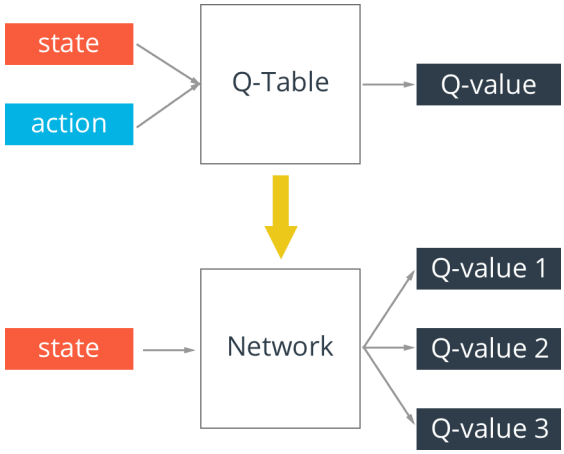
Lunar Lander

Introduction

The goal of this project is to create and train an agent using reinforcement learning to solve one of the simulated tasks from OpenAI Gym framework - “Lunar Lander”.
In this problem agent represents a landing spaceship. The state of the ship is represented by 6 continuous and 2 discrete values. Also agent can take one of 4 discrete actions at each state.

Learning model

Since transition model is unknown to solve Lunar Lander problem was chosen Q-learning that can learn based only on experience tuples of state, action, reward and next state. Traditional Q-learning uses table that maps states and actions to Q-value for each state-action pair which is then used to choose best possible action in particular state. But since Lunar Lander problem lies in continuous space usage of table is very problematic.
There are many ways to tackle this problem, but this project focuses on one of them in particular - deep Q-network (DQN). This approach replaces table in traditional Q-learner with neural network that returns Q-values for each action in any state. The big advantage of DQN is generalization which allows it to choose good actions even if exact input state hasn’t been seen before, but there were other similar states in the past experience tuples.



Neural network

In this project neural network used for learning Q-values consists of 3 fully connected layers with 64 neurons in each layer. Those neurons use ReLU activation function to better propagate errors back to first layers. Neural net also has linear output layer with 4 neurons, each neuron responsible for predicting Q-value for its action. Action with bigger Q-value is considered more favorable.
For more stable and faster training neural network is trained on small batches of experience tuples drawn from the memory buffer that accumulates those tuples during experiments. This modification is especially useful in reinforcement learning where correlation between adjacent states is high.

Experiments

Q-learner doesn't know anything about the environment, since it doesn't rely on transition or reward models, but it also means that it has to explore enough states and actions itself to make a good decisions in the future.
To address this issue in this project agent uses epsilon-greedy method. In the beginning of the experiment agent with high probability takes random actions to explore different states and actions, but during the experiment that probability decreases and at some point it uses only Q-values to make an educated choice of actions.
After taking new action agent adds new experience tuple to its memory buffer, draws new batch of experience tuples and trains neural network using those tuples. If Lunar Lander experiment episode has already ended (pod landed, crashed or exceeded maximum number of steps) then experiment is restarted. Process repeats until preset number of episodes is reached.

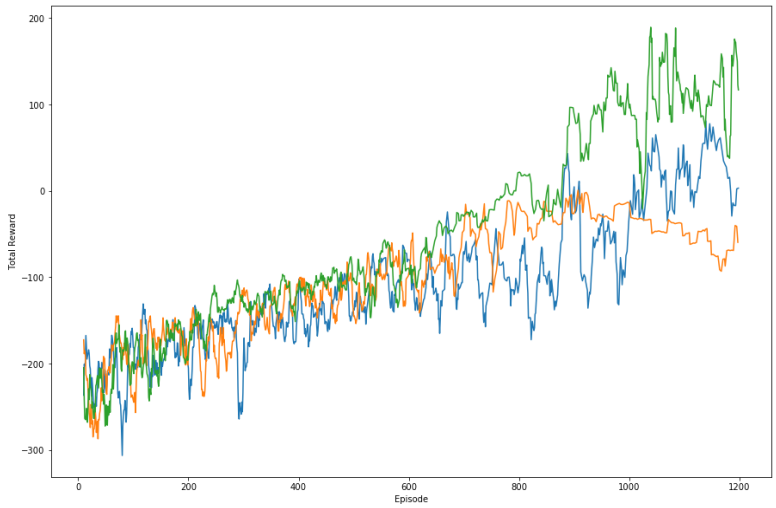
Overview

Parameters Tuning

One of the challenges of reinforcement learning and neural networks is a big number of hyperparameters. In this project most of those parameters were chosen to work together to produce good results. But for optimal performance those parameters should be tuned. In this section we explore how three of those parameters can affect agent performance.

