Reinforcement Deep Learning

Assignment 3

Andrea Nicolai - 1233407

03/09/2021

In this assignment we implement and test some neural networks to solve reinforcement learning problems. In particular, the *CartPole v1* environment is solved training a Deep Q-Learning agent that accepts as input either the state representation provided by the environment, or the screen pixels. Finally another Deep Q-Learning agent is trained to solve the *LunarLander v2*.

Introduction

Reinforcement learning tasks mainly consist of training an *agent* $\mathcal{A}$ which interacts with an environment $\mathcal{E}$. Every action performed causes a change in the state of the environment, which might either penalize or favor the agent assigning a *reward*. The agent naturally seeks to maximize its reward, therefore needs to learn the *optimal policy* which results in taking the best actions to obtain the largest reward given it finds itself in a certain state.
At every timestep t, the agent $\mathcal{A}$ performs an action a_t being in a given state s_t, receiving a reward r_t for it. The next state is denoted as s_{t+1}. Formally, the goal of the agent is to maximize a total return G_t, which is weighted according to some discount factor $\gamma \in (0, 1)$, that quantifies how much $\mathcal{A}$ should care about future rewards:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Gym environment

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms, providing several environments emulating classical problems or old-fashion games. In our work we will try to solve:

- **Cartpole v1**: the goal is to balance a pole attached to a cart, which can move without any friction either left or right. The episode ends when the pole falls, i.e. reaches an angle of 12° with respect to the vertical, or is considered "won" when 500 time steps have elapsed.
- **Lunarlander v2**: the goal is to make a star ship land onto a landing pad located at the center of the screen. The episode ends when either the starship crashes, comes at rest or exits the screen.

Methods

In Q-learning the agent $\mathcal{A}$ learns to associate a *value* Q, known as *long-term reward*, to every state-action pair $Q_\pi(s, a) = \mathbb{E}[G_t|s, a]$. Actions performed are sampled according to a policy, denoted by $\pi(s)$, which needs to be learned in such way that the actions chosen will come with the optimal $Q^*(s, a) = \max_\pi Q(s, a)$. The network therefore needs to be able to learn Q^*, which obeys the *Bellman*

Equation. However, since computations for it are too expensive and convergence might take too long, one may want to exploit a Deep neural network to directly approximate Q-values. The objective function, to be minimized through gradient descent algorithms, for this problem is:

$$L(\theta) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma\ \text{max}_{a'}Q(s', a', \theta^T) - Q(s, a, \theta)\right)^2\right]$$

where r is the reward, the second term refers to a **target** network whose task is to approximate $Q(s, a, \theta)$ given the *actual* parameters and is trained at every iteration, and the third term refers to a *policy* network updated every n steps and used for *action* selection. The introduction of the *target* network allows for a more stable training, and breaks the correlation between the target function and the Q-network.
In addition, in order to make the convergence occurring faster thus improving efficiency, it has been used **experience replay**. It basically acts as a *buffer*, and allows the learning from past experience once the network has gained "enough experience". Practically, this has been implemented using *deque* objects.
The actions to perform can usually be chosen according to either one of the following policies:

- **ϵ-greedy policy**: a non-optimal action is chosen with probability ϵ, while the optimal with probability $1 - \epsilon$.
- **softmax policy**: the action is chosen according to a *softmax* distribution of the Q-values, at a certain temperature. Temperature usually decreases over time starting from a "large" value, hence allowing at the beginning less theoretically favorable scenarios, while later choosing most likely actions that are known to return the largest Q-values.

Cartpole v1

Before proceeding with the description of the *Target* and *Policy* networks, which is the same, one should mention that the environment can be described using 4 quantities, namely ($x \in [-2.4, +2.4]$, $v \in \mathbb{R}$, $\theta \in [-15°, +15°]$, $\omega \in \mathbb{R}$). They are respectively cart *position*, *velocity*, pole *angle* and pole *angular velocity.*
The Target/Policy networks are implemented as it follows:

- **Input Layer**: 4 units, having the input state 4 elements.
- **First Hidden Layer**: 128 units, $Tanh$ activation function
- **Second Hidden Layer**: 128 units, $Tanh$ activation function
- **Output Layer**: 2 units, since the action can be only *left* or *right*

Before discussing the training, we should mention that as optimizer it was chosen the *SGD* with no momentum to improve stability and with learning rate 10^{-2}. In addition, it has been selected the *softmax* exploration policy, with the Temperature parameter varying according to some function. Indeed Temperatures profiles tried are different: either an exponential decay or a "linear" decay reaching almost 0 at half number of episodes devoted for training. In addition, some *Gaussian* perturbations to the Temperature are added in a number which is sampled uniformly between 0 and 4. The *loss* chosen for the problem is the *HuberLoss* function.
During the training, the agent receives a reward that is incremented by one for every steps in which the pole has not fallen, or the cart has not reached screen boundaries. Moreover, in order to keep the cart as much "centered" as possible, a linear penalty is applied when the cart moves away from the center.

