DEVELOPMENT AND OPTIMIZATION OF DEEP Q-NETWORKS FOR AUTONOMOUS LUNAR LANDER CONTROL

Krishnapriya Suresh
Anmol Kaul
Archit Saxena

BRIEF PROBLEM STATEMENT :

The project focuses on designing and optimizing a Deep Q-Network (DQN) to autonomously control a lunar lander in a simulated environment. The primary challenge is to enable the lander to:

Safely Reach the Target: Navigate and land on a specified location on the lunar surface.

Optimal Thrust Control: Manage the thrust to balance fuel efficiency and control during descent.

Precise Orientation: Adjust the lander's orientation to ensure a stable approach to the landing site.

Control Landing Speed: Minimize landing velocity to avoid crashes.

STATES AND ACTIONS

The input state of the Lunar Lander consists of following components:

1. Horizontal Position
2. Vertical Position
3. Horizontal Velocity
4. Vertical Velocity
5. Angle
6. Angular Velocity
7. Left Leg Contact
8. Right Leg Contact

The actions of the agents are:

1. Do Nothing
2. Fire Main Engine
3. Fire Left Engine
4. Fire Right Engine

The reward structure is as follows:

- +100/-100 for coming to rest or crashing.
- Ground contact for each leg is +10
- Firing engine is -0.3 The benchmark for winning the game is 200 points

CURRENT BENCHMARKS :

The benchmarks for DQN on the Lunar Lander problem include:

Original DQN (Mnih et al., 2015): Basic implementation with experience replay and target networks.

Double DQN: Reduces overestimation bias, leading to more stable training.

Dueling DQN: Separates the state value and action advantage, enhancing learning efficiency.

Rainbow DQN: Integrates multiple improvements such as prioritized experience replay, multi-step learning, and distributional RL.

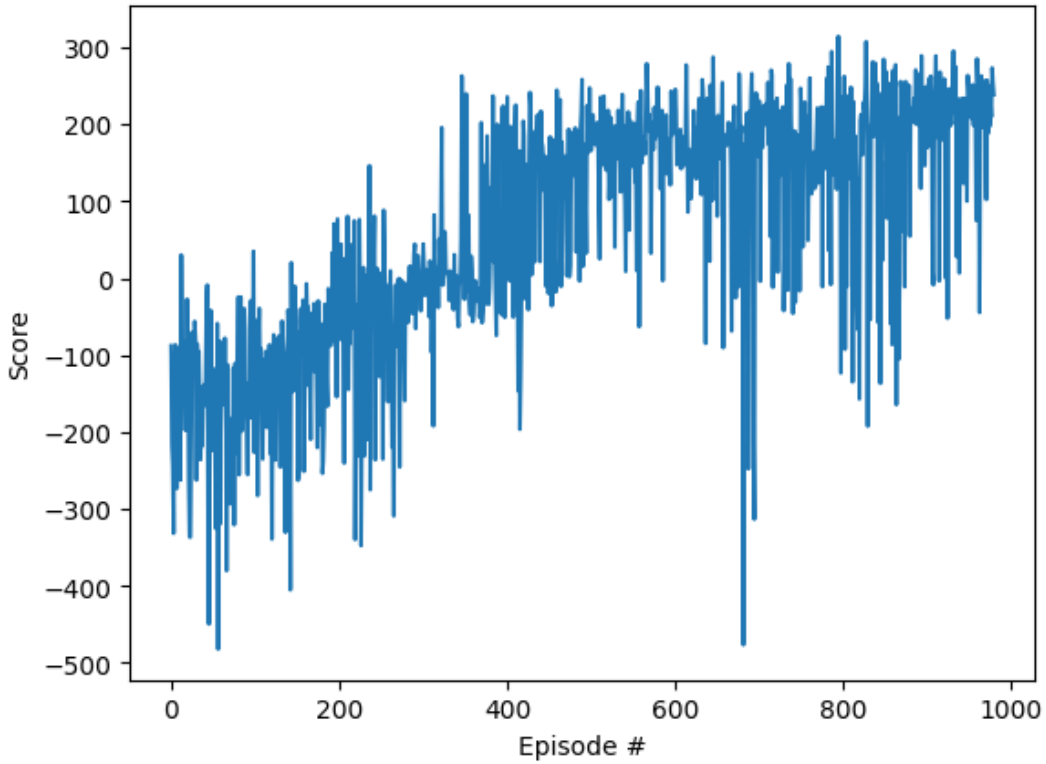
VANILA DQN

Vanilla DQN is a reinforcement learning algorithm that combines Q-learning with deep neural networks to approximate the Q-value function in high-dimensional state spaces. Instead of storing Q-values for each state-action pair, a neural network generalizes them, enabling efficient learning in complex environments.

Key features include **experience replay**, which stabilizes learning by sampling from past experiences, and a **target network** that reduces oscillations by periodically updating weights from the main network during training.