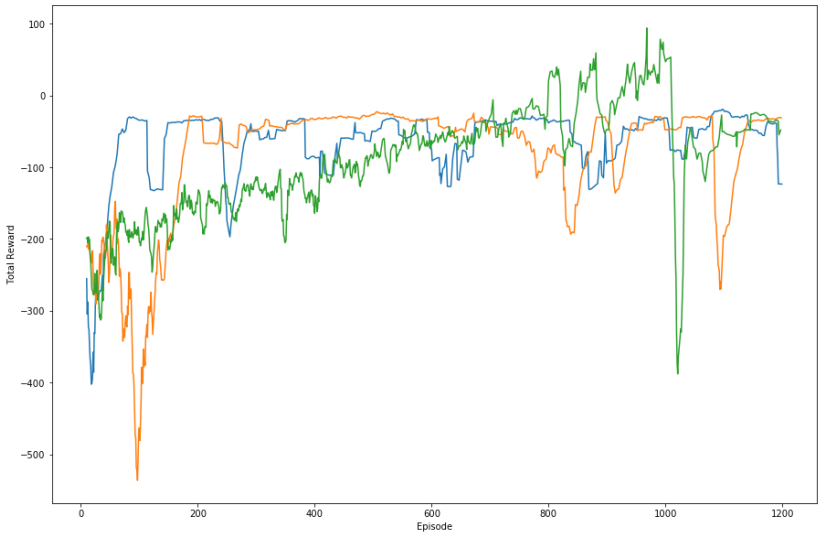
Rewards discount factor (gamma)

This parameter affects how far in the past effect of the current step reward can go. Too big gamma might lead to suboptimal strategies and might prevent agent from converging at all, but too small gammas means that agent might not know what to do at earlier steps as it doesn't know which action will lead to success.
Following graph shows training rewards of 3 gamma values: 0.99 (green), 0.9(blue), 0.8(red). Gamma equal 0.99 was the only one that could get close to successful landing. It can take the pod ~100 steps before it lands, which means that with discount factors of 0.9 and even worse 0.8 even the effect of the biggest reward given for landing in the center will be basically nullified once it reaches earlier steps.



Decay rate

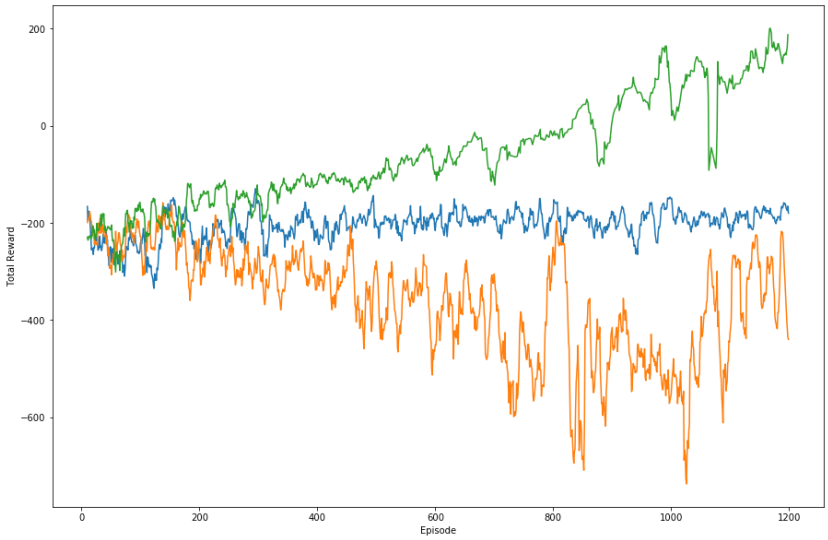
Decay rate is responsible for the speed at which probability of choosing the random action during the training will go to zero. If decay rate is too big then agent won't have enough time to explore the environment.



As we can see decay rate .00001 (green) was training slower than others in the beginning, since it was taking random actions more often, which is not very efficient. But it allowed it to continue improving longer than others as well.

