Developing an AI RL Agent for Space Invaders Using Deep Q-Networks

RL Space Invaders: ECE-472 Final Project

Georgiy Aleksanyan under the supervision of Professor Dr. Kristin Dana (ECE)

Department of Electrical and Computer Engineering, Rutgers University

Date: December 18, 2023

Introduction

Originally released in 1978, Space Invaders is a classic arcade game that provides an ideal platform for implementing reinforcement learning (RL) algorithms. This paper outlines the development of a custom environment for Space Invaders, utilizing Deep Q-Networks (DQN) to train an AI agent. The objective is to showcase the potential of DQN in mastering game strategies and outperforming a randomly acting agent.

Objectives

1. Develop and test the basic game.
2. Create and validate a custom OpenAI-Gym environment.
3. Implement a training script for the AI agent.
4. Train and evaluate the model against a random agent.

Methods

Framework and Methodology

The project involved several key stages:

Creating the Game Environment:

We recreated Space Invaders using PyGame and integrated it into an OpenAI-Gym environment. Parameters like game speed, observation space, action space, and rendering were optimized for training.

```
class SpaceInvadersEnv(gym.Env):
    # Initialization of the environment
    def __init__(self, render_mode=None, size=5):
        self.size = size
        self.width, self.height = 300 * size, 300 * size
        # ... Additional setup code ...

    def step(self, action):
        # Implementation of one step in the environment
        # ... Action handling and state updating code ...

    def reset(self):
        # Resetting the environment to the initial state
        # ... Environment reset code ...
```

- **Model Design and Training**: The training script was developed in Python using PyTorch. Key training parameters included BATCH_SIZE, GAMMA, EPS_START, EPS_END, EPS_DECAY, TAU, and LR. The Deep Q-Network (DQN) model was used, incorporating multiple convolutional layers for image processing. Images being processed were PyTorch tensors stored as observations by the environment. The training script utilized Torch tensors to store the trained model weights as well.

  ```
  class DQN(nn.Module):
      def __init__(self, observation, outputs):
          super(DQN, self).__init__()
          self.conv1 = nn.Conv2d(3, 8, kernel_size=5, stride=4)
          self.bn1 = nn.BatchNorm2d(8)
          self.conv2 = nn.Conv2d(8, 16, kernel_size=5, stride=4)
  ```

```
            self.bn2 = nn.BatchNorm2d(16)
            # ... Additional layers ...

        def forward(self, x):
            x = F.relu(self.bn1(self.conv1(x)))
            x = F.relu(self.bn2(self.conv2(x)))
            # ... Forward pass implementation ...
```

- **Optimization and Fine-Tuning**: The reward structure was varied, aiming to optimize performance and to ensure the agent was not overly constrained. Multiple training sessions with varied parameters were conducted to fine-tune the model. Multiple model variants were trained 50 episodes and 600 episodes. The reward structure was adapted for both 50 and 600 episode models to optimize performance. The 50-episode model focused on basic learning, while the 600-episode model emphasized long-term strategy and efficiency.

```
50 episode model:
def _compute_reward(self):
        # Initialize the reward for this step
        step_reward = 0

        # Add reward for destroyed invaders
        step_reward += self.invaders_destroyed * 15  # Assuming a reward of 10 for each i
        step_reward += self.moved_left_count * 0.5
        step_reward += self.moved_right_count * 0.5
        step_reward += self.shot_bullet_count * 0.001
        step_reward += self.current_level * 100
        #step_reward += (self.shot_bullet_count - self.invaders_destroyed) * -1
        # Subtract penalty for bullets that didn't hit anything
        # Assuming you keep track of bullets fired and invaders destroyed correctly
        #missed_shots = max(0, len(self.bullets) - self.invaders_destroyed)
        #step_reward -= missed_shots * 0.05  # Assuming a penalty of 0.05 for each missed

        # Reset the number of destroyed invaders after the reward is calculated
        self.invaders_destroyed = 0
        self.moved_left_count = 0
        self.moved_right_count = 0
        self.shot_bullet_count = 0

        return step_reward

600 episode model:
def _compute_reward(self):
        # Initialize the reward for this step
        step_reward = 0

        # Add reward for destroyed invaders
        step_reward += self.invaders_destroyed * 1  # Assuming a reward of 10 for each in
        step_reward += self.moved_left_count * 0.01
        step_reward += self.moved_right_count * 0.01
        step_reward += self.shot_bullet_count * 0.0001
        step_reward += self.current_level * 15000
        step_reward += (self.shot_bullet_count - self.invaders_destroyed) * -0.0004
        # Subtract penalty for bullets that didn't hit anything
        # Assuming you keep track of bullets fired and invaders destroyed correctly
        #missed_shots = max(0, len(self.bullets) - self.invaders_destroyed)
        #step_reward -= missed_shots * 0.05  # Assuming a penalty of 0.05 for each missed

        # Reset the number of destroyed invaders after the reward is calculated
        self.invaders_destroyed = 0
        self.moved_left_count = 0
```

```
        self.moved_right_count = 0
        self.shot_bullet_count = 0

        return step_reward
```

