REINFORCEMENT LEARNING PART 2

Denitsa Goranova

Contents

Introduction

Since we already have the foundation from part 1 for the Taxi-v3 environment, where I enabled a discrete agent to learn from a small state space, now I want to explore a bit more complex and dynamic setting, which is balancing a pole in the CartPole-v1 environment while using deep reinforcement learning.

The CartPole problem that I will now look into is working in a continuous work space. It involves a pole attached to a cart that is moving along in a horizontal track. The goal is to apply forces to keep the pole alanced upright for as long as possible.

There are four key variables:

1. Cart Position- horizontal position of the cart
2. Cart Velocity – speed at which the cart is moving
3. Pole Angle – angle of the pole
4. Pole Angular Velocity – the rate at which the pole is falling

The agent is receiving a reward of +1 for every timestep the pole remains upright so it can be encouraged to stay stable as long as possible.

Implementation

Since in the first part of this assignment I was using Taxi, I am now transitioning to a more complex environment such as CartPole. I started by implementing Q-Network. This neural network is receiving the environment's continuous state and outputs the Q-values for each possible action which in this environment is left or right. I used two hidden layers with ReLu activstions

Building the Deep Q-Network

```
def build_dqn_model(state_size, action_size):
    model = models.Sequential()
    model.add(layers.Dense(24, input_dim=state_size, activation='relu'))
    model.add(layers.Dense(24, activation='relu'))
    model.add(layers.Dense(action_size, activation='linear'))
    model.compile(loss='mse', optimizer=optimizers.Adam(learning_rate=0.001))
    return model

model = build_dqn_model(state_size, action_size)
model.summary()
```

14] Python

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_12 (Dense)            (None, 24)                120

 dense_13 (Dense)            (None, 24)                600

 dense_14 (Dense)            (None, 2)                 50

=================================================================
Total params: 770 (3.01 KB)
Trainable params: 770 (3.01 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 1- Building the DQN

Unlike the Taxi where each state-action pair could be stored in a table, here the situation is a but different. The CartPole environment has its spaces as continuous which means I can't put its values in a traditional Q-table. (the values are going to be infinite). Then the Deep Q-Network comes handy- it is learning to approximate the Q-values for any state-action pair.

Experience Relay

I also implemented a replay buffer that is storing the past experiences in a queue and is randomly sampling mini-batches during the training process.

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def add(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)

    def __len__(self):
        return len(self.buffer)

buffer_capacity = 2000
replay_buffer = ReplayBuffer(buffer_capacity)
```

Figure 2- Experience relay

This buffer is breaking the correlation between consecutive steps, making the training data more diverse and less biased. Why are we using this now and not in the Taxi environment? This model will learn from recent transitions only, which could be viable when we speak about overfitting. By reusing the past experience we have, the agent would learn more efficiently and it will be generelizing better.

Target Network and epsilon-greedy exploration startegy

I also added a target network that is copying the wights from the main DQN, which is helping to stabilize the Q-value updates.

```python
gamma = 0.95
epsilon = 1.0
epsilon_min = 0.01
epsilon_decay = 0.995
batch_size = 32
episodes = 500
target_update_freq = 10

target_model = build_dqn_model(state_size, action_size)
target_model.set_weights(model.get_weights())

