Homework 3:
Reinforcement Learning

Michele Guadagnini - ID 1230663

August 18, 2022

Abstract

In this final homework we explore Reinforcement Learning algorithms. In particular we deal with the *CartPole* problem from the OpenAI *gym* library. In the first part we are asked to study the effects of different exploration profiles (ε-*greedy* or *softmax*) on the learning curve and to improve the results obtained in the notebook of *LAB* 07 by tuning the hyper-parameters of the model. In the second part we try to solve the same problem but, instead of using the state variables of the environment, we build a convolutional network to solve it directly from raw pixels.

1 Introduction

Reinforcement learning framework is mainly composed by an *agent* that interacts with its *environment* by iteratively taking one of the possible actions and receiving from the environment its new state and the reward. The agent's aim is to maximize its own cumulative reward through the iterations by learning which actions to take when environment is in a particular state. Reinforcement learning presents several challenges with respect to other deep learning tasks: in general reward is sparse and delayed with respect to the action that lead to it and the sequence of state-action pair that are used for training are highly correlated.

In this work we make use of a neural network to approximate the *Q-value* function that represent the expected cumulative reward for every state-action pair and a given policy. In order to mitigate the difficulties stated above we adopt the use of an additional target network, whose weights are updated periodically, and of an experience replay memory. This approach is named *Deep Q-Network* (DQN).
We apply this method to the CartPole problem using the *OpenAI gym* library, where the task of the agent is to balance in vertical position a pole mounted on a cart that can move left or right. The state of the environment is represented by the four variables: *Cart Position, Cart Velocity, Pole Angle, Pole Angular Velocity*, while the action space contains the two options to push the cart *Left* or *Right*. The game is failed if the pole reaches an angle of 12° with respect to the vertical or if the cart exits the screen; the problem instead is considered solved when the agent has been able to take 500 consecutive action steps.
In the first part of the homework, described in Section 2, we start from the code of *LAB* 07 and we explore the impact of different exploration profiles. Then we set up an *Optuna* study in order to optimize the hyper-parameters of the model.
In the second part (Section 3) we use *PyTorch* and *PyTorch-Lightning* to build an agent that uses a convolutional network to approximate the Q-values not from the environment state variables, but from its rendered image.

2 CartPole from state variables

2.1 Method

To solve this first part of the homework we start by implementing the necessary tools. Most of them are taken from the *LAB* 07 notebook with little or no modifications, like the *Replay Memory* class, the *Q-network* class and the training routines. We then added the class *ActionChoice* that implements the policy of the agent to choose its next action, which can be set to ε-*greedy* or *softmax*. We implemented a function to add a penalty to

Parameter name	**LAB** 07 **value**	**Optuna sampling space**	**Optuna value**
replay memory capacity	10000	10000	10000
min. samples for training	1000	1000	1000
batch size	128	128	128
N episodes	1000	1000	1000
position penalty	1.	1.	1.
discount factor γ	0.97	(0.9, 0.99)	0.97893
learning rate	0.01	(1e-4, 1e-1)	0.080029
target_net update episodes	10	{2, 5, 10}	2
bad state penalty	0	{0, 1, 2}	0
DQN units	[128,128]	{[128,128], [256,64], [128,32], [64,64]}	[256, 64]
DQN activation	tanh	{tanh, relu}	tanh
optimizer	SGD	{SGD, Adam}	SGD
angle weight	0.	(0., 1.)	0.99966
softmax inititial temperature	5.	(2, 10)	2.8369
softmax decay constant	10	(4, 10)	8.2318

Table 1: Table of hyper-parameters of the first part of the homework. The first column reports the hyper-parameters used in *LAB* 07 and adopted also in the exploration profiles comparison. The second column contains the sampling ranges for the hyper-parameters to be used in the *Optuna* study, while the third column reports the best values resulting from the optimization. Note that fixed and not optimized parameters have been reported in *italics*.

the agent's reward based on the cart position (to force the agent to keep the cart inside the screen) and on the pole angle. We also defined some functions to create the decay profile of the policy parameter (either ε-greedy probability or softmax temperature). The three available decay profiles are: *exponential* decay, *linear* decay and *noisy* decay profile using random values from a half-normal distribution. In order to test the effects of different behaviour policies and exploration profiles we keep the hyper-parameters used in *LAB* 07, as reported in the first column of Table 1, and we define four decay profiles for each policy type using the functions described above:

