Alper Kutay ÖZBEK
2375467

Homework 3 - Reinforcement Learning

1. BENCHMARKING

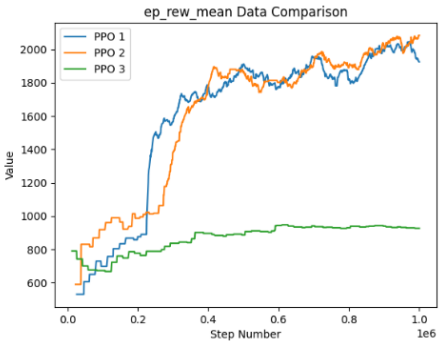

Figure 1 : ep_rew_mean plot of PPO

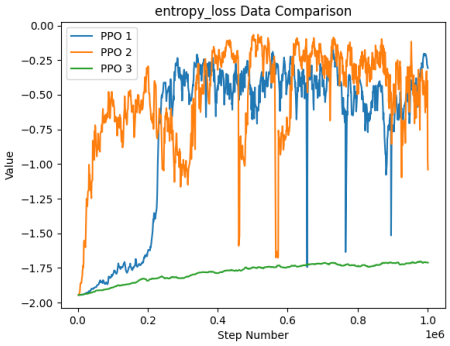

Figure 2 : entropy_loss plot of PPO

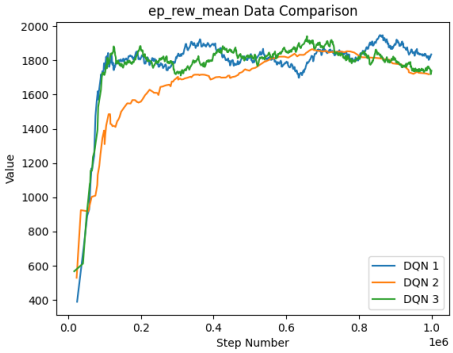

Figure 3 : ep_rew_mean plot of DQN

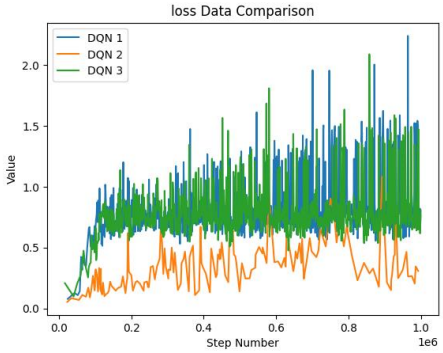

Figure 4 : loss plot of DQN

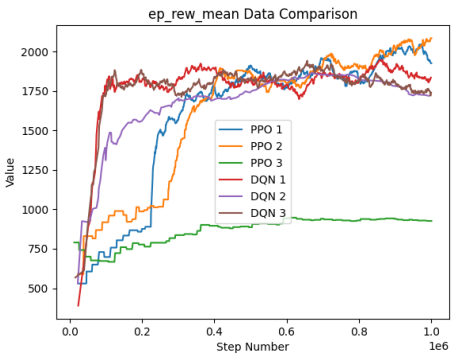

Figure 5 : ep_rew_mean plot of DQN and PPO

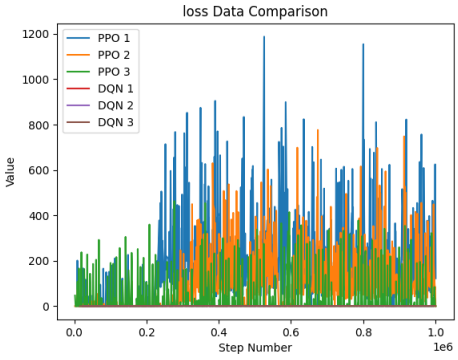

Figure 6 : loss plot of DQN and PPO

It can be seen from figure 5, PPO 2 is the best algorithm in the rest of all. Figures 7 and 8 show only 5 million steps of PPO_2 algorithm. I have seen 900 scores in the video of best algorithm. In addition, It is obvious that DQN algorithms have lower loss data than PPO algorithms.

