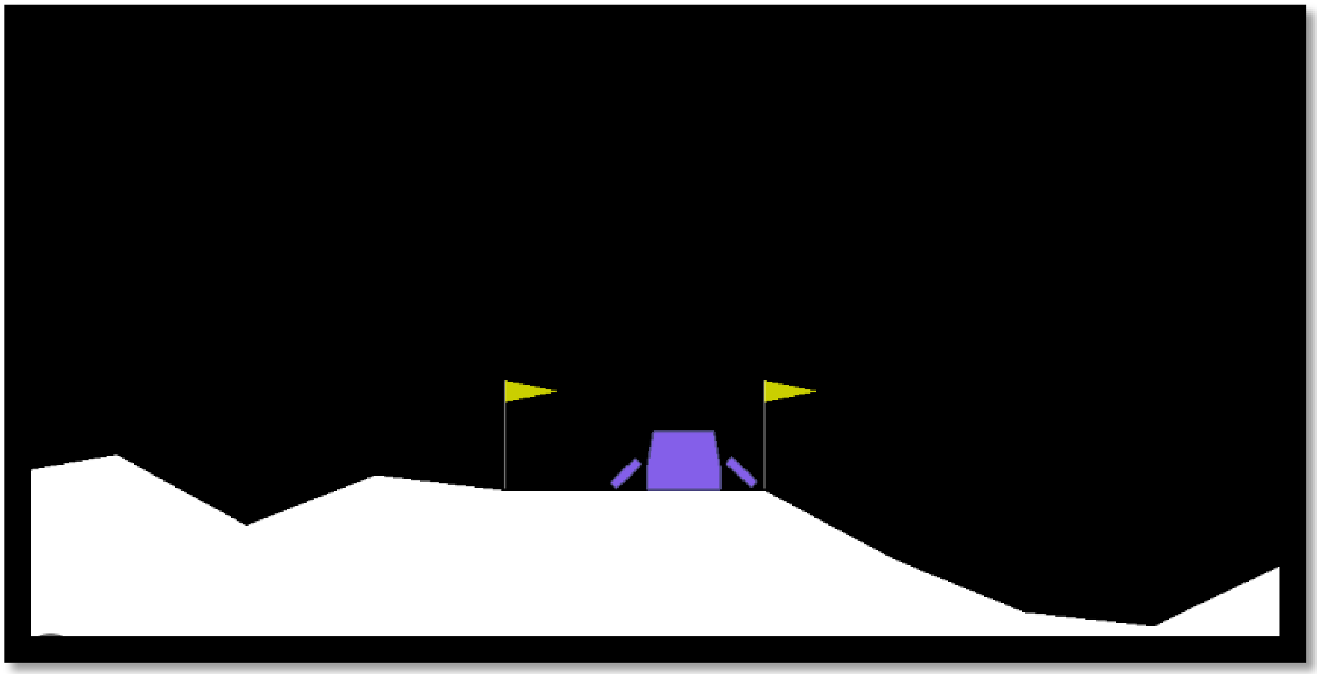
Jessica Carpenter

Intro to AI

Fall 2023

Final Project: OpenAI Gym



Resources:

https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

https://github.com/yuchen071/DQN-for-LunarLander-v2/blob/main/LunarLander.ipynb

https://www.gocoder.one/blog/rl-tutorial-with-openai-gym/

https://www.youtube.com/watch?v=F1Qm8TmDW84

https://github.com/yuchen071/DQN-for-LunarLander-v2/blob/main/LunarLander.ipynb

Another helpful source- comments on Piazza!

Project Goal and Training Method:

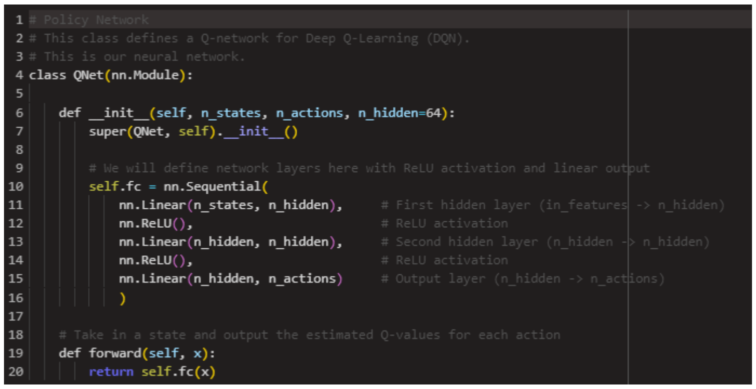
In this project, we will create an environment and train an agent using artificial intelligence techniques. In our coursework we discussed Q-learning, which will be utilized here.

