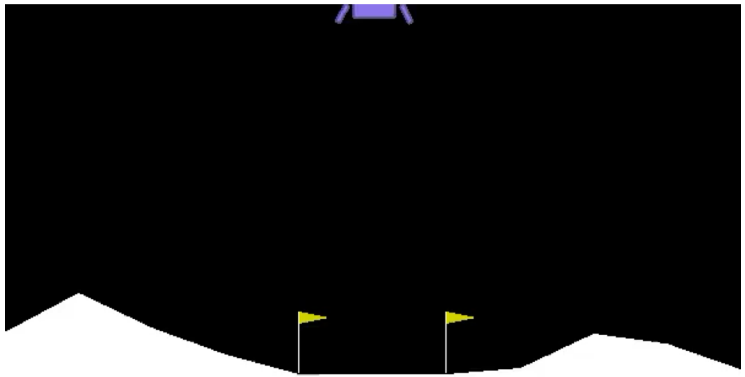
Deep Q-Learning for Lunar Lander

Carlton Brady

University of Delaware

Project Goal

- Apply Deep Q-Learning to train an agent that solves the LunarLander-v2 environment in OpenAI Gym
- Investigate different architectures for Deep Q-Networks
- Train an agent to solve the task in as few training episodes as possible

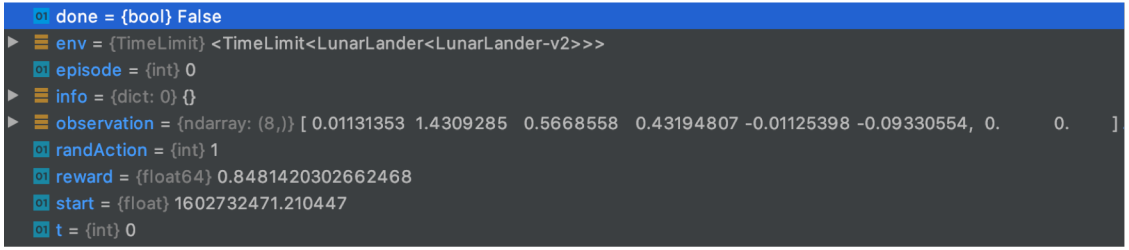


Environment

- OpenAI Gym provides a variety of standardized environments that can be used for reinforcement learning research
- The LunarLander-v2 environment is considered solved if the agent receives an average reward of +200 in the last 100 episodes

Data

- Action Space: 4 discrete values
  - 0 = do nothing
  - 1 = fire left engine
  - 2 = fire bottom engine
  - 3 = fire right engine



- State Space: 8 element ndarray of floats representing the position, velocity, and acceleration of the lander as well as the position of the landing pad
- Reward: a single float each frame, can be positive or negative. Larger magnitude rewards for success or failure at the end of each episode

Approach

- Implement Deep Q-Learning and Double Q-Learning with experience replay and epsilon decay
- Use hyper parameter values and a small network architecture that others have used as a baseline
- From existing solutions I knew that getting good performance in ~600 training episodes was reasonable
- Compare the performance of different network architectures

Deep Q-Learning

- Learn a function that maps environment states to the value of taking each possible action in that state
- With an accurate Q(s, a) function, the agent can then choose the action in each state that maximizes expected future reward

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Source: Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, & Martin Riedmiller. (2013). Playing Atari with Deep Reinforcement Learning.

Experience Replay

- The agent accumulates experience from environment transitions in the following form:
  - ($state_t$, $action_t$, $reward_t$, $state_{t+1}$)
- Consecutive experiences are highly correlated, so they don't contain as much information as a random sample of past experience
- By keeping an experience buffer and randomly sampling experiences from it for training updates, the correlations of consecutive experiences are broken up and the same experience can be use in more than one training update

Epsilon Decay

- Balancing exploration and exploitation is a core issue in RL
- To encourage exploration, the agent has a chance of taking a random action with probability ε
- Epsilon decay is a method for reducing that probability over time as the agent becomes more well trained
- At each timestep, do: $\varepsilon = \varepsilon$ * decay_rate
- Decay rate is a value < 1

Double Q-Learning

- Uses a target network to prevent Q-value overestimation and stabilize training
- τ is an added hyper parameter to control how much the regular networks parameters are weighted when updating the target network

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\, Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Source: Hado van Hasselt, Arthur Guez, & David Silver. (2015). Deep Reinforcement Learning with Double Q-learning.

