Homework 3 - Report

Fabio Vaccaro

06 February 2021

1 Exploration profile

1.1 Introduction

Exploration profile is very important as it affects learning curve. Exploration profile is basically a list of values from which at each iteration a new value is taken and is used as a temperature for the softmax policy function. Temperature parameter is crucial to balance exploration and exploitation driving learning to reach the best solution (exploitation) but at the same time not miss any better solution (exploration). Exploration profile should provide higher values at the beginning as in early iterations exploration is very important, more than in later steps.

1.2 Method

I have decided to implement a hyper-parameters tuning strategy using Optuna library to exploit the Bayesian search approach.

Exploration profile is calculated as follows:

```
exp_decay = np.exp(-np.log(initial_value) / num_iterations * steepness_value)
exploration_profile = [initial_value * (exp_decay ** i) for i in range(num_iterations)]
```

This led me to identify the hyper-parameters as:

- *initial_value*: it is the first value to be the temperature
- *steepness_value*: it controls the steepness of the curve. Bigger the number, quicker the convergence to 0.

I have chosen large intervals for hyper-parameters: *initial_value* in the interval [1, 9] and *steepness_value* in che interval [2, 10].

1.3 Results

Running the optimization 20 times, I realized that it was pretty difficult (see Figure 2 and Figure 3) to analyze graphs if all traces were plotted in the same graph for two reasons: the high number of traces and colors and the behavior of each trace (consequent episodes can reach very different scores translating to very fragmented lines). I solved the first problem implementing Plotly library which allows you to disable singular traces in an interactive way instead of the standard matplotlib library. I faced the second problem plotting two graphs, the first one as usual, the second with episodes scores averaged every 10 episodes. This gives smoother traces, easier to understand. So a first analysis is done on the the averaged graph and then the final, more punctual one is done on the normal one.

I have plotted two traces every trial, train score and test score. I have reasoned, taking theoretical concept from traditional machine learning, in the following way:

- *train score*: is the score (maximum number of steps before falling). This is considered at every episode.
- *test score*: is the average between scores of 5 test executions (each with a different seed). Each of these 5 test scores are calculated averaging 10 test episodes. I have changed the seed in comparison to the training to simulate a different scenario from training (like calculating the test loss on a different dataset for testing). Also different from each other to simulate different cases.
- *first_max_episodes*: is the number of the episodes in which for the first time the game is won (500 as test score). This value is calculated to help Optuna to find the best configuration. In fact it is programmed to minimize this value.

I have selected graphs that arrive to the maximum score of 500 points as soon as possible but at the same time maintaining that score to some extent. Selection results are shown in Figure 4 (averaged) and in Figure 5 (not averaged). See comparison[1] in Table 1. Other configurations hyper-parameters can be seen in Figure 8.

Name	initial_value	steepness_value	First iteration	Stable iteration
Trial 9	6	9	711	785
Trial 17	5	6	792	792
Trial 5	2	4	444	841
Trial 8	4	7	358	861
Trial 4	3	4	835	882

Table 1: Configurations comparison

[1]First iteration is the first iteration the game reaches 500. Stable iteration is the iteration from here on the score will remain stable at 500.

Best performing (Trial 9) can be seen in the Figure 7 and Figure 6 with both test and train graphs.

2 Reward function

2.1 Introduction

Reward function is one of the most important part of the learning strategy, so it is important to perform some kind of optimization of its parameters.

2.2 Method

Optimization phase is similar to the one described in section 1.3, similar procedures and reasonings have been done. The following are the hyper-parameters considered in this optimization:

- *pos_weight*: is the weight of the penalty based on the distance from the center. The interval used for optimization is [0.0, 3.0].
- *penalty_func*: is the function that penalizes the distance from the center. *abs* is the normal linear one, *exp* is the exponential version.

2.3 Results

Best configurations are reported in Table 2. Graphs of the two trials are shown in Figure 9 and Figure 10. The best performing (Trial 19) for this optimization is shown with both train and test scores in Figure 11 and Figure 12.

Name	pos_weight	penalty_func	First iteration	Stable iteration
Trial 19	0.8623	abs	291	801
Trial 10	0.5864	abs	313	835

Table 2: Configurations comparison 2

Interestingly stable iteration is obtained later than with configurations of the first optimization, but you can easily notice that first iteration is much reduced. I think that the motivation is to be found in the optimization value given to Optuna. To try to rate each trial in a numeric and easily computable way I have chosen to use the first_max_episodes (first iteration in the Table 2) but in this way Optuna converge to configurations that reaches the maximum score as soon as possible even if not in a stable way. To correct this behavior a new measure should be considered. Two approaches can be taken:

- Consider the first iteration valid only if it validates the same max value for at least n episodes (e.g. $n = 10$).
- Consider as a measure the *stable_iteration* maybe the first episode after which the test score does not go down the maximum till the end with n exceptions of episodes (e.g. $n = 3$).

3 Learning from pixels

3.1 Introduction

The most challenging task was to make the NN learn from pixels. Running the *gym* command *env.render(mode='rgb_array')*. The environment used was CartPole-v1.