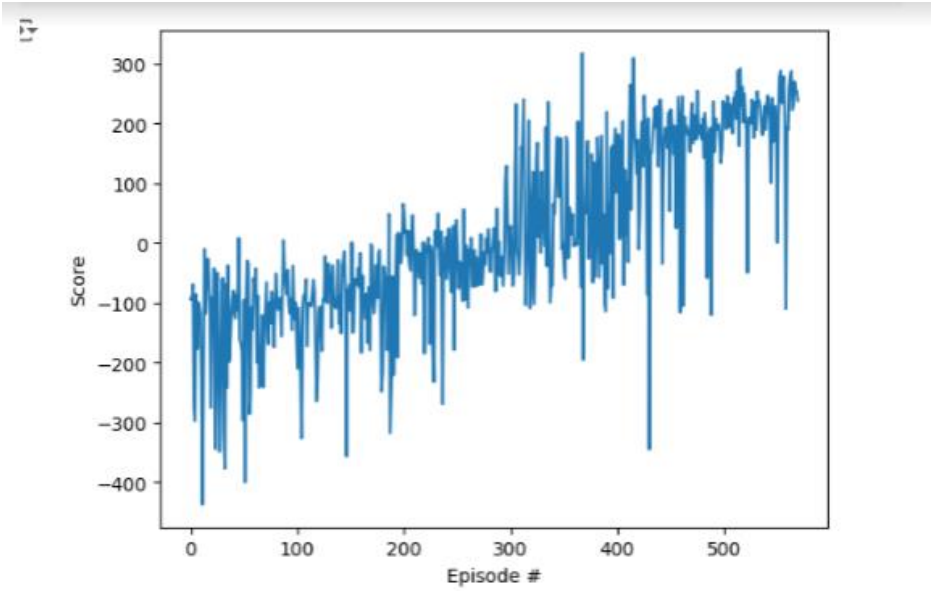
VANILLA DQN IMPLEMENTATION:

- We compute the loss and update the weights every 5 iterations with a batch size of 32. Our future value discount rate, gamma, is 0.99. The 200 average score threshold is reached at around 400 episodes.

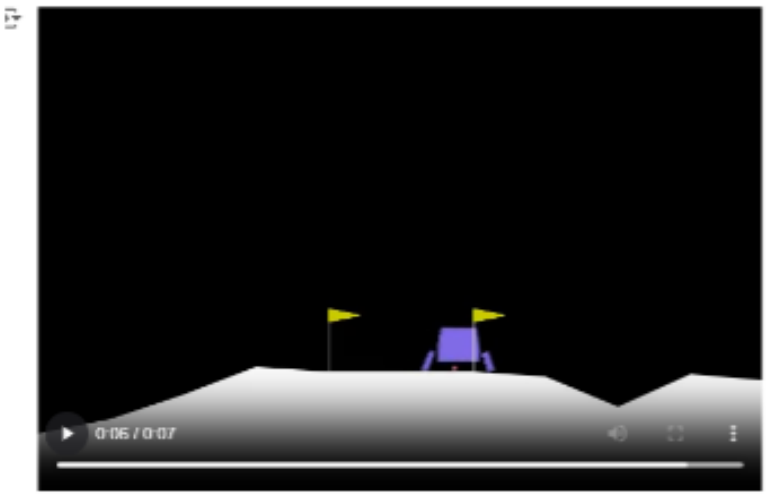


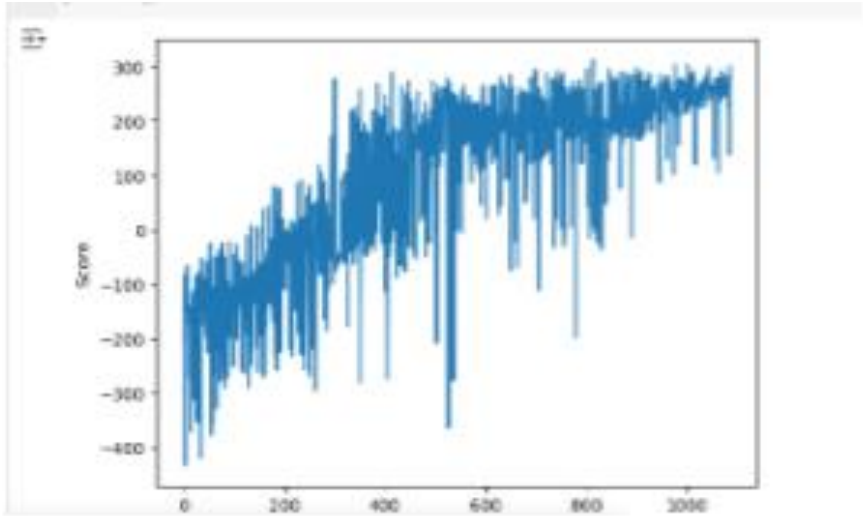
```python
class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = self.fc1(state)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        return self.fc3(x)
```

