Here I explain the notation used in the legend: b is the decay rate, T the period, ts the target network update steps and ms minimum number of sample for training.

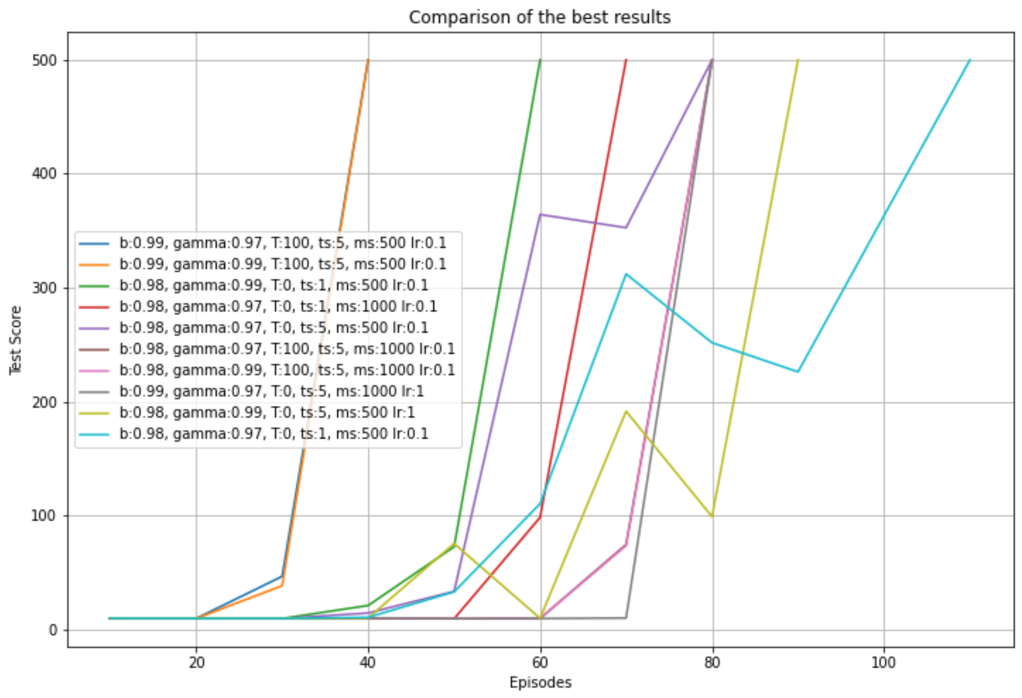


Figure 8: Learning curves of the best models with epsilon-greedy policy.