The assignment request is to improve the *convergence speed* to a solution for the CartPole environment. Thus, two metrics are defined: the *average score* and the *velocity.* The first is used with the idea that, the larger the average score, the more stable should be the network in providing a solution thus being able to mantain it "high". On the other hand, the velocity is defined as the *first* episode at which the network is able to solve the game, i.e. to reach a score of 500. If a given implementation is never able to do it, this value is set to be equal to the total number of episodes devoted to training: namely 1000.

A optimal hyperparameter search, using optuna, has been performed for 50 trials with the aim of *maximize* the average score and *minimize* velocity according to the following set:

- **profile type**: ['linear', 'exponential'], being this quantity the Temperature behavior in time
- $\mathbf{n}_{noises}$: number of Gaussian perturbations uniformly sampled between 0 and 4
- $\mathbf{T}_{initial}$: sampled *uniformly* between 1 and 15
- $\mathbf{n}_{update}$: $[5, 10, 15, 20, 25]$ number of episodes every which to update the *policy* network.
- γ: sampled *uniformly* $\in [0.90, 0.99]$

The best set of hyperparameters turned out to be: ['linear', $\mathbf{n}_{noises} = 0$,$\mathbf{T}_{initial} = 4$, $\mathbf{n}_{update} = 5$, $\gamma = 0.973$], solving the game at iteration 537. Some plots regarding this *multi-parameter* optimization can be inspected at Figg. 12, 13, 14, 15, 16 and 17. It is interesting to see as the *velocity* parameter depends mainly on the type of profile chosen for the Temperature and only secondly on γ. Whereas, the average score gives more importance to n_{steps} and secondly on γ. The number of Gaussian perturbations seems to play a *negative* role on two metrics (Figg. 18, 19) and from now will be neglected. The model trained with this optimal set of hyperparameters reaches the performances outlined in Fig. 1

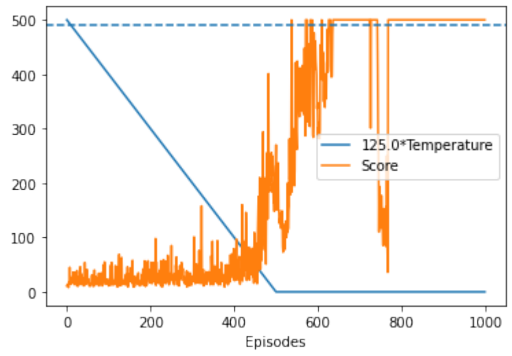

Figure (1) – Best model score wrt episodes in terms of velocity in converging *and* average high score.

Indeed one can see as the game can be solved thus reaching 500 steps, with some stability of the score later on. This network has been tested and some videos of it are attached to this report. One can compare this result with the ones in Figg. 18 and 19, where the network might be able to solve the game even faster, but not being a stable solution and thus has been discarded.

Pixels input

Then we try to solve the same environment changing the observation space, namely using the screen (i.e. pixels). However, frames are rendered as $3 \times 400 \times 600$ RGB-images which are computationally really expensive to handle: they end up filling the RAM very quickly. Therefore some functions have been implemented to *pre-process* the input data: the first one converting a RGB image to a black-white one, thus diminishing the number of channels from $3 \rightarrow 1$ and eventually resizing the image (see Fig. 4), not providing the color any useful information. In addition to this, one might want to crop the vertical image, just leaving the *horizontal belt* and screen boundaries, eventually rescaling the image (see Fig. 3). Finally the function that has actually been used for the training, since it turned out to be already much computationally expensive though being the most "compressing" one, is the one that, in addition to the aforementioned transformations, crops the image also *horizontally* only sparing the cart (see Fig. 2) and eventually resizing it.
The *resize* is performed using *cv*2 library, and the "output scale" might be actually a hyperparameter to be tweaked in order to improve network efficiency thus reducing the dimensionality of the problem. For our problem we have chosen to use only the latter function introduced, resizing the cropped-grey-scale image to 50% of its original size, thus having dimensions $(1 \times 80 \times 40)$.
Finally, in order give the network the opportunity to eventually learn from past frames, we feed it with a certain number of consecutive frames n_{frames}, which might vary from 1 (static input) to 5. Obviously, when $n_{frames} > 1$ we expect the network to be able to infer somehow velocities.

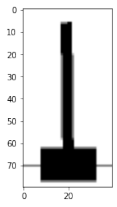

Figure (2) – Frames are converted to BW, cropped to be centered on the cart, and finally rescaled.