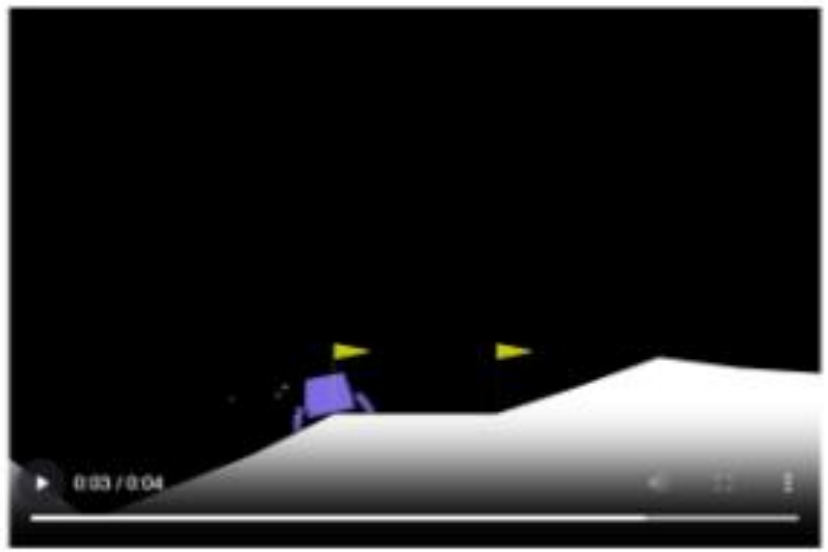
```python
[ ] BUFFER_SIZE = int(1e5)  # replay buffer size
    BATCH_SIZE = 64         # minibatch size
    GAMMA = 0.99            # discount factor
    TAU = 1e-3              # for soft update of target parameters
    #LR = 5e-4               # learning rate
    LR = 0.01               # learning rate
    UPDATE_EVERY = 4        # how often to update the network
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```



For LR = 0.008 , BATCH =4 , GAMMA=0.99 AND EPISODES THREASHOLD = 200



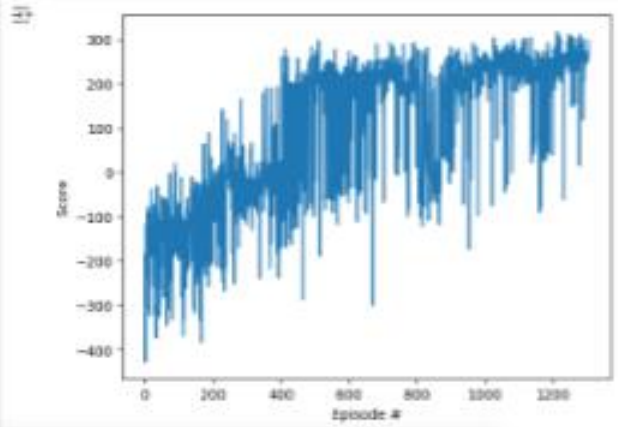




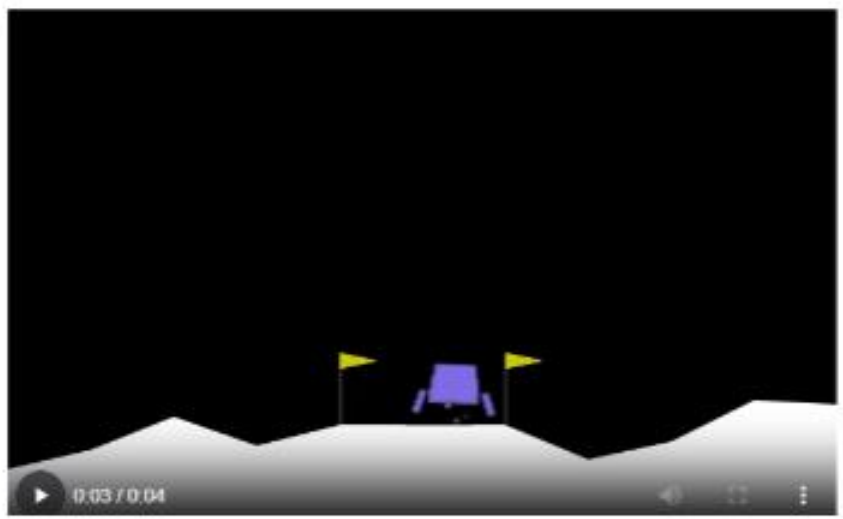

```
        return self.fc3(x)
```

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
#LR = 5e-4               # learning rate
LR = 0.0008              # learning rate
UPDATE_EVERY = 4        # how often to update the network
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```

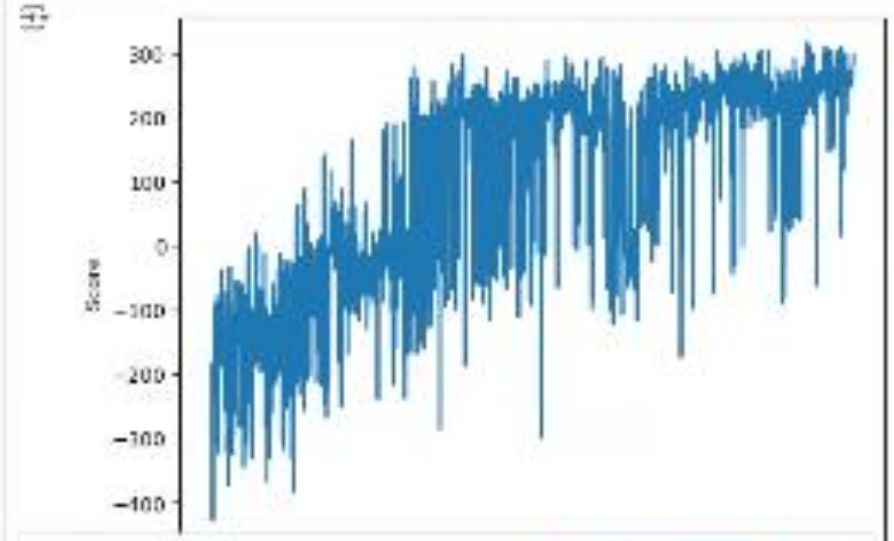