Neural Network Architectures

- Number of inputs must equal the number of values in the environment state (8 in this case)
- Number of outputs must equal the number of actions in the discrete action space (4 in this case)
- Relu was used as the activation function, 0.0001 was used as learning rate
- 8-32-64-4 (3 fully connected layers) was used as a baseline architecture since it was know that a network this small could solve Lunar-Lander-v2
- 8-32-64-128-4 and 8-64-128-4 were also trained and tested
- Implemented in PyTorch, trained only on a MacBook Pro with 3.1 GHz Dual-Core Intel Core i5 CPU

Demo

Results

	Architecture: 8-32-64-4						
Algorithm	**Deep Q-learning**	**Deep Q-learning**	**Deep Q-learning**		**Double Q-Learning**	**Double Q-Learning**	**Double Q-Learning**
Init epsilon	1.0	1.0	1.0		1.0	1.0	1.0
Min epsilon	0.1	0.1	0.1		0.1	0.1	0.1
Epsilon decay rate	0.99	0.999	0.995		0.99	0.995	0.999
Learning rate	0.0001	0.0001	0.0001		0.0001	0.0001	0.0001
Gamma	0.99	0.99	0.99		0.99	0.99	0.99
Tau	1.0	1.0	1.0		0.001	0.001	0.001
Training episodes	433	841	1035		1148	708	689
100 episode test score avg.	226.68	214.97	197.27		176.82	246.48	202.2

Results

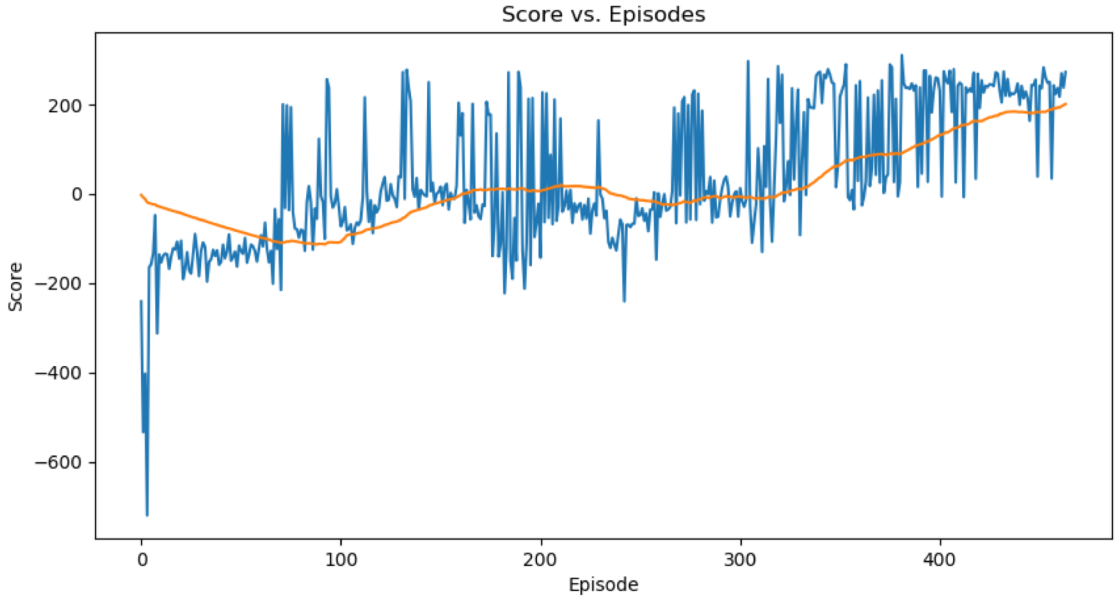
Architecture: 8-64-128-4 (wider)							
Algorithm	**Deep Q-learning**	**Deep Q-learning**	**Deep Q-learning**		**Double Q-Learning**	**Double Q-Learning**	**Double Q-Learning**
Init epsilon	1.0	1.0	1.0		1.0	1.0	1.0
Min epsilon	0.1	0.1	0.1		0.1	0.1	0.1
Epsilon decay rate	0.99	0.999	0.995		0.99	0.995	0.999
Learning rate	0.0001	0.0001	0.0001		0.0001	0.0001	0.0001
Gamma	0.99	0.99	0.99		0.99	0.99	0.99
Tau	1.0	1.0	1.0		0.001	0.001	0.001
Training episodes	464	346	1012		608	643	616
100 episode test score avg.	256.57	195.25	217.93		197.26	209.52	212.17

