METU EE 449
Computational Intelligence

Saving the Princess with a Reinforcement Learning Agent

Ozgur Gulsuna

[a]*Middle East Technical University, Electrical and Electronics Engineering, Ankara, Turkey*

Introduction

This report explores the reinforcement learning processes and applies it to a challenging task that is playing Mario. Two main algorithms are experimented with Proximal Policy Optimization (PPO) and Deep Q-Networks (DQN) are compared in terms of their hyperparametes and performance. The temporal interfacing layer is implemented with Cnn and Mlp and comparison based on prior knowledge is made. The figures are visualized with Tensorboard and the results are discussed.

Keywords: PyTorch; NumPy; Reinforcement Learning; SuperMarioBros; PPO; DQN; Tensorboard;

1. Basic Questions

1. **Agent:** The agent is the entity that interacts with the environment. It tries to maximize the reward it can acquire. In our case, the agent is the Mario character. In supervisied learning the agent can be the model that is trained with the data.

2. **Environment:** The environment is the world in which the agent lives. In other words, the system is the environment, it consist of problems or rewards. In our case, the environment is the Super Mario Bros game. In supervised learning the closest thing to the environment can be the data that is used to train the model.

3. **Reward:** The reward is the feedback that the agent receives from the environment. In each state the reward is calculated and the agent learns upon that reward trying to increase it. In our case, the reward is the score that Mario gets from the game. In supervised learning the reward can be the loss function that is used to train the model.

4. **Policy:** The policy is the series of mappings acting as a ”strategy” that the agent uses to determine the next action. In our case, the policy is the strategy that Mario uses to determine the next action. In supervised learning policy does not exist.

5. **Exploration:** The exploration is the process of the agent trying to find the best policy. In our case, the exploration is the process of Mario trying to find the best strategy to get the highest score. In supervised learning

E-mail address: ozgur.gulsuna@metu.edu.tr

the exploration can be the process of the model trying to find the best parameters to minimize the loss function.

6. **Exploitation:** The exploitation is the process of the agent using the best policy it found and not altering it but perfecting. In our case, the exploitation is the process of Mario tweaking the best strategy it has found to get the highest score but it does not search for a complete new strategy. In supervised learning a comparible term does not exist.

2. Benchmarking

2.1. PPO

The subjects of the experiments considering the PPO algorithm are the following. Each column represents hyperparamters of that algorithm. The hyperparameters are as follows:

Tag	Policy	Learning Rate	Preprocessing	Preprocessing
PPO_1	'CnnPolicy'	0.000001	4 Stack	Grayscale
PPO_2	'CnnPolicy'	0.0001	4 Stack	Grayscale
PPO_3	'CnnPolicy'	0.000001	8 Stack	Grayscale

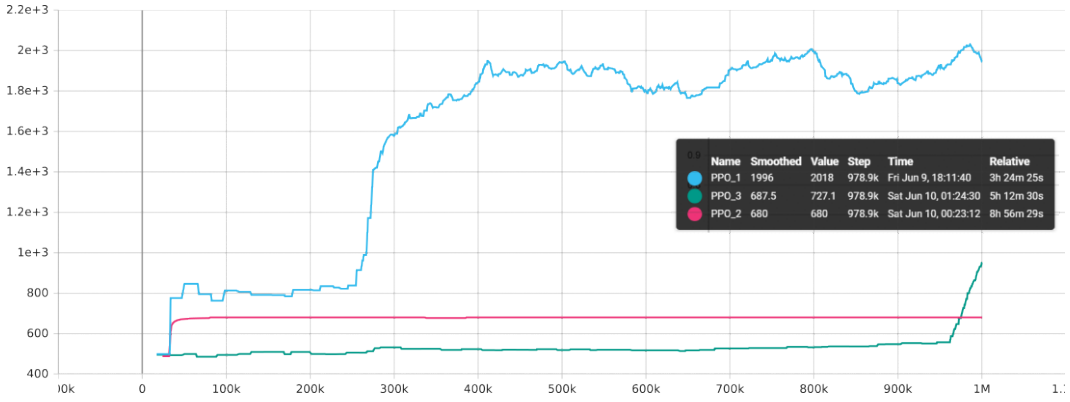

Fig. 1. r_ep_rew_mean vs timesteps (1M)