In the reinforcement learning model for Space Invaders, several critical hyperparameters were meticulously calibrated to optimize training effectiveness. The batch size was set to 128, allowing the model to process a sizable amount of experiences simultaneously, which is crucial for learning complex strategies. The discount factor GAMMA was tuned to 0.998, emphasizing the significance of future rewards. Exploration parameters EPS_START and EPS_END were set at 0.998 and 0.005, respectively, with a decay rate (EPS_DECAY) of 1000, facilitating a balance between exploration and exploitation during training. The learning rate (LR) was maintained at 1e-4, ensuring gradual but steady learning. Replay memory was expanded to hold 11,000 transitions, providing a rich set of experiences for training.

The `optimize_model` function samples a batch of transitions from the replay memory for training. The model calculates the current state-action values and estimates the next state values, which are then adjusted using the discount factor. The model employs a Smooth L1 Loss (Huber loss) as the criterion for calculating the difference between the estimated and actual Q-values, crucial for backpropagation and weight adjustments. Gradient clipping was also implemented to maintain stability in the learning process by preventing gradients from becoming too large.

```
# Hyperparameters
BATCH_SIZE = 128
GAMMA = 0.998
EPS_START = 0.998
EPS_END = 0.005
EPS_DECAY = 1000
TAU = 0.005
LR = 1e-4
memory = ReplayMemory(11000)

def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))

    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.next_state)),
    non_final_next_states = torch.cat([s for s in batch.next_state if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    state_action_values = policy_net(state_batch).gather(1, action_batch)

    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(non_final_next_states).max(1).valu
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()
```

Training the AI Agent:

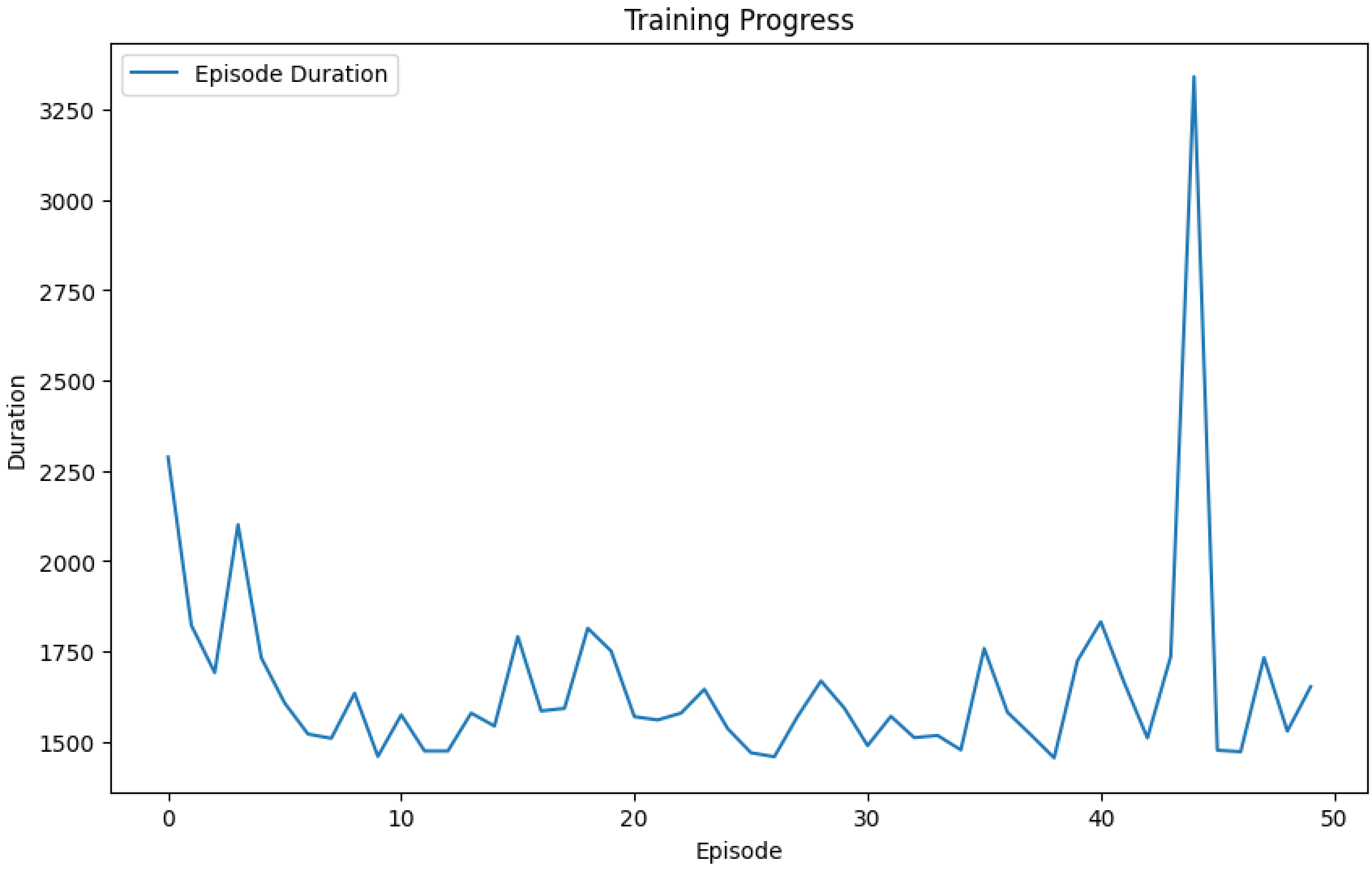
The DQN algorithm was used to train the AI agent. It involves using a neural network to approximate the Q-value function. The network predicts the value of taking an action in a given state. Training was conducted over a series of episodes, with the agent receiving feedback from the environment to update its policy.

```
# Main training loop 50 episode version
num_episodes = 50
for i_episode in range(num_episodes):
    env.reset()
    state = env._get_observation()
    state = torch.from_numpy(state).permute(2, 0, 1).unsqueeze(0).to(device).float()
    for t in count():
        action = select_action(state)
        observation, reward, done = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        if not done:
            next_state = observation
            next_state = torch.from_numpy(next_state).permute(2, 0, 1).unsqueeze(0).to(de
        else:
            next_state = None

        memory.push(state, action, next_state, reward)
        state = next_state

        optimize_model()
        if done:
            episode_durations.append(t + 1)
            break

    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

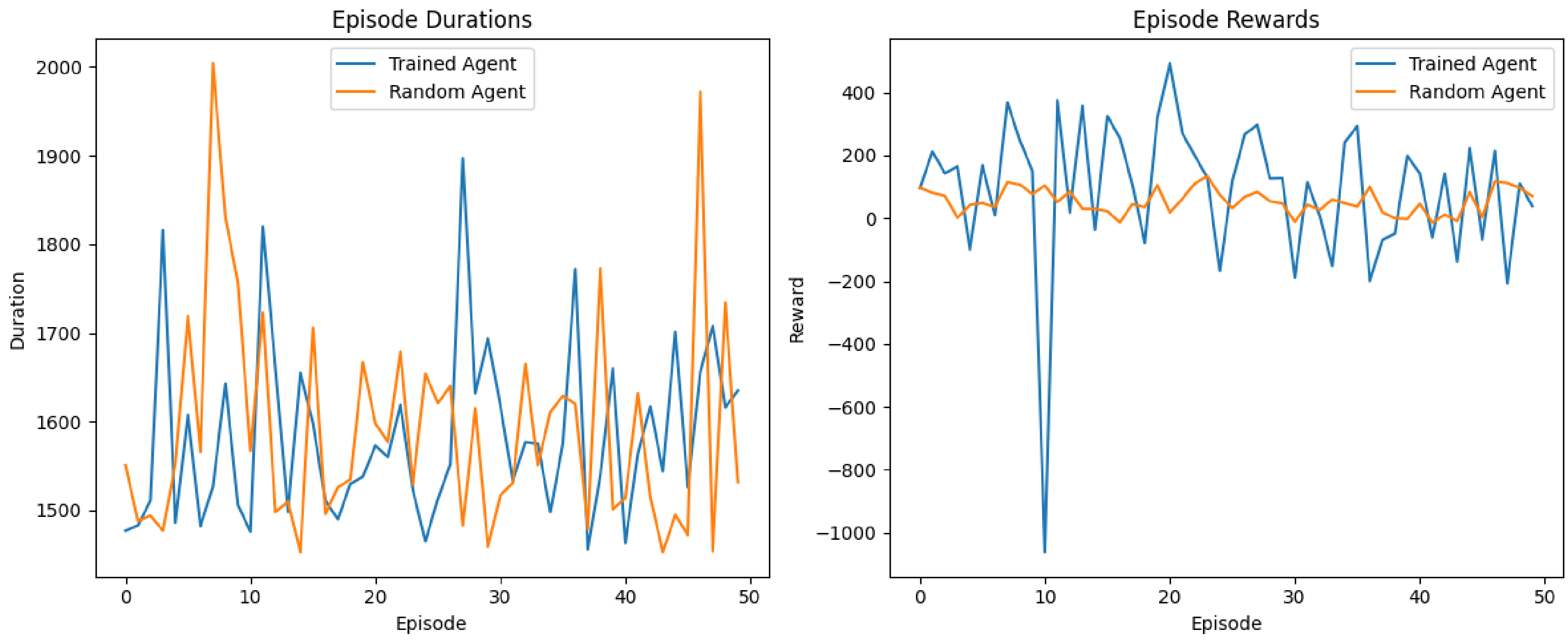
```
# Main training loop 600 episode version + Soft Update
if torch.cuda.is_available():
    num_episodes = 600
else:
    num_episodes = 50
for i_episode in range(num_episodes):
                ....SAME LOGIC AS ABOVE...