```
[19] BUFFER_SIZE = int(1e4)  # replay buffer size
     BATCH_SIZE = 64         # minibatch size
     GAMMA = 0.99            # discount factor
     TAU = 1e-3              # for soft update of target parameters
     #LR = 5e-4               # learning rate
     LR = 0.0008               # learning rate
     UPDATE_EVERY = 4        # how often to update the network
     device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```



```
Episode 200     Average Score: -124.03
Episode 300     Average Score: -47.15
Episode 400     Average Score:  27.38
Episode 500     Average Score: 81.17
Episode 600     Average Score: 145.66
Episode 700     Average Score: 173.09
Episode 800     Average Score: 191.71
Episode 900     Average Score: 85.46
Episode 1000    Average Score: 201.88
Episode 1100    Average Score: 232.66
Episode 1200    Average Score: 201.70
Episode 1300    Average Score: 248.03
Episode 1386    Average Score: 250.71
Environment solved in 1386 episodes!    Average Score: 250.71
```

```
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```



DOUBLE DQN

After implementing the DQN, we moved on to trying the **Double DQN model**. For background, a popular problem of the previous algorithm of DQN models is the overestimation of the action value, or Q-value.

Instead, the algorithm of Double Q-Learning solves the problem of overestimating the action value, or the Q-value.

This algorithm utilizes two action-value functions as estimators. The main difference between the Vanilla DQN and the Double DQN is the target equation, which is as follows:

$$Q^*(s_t, a_t) \approx r_t + \gamma Q(s_{t+1}.argmax_{a'} Q'(s_t, a_t))$$

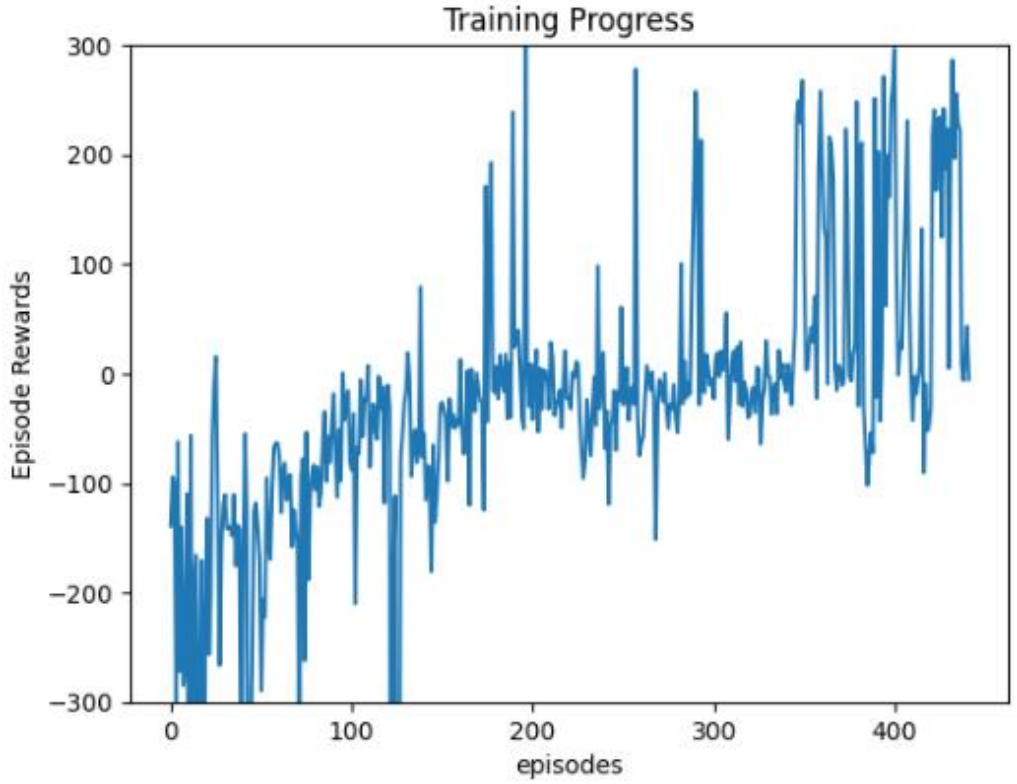
DDQN MODEL

```
# Configure and create the DQN model, which will use Double Q-learning by default
model = DQN("MlpPolicy",
            env,
            policy_kwargs=policy_kwargs,
            learning_rate=1e-3,
            batch_size=32,  # Use a larger batch size for better learning
            buffer_size=50000,  # A larger replay buffer is typically used
            learning_starts=1000,  # Start learning after filling some buffer
            gamma=0.99,  # Discount factor
            tau=1.0,  # Soft update coefficient for the target network
            target_update_interval=1000,  # How often to update the target network
            train_freq=(4, "step"),  # Train every 4 steps
            max_grad_norm=10,  # Gradient clipping
            exploration_initial_eps=1.0,  # Initial exploration rate
            exploration_fraction=0.1,  # Fraction over which exploration rate is reduced
            exploration_final_eps=0.01,  # Final exploration rate
            gradient_steps=1,  # Number of gradient steps
            seed=1,  # Seed for random number generators
            verbose=1)  # Set verbose to 1 to observe training logs
```

