Project 1: Navigation

April 2, 2021

Contents

1 Project Overview

In this project, we implement a deep q network to train an agent to learn how to navigate OpenAI Gym's Banana environment. In this episodic environment, the objective is to collect as many yellow bananas as possible while avoiding blue bananas. The agent is capable of moving forwards, backwards, left, or right, and perceives the environment via a 37-dimensional ray-based vector around the agent's forward direction. Each episode terminates after 1,000 time steps have been traversed, and we consider the agent to have learned the environment when it is able to achieve a score of +13.

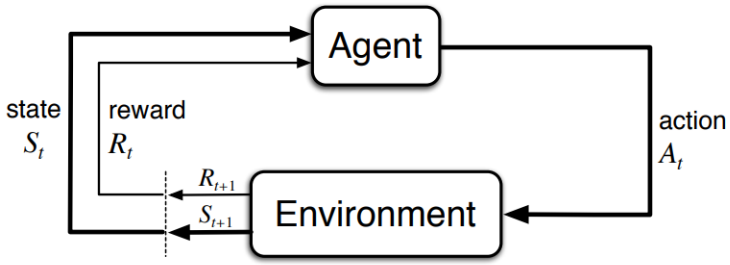


Figure 1:
Sutton and Barto (2018)

2 The Algorithm

2.1 Preliminaries

- s : 37-dimensional vector of where each dimension contains a double-precision value between 0 and 1.
- $\mathbb{A}(s) = \{left, right, forward, backward\}$: Set of all possible actions available to the agent in state s.
- $\pi(a|s) = \mathbb{P}(A_t = a|S_t = s)$: The policy function; a mapping from state s to the probability of taking some action $a \in \mathbb{A}$. This effectively chooses an action after observing the state.
- $Q_\pi(s, a)$: The value of taking action a in state s, provided that the policy π is used to choose the action. This can be thought of as a table, or a dictionary, that has 4 columns corresponding to the 4 possible actions and x rows corresponding to the x possible states.

2.2 Learning Algorithm

The agent interacts with an enviroment through a series of observations in its current **state**, **actions**, and **rewards**. The goal of the agent is to maximize its expected cumulative future **reward** over an **episode** by strategically choosing **actions** that transition the agent from its current **state** to the next.

In this example, an **episode** is defined as 1000 time steps and our agent is rewarded +1 for collecting a yellow bannana and -1 for collecting a blue bannana. But how does a naive agent learn to take **actions** strategically, especially considering the fact that it must often take intermediate non-rewareded steps prior to receiving any real feedback from the environment?

We start by considering the following objective function that brings together the concepts of **state**, **action**, **reward** over time:

$$
\begin{aligned}
Q_\pi(s,a) &= \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi\right] \\
\gamma &:= \text{intertemporal discount factor} \\
r_t &:= \text{reward at time } t \\
s_t &:= \text{state at time t} \\
\pi &:= \text{some policy} \\
Q_\pi(s,a) & \quad \text{cumulative expected reward from following policy } \pi
\end{aligned}
$$

It can be helpful to visualize the Q function in the following Q table, where in this case there are 37 columns corresponding to the 37 possible states and 4 rows corresponding to the 4 possible actions.

$Q^\pi(s_1a_1)$	$Q^\pi(s_2a_1)$	...	$Q^\pi(s_{36}a_1)$	$Q^\pi(s_{37}a_1)$
$Q^\pi(s_1a_2)$	$Q^\pi(s_2a_2)$		$Q^\pi(s_{36}a_2)$	$Q^\pi(s_{37}a_2)$
$Q^\pi(s_1a_3)$	$Q^\pi(s_2a_3)$		$Q^\pi(s_{36}a_3)$	$Q^\pi(s_{37}a_3)$
$Q^\pi(s_1a_4)$	$Q^\pi(s_2a_4)$		$Q^\pi(s_{36}a_4)$	$Q^\pi(s_{37}a_4)$

So we first find ourselves in a **state** (column) then consult our policy π to choose an **action** (row). Since our objective is to maximize our expected cumulative future reward, we want to choose the row that has the highest expected value Q^*. In theory there exists some optimal policy π^* which gives us a optimal Q table Q^*, which we approximate using a deep neural network.

$$
Q^*(s,a) = \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi^*\right]
$$

We assume the game to be markovian in nature, which enables us to represent the Q function as the famous Bellman equation:

$$
\underbrace{Q(s,a,w)}_{\text{Local Q Network}} = \underbrace{r(s,a) + \gamma \max_a Q(s',a)}_{\text{target Q network}}
$$

We also use a neural network to approximate the Q table. We notice that at each transition described above, there are 2 possiblities: either the equation holds or it does not. If the equation does not hold, we simply update Local Q network to match the target Q network. But what if it does hold? Then we know that we have discovered our optimal Q function, since the Q table already corresponds to the optimal action state pair. This leads us to our objective function, which is to minimize the squared difference between our local and target Q networks.