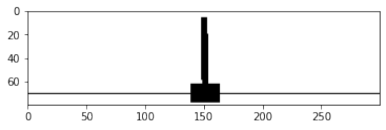

Figure (3) – Frames are converted to BW, cropped vertically to preserve only the horizontal belt, and eventually rescaled.

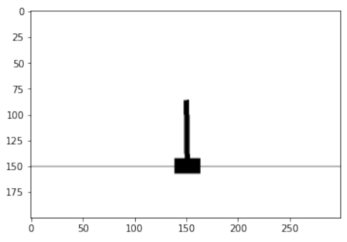

Figure (4) – Frames are converted to BW and eventually rescaled.

Figure (5) – Different types of frame preprocessing. All share that they were converted to a single channel, i.e. grey scale, to the reduce dimensionality of the problem which is already computationally really expensive.

The Deep Q-Convolutional Neural Network implemented for this task takes the architecture:

- **First Convolutional Layer**: having n_{frames} channel in input, $4 \cdot n_{frames}$ channels as output, *kernel* 8×8, *stride* 4. Activation function is *ReLU*.
- **Second Convolutional Layer**: having $4 \cdot n_{frames}$ channels as input, $2 \cdot 4 \cdot n_{frames}$ channels as output, *kernel* 4×4, *stride* 2. Activation function is *ReLU*.
- **Flattening layer**: with *ReLU* activation.
- **First Linear Layer**: with 256 units and *ReLU* activation.
- **Second Linear Layer**: with 128 units and no activation function.
- **Output Layer**: 2 units, since possible actions are 2.

Replay memory capacity here is set to 2000 and the minimum samples needed for training 200, due to the limited resources available (RAM). The network is trained with different Temperature profiles, namely the ones depicted in Fig. 6, with the *softmax* updating rule and using different values of frames $n = \{1, 3, 5\}$. The one that presents the higher average rolling mean seems to be the one exploiting the second exploration profile (i.e. the exponential) and using $n = 3$ consectuive frames. One should note that the networks exploiting the *static* frame input are the ones performing the worst, so "learning" velocity is useful.

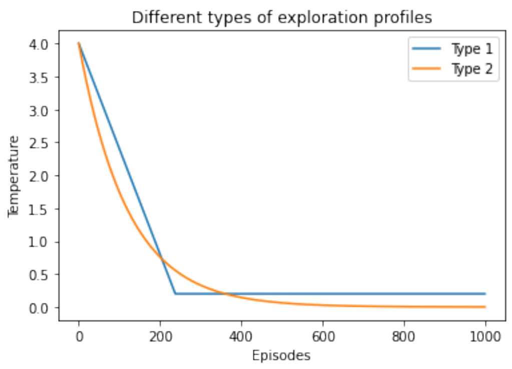

Figure (6) – The two different training profiles that have been used, the maximum temperature is set to 4.

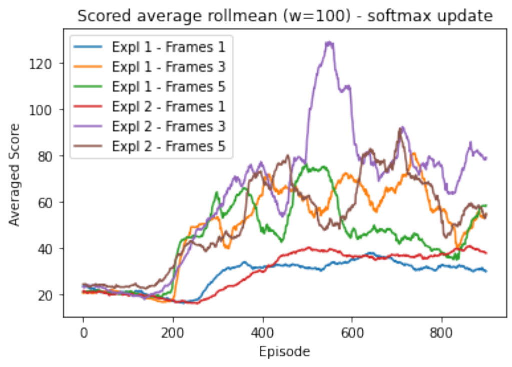

Figure (7) – The rolling mean score with a window 100 iterations for different number of frames and exploration profiles.

The results for such network can be inspected in Fig. 8. However, they seem to be really unstable despite the game was for good solved around episode 600.
Testing the network, one can see as the average score is 172 ± 11, which tells us that the network has been indeed able to learn something with respect to random agents results. At a further inspection of test videos (which are attached), one can see that, despite the linear penalty being present as in the