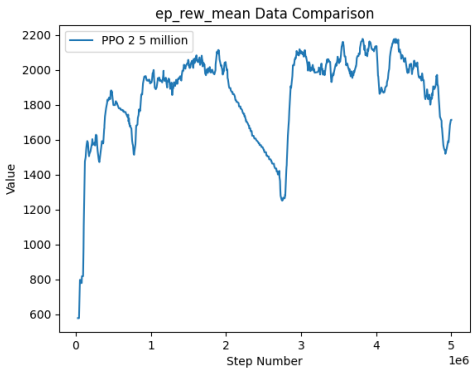

Figure 7 : ep_rew_mean plot of PPO_2

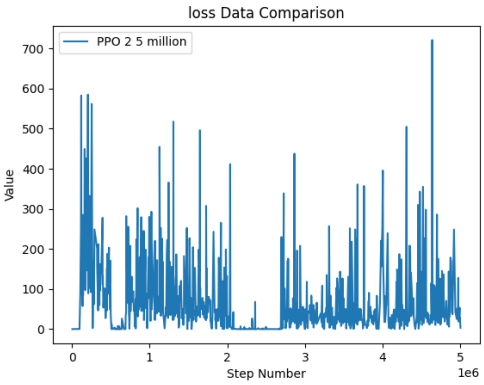

Figure 8 : loss plot of PPO_2

After evaluating the performance of my best algorithms for each type, I decided to change the environment to "Super Mario Bros Random Stages V0". However, upon further observation, it became evident that Mario needed help to learn and perform well on different maps. He encountered difficulties passing through pipes and avoiding lava, resulting in lower scores for the PPO_2 and DQN_1 algorithm.

Video link: https://youtu.be/MLSxFtr9w04

2. DISCUSSIONS

2.1 –

During the learning process, Mario's performance improves over time. Here are the observations and improvements based on the provided information at PPO_2:

At timestep 0 (random):

- Mario's performance could be more random and coordinated.
- He struggles to make successful jumps and frequently encounters obstacles.
- The agent achieves a low score, indicating poor gameplay.

At timestep 10,000:

- Mario's performance shows slight improvements compared to the random start.
- He manages to jump over some shorter pipes but struggles with longer ones.
- The highest scores achieved are modest, reflecting incremental progress.

At timestep 100,000:

- Mario's performance improves further and can pass the second pipe but struggles with the third, the highest pipe.
- He achieves higher scores compared to earlier timesteps, indicating improved gameplay.

At timestep 500,000:

- Mario's performance continues to improve.
- He can now jump over higher pipes more consistently.
- However, he has difficulty navigating and avoiding holes in the ground.

- The highest scores achieved are noticeably higher, demonstrating substantial progress.

At timestep 1 million:

- Mario's performance reaches a relatively stable and competent level.
- He can successfully jump over higher pipes and navigate through some holes.
- However, he still struggles with killing monsters and passing through holes entirely.

Additional training and fine-tuning may be necessary to improve Mario's performance further, focusing on killing monsters and successfully navigating holes. This can include adjusting the agent's reward structure, optimizing hyperparameters, or incorporating specific strategies for dealing with monsters and holes.

By refining the training process and incorporating specific objectives, Mario can enhance his skills in killing monsters and traversing holes, achieving even higher scores, and improving his overall gameplay performance.

By revising the description, we have highlighted the agent's progress in passing different pipes and observed that DQN outperforms PPO in overcoming these challenges at an earlier stage. However, both algorithms still face difficulties effectively dealing with monsters and passing holes. It's important to note that while the agent has shown progress, further refinement may be necessary to enhance its performance in these areas.

2.2-

Based on the observed learning curves of the PPO and DQN algorithms, the DQN algorithm learns faster initially than PPO. However, it reaches a point of diminishing returns and experiences a plateau in its performance. In contrast, PPO starts slower but continues to make steady progress and gradually increases the average episode reward over time.

This suggests that while DQN initially makes rapid improvements, its learning effectiveness begins to level off, and it struggles to achieve further significant enhancements. On the other hand, PPO demonstrates a more consistent and sustainable learning trajectory, continuously refining its performance and adapting to the environment.