- **exponential** decaying profile with decay constant equal to 10. Initial temperature is set to 5, while initial ε-greedy probability is set to 1.
- **linear** decaying profile, where the parameter decrease linearly towards 0 for the first 40% of the total episodes. Initial temperature and ε-greedy probability are set to same values as above.
- **noisy** profile, obtained combining a linear profile with a half-normal distribution. The noise level decreases until it reaches 0 at 60% of the total episodes. For the ε-greedy policy we also need to clip the values of the decaying profile to 1.
- **constant** profile, where the policy parameter have been fixed to 0.5 for the softmax policy, and to 0.1 for the ε-greedy policy.

For each of them (a total of 8 trainings) we train the agent for 1000 episodes, resetting the random state of the used libraries and of the *gym* environment to the same seed at the beginning of each iteration. At the end of the training we save in a dictionary the used decay profile, the loss value (Huber loss), the score and the cumulative reward for each episode in order to plot and compare the results. Note that score and cumulative reward can differ as score is just the number of steps completed by the agent during an episode, while the reward can have a different value due to the position penalty we applied with a weight of 1.

After the investigation of exploration profile effects, we want to optimize the hyper-parameters of the model in order to improve the agent's results. To do this we use the *Optuna* library to setup a study with the searching space reported in Table 1. Here we adopt a *softmax* exploration policy and try to optimize the parameters of the exponential temperature decay (initial temperature value and the decay constant). For each trial we reset the seed of the *gym* environment let the agent train for 1000 episodes. At the end of the trial we save a plot with the exploration profile, loss, score and cumulative reward. Since we aim in particular at accelerating the learning convergence, the criterion we choose to select the best hyper-parameters set is the number of solved episodes during the whole trial. We run the study for 40 trials, using random sampling of hyper-parameters for the first 25. Note that here, differently from what done in the previous homeworks, we do not use a pruner to