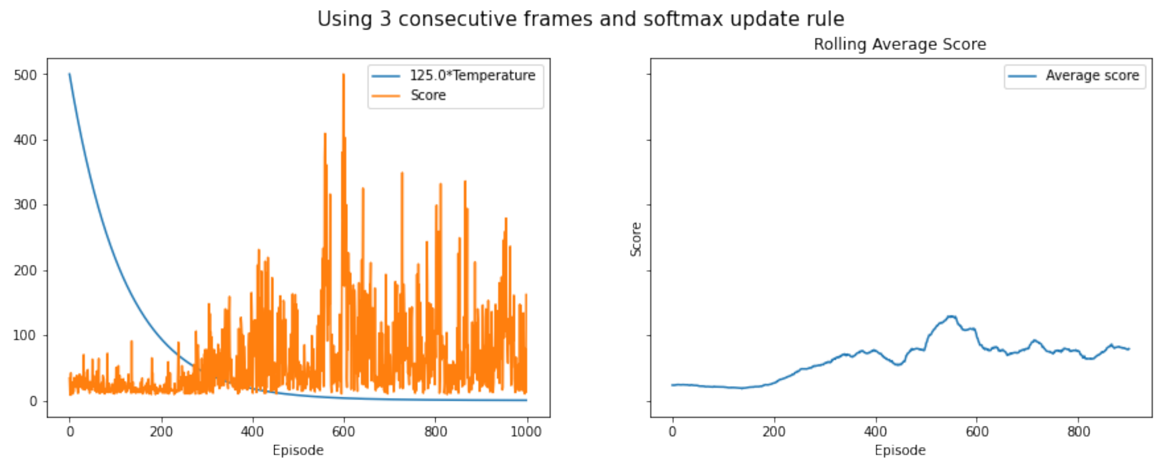

Figure (8) – Best model training using pixels as inputs.

first part of the assignment, the cart starts moving away from the center and the most of times exits the screen on the left side.
We might hypothesize that using the second function for screen preprocessing, namely the one whose effects are depicted in Fig. 3 (i.e. including screen boundaries) might have lead to better results, though sensitively increasing the computational demand. Or, alternatively, one might have wanted to try to increase the coefficient of the linear penalty.

LunarLander v2

As introduced before, the goal is to make the lander land onto a landing pad that is always at coordinates $(0, 0)$ (see Fig. 10). The reward for moving from the top of the screen to landing pad and zero speed is about $100, \dots, 140$ points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or $+100$ points. Each leg ground contact is $+10$. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete **actions** are available: do *nothing*, *fire left* orientation engine, *fire main* engine, *fire right* orientation engine.
The state returned by the environment consists of 8 variables, namely:

- X coordinate $\in \mathbb{R}$
- Y coordinate $\in \mathbb{R}$
- X velocity $\in \mathbb{R}$
- Y velocity $\in \mathbb{R}$
- Angle $\in \mathbb{R}$
- Angular velocity $\in \mathbb{R}$
- Left leg touching the ground (*bool*)
- Right leg touching the ground (*bool*)

Since dimensionality has changed, we double the number of units of the second hidden layer with respect to the implementation of the first network, and change input and output accordingly:

- **Input Layer**: 8 units, having the input state 8 elements.
- **First Hidden Layer**: 256 units, $Tanh$ activation function
- **Second Hidden Layer**: 128 units, $Tanh$ activation function
- **Output Layer**: 4 units, since the discrete action set has cardinality 4.

The training is performed with different penalties being added to the reward: we want to penalise states with large *angle*, since a “good” landing has null angle, X coordinate to make the lander stay in the center and finally Y coordinate to make it land in a faster way. Four different combinations of such penalties have been used during the training, namely:

- No penalty
- 1.0 penalty on the angle if this is negative. Note as we cannot take the absolute value of the angle since this would lead to "too much stability", with the lander that keeps on flying without descending since this would result in an angle.
- 1.0 penalty on the angle and 0.02 penalty on the absolute value of X coordinate
- 1.0 penalty on the angle and 0.02 penalty on the absolute values of X, Y coordinates

The result for such search are visible in Fig. 9. It has been decided to not proceed further, or to tune "better" the *values* of such penalties, or even to explore different Temperature profiles, due to the large computational demand even for such simple task. Some attempts were however performed introducing penalties on other variables or parametrizing the actual ones in a different way (e.g. penalty on the absolute value of the angle), but they were not promising as the ones presented here.

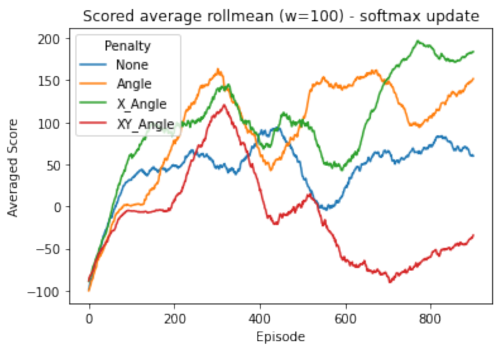

Figure (9) – The rolling mean score with a window 100 iterations for different penalty type.

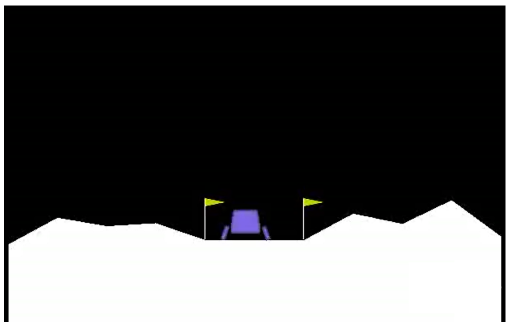

Figure (10) – A frame taken from a successful landing video.

