Training an Agent to Navigate a Game Environment

Bharat Raman

Udacity: Deep Reinforcement Learning Nanodegree

Abstract. In 2015, a group of researchers published a paper on their new model for reinforcement learning: Deep-Q Learning [2]. This model was used to train an agent to play various games on the Atari system. Their results show that the agent's scores were comparable to those of professional Atari players. Using this method of incorporating deep reinforcement learning, I trained an agent to navigate and collect bananas in a Unity ML-Agents environment.

1 Introduction

1.1 Lunar Landing Exercise

For my reference, I'm using the Deep-Q Learning exercise from this course; the exercise trains a DQN agent to operate in OpenAI Gym's LunarLander-v2 environment [3]. In the exercise's environment, each time step has a state space of size 8, and an action space of size 4. The Deep-Q model used a three-layered neural network to update Q-values. All three were linear fully-connected layers, followed by ReLU activation functions. To keep Q-values from oscillating or diverging, the exercise utilized a replay buffer to store the tuples $< S_t, A_t, R_{t+1}, S_{t+1} >$ for each time-step t. The resultant training shows a huge improvement in performance, from an average reward (among the 100 earliest episodes) of -208 to a reward a target average reward (among the 100 latest episodes) of 200 (shown in figure 1). I will be referring to this code to train my agent in this project.

Table 1. Layers for Exercise's Neural Network

Layer	Input Size	Output Size	Activation
Linear	8	64	ReLU
Linear	64	128	ReLU
Linear(output)	128	4	ReLU

1.2 Project Outline

My project will proceed as follows:

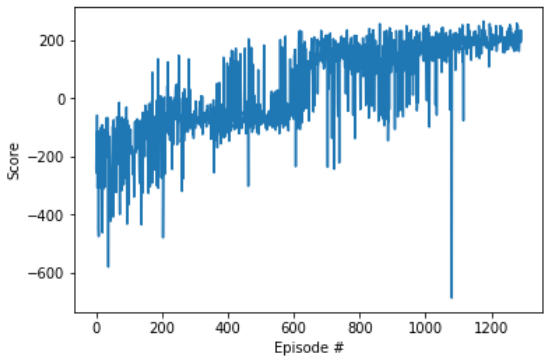

Fig. 1. Reward per episode plot for Lunar Lander exercise

1. Set up the Unity Environment
2. Create the Architecture (deep neural network for determining Q-values, a replay buffer, and an agent)
3. Train an agent
4. Evaluate Results

2 The Environment

The Unity ML-Agents environment, figure 2, is a walled-in plane scattered with yellow and blue bananas. If the agent reaches the location of a yellow banana, it attains a reward of +1. If it reaches a blue banana, it attains a reward of -1. Each state is an array of 37 elements; the elements represent the agent's velocity, and its ray-based perception of objects around its forward direction. For each state, there are 4 actions; move forward, move backward, turn left, turn right.

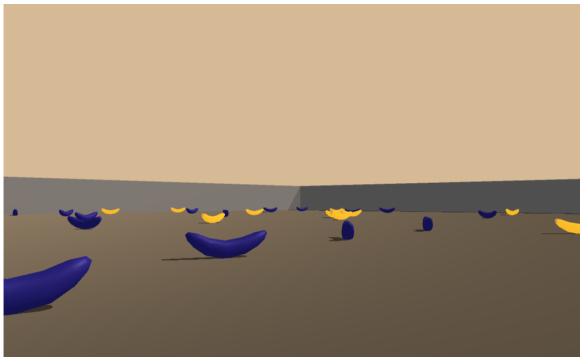

Fig. 2. The environment: A flat plane scattered with blue and yellow bananas

3 The Architecture

3.1 The Model

As stated earlier, I will be referring to the Lunar landing exercise's model and implementing a similar one in this project. My model uses an extra fully connected layer with ReLU activation, as shown in table 2.

Table 2. Layers for My Project's Neural Network

Layer	Input Size	Output Size	Activation
Linear	37	64	ReLU
Linear	64	128	ReLU
Linear	128	256	ReLU
Linear(output)	256	4	ReLU

The weights and biases for each layer are set at random, and will refine itself via gradient descent during training.

3.2 The Agent

The agent utilizes two models: a training model, and a target model. The training model is what's used to select actions for the agent. The target model is utilized as a reference for evaluating loss and updating weights of the training model. I'm using a second model in order to keep weights fixed during the learning process. This prevents the agent from establishing wrong correlations with time-dependent Q-values.

Selecting an Action: Like the model, the agent used for this project is taken from the Lunar Landing exercise code. To select an action, the agent uses the model to find action values for the current state S_t. With four values, the agent then picks one using the epsilon-greedy algorithm. Using this action A_t, the agent will proceed accordingly in the environment, and will attain a reward R_{t+1} at state S_{t+1}. Hence, we get a complete tuple, $< S_t, A_t, R_{t+1}, S_{t+1} >$, to store into the replay buffer.