The learning curves indicate that PPO has a higher potential for long-term learning and can adapt more effectively to complex and challenging scenarios. While DQN may have a faster initial learning rate, it cannot make continued progress and refine its performance beyond a certain point.

It's important to note that various factors, including hyperparameter settings, reward structures, and the specific characteristics of the environment can influence the performance and efficiency of these algorithms. Further experimentation and analysis can provide deeper insights into the comparative strengths and weaknesses of PPO and DQN in specific contexts.

2.3-

The PPO and DQN algorithms approach exploration and exploitation differently in learning. Using on-policy learning, PPO maintains a balance between exploration and exploitation by constraining policy updates within a certain range. This allows PPO to adaptively explore and exploit the environment, resulting in a controlled and balanced learning process. On the other hand, DQN, employing off-policy learning, uses an epsilon-greedy exploration strategy to balance exploration and exploitation. Initially emphasizing more exploration, DQN gradually shifts towards exploitation as the learning progresses. However, DQN may struggle to strike an ideal balance between exploration and exploitation, potentially leading to suboptimal actions even after significant learning. Overall, PPO is considered better at balancing these aspects throughout the learning process, but the effectiveness of exploration and exploitation strategies may vary depending on the specific problem and environment. Experimentation and fine-tuning hyperparameters are crucial to achieving the desired balance between exploration and exploitation.

2.4-

Based on my observation, DQN appears to be a faster learner and performs better during the early stages of training, specifically before 300,000 timesteps. DQN demonstrates a quicker ability to generalize to new environments or unseen game levels, indicating its robustness and adaptability in handling different scenarios.

On the other hand, while PPO may start slower in terms of performance, it eventually catches up and ends up with higher points than DQN. This suggests that PPO has a higher potential for achieving higher scores and optimizing its gameplay strategy over longer training periods. PPO's slower progress could be attributed to its more cautious and stable learning approach, which allows for a thorough exploration of the policy space.

It's important to note that the comparative performance of these algorithms can vary depending on the specific task, hyperparameter settings, and training duration. The strengths and weaknesses of PPO and DQN make them suitable for different learning scenarios and objectives.

In summary, DQN demonstrates faster learning and better early-stage performance, making it adaptable to new environments or unseen levels. However, PPO showcases its strengths over extended training periods, eventually achieving higher scores. The choice between these algorithms depends on the specific requirements of the task and the desired trade-offs between speed and final performance.

2.5-

According to my observations and figures above, increasing the batch size and buffer size in DQN leads to decreased performance, resulting in slower and delayed learning. This suggests that larger batch and buffer sizes may hinder the algorithm's ability to effectively learn and update its Q-values.Additionally, when comparing the MLP policy to the CNN policy in DQN, it is evident that the MLP policy performs much worse. This highlights the importance of using a CNN policy in DQN, as it is better suited for capturing spatial information and extracting meaningful features from the game environment.In terms of learning rate, increasing it by 50% in DQN with the MLP policy does not significantly impact the learning speed compared to the default learning rate. However, it is worth noting that the default parameters result in higher scores in DQN. This indicates that the default learning rate suits the given task and environment better.

On the other hand, in PPO, increasing the learning rate by 500% and n_steps by 50% leads to higher scores at the end of training. This suggests that larger learning rates and n_steps can accelerate the learning process and improve PPO performance.It's important to remember that the impact of hyperparameter settings can vary depending on the specific task and environment. The optimal hyperparameter values for different algorithms and tasks often require experimentation and tuning to find the best combination.

2.6-

When comparing the computational complexity of PPO and DQN, there are a few key observations. PPO typically has a longer learning curve and takes more time to converge than DQN. This is because PPO involves multiple iterations of policy optimization. However, in terms of computational demands, PPO generally has lower complexity. It does not require a replay buffer or the computation of Q-values like DQN does. This lower complexity makes PPO more practical and scalable in terms of computational resources required. It can be trained on less powerful hardware or in larger-scale training setups. However, it's important to note that these observations are general tendencies and can be influenced by implementation details and task complexity.

2.7-

I used both MLP and CNN policies in both PPO and DQN algorithms, the observations from the figures indicate that the choice of policy has different effects on the performance of the two algorithms.