3.2 Method

I have used two approaches:

3.2.1 CNN

Version 1: In this case I made a CNN able to take the screen and decide the action to do. In the first attempt I gave the CNN a single screenshot of a modified version of the pixels. The image was converted to grayscale, then scaled down with a scale factor of 6, then padded to an height of 100px, normalized and then the array was shaped to the proper shape recognized by the CNN (structure in Figure 13).

Version 2: I soon realized then with only one frame the network would not be allowed to "understand time" and so infer velocity. So the way to do this was to create a structure to maintain last n frames. So I created the Frames class. It was designed to store 8 real frames but to consider them as 4 frames (every time a state was asked, 4 frames were generated making the pixel by pixel maximum between two subsequent frames, this was intended to reduce anti-aliasing effect). With this approach I decided to pre-process images in another way. The image was converted to grayscale, then scaled down with a scaling factor of 5, padded to an height of 120px and then normalized. The CNN structure is in Figure 14.

Version 3: Thinking that maybe It was not a great idea to solve anti-aliasing problem in that way, I have taken this new approach, dealing with anti-aliasing in the pre-processing. This was done as follows: the image was first converted to grayscale, then scaled with a 5 scaling factor, padded to 120px height, then every pixel under a certain threshold of black was deleted (this is intended to solve anti-aliasing effect), and then the image was normalized. In this case the Frames class was much cleaner. The CNN structure was the same as the previous approach (Figure 14) as the image pre-processed was the same.

Version 4: I then thought that maybe the problem was the pre-processing. So I thought that, as the reward function was modified to maintain the cart in the center the important part of the frame was the center so I used the following pre-processing: the image was first converted to grayscale, then reduced to 120x120px taking only the center of the image (centered with the starting point of the cart), then filtered anti-aliasing noise and then normalized. The CNN structure was the same as before, shown in Figure 14.

3.2.2 State extraction from pixels

I created a structure able to store subsequent frames and calculate Cart Position, Cart Velocity, Pole Angle and Pole Angular Velocity, in a quite precise way. Each time a new action was done, a new frame was requested to gym and then added to the Frames object. Each time a state was needed, a state was requested to the Frames object, it would be calculated and then returned. The NN structure was the same as the original one (see it in Figure 15).

3.3 Results

3.3.1 CNN

It was not clear if the failure with this approach was caused by the CNN structure or the pre-processing. I suspect that the problem is only of hyper-parameters tuning that, due to computational limitations and heavy CNN calculations, was done in a very basic way.

3.3.2 State extraction from pixels

With this approach the network was able to win the game at episode 720 and then consistent winning starts from episode 895. Considering that no optimization was done (a part from the increase of *replay_memory_capacity* from 10000 to 100000) and that the state is taken from a small pixel representation, the result was great.

4 MountainCar-v0 environment

4.1 Introduction

In this section I will solve the game with a different environment, MountainCar-v0. The goal of this game is to drive a 2D car up to the top of the right hill. The limitation is that the power of the engine is limited and not sufficient to drive to the hill so to get to the flag you have to use momentum.

4.2 Method

The key idea to solve this problem is to modify the reward function to encourage the system to find the right strategy.

The first solution, the obvious one would be to increment the reward proportionally to the distance from the lowest point. But this way the car tries to drive up to one hill to gain bigger rewards but when it goes down due to force of gravity it suffers the reduction in reward and so at the slowest point it stops and tries to climb the same hill again, wasting momentum acquired.

Second solution was to reward the system only if it decides to drive the motors in the same direction of the current speed direction. So, if the car is driving upon a hill the reward will be proportional to the distance to the lowest point and present only if the the motors spin in the hill's peak direction. When the car will stop due to force of gravity, the reward will be still present only if the car will drive the motors to the direction of the downhill. In this way the car will try to climb the hill, the left or the right one, and when it reaches the maximum height possible it will drive down as fast as possible and it will climb the other hill. The code used to force this behavior is written in Figure 1.

```
pos_weight = 6
if (action == 0 and state[1] < 0) or (action == 2 and state[1] > 0):
    # Motor direction and velocity same sign
    reward = reward + pos_weight * np.abs(state[0] + 0.5)
else:
    reward = reward - 2
```

Figure 1: Reward function

4.3 Results

First solution, obviously, did not lead to good results: the car tries to climb one of the two hills and when it gets to the highest height possible it goes down and near the lowest point it brakes. Then it tries the same hill, again and again. This behavior can be visualized in Figure 17.
Second solution, instead, is working perfectly: the car drives up a hill and when it reaches the maximum height possible it drives down as fast as possible, gaining momentum to drive up on the other hill. This behavior can be visualized in Figure 18.