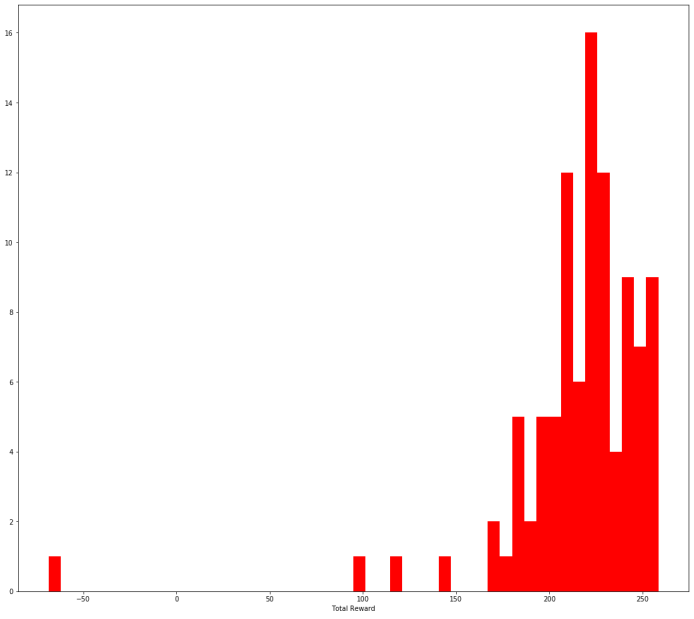
Learning rate

One of the most important parameter is learning rate. Too small learning rate will require agent more time to converge, but big learning rates might cause agent to converge to suboptimal strategy, never converge at all or even diverge at some point. And that's exactly what we see on the following graph, where learning rate .0001(green) is the only one that keeps improving while bigger values converge to suboptimal solution (blue) or diverge (red).



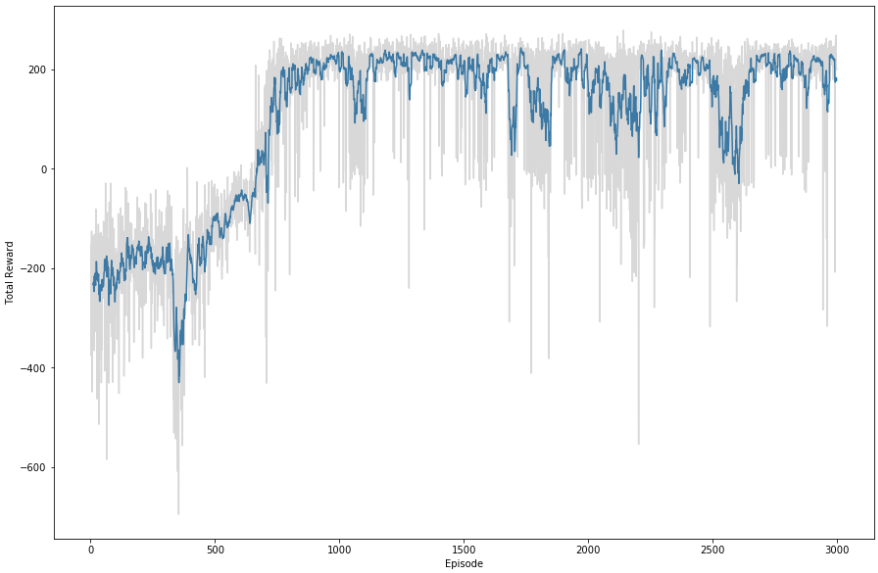
Results

Using described methods and after tuning DQN parameters agent was able to land a ship in most cases and do it efficiently (reward > 200) in ~80% cases. There were still some cases when agent couldn't land, most of these cases relate to randomly generated environment discussed in the following pitfalls section. Following histogram of total rewards per episode confirms this and shows us that on average agent performed even better that required by the task to be considered successful which is to gain 200 points.



Training

Despite good results agents training wasn't very smooth.



The graph above represents accumulated rewards gained by the agent during each training episode (grey). To better demonstrate overall trend a rolling mean was also added to the plot (blue).
We can see that even after many episodes rewards are still very volatile and it doesn't seem to improve after ~1000 episodes. But model also doesn't overfit as on average rewards pretty much converge and don't decrease as number of episodes grows.

Pitfalls

Randomly generated environment

In this project agent was able to learn right actions depending on its state to successfully land the pod in designated area in most cases. But since it has no information about the terrain it's difficult for agent to adapt to randomly generated terrain during each episode. That lead to the problem that in some cases terrain interfered with agent work and prevented it from successful landing.
One possible solution in this case is to keep training the agent until it learns straight vertical landing, in that case terrain wouldn't be a problem. But that would require either a lot more training episodes or some method that would focus on failed episodes forcing agent to learn new strategies.

Local optima

It was noted that with some parameters agent can actually learn suboptimal strategy when pod just simply levitates above ground even though using engines is penalized by the environment. This usually happens during bad explorations when agent hasn't performed enough successful landings before learning the strategy that forces pod to avoid heavily penalized crashes at all costs, even if it means to stay in the air indefinitely.