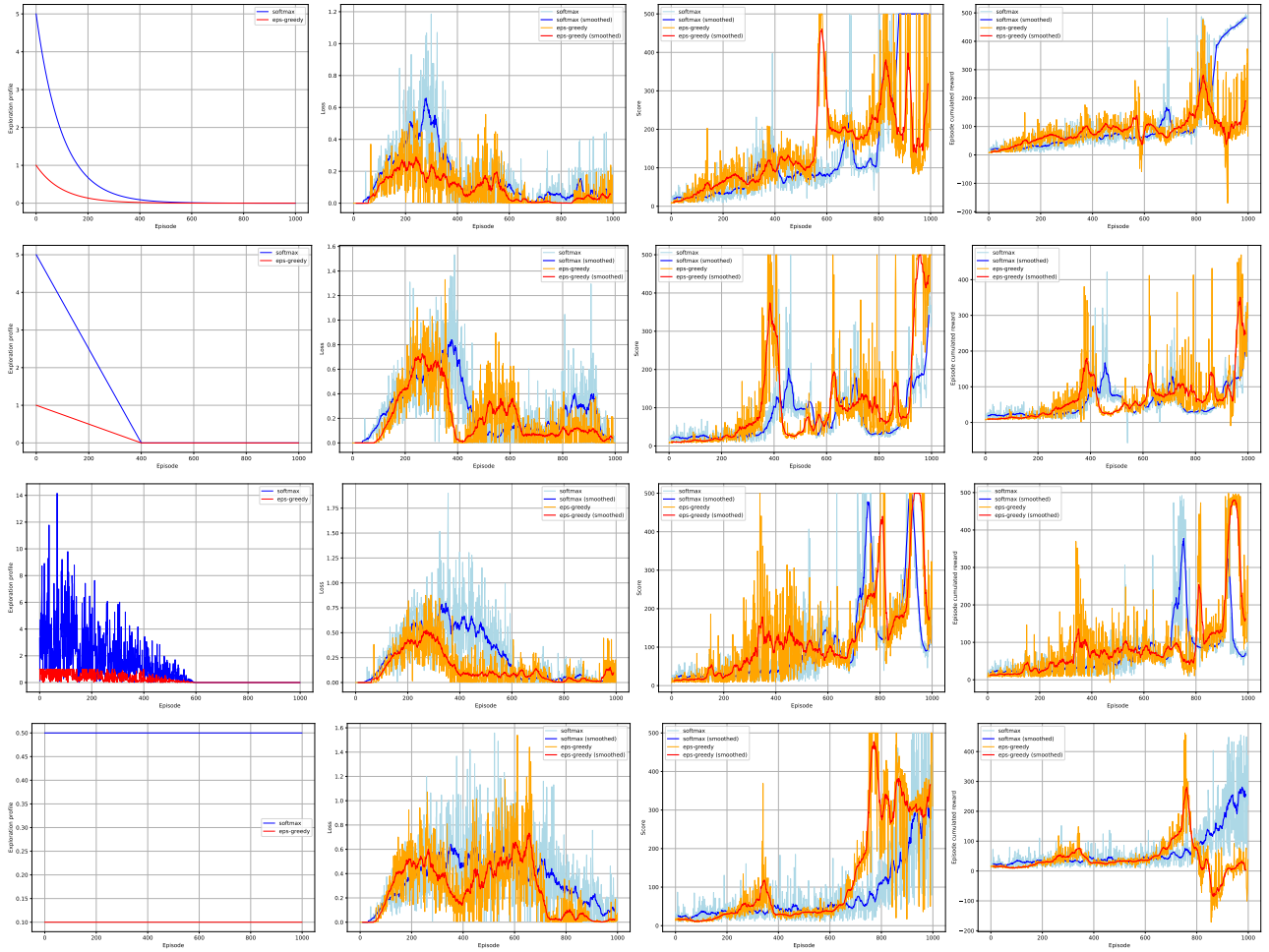

Figure 1: Comparison of softmax and ε-greedy policies results for different exploration profiles. For each profile (reported in the left figure of each line) the plots of Huber loss, score and cumulative reward at the end of each episode are reported. The lines with darker colour represent the moving average of the corresponding plots with a window of 20 episodes.

stop unpromising trials. Once completed the hyper-parameters optimization, we load the best model and test it for few episodes.

2.2 Results

The results of the exploration profiles evaluation are best summarized in Figure 1. From these plots, in particular looking at the score and reward graphs, it can be noticed that usually the softmax policy have a more regular behaviour, while ε-greedy policy is more noisy. We can also see that the softmax policy with exponential temperature decay (first line of plots in Figure 1) is the one with the best results. Indeed, looking at the score plot we can see that the last about 150 episodes are correctly solved by the agent. However, it must be said that the parameters fo the decay profiles have been chosen arbitrarily and not optimized for the task. Additionally, we noticed that these results are heavily dependent on the random seed used to initialize the network and the environment, leading to substantial changes if modified.

In the third column of Table 1 we report the hyper-parameters set resulting from the *Optuna* study. It can be noticed that some hyper-parameters have been set to values very similar to the ones of LAB 07, like the discount factor γ, the bad state penalty, the network activation and the optimizer, while others have assumed different values, like learning rate, the interval between target network updates, the number of units of network's layers and the angle weight, which assumed a value very close to 1. It is interesting to notice that, differently from the tasks of the previous homeworks, the *Adam* optimizer is performing worse than a simple *SGD* without momentum. The same can be noticed also for the activation function, given that *Optuna* have preferred *tanh* activation against *ReLU*.

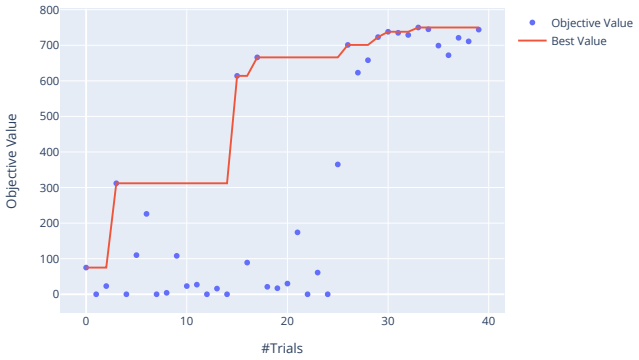

Figure 2: *Optuna* optimization history. Random sampling of hyper-parameters has been used for the first 25 trials of the study. The objective value is the number of solved episodes during the whole training.