rewards_list = []
loss_history = []
```

Figure 3- Target network

In order to balance the exploration and exploitation, I used an epsilon-greedy strategy, starting with full exploration and decaying over time.

Monitoring Rewards and training loss

I also tracked rewards pers episode and training loss per update step.

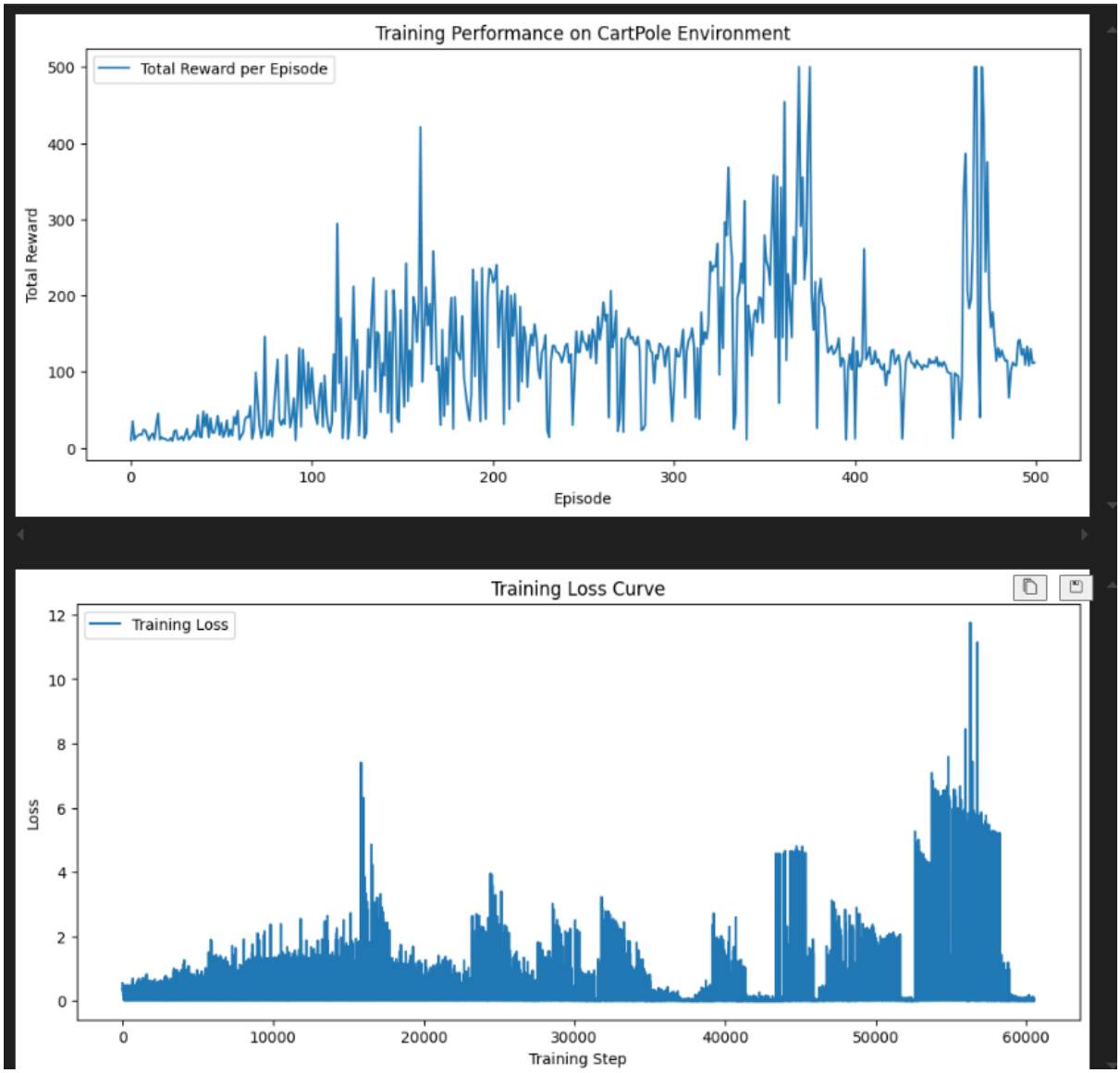

Figure 4- Performance of model during episodes and steps

What we can say based on those graphs is the following:

The evolution of the agent's overall reward across 500 training events is seen in this plot. Because to random exploration and untrained Q-values, rewards are minimal in the early stages (Episodes 0–100). We see a steady increasing trend beginning at Episode 100, suggesting that the agent is improving at balancing the pole. Peaks over 300 and 400 reward show that the agent sometimes performs almost flawlessly. The policy isn't yet completely stable, though, as seen by the jumps and falls, which still change based on state initializations and little changes in the learnt Q-function.
By steadily improving performance and regularly getting large rewards, the agent exhibits evident learning. Nonetheless, some instability persists, indicating that more occurrences might lead to improvements.

The prediction error of the neural network changed throughout training, as seen by the loss curve. Moments when the agent experiences states or transitions that differ from its present expectations are represented as jumps in the loss; these usually occur when the policy experiences major modifications or when the agent moves from exploration to exploitation. We

see lower, more consistent variations between steps 10,000 and 40,000, suggesting a more stable learning period. Large spikes in the last section, however, point to either insufficient buffer variety, policy changes, or overfitting.

The network is learning as seen by the loss, which occasionally varies greatly but typically tends downward. Because the learning objective is non-stationary the spikes are helpful markers of instability or unexpected transitions, which are frequently observed in reinforcement learning.

Moving average for smoothed rewards