$$
\mathcal{L} = \left(r(s,a) + \gamma \max_a Q(s',a) - Q(s,a)\right)^2
$$

2.3 Deep Q Network Architecture and algorithm

The deep Q network is designed with 3 layers. The first is a 37-dimensional input layer of the agent's ray-based perception of the environmentfollowed by two hidden layers of 37 dimensions apiece and a 4-dimensional output layer where each dimension corresponds to one of the 4 possible actions. We note that Q learning can be unstable and even diverge. To address these challenges, we implement several techniques to smooth the learning process. We implement a replay buffer of 10,000 episodes to store past experiences and correct for correlations when training our network by randomly sampling from these experiences during training. We utilize a soft-update paramater τ to reduce the variance of the network weight updates. The full algorithm is as follows:

- Initialization: Randomly initlialize the weights θ in the Q network.
- Initialize a replay buffer D with capacity N
- Initialize target network $\hat{Q}$ weights θ^{-}

For $t = 1, T$ **do**:

1. Observe current state s_t
2. Given state $s_t$choose the action a_t that maximizes the expected reward as per the local Q network.
   (a) $a_t = argmax_a Q(s_t, a; \theta)$We choose the action in an epsilon-greedy manner where epsilon decays at a rate of .99 from 1 to .01.
3. Perform the action a_t and step into state s_{t+1} and receive any relevant reward r_{t+1}
   (a) add experience (s_t, a_t, r_t, s_{t+1}) to replay memory
   (b) Sample a batch of 32 experiences (s_j, a_j, r_j, s_{j+1}) from D
   (c) Set $y_j = \begin{cases} r_j & \text{if eposide terminates at step j+1} \\ r_j + \gamma max_{a'} \hat{Q}(s_{t+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
   (d) Perform gradient descent on $L(\theta) = (y_j - Q(s_j, a_j; \theta))^2$ to optimze the network paramaters θ with learning rate α
   (e) Every 4 steps, update $\hat{Q} = Q$.

Steps 1-3 comprise an episode. Repeat eposides until a target reward of 13 has been achieved.

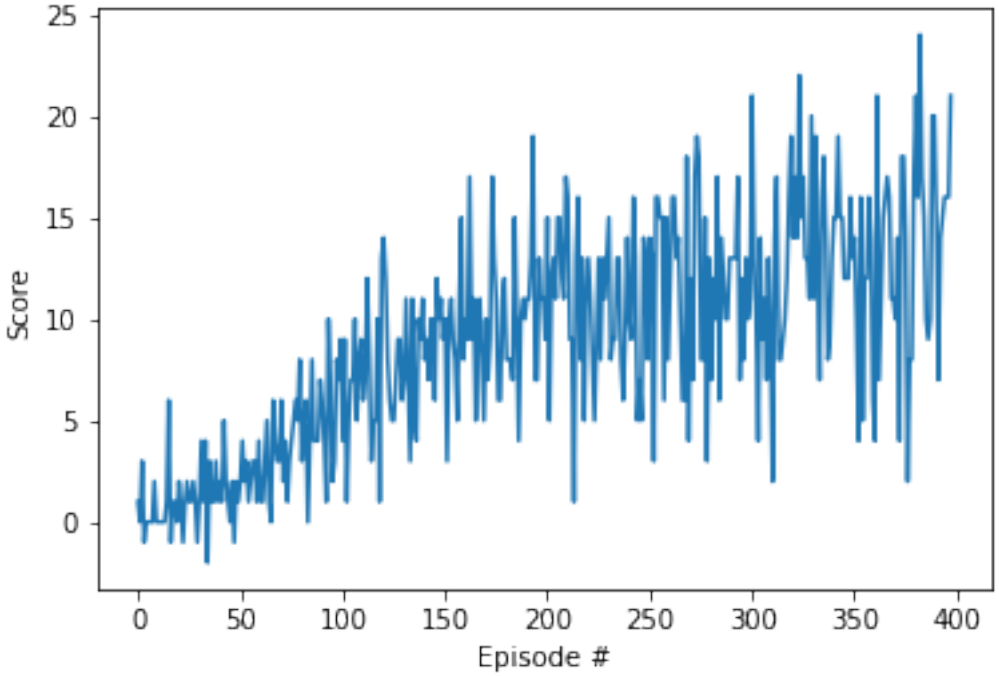
2.4 Paramaters

- Max Episodes: 2,000 (maximum number of training episodes)
- max_t: 1000 (maximum number of training steps per episode)

- ϵ: [1.0,0.01] Noise paramater for epsilon-greedy selection; Decays at a rate of .99 from 1 to .01.
- Replay Buffer Size: 10,000: Number of episodes to retain in replay buffer.
- Batch Size: 32
- $\gamma = .99$: Discount factor for future rewards
- $\tau = .001$: Soft update paramater
- α : Learning Rate = .00005: Rate at which algorithm learns during the gradient descent step.
- Update Every: 4: How often to update the Q network with updated values
- γ: .99 (intertemporal discount factor)

3 Learning Curve

The aforementioned algorithm was able to solve the environment in 298 episodes.



4 Future Work

An interesting direction to take this research in would be feed to agent a sequence of images from the environment rather than a simple ray-based percetion and leverage the tools from computer vision process the raw pixels before training the network. Additional work could also by integrating multiple agents into the environment, or applying an actor-critic method