Deep Reinforcement Learning Nanodegree

Project 1 : Navigation

Introduction

The goal of this project is to train a deep Q net agent to walk across a large banana world, collecting yellow bananas, and doing so while avoiding blue bananas, in a way, reward collected over a range of time frames is above a certain specified threshold, here +13.

The agent percepts a 37 dim vector from the environment including its velocity, ray based perception of objects, and given that state vector, we would like the Neural net to map that vector into best action to take (turn left ,right, forward, backward)

Learning Algorithm

At the heart of our Q-learning based algorithm consists of optimizing a neural network that maps 37 dim vectors into the action space of an agent.

The deep neural network is trained to be able to assign the action value of each action given a state input. As net is randomly initialized at first, and we are using an estimate of an estimate to train the network, we keep two separate networks to assign action values, local and target.

The local network generates action value estimates for current state, while the target network generates estimates for target action values.

In order to add some stability in learning, especially as the beginning phase, we keep target network weights constant (not exactly) . For our case, rather than pulling off a hard update of the target network after some few thousand time steps as David Silver mentions in his atari solving DQN paper , we blend local net weights into that of the target net slowly (of 0.0001 %), at every 4 time steps, controlled by parameter *tau*.

Another important modification to Original q learning algorithm was to collect the samples of agents' experiences (state_n, action_n , reward_n+1 ,state_n+1, done) in a Replay Buffer of size : 10^5. Then after buffer size is at least as large as our specified batch size of 64, we sample from that buffer and train the network.

This significantly eliminates the sequential dependability of the network.

Another slight tweak to prevent suboptimal policy because of high variance rewards, was use of double dqn technique as adviced in the course, which accelerated learning in my code. Basically, now the model is using local networks best actions for predicting target next state values.

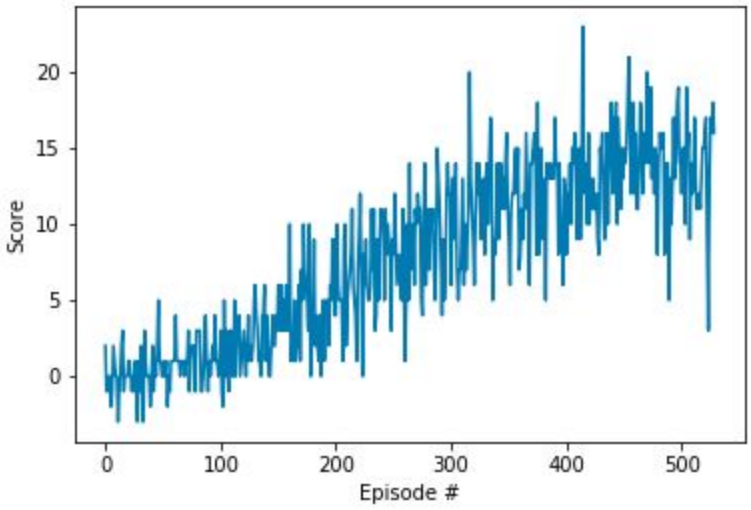
Neural Network Architecture

The architecture is a fundamental feedforward net of two 64 node hidden layers, both RELU activated, and finally an output layer equal to dimension of the action space of the model. I didn't use Relu activation on the last layer, thinking, just a linear combination was a better way to estimate action values. But, at end of day, gradient descent works and optimizes, irrespective of using activation or not.

Hyperparameters

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate of optimizer, we choose ADAM
UPDATE_EVERY = 4        # time steps gap to soft update target network
```

Plot of Rewards



Ideas for Future work

Currently, we are hoping that, on an infinite run, agents will pick up a reward signal that effectively estimates the action values, and bootstrapping from that point on, learning will grow rapidly. While there's nothing wrong with it, in the real world problem with insurmountable state space, we need to make sure agents make best use of the experiences it has had.

Different ways like Dueling network, n-step bootstrapping , Distributional RL with parallel agents sampling their own experiences of the environment will escalate learning. But furthermore, there’s something limited with the notion of using just DQN, since it doesn't necessarily address maximizing over a trajectory. i.e, a set of actions that will result in better performances in problems associated with rewards after long time periods. Example, Dota 2 , go e.t.c.