```
Using cpu device
Wrapping the env in a DummyVecEnv
```

DOUBLE DQN IMPLEMENTATION:

- In addition, the 200 average score threshold is reached at around 180 episodes which is faster than with the Vanilla DQN.



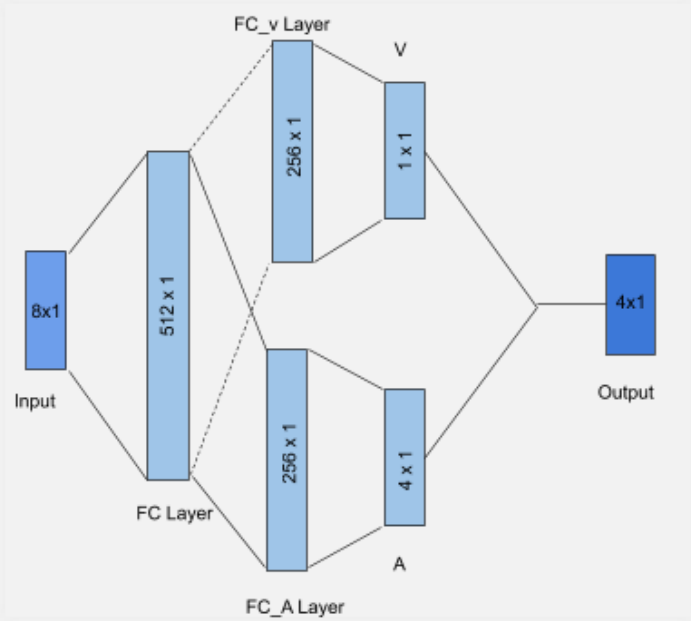
DUELING DQN

The Dueling DQN is an alternative variation of the standard vanilla DQN utilizing a novel dueling architecture.

This structure utilizes two separate streams to explicitly separate the representation of state values and state-dependent action advantages.

The two streams represent the value and advantage functions, and share a common convolutional feature learning module (Wang et al.).

DUELING DQN MODEL



The key part of the architecture, the aggregating module (for the two streams), implements the forward mapping:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - max_{a'\epsilon|A|}A(s,a';\theta,\alpha))$$

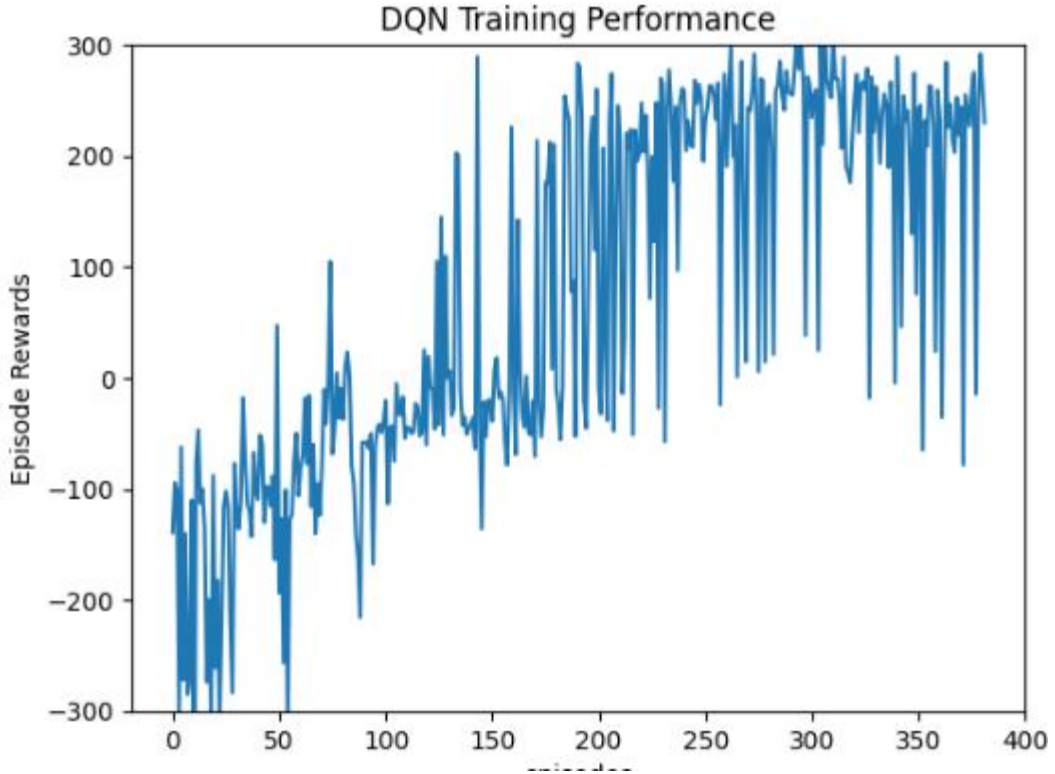
where V is the state value and A is the advantage.

DUELING DQN MODEL

```python
# Custom policy kwargs for dueling architecture
policy_kwargs = dict(
    net_arch=[512, 256],
)