The best penalty to be chosen, according to the higher average score, is the one penalising angles and on absolute values of X. The performances for such network can be seen in Fig. 11. During the test, the average score over 10 episodes has been 147 ± 93. The large standard deviation tells us that the network sometimes is able to solve the game, whereas sometimes is not able. Further inspecting at the videos attached, it seems like the ones with low score do not end up with the lander crashing, rather than with the latter wasting fuel to reach the central point. This might suggest some more optimization is needed, though the training has been successfully performed: the lander is now able to land onto the landing pad in most of the cases without crashing.

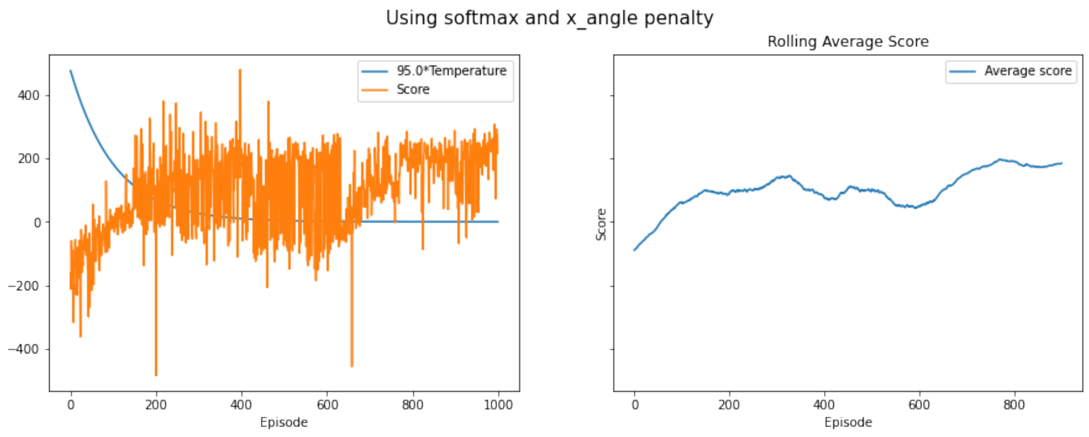

Figure (11) – Best model with penalty on X coordinate (0.02) and on the angle (1.0). Temperature profile is also visible.

Appendix

This section contains some additional figures, not fundamental for the drafting of the report, nevertheless worth to be showed.

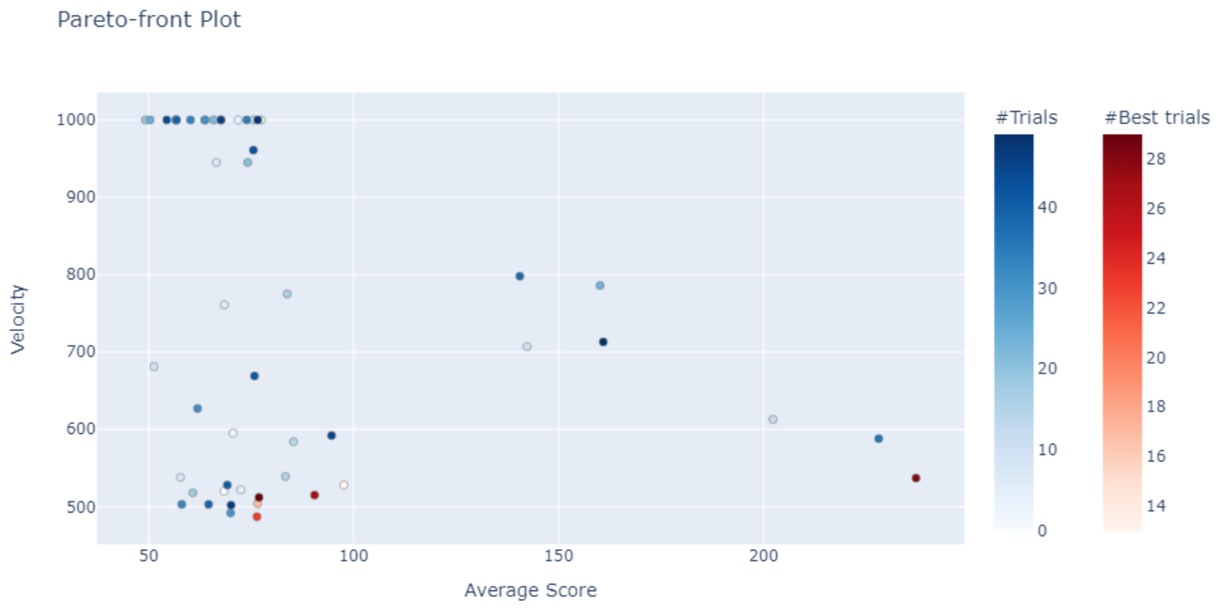

Figure (12) – Pareto plot for the Cartpole optimization for velocity in convergence and average high score.