To better interpret trends in the noisy reward data, I plotted a moving average over the rewards

Figure 5- Smoothed training performance

A clear picture of the agent's learning development throughout training is given by the smoothed reward curve. At first, the agent shows consistent progress, with average rewards progressively rising as it gains pole-balancing expertise. We see a decline in performance at episode 150, which is probably caused by instability during the shift from exploration to exploitation. This is followed by a robust recovery, which results in peak performance between episodes 300 and 350, when rewards surpass 250. This indicates the agent has picked up a useful policy. Nevertheless, the plot also exhibits obvious spikes in later phases, such as a sharp decline following episode 350 and another toward the conclusion of training. Policy instability or continued exploration might be the cause of these declines.

Dueling Double DQN

To further enhance learning stability and performance, I implemented a Dueling Double DQN. This separates the estimation of state values and action advantages, helping the model learn better representations of the environment's value dynamics

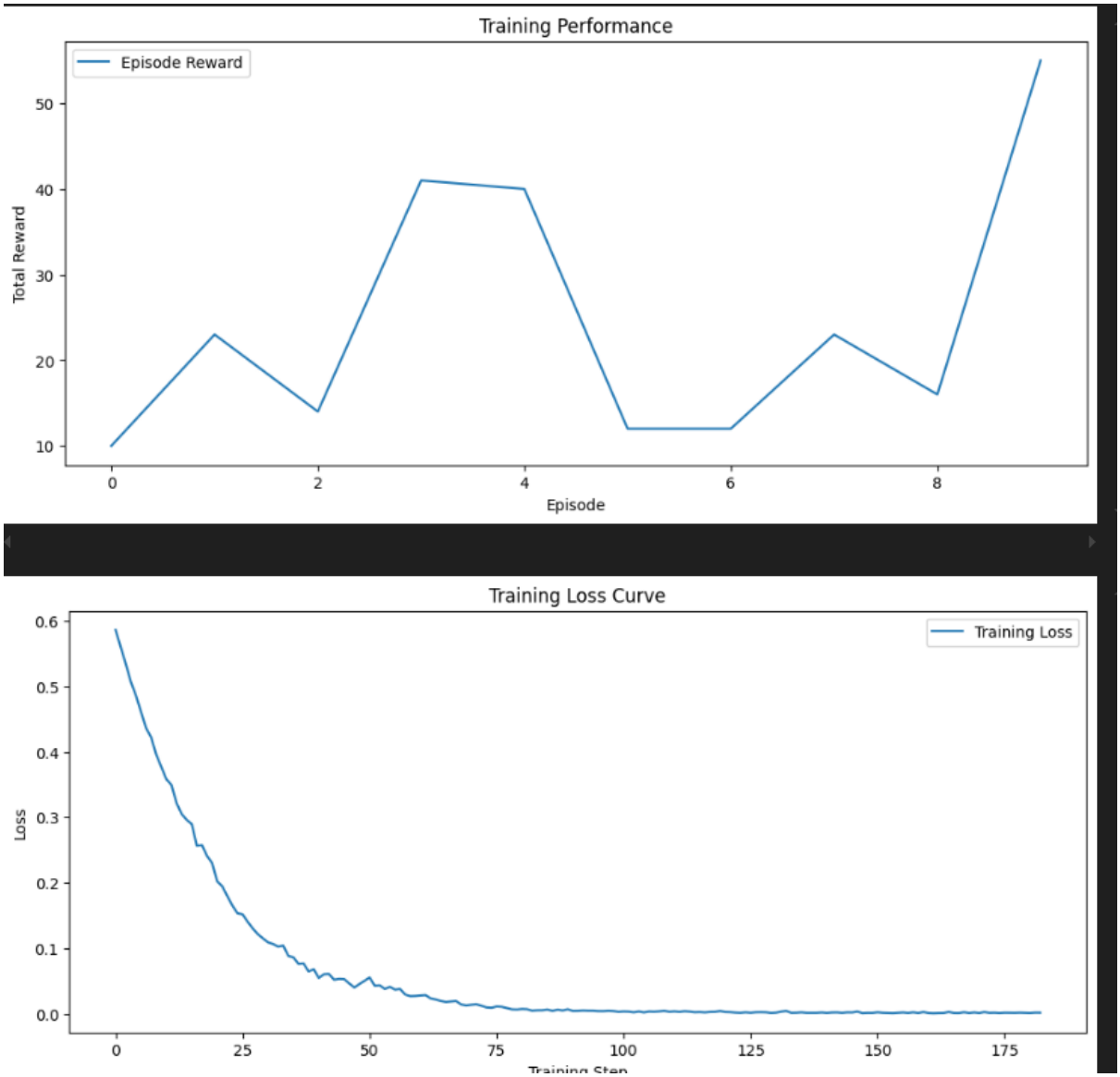

Figure 6- Training loss and rewards during process

In the Training Performance plot (top), we observe episode rewards over the first 10 episodes. The agent starts with relatively low performance, and although there's a noticeable spike around Episode 3–4 (reaching ~40+ reward), the performance is inconsistent. This is expected during the initial exploration phase, where the agent is still randomly sampling actions and learning the environment dynamics. The final episode shows a significant jump (above 50), which may indicate the early signs of learning. In the Training Loss Curve (bottom), the model's loss decreases rapidly and smoothly. Starting around 0.6, it drops below 0.1 within the first 50 steps and continues to decline toward zero. This indicates that the neural network is effectively learning to minimize prediction error on Q-values, even if this hasn't yet fully translated into stable performance.

Conclusion