# Configure and create the DQN model
model = DQN("MlpPolicy",
            env,
            policy_kwargs=policy_kwargs,
            learning_rate=1e-3,
            batch_size=32,
            buffer_size=50000,
            learning_starts=1000,
            gamma=0.99,
            tau=1.0,
            target_update_interval=1000,
            train_freq=(4, "step"),
            max_grad_norm=10,
            exploration_initial_eps=1.0,
            exploration_fraction=0.1,
            exploration_final_eps=0.01,
            gradient_steps=1,
            seed=1,
            verbose=1)
```

DUELING DQN IMPLEMENTATION:

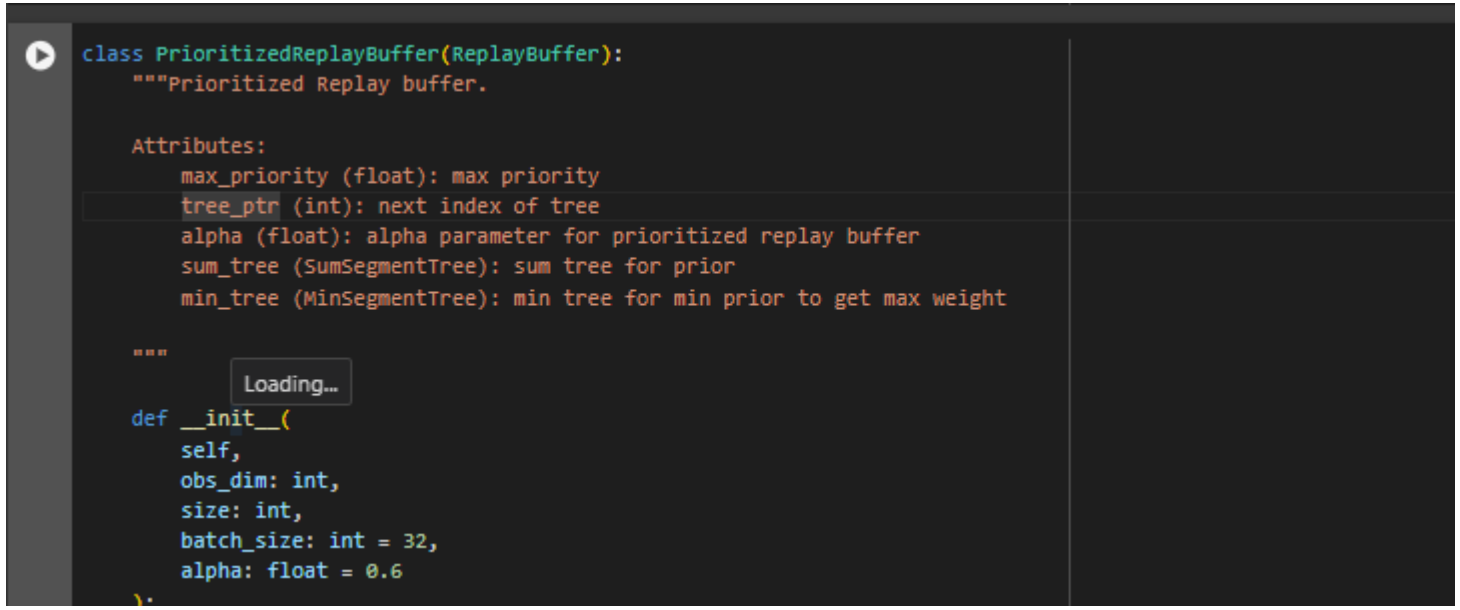
- In addition, the 200 average score threshold is reached at around 150 episodes which is faster than with the Vanilla DQN.



RAINBOW DQN

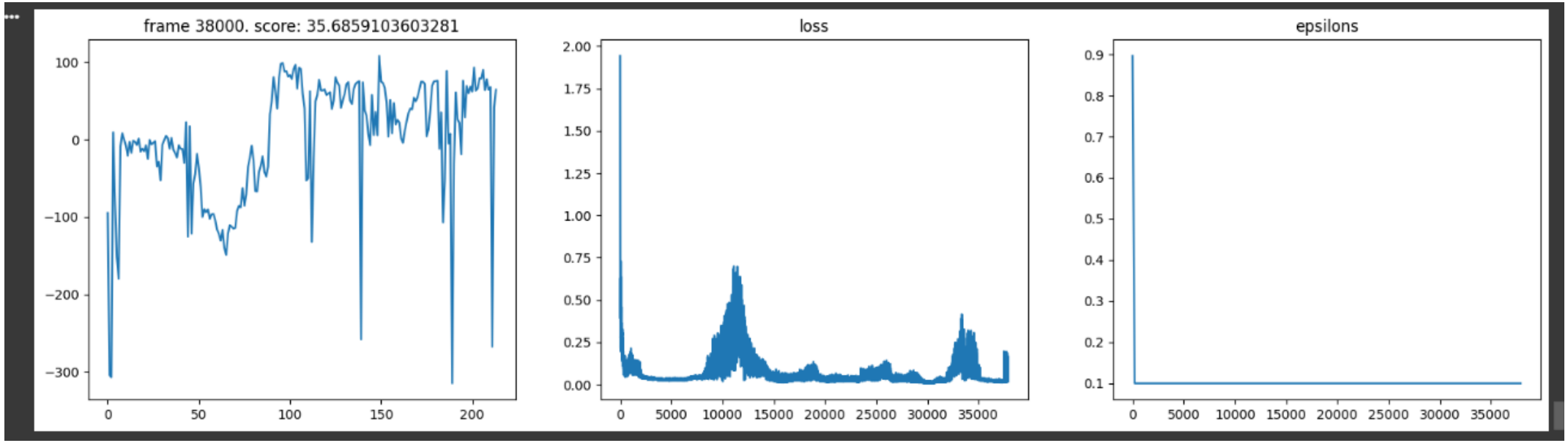
Rainbow DQN is a DQN variant that combines improvements of many different variations of DQN into a single model. Building on top of the original DQN, the original paper for Rainbow selected six specific improvements to aggregate into one model (Hessel et al.):

1. Double Q-Learning
2. Prioritized Experience Replay
3. Dueling Networks
4. Multi-step learning
5. Distributional RL
6. Noisy Nets

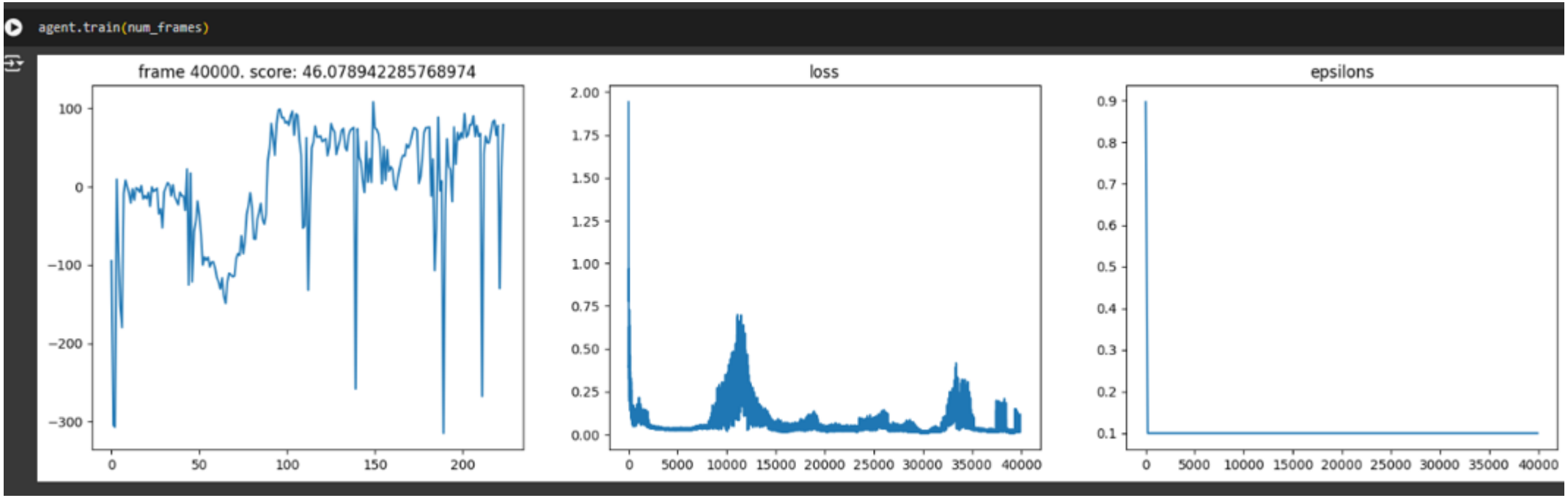

class PrioritizedReplayBuffer(ReplayBuffer):
"""Prioritized Replay buffer.
Attributes:
max_priority (float): max priority
tree_ptr (int): next index of tree
alpha (float): alpha parameter for prioritized replay buffer
sum_tree (SumSegmentTree): sum tree for prior
min_tree (MinSegmentTree): min tree for min prior to get max weight
"""
Loading...
def __init__(
self,
obs_dim: int,
size: int,
batch_size: int = 32,
alpha: float = 0.6
):

Try 1:

```
# parameters
num_frames = 20000
memory_size = 1000
batch_size = 100
target_update = 50
epsilon_decay = 1/200

# train
agent = DQNAgent(env, memory_size, batch_size, target_update, epsilon_decay, seed)
```



Try 2: *with frames 40000*

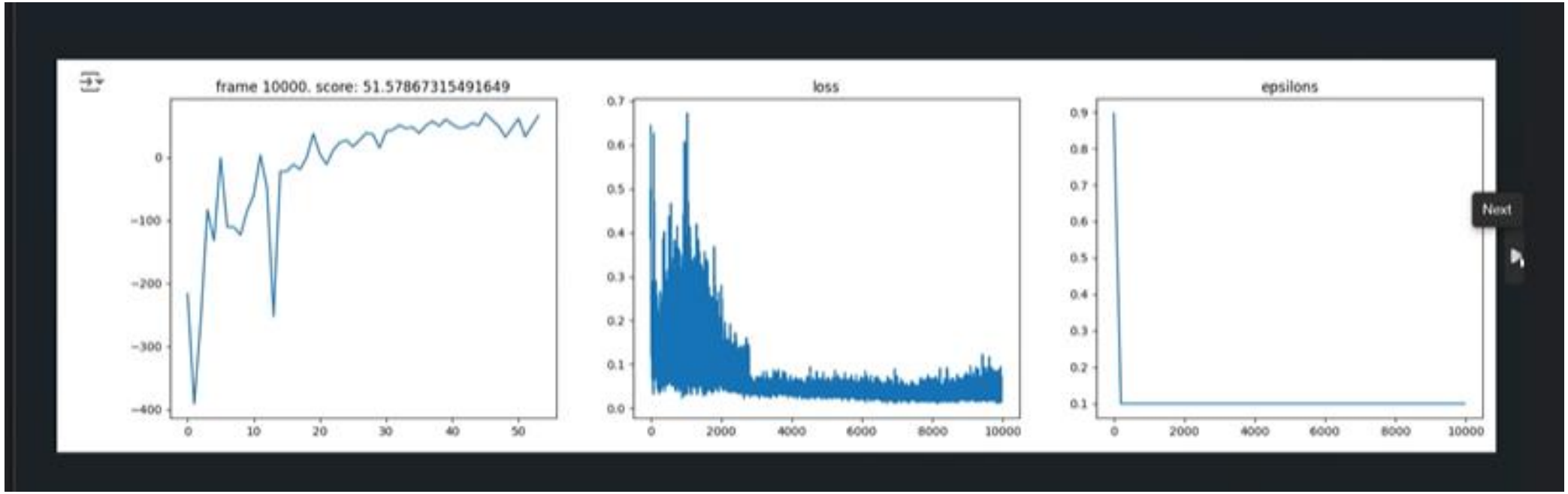


When applied to Lunar Lander, however, the training proved to be fairly unstable. When applied to the Lunar Lander environment, the mean reward would not converge to a value of 200, instead it produces equivalent score of 35 in first try, and 46 in another.

TRY 3: *WITH FRAMES 10000*

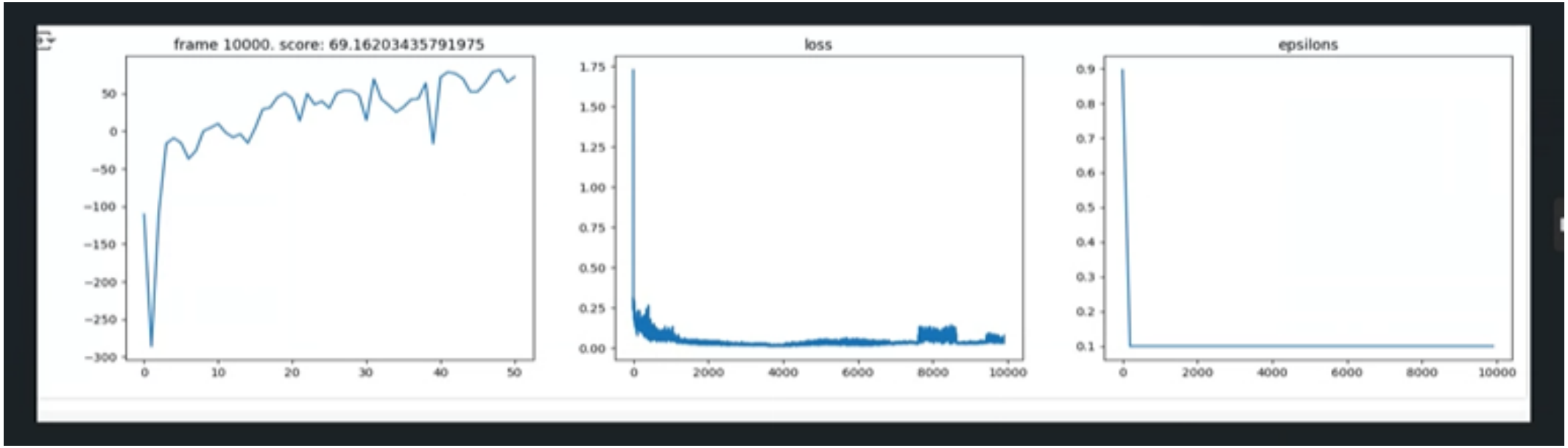
```
[41] # parameters
     num_frames = 10000
     memory_size = 1000
     batch_size = 32
     target_update = 150
     epsilon_decay = 1 / 200