A Appendix

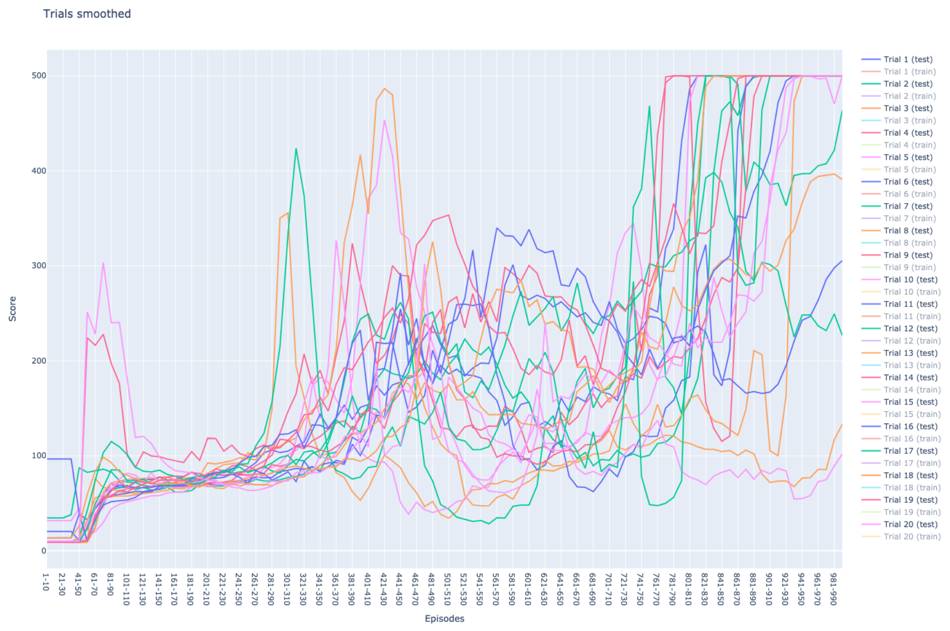

Figure 2: Complete averaged graphs

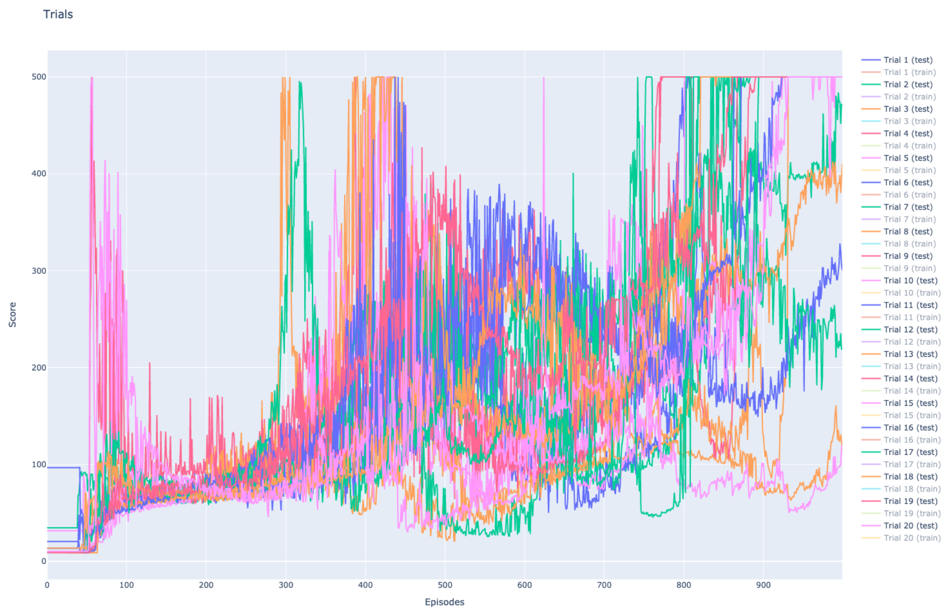

Figure 3: Complete graphs

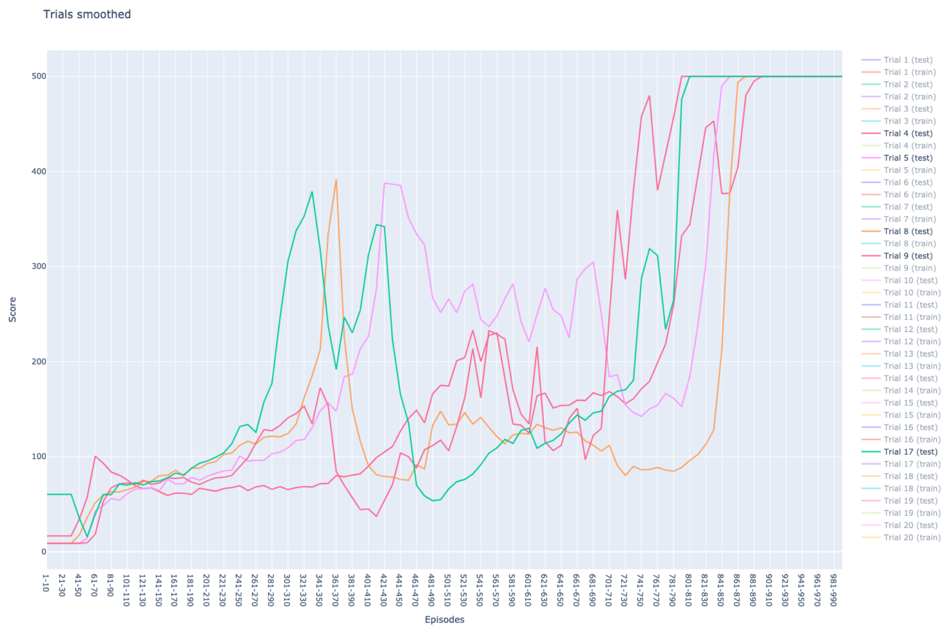

Figure 4: Averaged graphs - Selected trials

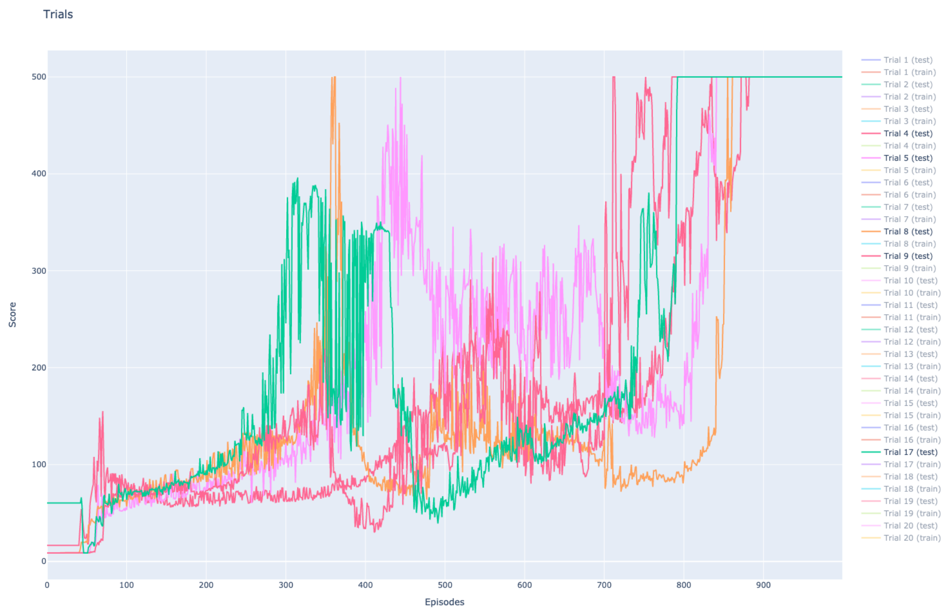

Figure 5: Graphs - Selected trials

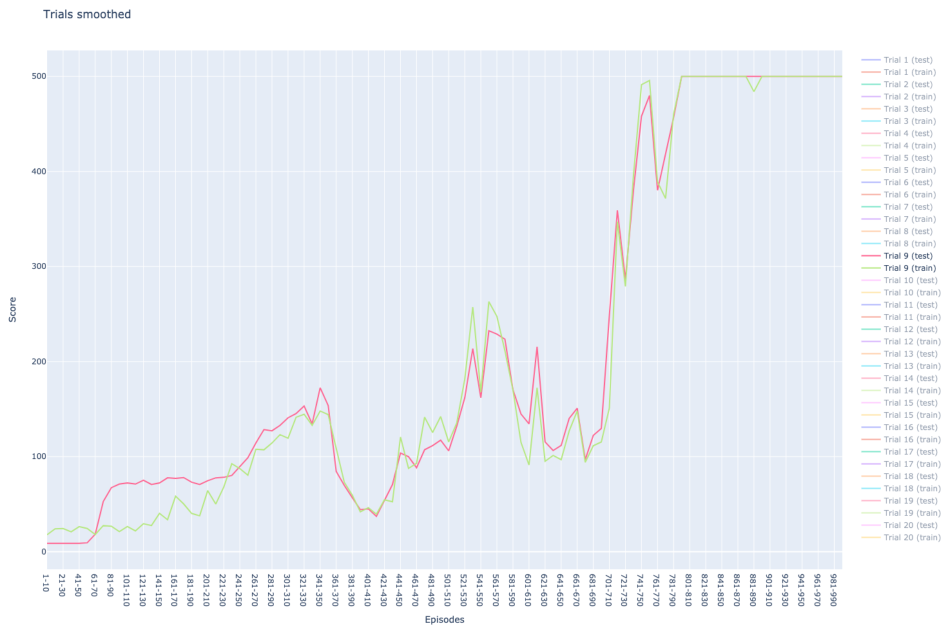

Figure 6: Averaged graphs - Best

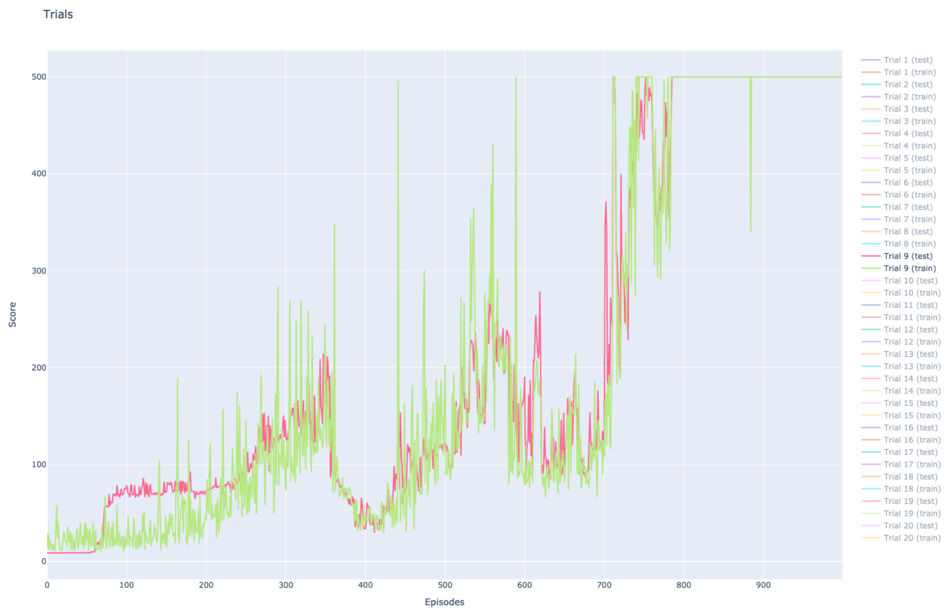

Figure 7: Graphs - Best

```
Trial 1, First episode: 575.0, Hyperparameters: {'initial_value': 7, 'steepness_value': 9}
Trial 2, First episode: 409.0, Hyperparameters: {'initial_value': 7, 'steepness_value': 5}
Trial 3, First episode: 305.0, Hyperparameters: {'initial_value': 7, 'steepness_value': 8}
