EARIN
Laboratory report
EXERCISE 6: Reinforcement learning

Bartłomiej Mastej & Paweł Borsukiewicz

Warsaw University of Technology, Warsaw, Poland

1 Introduction

At the very beginning, there was prepered the environment using the openAI gym for the 'CarRacing-v0' problem. Further, there was created the Proximal Policy Optimization model which was further tuned with the usage of the Grid Search algorithm.

2 Implementation

First of all, there was prepared environment using the openAI gym environment. It was decided to use 'CarRacing-v0' gym instead of 'CarRacing-v1' due to high availability of libraries supporting reinforcement learning.

2.1 Model & learning procedure

It was decided to use the PPO - Proximal Policy Optimization model from the *stable_baselines*3 python module due to the ease of implementation and tuning. Furthermore, this module is implemented with the usage of the *TensorFlow* module, hence the learning can be done with the GPU computational support. The implementation of the model creation with the parameters tuning is presented on the Listing. 1.1. Further, there was used the *learn* method of the module which take the timestep as the parameter. The duration of the model learning is being saved as shown on the Listing. 1.1. Finally, there is a need to give a reward for the given simulation as well as the standard deviation. Therefore, there was used the *evaluate_policy* function also from the *stable_baselines*3 module. As described in the subsection 3 the best model is being saved to a zip file.

Listing 1.1: Model creation

```
#Prepare model
model = PPO(params['policy'], \
            env,\
            verbose=0, \
            learning_rate=params['l_rate'], \
```

```
                tensorboard_log=log_path, seed=2137, \
                n_steps=params['n_steps'], \
                n_epochs=params['n_epochs'])

#Perform learning procedure
print("LEARNING", iter, "/", len(grid))
start = timeit.default_timer()
model.learn(total_timesteps=params['timesteps'])
stop = timeit.default_timer()
l_time = "{0:0.3f}".format(stop - start)

#Evaluate learning outcomes
print("EVALUATING", iter, "/", len(grid))
#Returns (Mean reward, Standard deviation)
#Change render to true to see results
mean, std = evaluate_policy(model, env, n_eval_episodes=3, render=False)
print("Result:", mean)
```

2.2 Grid Search parameters tuning

For the scope of tuning there was used Grid-Search process the following parameters were checked: learning rate, policy, number of steps, number of epochs, and the timestep. The implementation of the Grid Search was done with the usage of the *sklearn* module and it can be seen on the Listing. 1.2. It was decided to use two different policies: *MlpPolicy* (multi-layer perceptron, which was described in detail in the previous laboratory report) and *CnnPolicy* (convolutional neural networks which the most common application is to analyze visual imagery, hence it was decided to test them with the problem). Moreover, there are different learning rates applied so as to find the more optimal solution (the duration of learning process vs optimal weights changing). Next parameters is *n_steps* are and *n_epochs* where the fist one determines the number of steps to process the one batch of data and the second one determines the number of complete passes through the training data set. Finally, the *timesteps* is the number of steps during each the single action will take place and the reward is being given.

Listing 1.2: Grid Seach parameters

```
#Variables for grid search
param_grid = {'l_rate': [0.0001, 0.001, 0.01, 0.1], \
              'policy' : ["MlpPolicy", "CnnPolicy"], \
              'n_steps': [1024, 2048, 4096], \
              'n_epochs': [5, 10, 20], \
              'timesteps': [1000, 10000]}

grid = ParameterGrid(param_grid)
```

Further, there was created the main loop which iterated through all the grid search parameters. The code that is presented on the Listing. 1.1 and Listing 1.4 are inside that loop.

2.3 Results saving/loading

Before main loop of the Grid Search was performed, there was created the *results.csv* file that is presented on the Listing. 1.3 in which there are saved the parameters of the given iteration as well as the mean reward value and the network learning time. The save of the current results takes place just after learning outcomes are achieved Listing. 1.4. Furthermore, it was decided to save only the best pretrained model of the given program execution as can be seen on the Listing. 1.4. The model is being saved in the zip format. The pretrained network can be easily loaded from the file as presented on the Listing. 1.5.

Listing 1.3: Log files creation

```
#Save logs to file
log_path = ('Logs')
filename = "Logs/Results3.csv"
```

Listing 1.4: Pretrained model and parameters saving

```
#Save results
with open(filename,"a") as my_csv:
    csvWriter = csv.writer(my_csv,delimiter=';')
    csvWriter.writerow([params['l_rate'], \
                        params['policy'], \
                        params['n_steps'], \
                        params['n_epochs'], \
                        params['timesteps'], \
                        l_time, mean])

#Check if it is currently the best model
if mean > highscore:
    highscore = mean
    print("New highscore: ", highscore)
    #Save the best model to file in a .zip format
    print("SAVING TO FILE")
    model.save('Logs/Model')
```

Listing 1.5: Loading pretrained network

```
model.load('Logs/Model')
```

3 Resuls & conclusions