Through my research while planning this project, I found a method that uses Q-learning combined with deep neural networks, called Deep Q-learning. This algorithm allows machines to learn how to take optimal actions in complex environments through trial and error, using a deep neural network to estimate the long-term rewards of different choices. OpenAI Gym offers complex environments and rewards, and Deep Q-Learning (DQN) is a perfect fit for several reasons. First, its neural networks can extract features from these complex states, making it adaptable. Second, it learns from past experiences stored in a replay memory, allowing it to explore and discover better solutions beyond immediate rewards. Third, it prioritizes high-valued experiences for efficient learning, especially useful in environments with sparse rewards. Furthermore, DQN uses two separate networks, promoting stable and robust learning. Finally, it's flexible, adapting to different tasks by modifying its architecture and hyperparameters. While DQN has a high computational cost and many tuning requirements, its strengths make it an effective choice for training models in environments like Lunar Landing in OpenAI gym.

In the Lunar Landing environment, the lander starts above the lunar surface with fuel and two thrusters. Each action controls the thrust of the respective thruster, affecting the lander's movement. Observations consist of various measurements like altitude, velocity, angle, and fuel remaining. The episode ends when the lander lands softly within a designated area, crashes, or runs out of fuel. Reward encourages safe landing: points are awarded for touching down gently, with penalties for crashing or exceeding time limits.

Code Snippets and Results:

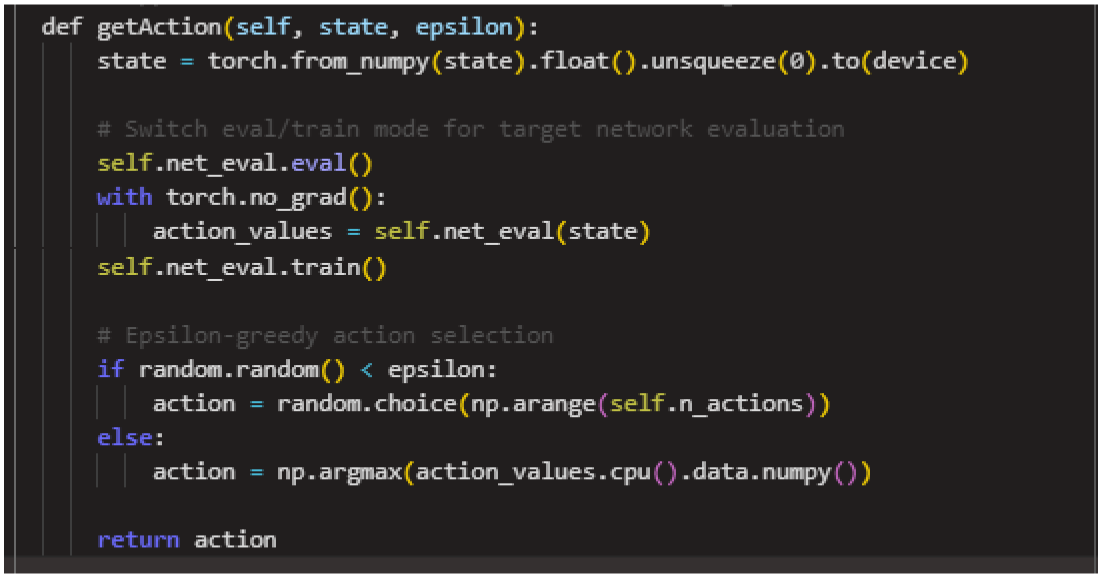
For our policy, we create a neural network with Q-learning. We use three hidden layers and a ReLU activation.



```python
# Policy Network
# This class defines a Q-network for Deep Q-Learning (DQN).
# This is our neural network.
class QNet(nn.Module):

    def __init__(self, n_states, n_actions, n_hidden=64):
        super(QNet, self).__init__()

        # We will define network layers here with ReLU activation and linear output
        self.fc = nn.Sequential(
            nn.Linear(n_states, n_hidden),     # First hidden layer (in_features -> n_hidden)
            nn.ReLU(),                         # ReLU activation
            nn.Linear(n_hidden, n_hidden),     # Second hidden layer (n_hidden -> n_hidden)
            nn.ReLU(),                         # ReLU activation
            nn.Linear(n_hidden, n_actions)     # Output layer (n_hidden -> n_actions)
            )

    # Take in a state and output the estimated Q-values for each action
    def forward(self, x):
        return self.fc(x)
```