For PPO, it can be observed that the CNN policy (PPO_1) has a better learning curve and achieves higher performance than the MLP policy (PPO_3). This suggests that PPO benefits from the

spatial processing capabilities of the CNN policy, enabling it to capture and learn meaningful representations from the high-dimensional image inputs. The CNN policy is better suited for handling image-based tasks like Super Mario Bros, as it can effectively extract spatial features and generalize across different levels.

On the other hand, the performance of DQN with MLP policy (DQN_3) surpasses that of DQN with CNN policy (DQN_1) based on the figures. This implies that, contrary to PPO, DQN benefits more from the fully connected layers of the MLP policy rather than the spatial processing capabilities of the CNN policy. The MLP policy might be more effective for DQN in capturing and learning the necessary features from the input states.

3. CODES

```
import torch
import gym_super_mario_bros
from nes_py.wrappers import JoypadSpace
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT
import utils
from gym.wrappers import GrayScaleObservation
from stable_baselines3.common.vec_env import VecFrameStack, DummyVecEnv, VecMonitor
from matplotlib import pyplot as plt
from utils import SaveOnBestTrainingRewardCallback
from stable_baselines3 import PPO
from stable_baselines3 import DQN
for i in range (1,2):
    CHECKPOINT_DIR = "./train"+str(i)+"/"
    LOG_DIR = "./logs/"
    # Start the environment
    # Set the device to use
    env = gym_super_mario_bros.make("SuperMarioBros-v0") # Generates the environment
    env = JoypadSpace(env, SIMPLE_MOVEMENT) # Limits the joypads moves with important moves
    #utils.startGameRand(env)
    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, CHECKPOINT_DIR+"TestMonitor") # Monitor your progress
    callback = SaveOnBestTrainingRewardCallback(save_freq=100000,
check_freq=1000,chk_dir=CHECKPOINT_DIR)
    if(i == 0):
        model = PPO("CnnPolicy", env, verbose=1, tensorboard_log=LOG_DIR,
learning_rate=0.000001,n_steps=512)
    elif( i == 1):
        model = PPO("CnnPolicy", env, verbose=1, tensorboard_log=LOG_DIR,
learning_rate=0.000001*5,n_steps=768)
```

```
    elif(i == 2):
        model = PPO("MlpPolicy", env, verbose=1, tensorboard_log=LOG_DIR,
learning_rate=0.000001*5,n_steps=512)
    model.learn(total_timesteps=1000000, callback=callback)
```

This code is written to train model bu using PPO algorithms.

```
import torch
import gym_super_mario_bros
from nes_py.wrappers import JoypadSpace
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT
import utils
from gym.wrappers import GrayScaleObservation
from stable_baselines3.common.vec_env import VecFrameStack, DummyVecEnv, VecMonitor
from matplotlib import pyplot as plt
from utils import SaveOnBestTrainingRewardCallback
from stable_baselines3 import PPO
from stable_baselines3 import DQN
for i in range (3,6):
    CHECKPOINT_DIR = "./train"+str(i)+"/"
    print(CHECKPOINT_DIR)
    LOG_DIR = "./logs/"
    # Start the environment
    # Set the device to use
    env = gym_super_mario_bros.make("SuperMarioBros-v0") # Generates the environment
    env = JoypadSpace(env, SIMPLE_MOVEMENT) # Limits the joypads moves with important moves
    #utils.startGameRand(env)
    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, CHECKPOINT_DIR+"TestMonitor") # Monitor your progress
    callback = SaveOnBestTrainingRewardCallback(save_freq=100000,
check_freq=1000,chk_dir=CHECKPOINT_DIR)
    if(i == 3):
        model = DQN("CnnPolicy",
                    env,
                    batch_size=192,
                    verbose=1,
                    learning_starts=10000,
                    learning_rate=0.001,
                    exploration_fraction=0.1,
                    exploration_initial_eps=1.0,
                    exploration_final_eps=0.1,
                    train_freq=8,
                    buffer_size=10000,
                    tensorboard_log=LOG_DIR)
```

```
    elif( i == 4):
        model = DQN("CnnPolicy",
                    env,
                    batch_size=216,
                    verbose=1,
                    learning_starts=5000,
                    learning_rate=5e-3,
                    exploration_fraction=0.1,
                    exploration_initial_eps=1.0,
                    exploration_final_eps=0.1,
                    train_freq=8,
                    buffer_size=12000,
                    tensorboard_log=LOG_DIR)
    elif(i == 5):
        model = DQN("MlpPolicy",
                    env,
                    batch_size=192,
                    verbose=1,
                    learning_starts=10000,
                    learning_rate=8e-3,
                    exploration_fraction=0.1,
                    exploration_initial_eps=1.0,
                    exploration_final_eps=0.1,
                    train_freq=8,
                    buffer_size=10000,
                    tensorboard_log=LOG_DIR)
    model.learn(total_timesteps=1000000, log_interval=1, callback=callback)
```