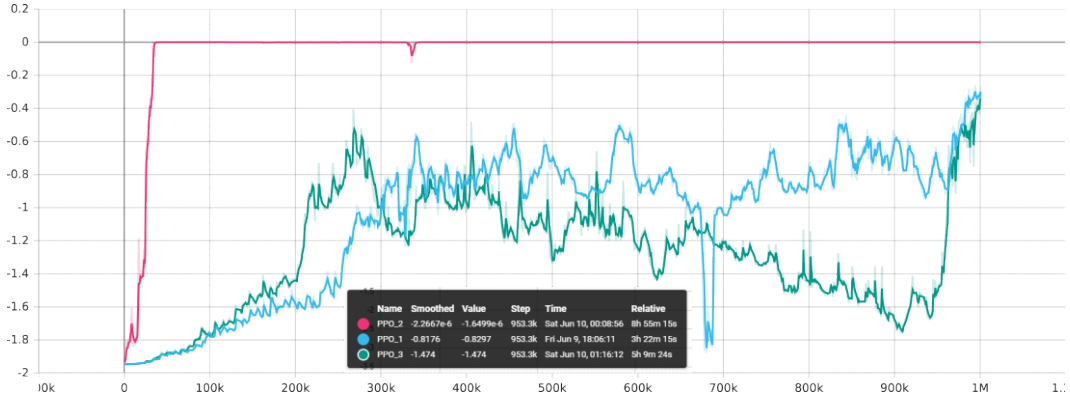

Fig. 2. entropy_loss vs timesteps (1M)

2.2. *DQN*

The subjects of the experiments considering the DQN algorithm are the following. Each column represents hyperparamters of that algorithm. The hyperparameters are as follows:

Tag	Policy	Batch Size	Preprocessing	Buffer Size
DQN_2	'CnnPolicy'	192	4 Stack	10000
DQN_4	'CnnPolicy'	192	4 Stack	20000
DQN_1	'CnnPolicy'	192	8 Stack	10000

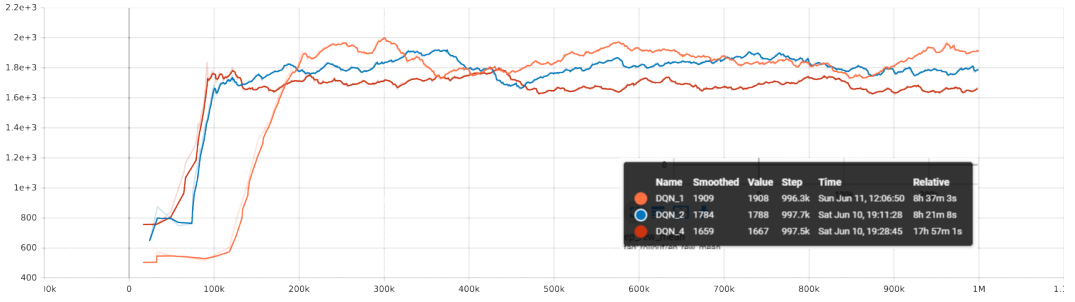

Fig. 3. r_ep_rew_mean vs timesteps (1M)

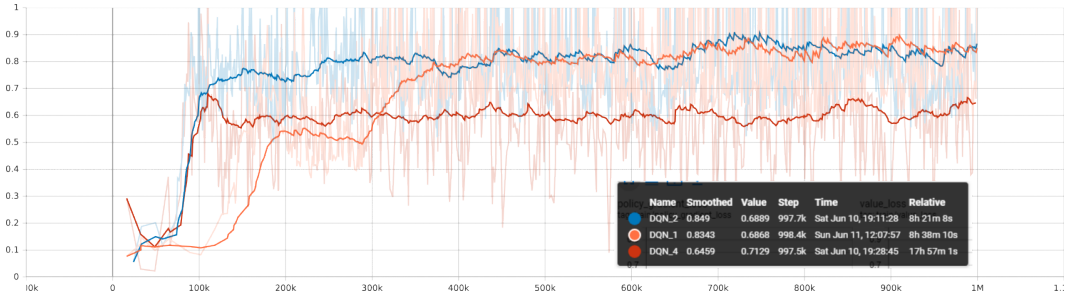

Fig. 4. loss vs timesteps (1M)

2.3. *Comparison*

The comparison of the two algorithms is shown in the following figures.

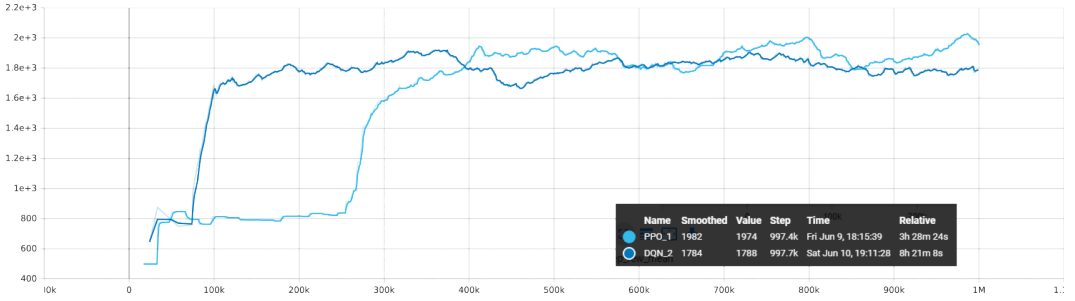

Fig. 5. r_ep_rew_mean vs timesteps (1M)

