Virtual World Navigation Using Deep Reinforcement Learning

Preliminaries

What is reinforcement learning ?

According to wikipedia:

Reinforcement learning (**RL**) is an area of machine learning concerned with how software agents ought to take *actions* in an *environment* so as to maximize some notion of cumulative *reward*. The problem, due to its generality, is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics and genetic algorithms. In the operations research and control literature, reinforcement learning is called *approximate dynamic programming,* or *neuro-dynamic programming.* The problems of interest in reinforcement learning have also been studied in the theory of optimal control, which is concerned mostly with the existence and characterization of optimal solutions, and algorithms for their exact computation, and less with learning or approximation, particularly in the absence of a mathematical model of the environment. In economics and game theory, reinforcement learning may be used to explain how equilibrium may arise under bounded rationality.

In machine learning, the environment is typically formulated as a Markov Decision Process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques.The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible.

Reinforcement learning differs from standard supervised learning in that correct input/output pairs need not be presented, and sub-optimal actions need not be explicitly corrected. Instead the focus is on performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). The exploration vs. exploitation trade-off has been most thoroughly studied through the multi-armed bandit problem and in finite MDPs

Theoretical base for the project

The vanilla DQN used in this project is described by this paper:

> Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Belle- mare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, February 2015

Link to the original article - https://www.nature.com/articles/nature14236

The paper introduced using a experience replay buffer that stores past observations and uses them as training input to reduce correlations between data samples. They also used a separate target network consisting of weights at a past time step for calculating the target Q value. These weights are periodically updated to match the updated, latest set of weights on the main Q network. This reduces the correlation between the target and current Q values. Q target is calculated as below.

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a\right]$$

This is only a 10,000 feet overview, for details please read the paper, it will be worthy :)

In layman's words

The typical loop for a DQN looks like this (code biased towards pytorch :) …)

```
model = MyAI()
optimizer = torch.optim.Adam(model.parameters())
env.reset()
for step in range(0, nEpisodes):
 action = model.act(state, epsilon)
 next_state, reward, done, _ = env.step(action)
 replay_buffer.append(state, action, reward, next_state, done)
 state = next_state
 if done:
   env.reset()
 if len(replay_buffer) > batch_size: # won or died
   loss = calc_loss_over_episode(replay_buffer)
   optimizer.zero_grad()
   loss.backward()
   optimizer.step()
```

Two attention points:

- MyAI is generally a simple feed-forward network; caution to be taken, in order to avoid agent remaining stuck in a local minima, epsilon greedy strategy should be applied (think simulated annealing)
- The calc_loss_over_episode should be an implementation of the above equation, i.e. predict the Q values given current state, predict the Q values for the actual next state after we took the action predicted by the previous step, Discount the Q value a bit (multiply by gamma) before adding it to the expected reward, find the mean squared error

See the code for more details

Project specifics

Environment

The environment used for this project is the Udacity version of the Banana Collector environment, from Unity Technologies

The goal of the agent is to collect as many yellow bananas as possible while avoiding blue bananas. The task is episodic, and to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. The state space has 37 dimensions and contains the agent's velocity, along with a ray-based perception of objects around agent's forward direction.

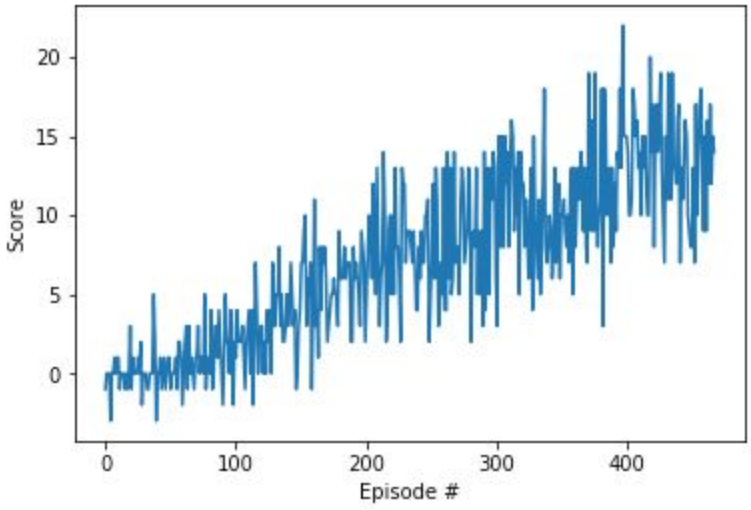
Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to moving forward, backward, turn left and right

(Some) Implementation details

In the Jupyter notebook you will find exactly this algorithm implemented. I did some juggling around with the hyperparameters and deep network layout, the ones in the book are the best I can come with. Feel free to modify and see what happens. My final training weights are checkpointed, so you can compare performance.
Side note, on most DQNs I studied around, the network layout is generally (env space) -> N -> N -> (action space) . N being a random value, but duplicated. I have not found a reasonable explanation as for why this is used, what I noticed is that, IN THIS SPECIFIC CASE, it converges faster if I take a "funnel" approach, which in my case is (env space) -> 128 -> 64 -> 32 -> (action space).

Given this approach, the environment is solved in 450-470 episodes (467 in the saved notebook)

Convergence looks like this :



Further investigation

- The successors of the vanilla DQN are next on my list : duelling DQN, prioritized replay, Rainbow DQN
- An interesting approach to RL is presented by Uber engineers in this paper : http://eng.uber.com/wp-content/uploads/2017/12/deep-ga-arxiv.pdf , I would surely like to test it in this environment, as it presents contradictory results depending on environments
- A somehow newer approach is augmented random search described here https://arxiv.org/abs/1803.07055 , I tested it only on the half cheetah, would be very curious on how it performs on this task.