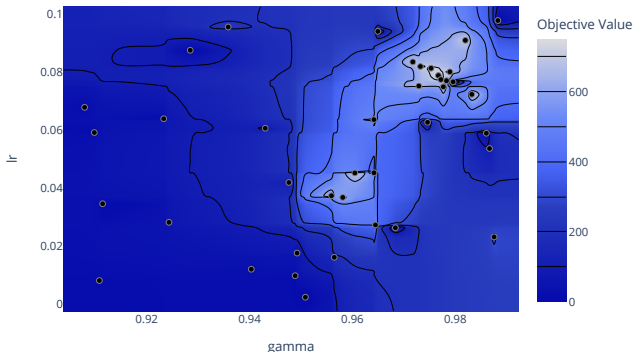

Figure 3: Contour plot of the number of solved episodes as function of the learning rate and the discount factor γ. The objective value is the number of solved episodes during the whole training.

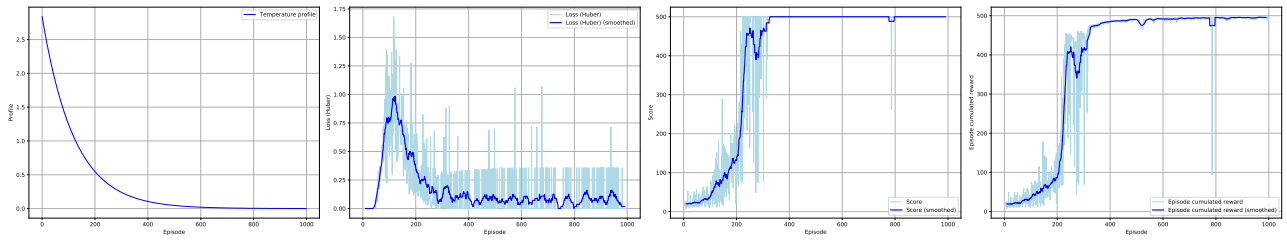

Figure 4: Comparison of the exploration profile, loss, score and cumulative reward per episode for trial 33. Blue solid lines represent moving averages of the corresponding quantities with a window of 20 episodes.

The best set of hyper-parameters is obtained at trial 33, as visible in Figure 2. As can be seen in Figure 4, this agent has been able to solve 750 episodes on a total of 1000 during training, solving the first one just after 200 episodes and almost all the episodes after about 300. Comparing its score and cumulative reward, we can see that the latter is progressively approaching the maximum value of 500, meaning that the agent is learning to keep as much as possible the cart at the center and the pole vertical, minimizing the penalties.

3 CartPole from raw pixels

3.1 Method

The second part of the homework consists in training an agent that can solve the *CartPole* problem using the rendered images instead that state variables. We decide to organize the code in a unique class called *DQNAgent* all the parts needed to train the agent, like memory, behaviour policy and Q-Network. We also encapsulate the class within *PyTorch-Lightning* framework in order to the implementation of the training logic and simplify hyper-parameters saving when checkpointing the model.
The *ReplayMemory* class inside the file *memory.py* is similar to the one used before. The main difference is that we need to enclose it inside an *IterableDataset* class in order to use it within *PyTorch-Lightning*. The class *SoftmaxBehaviour* implements the softmax exploration profile with exponential decay of the temperature parameter. Initial temperature and the decay constant can be passed at initialization. To adapt the code to this new task we need to change the Q-Network into a convolutional model to efficiently work with images. For this reason we implemented the class *PolicyNetwork* inside the file *components.py*. The class can receive at initialization:

- the convolutional configuration with number of channels per layer and per-layer configuration with *kernel size*, *stride* and *padding*. Optionally, batch normalization can be inserted after each convolutional layer.
- the linear configuration, consisting in a list with the number of units for each linear layer. Dropout can be added after each linear layer, although we will not test it due to lack of time.
- the activation function, for which we will test *tanh* and *ReLU*.