Trial 4, First episode: 835.0, Hyperparameters: {'initial_value': 3, 'steepness_value': 4}
Trial 5, First episode: 444.0, Hyperparameters: {'initial_value': 2, 'steepness_value': 4}
Trial 6, First episode: 333.0, Hyperparameters: {'initial_value': 4, 'steepness_value': 7}
Trial 7, First episode: 307.0, Hyperparameters: {'initial_value': 9, 'steepness_value': 9}
Trial 8, First episode: 358.0, Hyperparameters: {'initial_value': 4, 'steepness_value': 7}
Trial 9, First episode: 711.0, Hyperparameters: {'initial_value': 6, 'steepness_value': 9}
Trial 10, First episode: 649.0, Hyperparameters: {'initial_value': 7, 'steepness_value': 9}
Trial 11, First episode: 964.0, Hyperparameters: {'initial_value': 9, 'steepness_value': 2}
Trial 12, First episode: 319.0, Hyperparameters: {'initial_value': 9, 'steepness_value': 7}
Trial 13, First episode: 800.0, Hyperparameters: {'initial_value': 9, 'steepness_value': 8}
Trial 14, First episode: 940.0, Hyperparameters: {'initial_value': 8, 'steepness_value': 10}
Trial 15, First episode: 898.0, Hyperparameters: {'initial_value': 6, 'steepness_value': 10}
Trial 16, First episode: 898.0, Hyperparameters: {'initial_value': 8, 'steepness_value': 8}
Trial 17, First episode: 792.0, Hyperparameters: {'initial_value': 5, 'steepness_value': 6}
Trial 18, First episode: 950.0, Hyperparameters: {'initial_value': 8, 'steepness_value': 10}
Trial 19, First episode: 688.0, Hyperparameters: {'initial_value': 6, 'steepness_value': 8}
Trial 20, First episode: 286.0, Hyperparameters: {'initial_value': 9, 'steepness_value': 9}
```