     # train
     agent = DQNAgent(env, memory_size, batch_size, target_update, epsilon_decay,
```

```
cpu
```



TRY 4: *WITH FRAMES 10000*

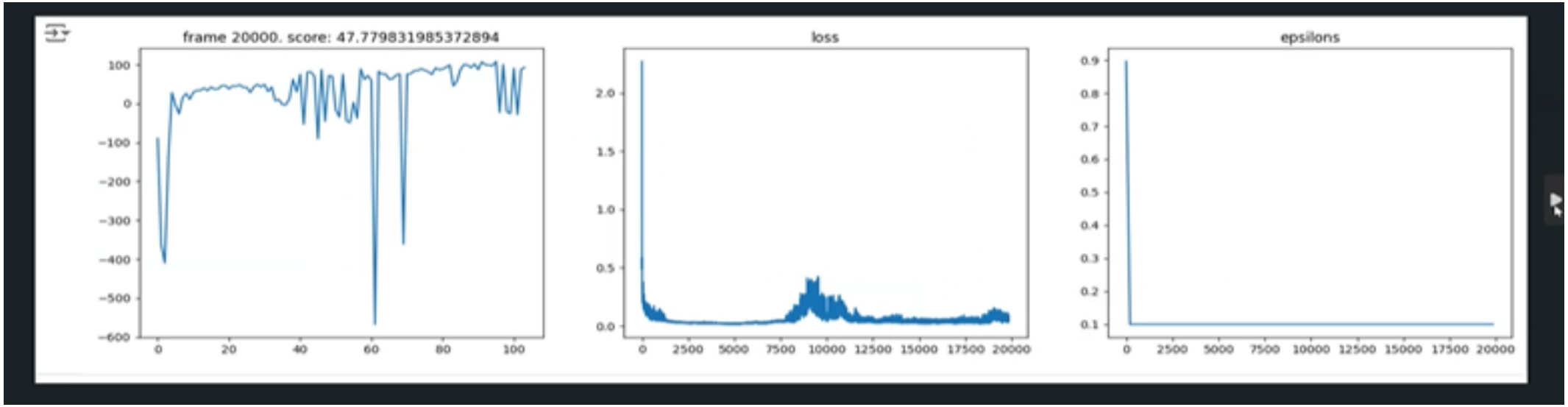
```
[49] # parameters
     num_frames = 10000
     memory_size = 1000
     batch_size = 100
     target_update = 50
     epsilon_decay = 1/200

     # train
     agent = DQNAgent(env, memory_size, batch_size, target_update, epsilon_decay, seed)

     cpu
```



TRY 5: *WITH FRAMES 20000*

```
# parameters
num_frames = 20000
memory_size = 1000
batch_size = 150
target_update = 50
epsilon_decay = 1/200

# train
agent = DQNAgent(env, memory_size, batch_size, target_update, epsilon_decay, seed)
```

```
cpu
```



RESULTS

Summary of Performance

Key Insights:

Vanilla DQN: is effective but slower to converge compared to the improved variants.

Double DQN: reduces overestimation bias, speeding up convergence and leading to better performance.

Dueling DQN: achieves the best performance, with faster convergence due to the separation of state value and action advantage.

Rainbow DQN: Using Rainbow we were able to reach reasonable scores. To fine tune the execution : prioritisation parameters alpha and beta and their impact can be explore further.

CONCLUSION

This project demonstrates the impact of advanced DQN variants in solving the Lunar Lander control problem.

The transition from Vanilla DQN to Rainbow DQN shows significant improvements in training speed, stability, and overall performance.

Rainbow DQN, by integrating multiple enhancements, offers the most robust solution, achieving fast convergence and high success rates.

REFERENCE :

Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). "Human-level control through deep reinforcement learning." *Nature*, 518(7540), 529-533.

Wang, Z., Schaul, T., Hessel, M., et al. (2016). "Dueling Network Architectures for Deep Reinforcement Learning." *arXiv preprint arXiv:1511.06581*.

Hessel, M., Modayil, J., Van Hasselt, H., et al. (2018). "Rainbow: Combining Improvements in Deep Reinforcement Learning." *AAAI Conference on Artificial Intelligence*.