Name	*Value*	**Name**	*Value*
convolutional channels	[16,32]	target sync rate (steps)	200
convolutional config.	[[8,4,0],[4,2,0]]	discount factor γ	0.97
batch normalization	TRUE	warm-up steps	5120
linear units	[512,128]	batch size	128
dropout probability	0	optimizer	SGD
initial softmax temperature	4	learning rate	0.01
softmax decay constant	8	L2 penalty	0
memory capacity	15360	momentum	0

Table 2: Set of fixed hyper-parameters. Note that here *target sync rate* is assigned in steps and not in episodes as done in Section 2. *warm-up steps* represents the number of steps taken to fill up the memory before starting the training.

Another crucial modification that we need to adapt for the pixels input is the addition of the *_preprocess_input* method. This function takes every rendered frame and applies a sequence of transformations:

- it converts the picture to a *PyTorch* tensor and to gray-scale in order to reduce the image from 3 channels to 1.
- it crops the image from a size of 400×600 to 150×200 with top-left corner placed at $(168, 200)$. This way it removes regions of the image that are always white and not informative.
- since the pole angle can be evaluated from the image by just looking at the cart and at the top of the pole, we decided to crop out 32 lines of pixels on the top and on the bottom and then stack them together, obtaining an image of size 64×200.
- finally, it resizes the image to 32×100 to reduce the computational complexity.

The environment state includes information about the velocity of cart and pole which can not be deduced from the single rendered frame. To let the network infer this information, we decided to implement a buffer of 4 consecutive frames using the built-in python object *deque*, similarly to what have been done for the memory. This solution should be more robust with respect to the usage of other tricks like consecutive frames difference and is very easy to implement within *PyTorch*, as we simply need to increase the number of input channels from 1 to 4. An example on how the model is transforming the input image is visible in Appendix in Figure 6.
We include in the class *DQNAgent* also a method to apply a penalty based on cart position and pole angle to the reward in order to guide the agent through learning. We test two types of penalty, the first using position and angle from the environment state variables and the second using estimates of these quantities extracted from pixels. In this second case the cart position and the pole-top displacement (which is related to the pole angle) are estimated by computing the center of mass of the top and bottom images found in the preprocessing method.
As usual with *PyTorch-Lightning*, the training logic is implemented inside the method *training_step* and the procedure is still similar to what have been done in LAB 07. As before, we adopt the Huber loss. We create also two custom callbacks to store some results during training and to stop the training when reached the assigned number of episodes. To test the agent after training instead, we add the method *run* that can start a variable number of episodes and optionally save their recordings to disk.

Due to the high number of hyper-parameters and the complexity of the task we decide to not implement an *Optuna* study as done in the previous part, but we simply try few configurations and select an acceptable values. Instead, given that the agent trained on state variables can easily solve the task, we decide to use it as a *teacher* for the new agent and see if this trick produces an improvement. As teacher model we take the best agent resulting from the *Optuna* study of the previous part. At the beginning of the training, the teacher model selects the action in place of the new agent with a probability of 0.6. This probability is progressively reduced until it reaches 0 when completed the 60% of the assigned episodes, then the new agent selects its actions always by its own. This should help the agent to construct better training memories.

In Table 2 we report the hyper-parameters we have fixed before running a complete training of an agent model. We adopt an architecture with 2 convolutional and 2 linear layers. Other parameters values have been adapted from the ones obtained from the *Optuna* study of Section 2. For example we have set the decay constant of the softmax policy to 8, which is similar to the value of the optimization, but we increased the initial temperature to 4 as we expect the agent to need more exploration at beginning with this new task.

Trial #	N episodes	Activation	Penalty type	Teacher network	Total steps	Avg. score
1	1000	ReLU	state	FALSE	76201	103 ± 62
2	1000	tanh	state	FALSE	28869	46 ± 21
3	1000	tanh	pixels	TRUE	101000	73 ± 17
4	1000	tanh	pixels	FALSE	47259	64 ± 25
5	1000	ReLU	pixels	FALSE	30658	51 ± 15
6	2000	ReLU	state	FALSE	343000	423 ± 134
7	1000	ReLU	none	FALSE	72291	139 ± 84
8	1000	ReLU	state	TRUE	78265	404 ± 104
9	2000	ReLU	none	FALSE	147000	63 ± 26
10	1000	ReLU	none	TRUE	133000	445 ± 111

Table 3: Tested configurations together with the total number of training steps and the average score obtained from 100 episodes. During the final testing only three agents have managed to solve at least one episodes: agents trained in trial 6, 8 and 10 have solved 67, 42 and 72 episodes, respectively. Note that when using the teacher network we change also the softmax decay constant from 8 to 12.