        target_net_state_dict = target_net.state_dict()
        policy_net_state_dict = policy_net.state_dict()
        for key in policy_net_state_dict:
            target_net_state_dict[key] = policy_net_state_dict[key]*TAU + target_net_stat
        target_net.load_state_dict(target_net_state_dict)

        if done:
            episode_durations.append(t + 1)
            plot_durations()
            break
```

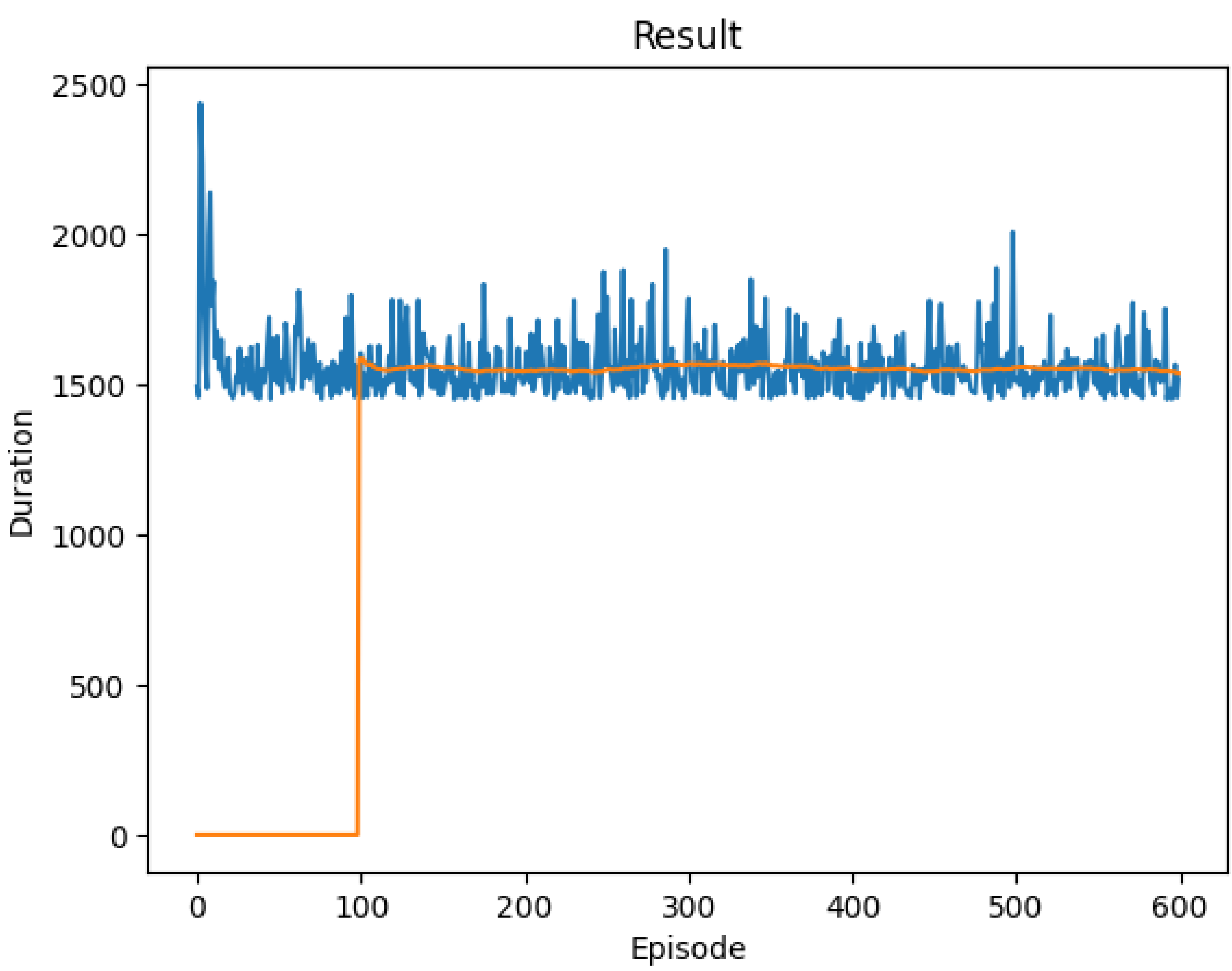
Results