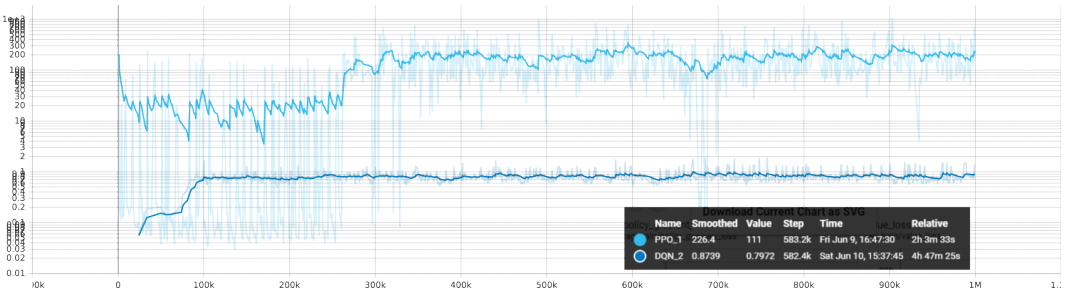

Fig. 6. loss vs timesteps (1M)

3. Discussions

1. Mario struggles to jump over the pipes up to the iteration number between 100000 and 200000. At around 200000 it can get points from blocks and killing enemies at maximum it could get 700 points. In this time period it did not explored much to get points from blocks nor it knows the question marked blocks.

2. DQN is shown to be perform better faster than PPO. But on the end they both saturated around the same reward. When we compare the training time, PPO is trained faster than DQN. Then it is a matter of computation power on the training time.

3. Exploration and exploitation performance can be compared with the reward plot. The DQN exploit more hence it converges faster but the peak performance is limited. PPO explores more and it converge more slow but the peak performance is higher than DQN.

4. In terms of generalization performance, both algoritms are seem to be performing bad. The DQN is little bit better than PPO. PPO likes to jump continuously in places that it is not familiar with. Again it is not easily observer but DQN is little bit better than PPO in terms of both generalization and adaptation.

5. Only a couple hyperparameters are compared. Starting with the PPO, default parameters worked best, increased learning rate made the performance worse. In terms of preprocessing stack size increased to 8 made the training take longer hence its performance can not be compared since it needed more training steps.

   On the DQN case, in terms of reward buffersize does not make a difference. when the losses are compared in buffersize experiment, the 20000 buffer size is better than 10000 buffer size. The higher the buffersize more the RAM required for the training process. In terms of stack size, 8 stack size has a better end game performance but again it took more timesteps to train.

6. DQN consist of deep Q-networks and the training process is more computationally expensive than PPO. The training performance shows that although it is slower to train, it learned in less steps to play Mario. The practical comparison tell us that PPO is better for more ”realtime” applications where DQN is better for more ”offline” applications.

7. Comparison of ’MlpPolicy’ and ’CnnPolicy’ without making an experimentation is possible since it is known that for more complex temporal inputs such as complex game screen, the ’CnnPolicy’ is better since it can understand the important aspects of the complex imputs such as enemies and different textured blocks. The ’MlpPolicy’ is more suited to clean binary type inputs.