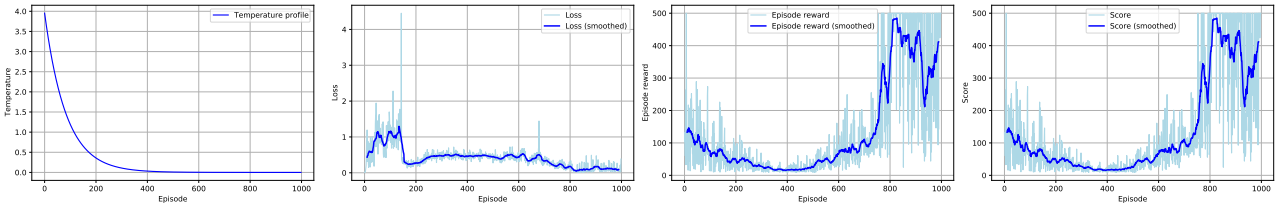

Figure 5: Plots of the exploration profile, loss, cumulative reward and score per episode for trial 10 of Table 3. Blue solid lines represent moving averages of the corresponding quantities with a window of 20 episodes. Note that here no penalty was applied and consequently score and cumulative reward are identical.

3.2 Results

We managed to run 10 complete trainings with some variations in few hyper-parameters: maximum number of episodes, activation function, penalty type and the usage of the teacher network. In Table 3 we report the adopted settings together with the total number of steps taken during training and the average score obtained with a short test of 100 episodes after training. A comparison between trials 1 and 2 suggests that *ReLU* activation works better than *tanh*. This is expected, as this time we are training a deeper and convolutional model. Regarding the penalty type, we can compare the trials 1, 5 and 7 and notice that the penalty extracted from pixels (trial 5) is worsening the performance with respect to the penalty based on state variables (trial 1), but also with respect to the case without penalty. Comparing trials 6 and 9, we see that the penalty based on state variables have produced a good improvement in agent's performance. In contrast, comparing trials 8 and 10 where we have used the pretrained network as *teacher*, we notice that the model trained without penalty have slightly better performance. All the trials where we applied the teacher network have produced at least a small gain in agent's results. Indeed, the best performing agent is the one of trial 10, for which we report in Figure 5 the training history with loss, score and cumulative reward per episode.

However, we must say that in all the trials we have used the same seed to initialize random libraries and the *gym* environment. For a more fair evaluation we should run the trials several times with a set of different seeds, which unfortunately would have required too much time. Additionally, hyper-parameters have not been optimized but chosen arbitrarily and this can affect the results of each trial differently.

4 Conclusions

In this work we have tackled the *CartPole* problem using Reinforcement Learning and in particular a *DQN* approach. When using the environment state variables we have seen that it is quite easy to train a successful agent, while when using the rendered image as input the problem becomes more tricky and computationally demanding. Hyper-parameter optimization have been possible only in the first case, so we expect that the results of the second part of the homework can be improved with greater time and computational efforts.