The AI agent's performance improved significantly over training episodes. Initial episodes showed random movements with low scores. However, as training progressed, the agent learned to dodge bullets and effectively target enemies. The following plots illustrate the agent's learning curve, showing the increasing trend in scores and better game duration management over time. Data analysis focused on comparing mean values, episode durations, and rewards between the trained and random agents. Weights collected during training are provided in the files with this paper.



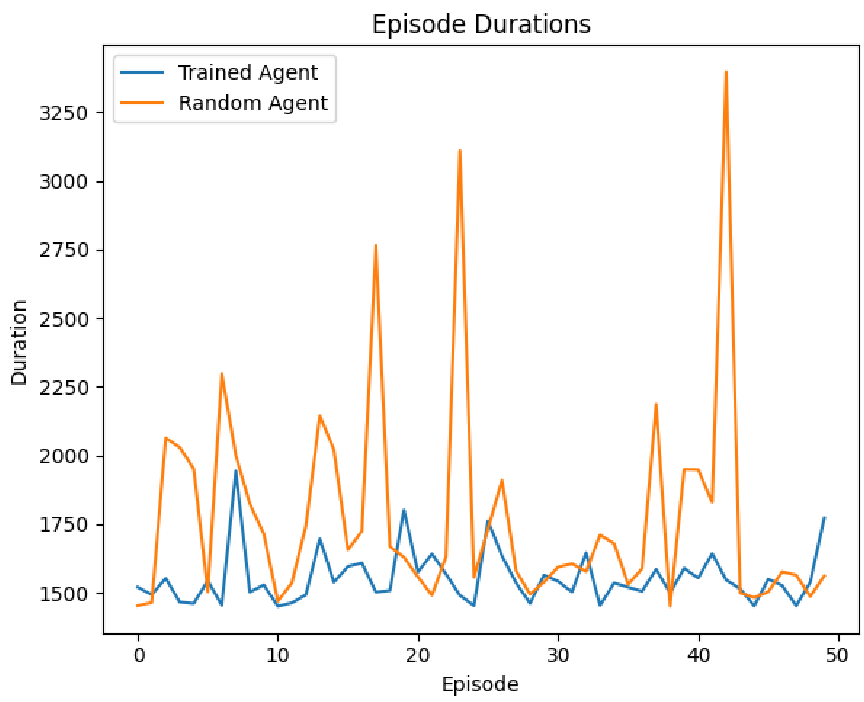