Figure 8: Optimization 1 configurations

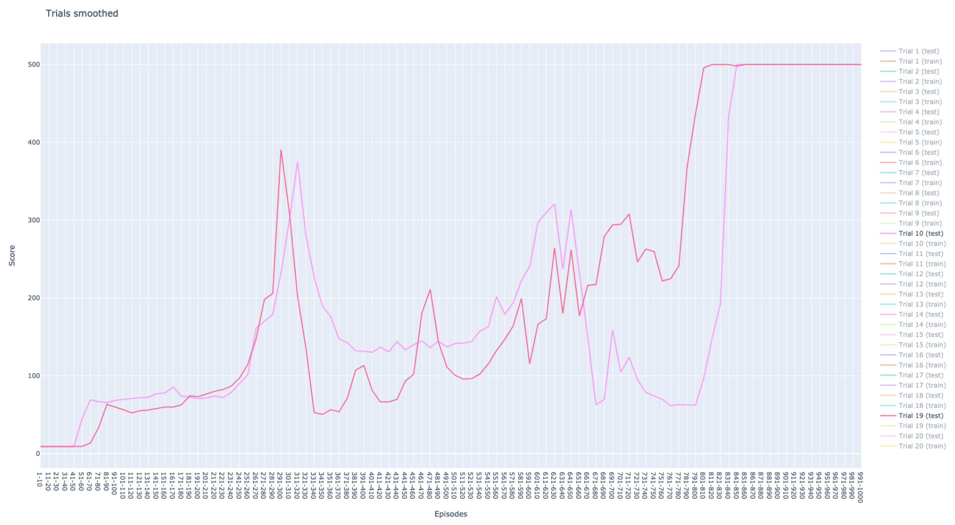

Figure 9: Averaged graphs 2 - Selected trials

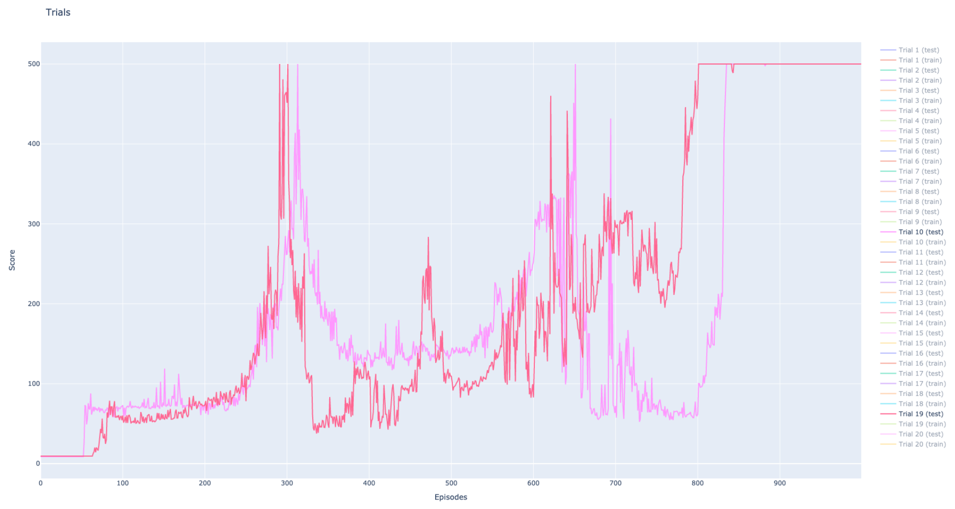

Figure 10: Graphs 2 - Selected trials

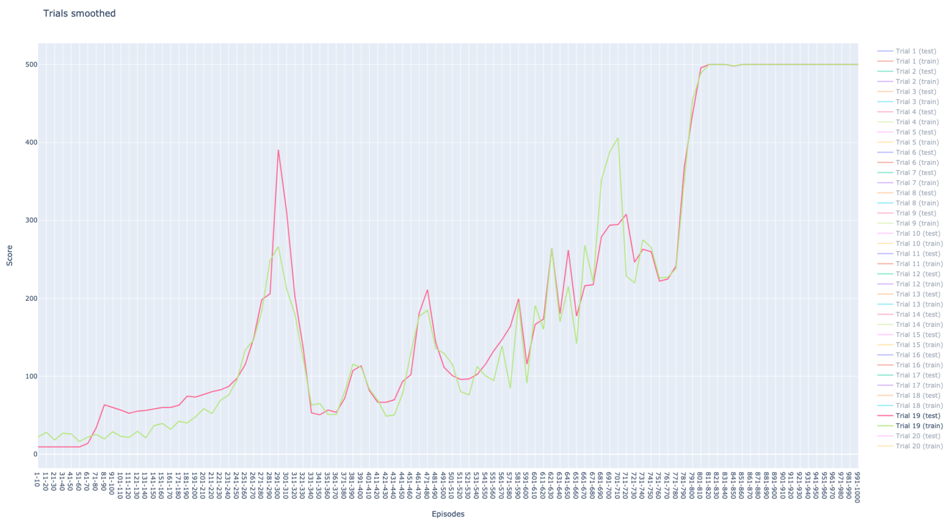

Figure 11: Averaged graphs 2 - Best

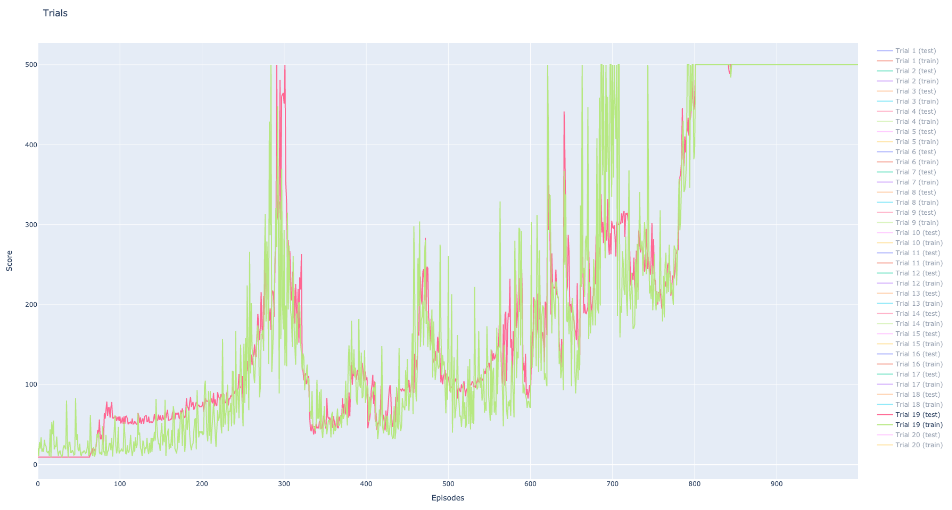

Figure 12: Graphs 2 - Best

```
self.cnn_model = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=10, kernel_size=5, stride=3, padding=2),
    nn.ReLU(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(in_channels=10, out_channels=20, kernel_size=5, stride=2, padding=0),
    nn.ReLU(),
    nn.AvgPool2d(kernel_size=2, stride=1)
)

self.fc_model = nn.Sequential(
    nn.Linear(in_features=720, out_features=160),
    nn.ReLU(),
    nn.Linear(in_features=160, out_features=action_space_dim),
)
```