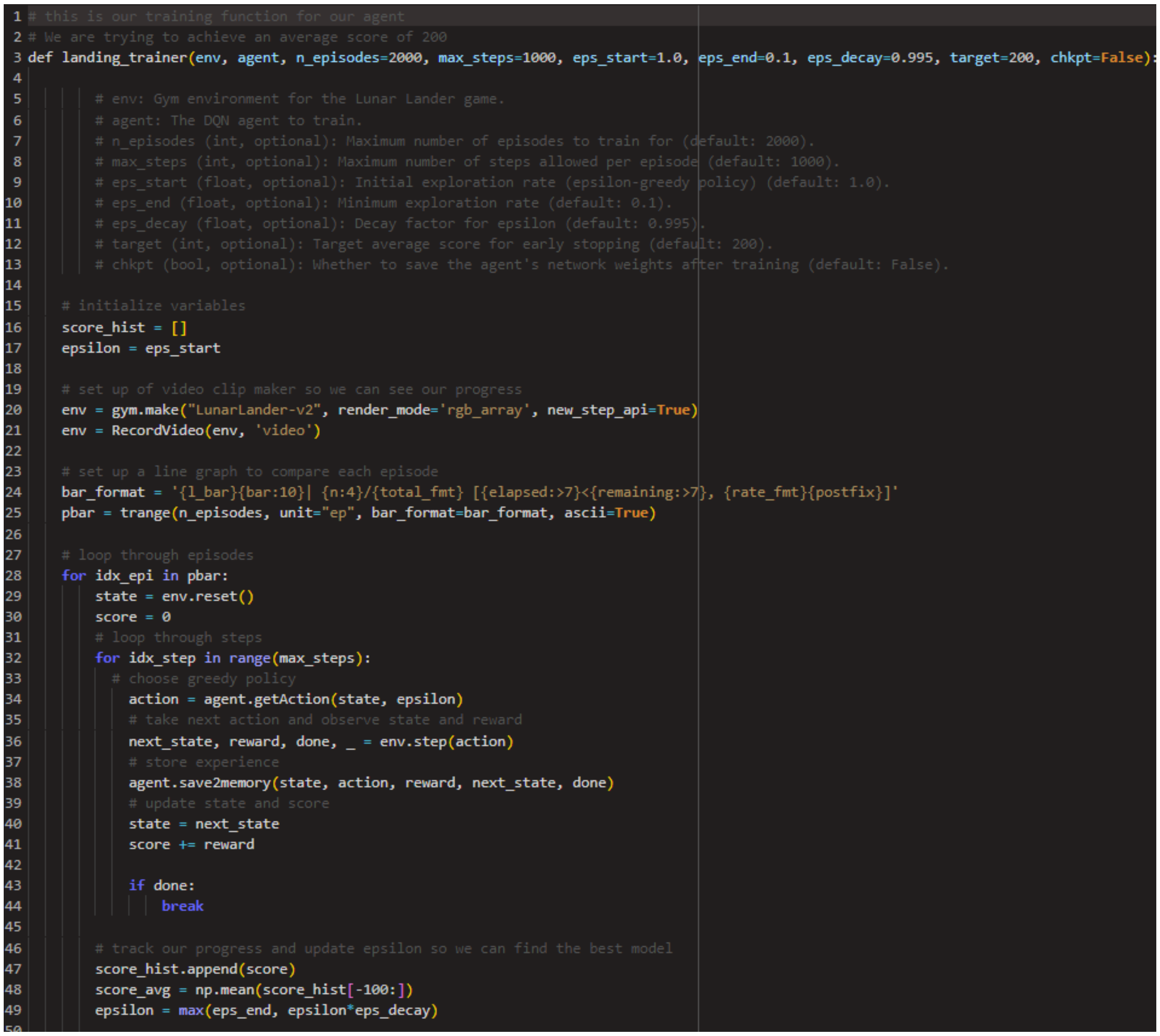
In Deep Q-Learning, the Q-Network (QNet) is the brain, estimating future rewards for each action the agent can take in a given state. The DQN algorithm is the trainer, using past experiences to refine the QNet's predictions. It samples past experiences, calculates optimal future rewards ("targets"), and teaches the QNet to improve its estimates through adjustments based on its mistakes. This continuous partnership allows the agent to gradually master the environment, choosing actions that lead to the highest long-term rewards.