From discrete Q-learning in the Taxi-v3 environment to deep reinforcement learning in the CartPole-v1 setting, this project offered an insightful educational experience that covered the

foundations and development of reinforcement learning.

Using a straightforward, limited state space, I examined the fundamentals of Q-learning in tabular form in Part 1. This helped me understand important ideas like reward shaping, exploration-exploitation trade-offs (epsilon-greedy vs. Boltzmann), state-action value estimate, and the implications of various hyperparameters. I learned how an agent gradually improves its behavior with just rewards and punishments through experiments, visualizations, and policy assessments. I also learned how design decisions may direct or speed up the learning process.

Moving on to Part 2, I used those fundamental concepts in CartPole, a more dynamic and complicated environment where the continuous structure of the state space renders typical Q-tables impractical. In order to achieve consistent learning, I developed a Deep Q-Network (DQN) here, incorporating crucial processes like epsilon decay, target networks, and experience replay. I witnessed the whole learning lifecycle—early exploration, increasing performance, actions refining, and the random instability that deep agents encounter—by experimenting with network designs, smoothing reward monitoring, and ultimately the usage of a Dueling Double DQN.

In both sections, I discovered that reinforcement learning is a continuous adaptation process in which agents get better by trial-and-error feedback in unpredictable situations rather than direct teaching. I also seen firsthand how important design choices directly affect the agent's learning effectiveness, such as how we express state-action values, explore, and stabilize training.

I gained a solid mathematical and practical understanding of RL through this journey from simple to complicated, which connected theory to actual application. These experiences have equipped me to investigate even more complex RL subjects in the future, such continuous action spaces, policy gradient techniques, and practical applications in robotics and gaming AI.

GIT repository:
https://github.com/goranova14/AI_Advanced