Figure 13: CNN structure - v1

```
self.cnn_model = nn.Sequential(
    nn.Conv2d(in_channels=4, out_channels=32, kernel_size=8, stride=4, padding=0),
    nn.ReLU(),
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=3, padding=1),
    nn.ReLU(),
    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=0),
    nn.ReLU()
)

self.fc_model = nn.Sequential(
    nn.Linear(in_features=4096, out_features=512),
    nn.ReLU(),
    nn.Linear(in_features=512, out_features=action_space_dim)
)
```

Figure 14: CNN structure - v2, v3 and v4

```
self.linear = nn.Sequential(
    nn.Linear(state_space_dim, 128),
    nn.Tanh(),
    nn.Linear(128, 128),
    nn.Tanh(),
    nn.Linear(128, action_space_dim)
)
```

Figure 15: NN structure - v1

```
Trial 1, First episode: 877.0, Hyperparameters: {'pos_weight': 0.6967633493304056, 'penalty_func': 'exp'}
Trial 2, First episode: 661.0, Hyperparameters: {'pos_weight': 1.6943653083707781, 'penalty_func': 'abs'}
Trial 3, First episode: 688.0, Hyperparameters: {'pos_weight': 0.9634127936868694, 'penalty_func': 'abs'}
Trial 4, First episode: 1001.0, Hyperparameters: {'pos_weight': 0.6361157337308061, 'penalty_func': 'abs'}
Trial 5, First episode: 1001.0, Hyperparameters: {'pos_weight': 0.9917453599882305, 'penalty_func': 'abs'}
Trial 6, First episode: 670.0, Hyperparameters: {'pos_weight': 0.52445295994449, 'penalty_func': 'abs'}
Trial 7, First episode: 1001.0, Hyperparameters: {'pos_weight': 1.9269991720041624, 'penalty_func': 'exp'}
Trial 8, First episode: 1001.0, Hyperparameters: {'pos_weight': 1.2009732992770235, 'penalty_func': 'abs'}
Trial 9, First episode: 1001.0, Hyperparameters: {'pos_weight': 2.6534713831880317, 'penalty_func': 'exp'}
Trial 10, First episode: 313.0, Hyperparameters: {'pos_weight': 0.5864385489732975, 'penalty_func': 'abs'}
Trial 11, First episode: 498.0, Hyperparameters: {'pos_weight': 0.0024408227722856335, 'penalty_func': 'abs'}
Trial 12, First episode: 351.0, Hyperparameters: {'pos_weight': 0.033696321158909984, 'penalty_func': 'abs'}
Trial 13, First episode: 1001.0, Hyperparameters: {'pos_weight': 0.09736191213835121, 'penalty_func': 'abs'}
Trial 14, First episode: 1001.0, Hyperparameters: {'pos_weight': 0.04702815380427134, 'penalty_func': 'abs'}
Trial 15, First episode: 702.0, Hyperparameters: {'pos_weight': 0.25813748695276173, 'penalty_func': 'abs'}
Trial 16, First episode: 564.0, Hyperparameters: {'pos_weight': 2.225616071337061, 'penalty_func': 'abs'}
Trial 17, First episode: 663.0, Hyperparameters: {'pos_weight': 1.3993168507419824, 'penalty_func': 'abs'}
Trial 18, First episode: 962.0, Hyperparameters: {'pos_weight': 0.3741540881943539, 'penalty_func': 'exp'}
Trial 19, First episode: 291.0, Hyperparameters: {'pos_weight': 0.862277838440908, 'penalty_func': 'abs'}
Trial 20, First episode: 982.0, Hyperparameters: {'pos_weight': 0.9114283940353922, 'penalty_func': 'abs'}
```