Our DQN class and multiple functions. The getAction function uses a greedy action selection based on the current state and exploration rate. Our agent will take advantage of prior knowledge and search for better options. This approach selects the action with the highest reward most of the time.



```python
def getAction(self, state, epsilon):
    state = torch.from_numpy(state).float().unsqueeze(0).to(device)

    # Switch eval/train mode for target network evaluation
    self.net_eval.eval()
    with torch.no_grad():
        action_values = self.net_eval(state)
    self.net_eval.train()

    # Epsilon-greedy action selection
    if random.random() < epsilon:
        action = random.choice(np.arange(self.n_actions))
    else:
        action = np.argmax(action_values.cpu().data.numpy())

    return action
```

Next, we use samples to update experiences from memory and store experiences for later sampling. To do this, we apply functions learn and buffer. After we have set up how our agent will track its training and rewards, we are ready to begin training. We set our agent up to try to reach a score of 200. We add a recording feature to the training to capture clips of the process. We loop through the episodes and use the greedy epsilon policy we set up earlier to make choices. We track the state and reward for each episode and update epsilon so we can find the best weights to achieve our goal.



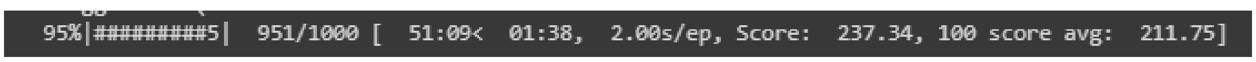
```python
# this is our training function for our agent
# We are trying to achieve an average score of 200
def landing_trainer(env, agent, n_episodes=2000, max_steps=1000, eps_start=1.0, eps_end=0.1, eps_decay=0.995, target=200, chkpt=False):

        # env: Gym environment for the Lunar Lander game.
        # agent: The DQN agent to train.
        # n_episodes (int, optional): Maximum number of episodes to train for (default: 2000).
        # max_steps (int, optional): Maximum number of steps allowed per episode (default: 1000).
        # eps_start (float, optional): Initial exploration rate (epsilon-greedy policy) (default: 1.0).
        # eps_end (float, optional): Minimum exploration rate (default: 0.1).
        # eps_decay (float, optional): Decay factor for epsilon (default: 0.995).
        # target (int, optional): Target average score for early stopping (default: 200).
        # chkpt (bool, optional): Whether to save the agent's network weights after training (default: False).

    # initialize variables
    score_hist = []
    epsilon = eps_start

    # set up of video clip maker so we can see our progress
    env = gym.make("LunarLander-v2", render_mode='rgb_array', new_step_api=True)
    env = RecordVideo(env, 'video')

    # set up a line graph to compare each episode
    bar_format = '{l_bar}{bar:10}| {n:4}/{total_fmt} [{elapsed:>7}<{remaining:>7}, {rate_fmt}{postfix}]'
    pbar = trange(n_episodes, unit="ep", bar_format=bar_format, ascii=True)

    # loop through episodes
    for idx_epi in pbar:
        state = env.reset()
        score = 0
        # loop through steps
        for idx_step in range(max_steps):
            # choose greedy policy
            action = agent.getAction(state, epsilon)
            # take next action and observe state and reward
            next_state, reward, done, _ = env.step(action)
            # store experience
            agent.save2memory(state, action, reward, next_state, done)
            # update state and score
            state = next_state
            score += reward

            if done:
                break

        # track our progress and update epsilon so we can find the best model
        score_hist.append(score)
        score_avg = np.mean(score_hist[-100:])
        epsilon = max(eps_end, epsilon*eps_decay)
```

Troubleshooting and Results:

I tried a number of training episodes, and to reach my goal I needed to train for at least 1000 episodes. This was quite time consuming! I found that setting the target score to 250 created more consistent results and higher overall scores. When I experimented with lower values, the final results had more misses. However, when I tried to set a higher target score, the agent was not able to reach the goal and took too long to train because of this. To handle how the model prioritizes the reward versus the potential future rewards, I tried tweaking the discount factor, gamma. By making gamma closer to 1, I forced the agent to be more "farsighted" and consider potential long term rewards instead of immediate ones. However, when I set the gamma high than 1, the agent's score fell steadily after about 500 episodes. Using .9 worked the best for the 1000 episode training I used in my final version.