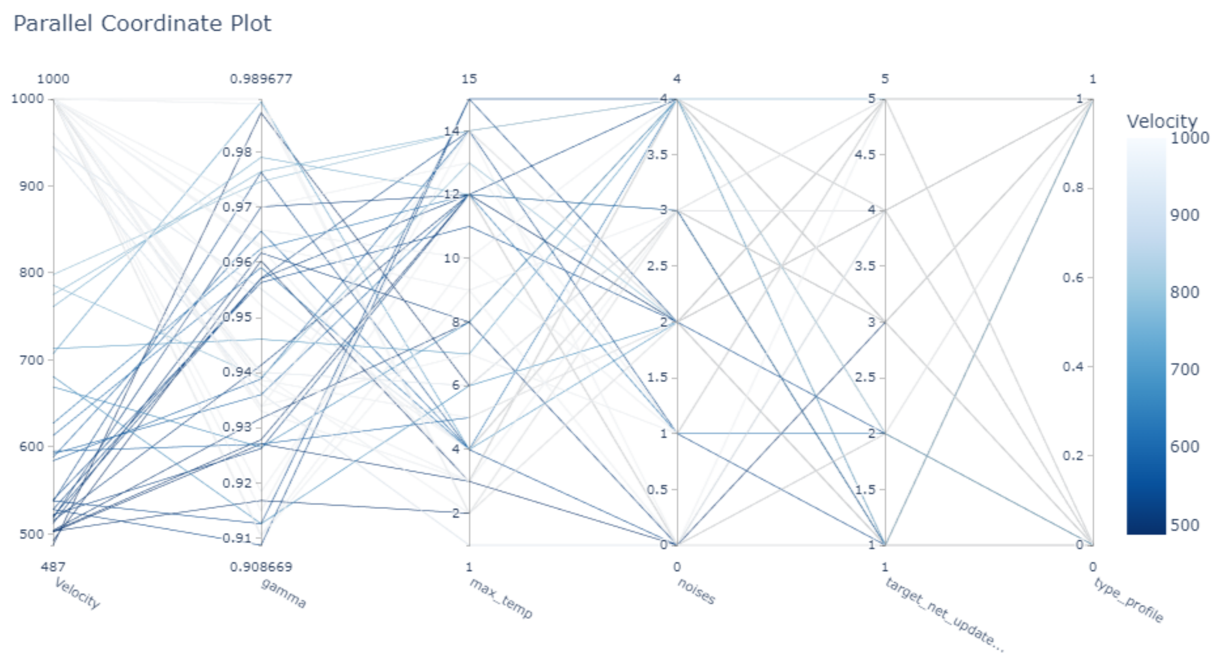

Figure (13) – Parallel plot for velocity in convergence wrt the choice of other hyperparameters.

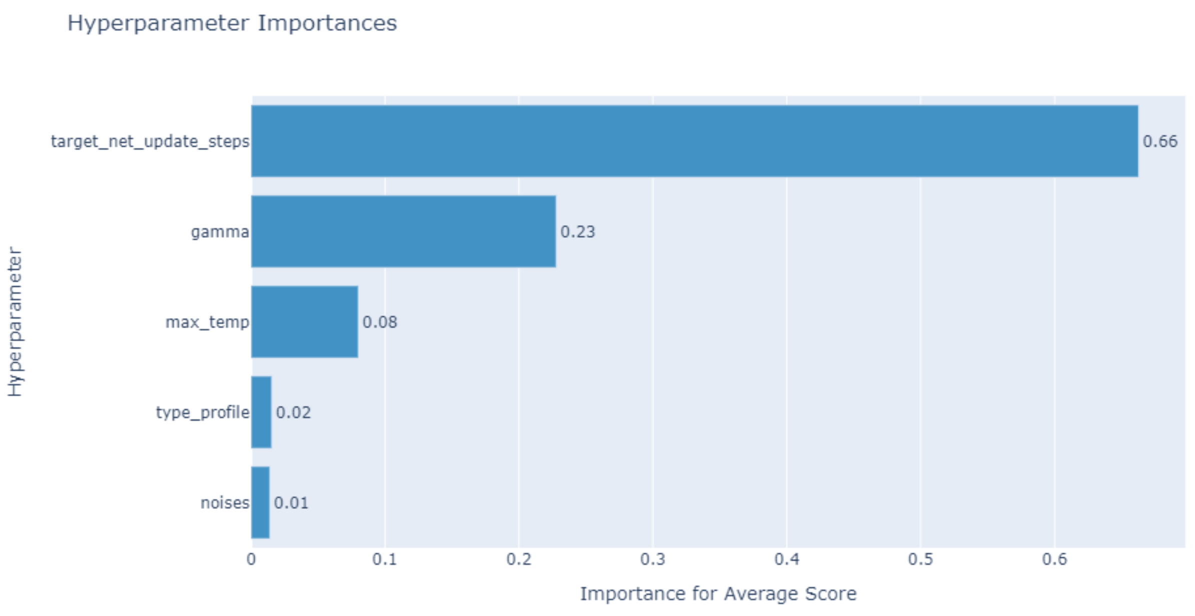

Figure (14) – How average score depends on the hyperparameters inspected.