Figure 16: Optimization 2 configurations

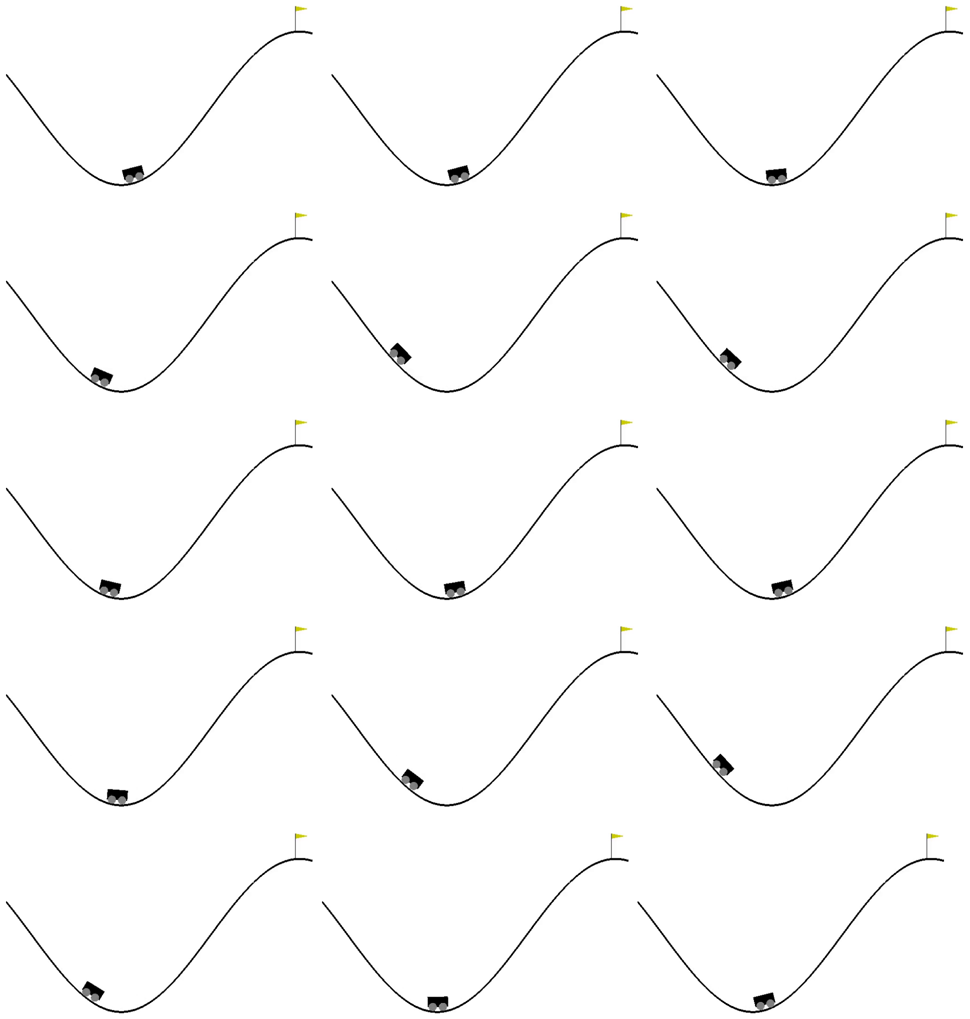

Figure 17: Animation for version 1

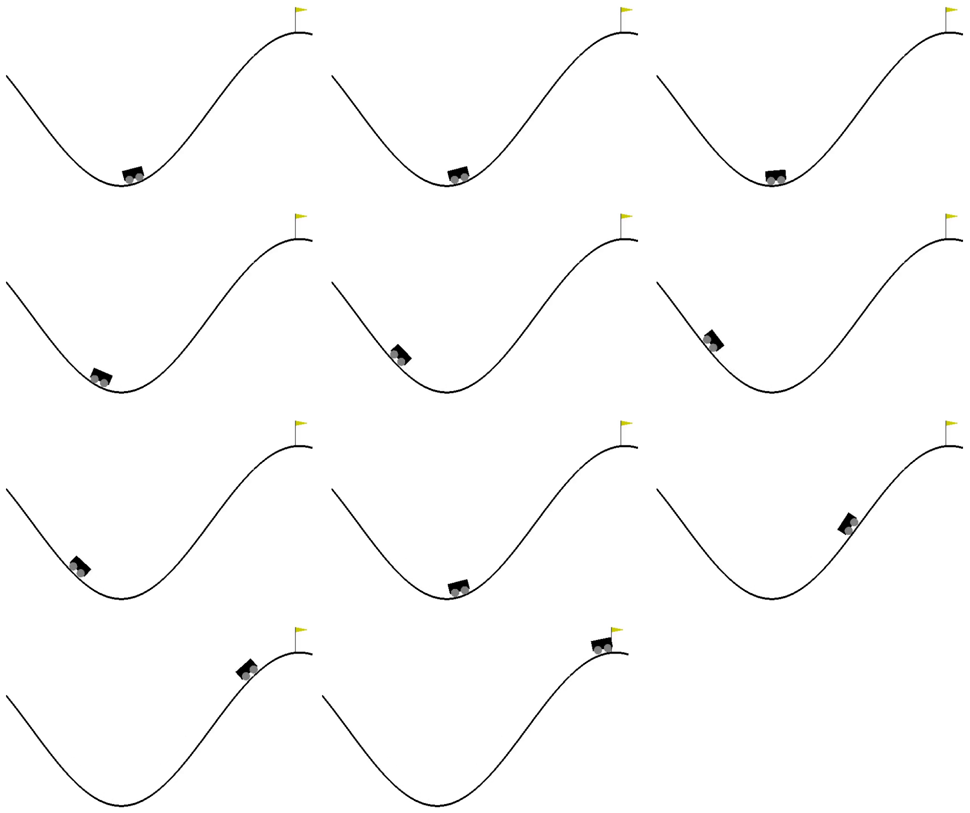

Figure 18: Animation for version 2