This code is written to train model bu using DQN algorithms.

```
#this code is written to start best game model
import torch
import gym_super_mario_bros
from nes_py.wrappers import JoypadSpace
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT
import utils
from gym.wrappers import GrayScaleObservation
from stable_baselines3.common.vec_env import VecFrameStack, DummyVecEnv, VecMonitor
from matplotlib import pyplot as plt
from utils import SaveOnBestTrainingRewardCallback
from stable_baselines3 import PPO
from stable_baselines3 import DQN
env = gym_super_mario_bros.make("SuperMarioBros-v0") # Generates the environment
env = JoypadSpace(env, SIMPLE_MOVEMENT) # Limits the joypads moves with important moves
#utils.startGameRand(env)
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, "train5_million/TestMonitor") # Monitor your progress
model = PPO.load("./train5_million/best_model")
utils.startGameModel(env, model)
```

```
#this code is written to plot ep_rew_mean and loss of DQN algorithms
import json
import matplotlib.pyplot as plt
dir = "loss/"
# Read the JSON files
with open(dir+'DQN_1.json') as file:
    data1 = json.load(file)
with open(dir+'DQN_2.json') as file:
    data2 = json.load(file)
with open(dir+'DQN_3.json') as file:
    data3= json.load(file)

# Extract the data
x1 = [item[1] for item in data1]
y1 = [item[2] for item in data1]

x2 = [item[1] for item in data2]
y2 = [item[2] for item in data2]

x3 = [item[1] for item in data3]
y3 = [item[2] for item in data3]

# Create the figure and axis objects
fig, ax = plt.subplots()

# Plot the data on the axis
ax.plot(x1, y1, label='DQN 1')
ax.plot(x2, y2, label='DQN 2')
ax.plot(x3, y3, label='DQN 3')

# Set the labels and title
ax.set_xlabel('Step Number')
ax.set_ylabel('Value')
ax.set_title('loss Data Comparison')

# Display the legend
ax.legend()
plt.savefig(dir+" DQN.png")
# Show the plot
plt.show()
```

```python
#this code is written to plot ep_rew_mean and entropy loss of PPO algorithms
import json
import matplotlib.pyplot as plt
dir = "ep_rew_mean/"
# Read the JSON files
with open(dir+'PPO_1.json') as file:
    data1 = json.load(file)
with open(dir+'PPO_2.json') as file:
    data2 = json.load(file)
with open(dir+'PPO_3.json') as file:
    data3= json.load(file)

# Extract the data
x1 = [item[1] for item in data1]
y1 = [item[2] for item in data1]

x2 = [item[1] for item in data2]
y2 = [item[2] for item in data2]

x3 = [item[1] for item in data3]
y3 = [item[2] for item in data3]

# Create the figure and axis objects
fig, ax = plt.subplots()

# Plot the data on the axis
ax.plot(x1, y1, label='PPO 1')
ax.plot(x2, y2, label='PPO 2')
ax.plot(x3, y3, label='PPO 3')

# Set the labels and title
ax.set_xlabel('Step Number')
ax.set_ylabel('Value')
ax.set_title('ep_rew_mean Data Comparison')

# Display the legend
ax.legend()
plt.savefig(dir+" PPO.png")
# Show the plot
plt.show()
```