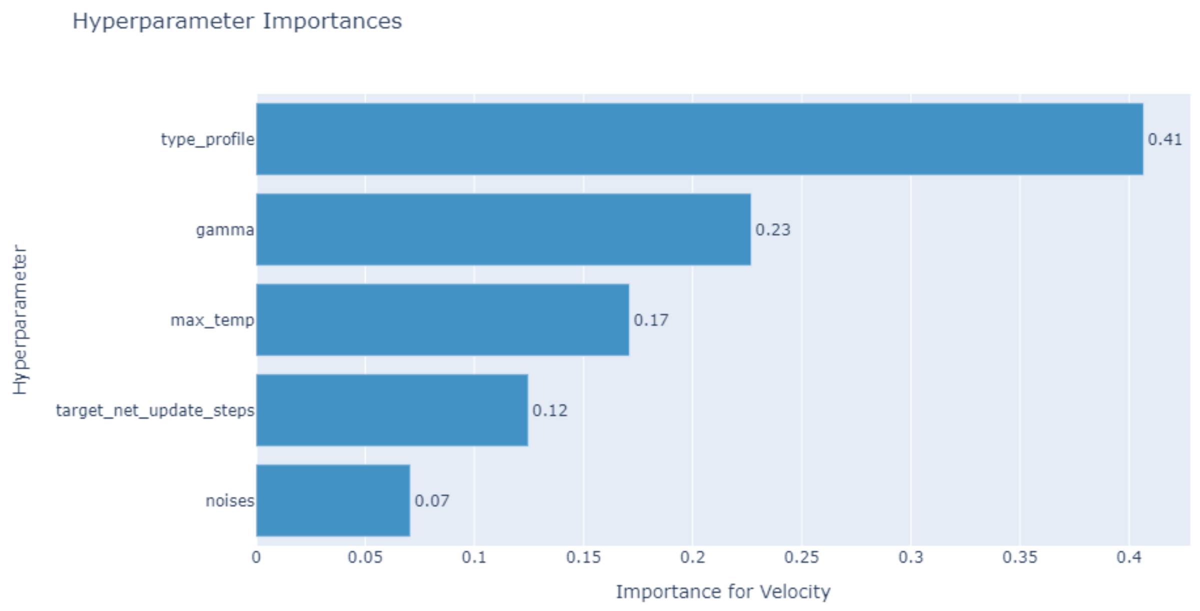

Figure (15) – How velocity depends on the hyperparameters inspected.