Appendix

Example of input image preprocessing

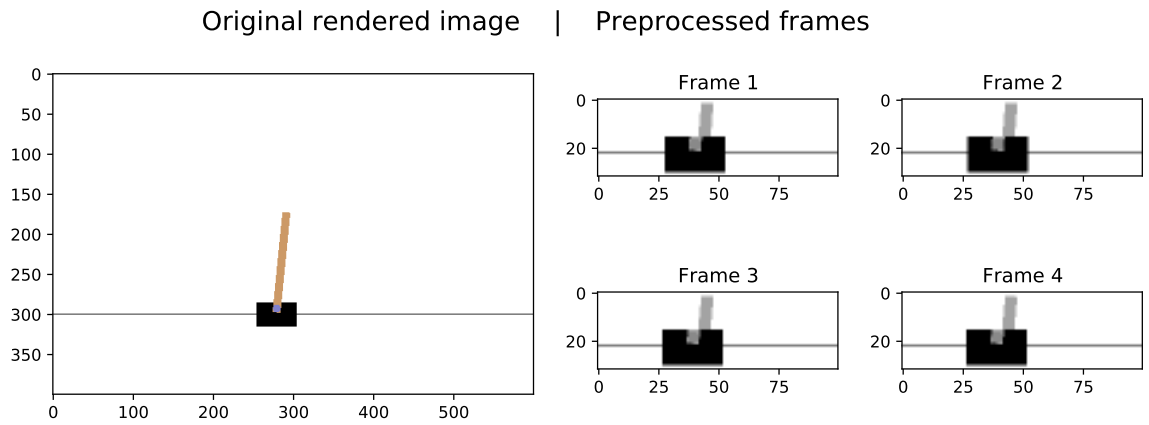

Figure 6: Example of the transformations done by agent's preprocessing method in order to reduce the computational complexity and consider the temporal context. Frames are numbered in consecutive order (frame #4 is the most recent one).

Train history of some agents trained on pixels input

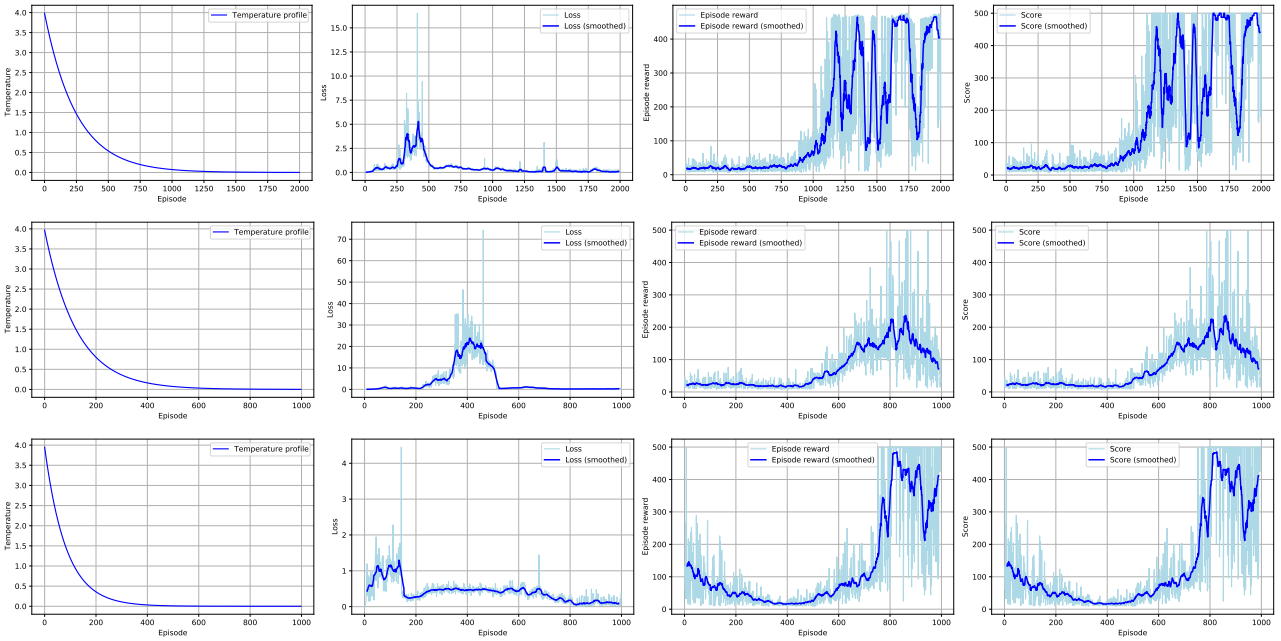

Figure 7: Comparison of training histories for trials 6, 7 and 10 in top-bottom order. Each line of plots reports: temperature parameter profile, Huber loss, cumulative reward and score per episode. The lines with darker colour represent the moving average of the corresponding plots with a window of 20 episodes.

Reference to additional media included in this work

With this homework we include also the following files inside the folder *AdditionalMedia*:

- **StateVariables_agent.mp4**; short video of an episode performed by the best agent resulting from the *Optuna* study of Section 2.
- **PixelsInput_agent.mp4**; short video of an episode performed by the agent trained in trial 10 from pixels.
- **Rendered_TransformedInput.gif**; animation that shows side by side the rendered episode and the transformed input that the agent is using to evaluate its action. Again we have used the agent of trial 10.