Results

Architecture: 8-32-64-128-4 (deeper)							
Algorithm	**Deep Q-learning**	**Deep Q-learning**	**Deep Q-learning**		**Double Q-Learning**	**Double Q-Learning**	**Double Q-Learning**
Init epsilon	1.0	1.0	1.0		1.0	1.0	1.0
Min epsilon	0.1	0.1	0.1		0.1	0.1	0.1
Epsilon decay rate	0.99	0.999	0.995		0.99	0.995	0.999
Learning rate	0.0001	0.0001	0.0001		0.0001	0.0001	0.0001
Gamma	0.99	0.99	0.99		0.99	0.99	0.99
Tau	1.0	1.0	1.0		0.001	0.001	0.001
Training episodes	1215	238	2501		576	1073	867
100 episode test score avg.	186.74	169.82	235.09		167.77	213.33	245.38

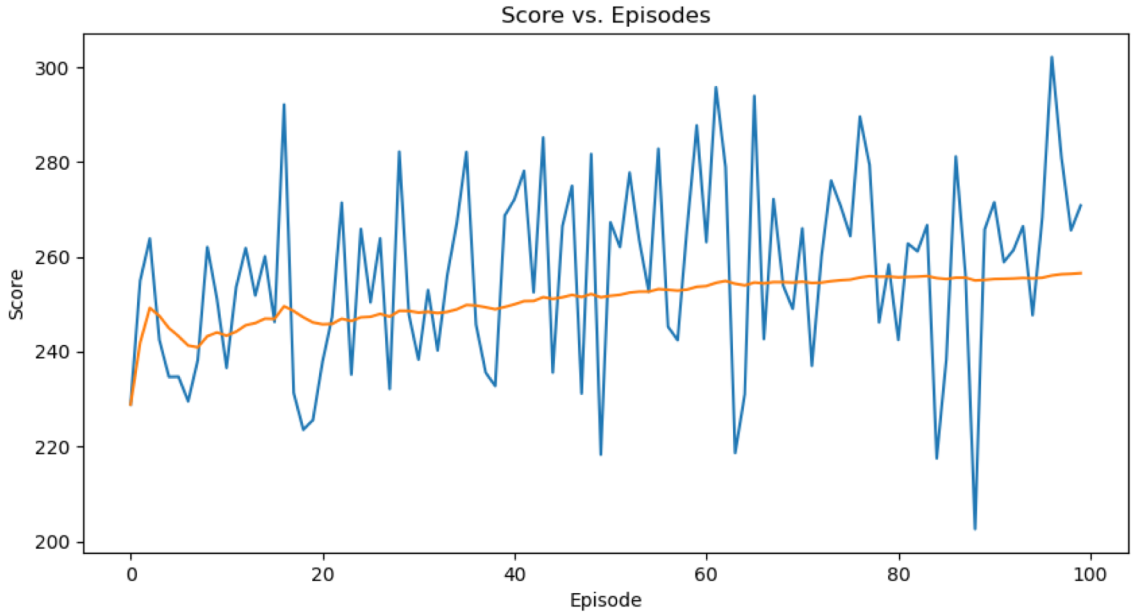
Results

- 8-64-128-8 example training plot ->



Results

- 8-64-128-8 example testing plot ->



Conclusion

- Lunar-Lander-v2 can be solved using a small neural net, its very feasible without a GPU
- How quickly the agent reaches a ”solved” level of performance depends partly on exploration luck
- Agent could not solve it with learning rates larger than 0.0001, training is volatile due to the bootstrapping nature of Deep Q-Learning
- Deeper networks are harder to train without much performance benefit in this simple environment
- Double Q-Learning has a significant stabilizing effect on the training

Questions?