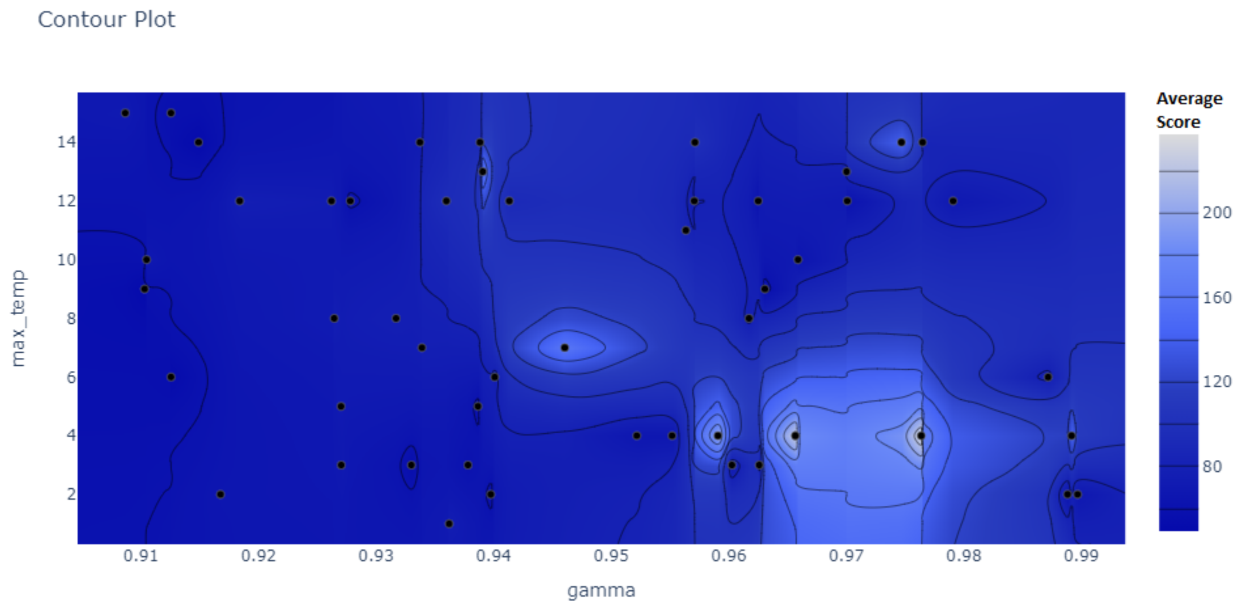

Figure (16) – Countour plot of average score versus temperature and future reward weight γ. It seems that there is an optimal region for γ that maximizes the average score.

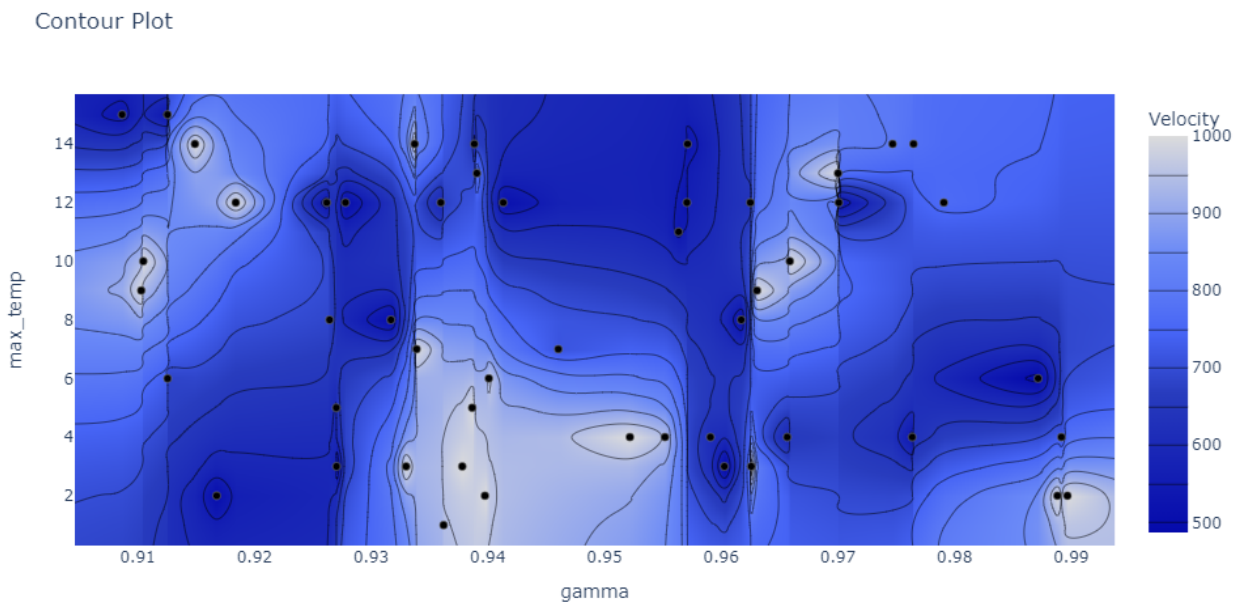

Figure (17) – Countour plot of velocity score versus temperature and future reward weight γ.

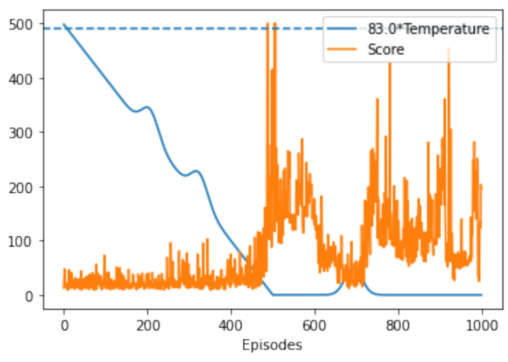

Figure (18) – Training score for the second best network. Gaussian noises do not help at all in improving performances.

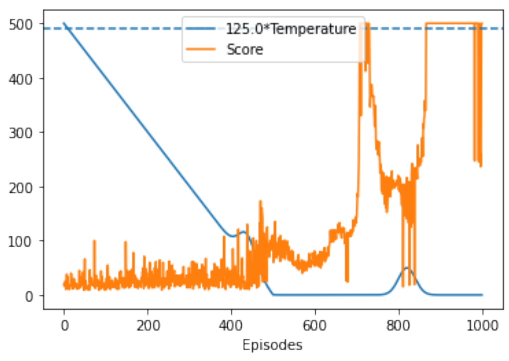

Figure (19) – Another network proposed in the CartPole environment with actual state as input. Gaussian noises are seen to have disrupting effect.