Learning: After 64 time steps, the agent begins learning by taking a sample of tuples from the buffer, and performing gradient descent on the parameters. To update the parameters, the agent evaluates the mean squared error between the expected Q-value, and the target Q-value. The expected Q-value, $Q(S_i, A_i, w)$, is simply the value for A_i, attained by the expected-Q model with input S_i. i represents an arbitrary tuple in the batch sample.

Target Q-values are attained in the following function:

$$R_{t+1} + \gamma * max_a Q(S_{t+1}, a, w^-)$$

- γ is the discount factor, which is set to 0.99 for this project.
- $max_a Q(S_{t+1}, a, w^-)$ is the maximum Q-value possible for an action a in the next time-step
- w^- are the weights unique to the target model, and are updated to the expected-Q's weights after expected-Q has learned from the entire batch sample of tuples.

Once loss is evaluated for the current tuple, the weights are updated for the training model, and the process repeats for each tuple in the batch sample.

4 Training

For training the agent, I set the maximum number of episodes to 1000. For each episode, the maximum number of time steps is 1000. Epsilon, used for the epsilon-greedy function, is instantiated at 1.0. After each episode, epsilon decays via a multiplicative factor = 0.995 until it reaches 0.1. The criteria for success is if the agent attains an average score greater than or equal to 13.0 among the 100 latest episodes. The training ends if the average reaches 14.0.

4.1 Results

Training session 1: After 713 episodes, the agent's average was at 14.01; therefore, training ended at 713 episodes. The progression of the scores can be seen in figure 3.

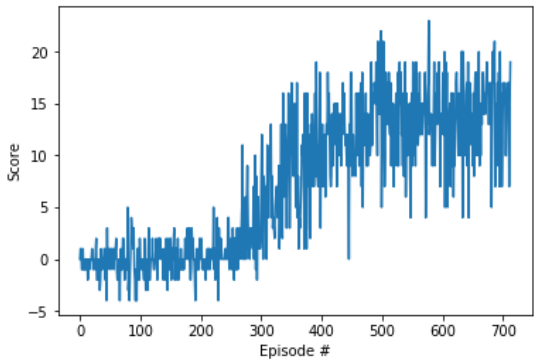

Fig. 3. Session 1: Score per Episode

Training session 2: After 579 episodes, the agent's average was at 14.03, The progression of the scores can be seen in figure 4.

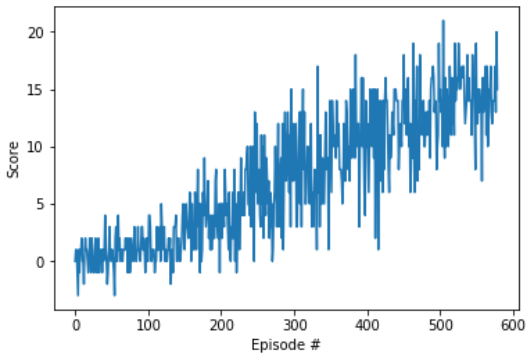

Fig. 4. Session 2: Score per Episode

Training session 3: After 935 episodes, the agent's average was at 14.02, The progression of the scores can be seen in figure 4.

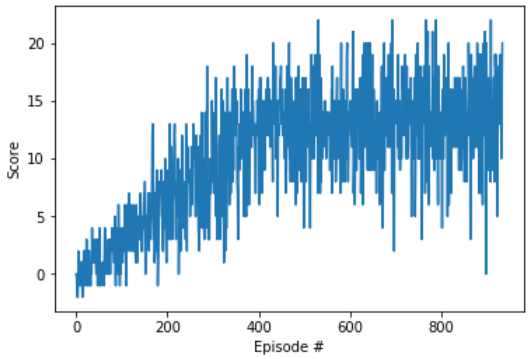

Fig. 5. Session 3: Score per Episode

5 Conclusion

That the agent has managed to attain a reward average of 14 before 1000, it is safe to say that the neural network implemented is robust for this project. It's worth noting that the third trial took much longer to reach an average score of 14 than the previous two. That's to be expected since each session had its own progression of states and actions. A possible improvement to the model is an implementation of a prioritized experience-replay, where experiences are sampled based on their TD-error $(R_{t+1} + \gamma * max_a Q(S_{t+1}, a, w) - Q(S_t, A_t, w))$, instead of at random [4]. Other options include the Double DQN, which mitigates

overestimation of the current model [1], or the Dueling DQN, which identifies which states are valuable for training the agent [5]. The above three suggestions could also yield a more consistent training progression, than what's seen here.

References

1. Hasselt, Guez, S.: Deep reinforcement learning with double q-learning (2015), https://arxiv.org/abs/1509.06461
2. Mnih, Kavukcuoglu, e.a.: Human-level control through deep reinforcement learning (2015), https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf
3. OpenAI: Lunarlander-v2, https://gym.openai.com/envs/LunarLander-v2/
4. Schaul, Quan, e.a.: Prioritized experience replay (2016), https://arxiv.org/abs/1511.05952
5. Wang, Schaul, e.a.: Dueling network architectures for deep reinforcement learning (2016), https://arxiv.org/abs/1511.06581