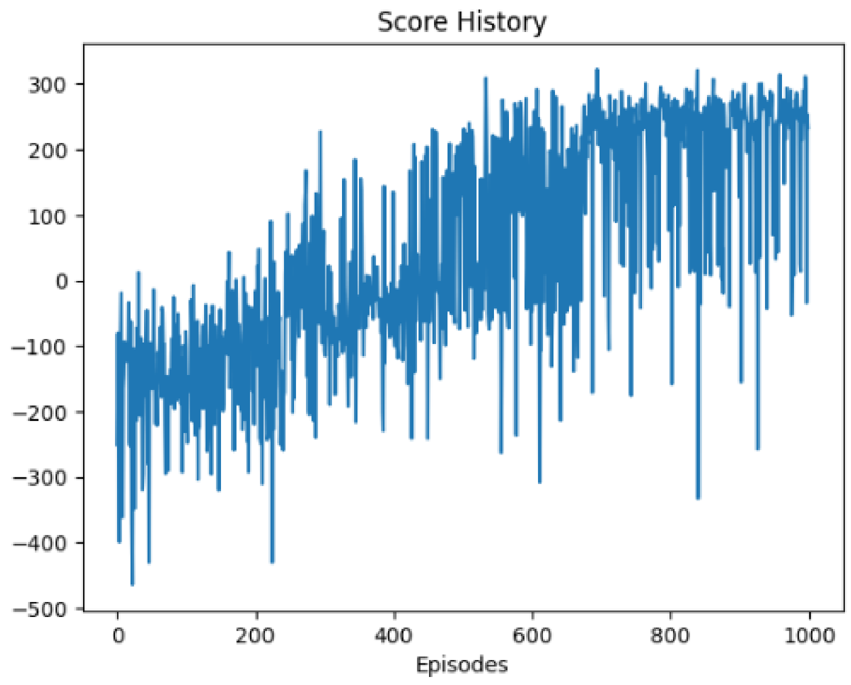
Examples:

By episode 951, our agent had an average score of 211.

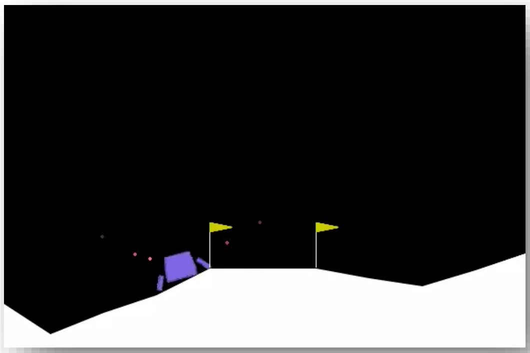


```
 95%|#########5|  951/1000 [  51:09<  01:38,  2.00s/ep, Score:  237.34, 100 score avg:  211.75]
```

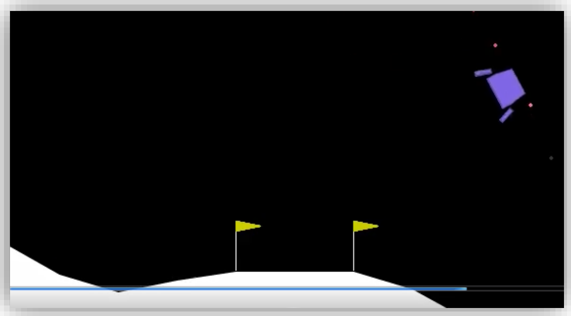
This graph shows our progress.



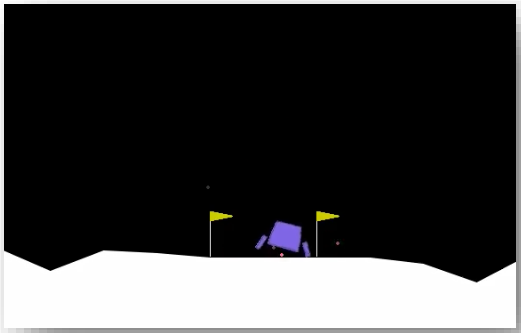
Episodes: 500 Target: 200 Gamma: 1



Episodes: 500 Target: 300 Gamma: 0.9



Episodes: 1000 Target: 250 Gamma: 0.9



Episodes: 1000 Target: 200 Gamma: 0.9