4. Performance

Watch @ https://youtu.be/vjRhMvMw6Xc

```
# #@markdown #**Anti-Disconnect for Google Colab**
# #@markdown ## Run this to stop it from disconnecting automatically
# #@markdown  **(It will anyhow disconnect after 6 - 12 hrs for using the free version of Colab.)**
# #@markdown  *(Colab Pro users will get about 24 hrs usage time)*
# #@markdown ---

# import IPython
# js_code = '''
# function ClickConnect(){
# console.log("Working");
# document.querySelector("colab-toolbar-button#connect").click()
# }
# setInterval(ClickConnect,60000)
# '''
# display(IPython.display.Javascript(js_code))

# !pip install setuptools==65.5.0
# !pip install wheel==0.38.4

# !pip install gym_super_mario_bros==7.3.0
# !pip install nes_py stable-baselines3[extra]

import time
from glob import glob


import numpy as np
import os
from stable_baselines3.common.results_plotter import load_results, ts2xy
from stable_baselines3.common.callbacks import BaseCallback
from gym.wrappers import RecordVideo
from time import time

class SaveOnBestTrainingRewardCallback(BaseCallback):
    """
    Taken from https://stable-baselines3.readthedocs.io/en/master/guide/examples.html
    Callback for saving a model (the check is done every ``check_freq`` steps)
    based on the training reward (in practice, we recommend using ``EvalCallback``).
    :param check_freq:
    :param chk_dir: Path to the folder where the model will be saved.
      It must contains the file created by the ``Monitor`` wrapper.
    :param verbose: Verbosity level.
    """
    def __init__(self, save_freq: int, check_freq: int, chk_dir: str, verbose: int = 1):
        super(SaveOnBestTrainingRewardCallback, self).__init__(verbose)
        self.check_freq = check_freq
        self.save_freq = save_freq
        self.chk_dir = chk_dir
        self.save_path = os.path.join(chk_dir, 'models')
        self.best_mean_reward = -np.inf

    def _init_callback(self) -> None:
        # Create folder if needed
        if self.save_path is not None:
            os.makedirs(self.save_path, exist_ok=True)

    def _on_step(self) -> bool:
        if self.n_calls % self.save_freq == 0:
            if self.verbose > 0:
                print(f"Saving current model to {os.path.join(self.chk_dir, 'models')}")
            self.model.save(os.path.join(self.save_path, f'iter_{self.n_calls}'))

        if self.n_calls % self.check_freq == 0:
```

```
          # Retrieve training reward
          x, y = ts2xy(load_results(self.chk_dir), 'timesteps')
          if len(x) > 0:
              # Mean training reward over the last 100 episodes
              mean_reward = np.mean(y[-100:])
              if self.verbose > 0:
                print(f"Num timesteps: {self.num_timesteps}")
                print(f"Best mean reward: {self.best_mean_reward:.2f} - Last mean reward per episode:
{mean_reward:.2f}")

              # New best model, you could save the agent here
              if mean_reward > self.best_mean_reward:
                  self.best_mean_reward = mean_reward
                  # Example for saving best modell
                  if self.verbose > 0:
                    print(f"Saving new best model to {os.path.join(self.chk_dir, 'best_model')}")
                  self.model.save(os.path.join(self.chk_dir, 'best_model'))

        return True

def startGameRand(env):
    fin = True
    for step in range(100000):
        if fin:
            env.reset()
        state, reward, fin, info = env.step(env.action_space.sample())
        env.render()
    env.close()

def startGameModel(env, model):
    state = env.reset()
    while True:
        action, _ = model.predict(state)
        state, _, _, _ = env.step(action)
        env.render()

def saveGameRand(env, len = 100000, dir = './videos/'):
    env = RecordVideo(env, dir + str(time()) + '/')
    fin = True
    for step in range(len):
        if fin:
            env.reset()
        state, reward, fin, info = env.step(env.action_space.sample())
    env.close()

def saveGameModel(env, model, len = 100000, dir = './videos/'):
    env = RecordVideo(env, dir + str(time()) + '/')
    fin = True
    for step in range(len):
        if fin:
            state = env.reset()
        action, _ = model.predict(state)
        state, _, fin, _ = env.step(action)
    env.close()

# Import Environment libraries
import gym_super_mario_bros
from nes_py.wrappers import JoypadSpace
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT

# Start the environment
env = gym_super_mario_bros.make('SuperMarioBros-v0') # Generates the environment
# env = gym_super_mario_bros.make('SuperMarioBrosRandomStages-v0') # Generates the environmen
env = JoypadSpace(env, SIMPLE_MOVEMENT) # Limits the joypads moves with important moves
```

```
# Import preprocessing wrappers
from gym.wrappers import GrayScaleObservation
from stable_baselines3.common.vec_env import VecFrameStack, DummyVecEnv, VecMonitor
from matplotlib import pyplot as plt

# Apply the preprocessing
env = GrayScaleObservation(env, keep_dim=True) # Convert to grayscale to reduce dimensionality
env = DummyVecEnv([lambda: env])

# Alternatively, you may use SubprocVecEnv for multiple CPU processors
env = VecFrameStack(env, 4, channels_order='last') # Stack frames
env = VecMonitor(env, "./train/TestMonitor") # Monitor

#from utils import SaveOnBestTrainingRewardCallback

from stable_baselines3 import DQN
from stable_baselines3 import PPO

# CHECKPOINT_DIR = './train/'
# LOG_DIR = './logs/'

# # #@markdown #**PPO**

# callback = SaveOnBestTrainingRewardCallback(save_freq=100000,
check_freq=1000,chk_dir=CHECKPOINT_DIR)

# model = DQN('CnnPolicy',
#             env,
#             batch_size=192,
#             verbose=1,
#             learning_starts=10000,
#             learning_rate=5e-3,
#             exploration_fraction=0.1,
#             exploration_initial_eps=1.0,
#             exploration_final_eps=0.1,
#             train_freq=8,
#             buffer_size=1000,
#             tensorboard_log=LOG_DIR
#             )
# model.learn(total_timesteps=1000000, log_interval=1, callback=callback)

# model =DQN.load('./DQN_4/train/best_model')
model =DQN.load('./DQN_4/train/models/iter_1000000')
startGameModel(env, model)
# saveGameModel(env,model)
```