```
Trained Agent First 10 Episodes for a 50 episode model:

Episode: 1, Agent: trained, Total Reward: 95.72800000000055, Duration: 1477
Episode: 2, Agent: trained, Total Reward: 212.07800000000037, Duration: 1483
Episode: 3, Agent: trained, Total Reward: 142.70100000000096, Duration: 1511
Episode: 4, Agent: trained, Total Reward: 165.30100000000076, Duration: 1816
Episode: 5, Agent: trained, Total Reward: -100.67499999999927, Duration: 1486
Episode: 6, Agent: trained, Total Reward: 168.6270000000006, Duration: 1608
Episode: 7, Agent: trained, Total Reward: 9.753000000000034, Duration: 1482
Episode: 8, Agent: trained, Total Reward: 368.55200000000025, Duration: 1527
Episode: 9, Agent: trained, Total Reward: 247.1580000000005, Duration: 1643
Episode: 10, Agent: trained, Total Reward: 151.0, Duration: 1506

Trained Agent First 10 Episodes for a 50 episode model:
Episode: 1, Agent: random, Total Reward: 96.84600000000046, Duration: 1551
Episode: 2, Agent: random, Total Reward: 80.85900000000012, Duration: 1488
Episode: 3, Agent: random, Total Reward: 71.36099999999958, Duration: 1494
Episode: 4, Agent: random, Total Reward: 1.3970000000000553, Duration: 1477
Episode: 5, Agent: random, Total Reward: 42.41399999999988, Duration: 1553
Episode: 6, Agent: random, Total Reward: 48.42499999999927, Duration: 1719
Episode: 7, Agent: random, Total Reward: 35.417999999999935, Duration: 1566
Episode: 8, Agent: random, Total Reward: 115.01200000000043, Duration: 2004
Episode: 9, Agent: random, Total Reward: 106.9690000000015, Duration: 1830
Episode: 10, Agent: random, Total Reward: 76.94300000000035, Duration: 1757
```

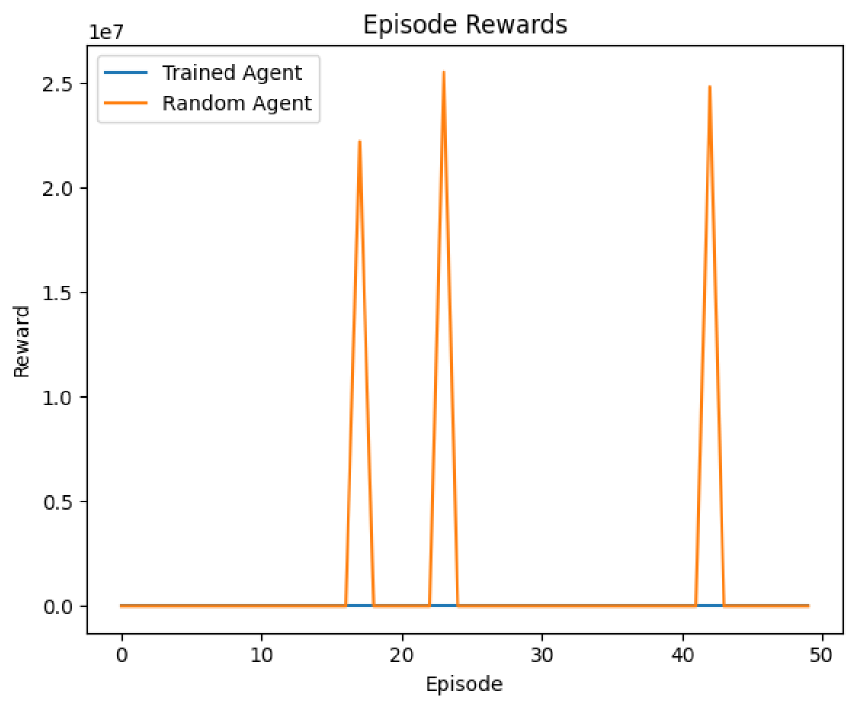





```
Trained Agent First 10 Episodes for a 600 episode model with adjusted reward and parms:

Episode: 1, Agent: trained, Total Reward: -0.4560000000000124, Duration: 1520
Episode: 2, Agent: trained, Total Reward: -0.4479000000000118, Duration: 1493
Episode: 3, Agent: trained, Total Reward: 0.5348000000000251, Duration: 1552
Episode: 4, Agent: trained, Total Reward: -0.4398000000000112, Duration: 1466
Episode: 5, Agent: trained, Total Reward: -0.43830000000001107, Duration: 1461
Episode: 6, Agent: trained, Total Reward: 0.5372000000000353, Duration: 1544
Episode: 7, Agent: trained, Total Reward: 1.5646000000000442, Duration: 1454
Episode: 8, Agent: trained, Total Reward: 1.4179000000000588, Duration: 1943
Episode: 9, Agent: trained, Total Reward: -0.450600000000012, Duration: 1502
Episode: 10, Agent: trained, Total Reward: 0.5420000000000271, Duration: 1528

Random Agent First 10 Episodes for a 600 episode model with adjusted reward and parms:

Episode: 1, Agent: random, Total Reward: 11.077599999999945, Duration: 1453
Episode: 2, Agent: random, Total Reward: 10.137799999999936, Duration: 1464
```

```
Episode: 3, Agent: random, Total Reward: 12.527100000000008, Duration: 2063
Episode: 4, Agent: random, Total Reward: 16.780900000000155, Duration: 2029
Episode: 5, Agent: random, Total Reward: 17.26680000000029, Duration: 1951
Episode: 6, Agent: random, Total Reward: 10.70769999999995, Duration: 1503
Episode: 7, Agent: random, Total Reward: 20.299200000000276, Duration: 2298
Episode: 8, Agent: random, Total Reward: 14.597199999999933, Duration: 2000
Episode: 9, Agent: random, Total Reward: 14.730700000000004, Duration: 1824
Episode: 10, Agent: random, Total Reward: 13.638899999999985, Duration: 1714
```





Conclusion

The application of DQN in training an AI agent to play Space Invaders demonstrated some and most cases limited learning and performance improvement. This implemented approach needs to be further investigated to achieve desired performance and scenarios. Undoubtedly, the versatility and effectiveness of deep reinforcement learning in gaming and decision-making processes has been demonstrated. The future scope includes experimenting with different network architectures and training parameters to further enhance the agent's performance.