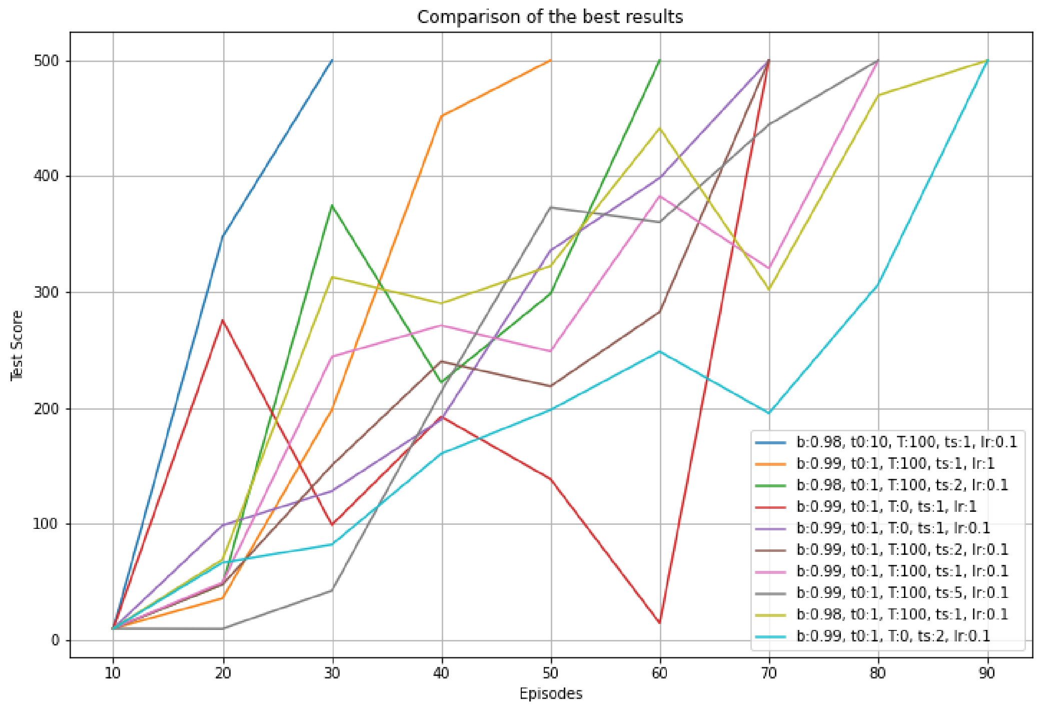


Figure 9: Learning curves of the best models with soft-max policy.

Task 3

The following figure shows the learning curves of the best models for the MountainCar environment in terms of the best position achieved (the target position is 0.5 and it is represented by a dashed line). Here I explain the notation used in the legend: rate is the decay rate, g is gamma, pw is the position weight (coefficient in the reward formula), sw is the speed weight, tnsu is the target network update steps and lr the learning rate.

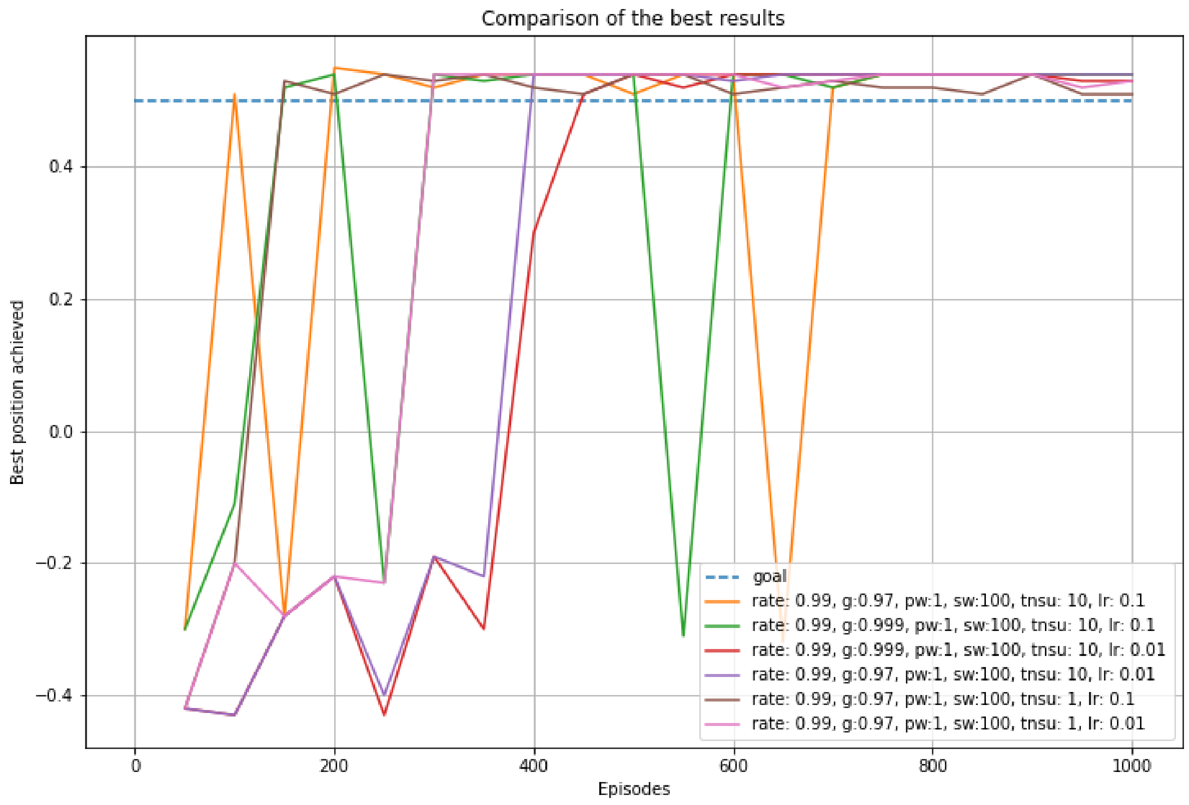


Figure 10: Learning curves of the best models for the MountainCar environment.

More figures and results can be found in the notebooks.