report

January 25, 2020

1 Navigation Project Report

1.0.1 1. Project Overview

In this project, a reinforcement learning (RL) agent is trained to navigate an environment the collect bananas:

The simulation contains a single agent that navigates a large environment. At each time step, it has four actions at its disposal:

- 0 - walk forward
- 1 - walk backward
- 2 - turn left
- 3 - turn right

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around agent's forward direction. A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. So the goal is to collect as many yellow bananas as possible while avoiding blue bananas. In order to solve the environment, the agent must achieve an average score of +13 over 100 consecutive episodes.

1.0.2 2. Learning Algorithm

We apply Q-learning algorithm to train the agent. **Q-learning** is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

2.1. Introduction General Description

The general ideas for the algorithm are described in the following:

The weight for a step from a state Δt steps into the future is calculated as $\gamma^{\Delta t}$, where γ (the discount factor) is a number between 0 and 1 ($0 \leq \gamma \leq 1$) and has the effect of valuing rewards received earlier higher than those received later (reflecting the value of a "good start"). γ may also be interpreted as the probability to succeed (or survive) at every step Δt.

The algorithm, therefore, has a function that calculates the quality of a state-action combination:

$Q : S \times A \rightarrow \mathbb{R}$.

Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1} (that may depend on both the previous state s_t and the selected action), and Q is updated. The core of the algorithm is a simple value iteration update, using the weighted average of the old value and the new information:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

where r_t is the reward received when moving from the states$_t$ to the state s_{t+1}, and α is the learning rate ($0 < \alpha \leq 1$).

An episode of the algorithm ends when state s_{t+1} is a final or terminal state. However, Q-learning can also learn in non-episodic tasks. If the discount factor is lower than 1, the action values are finite even if the problem can contain infinite loops.

For all final states s_f, $Q(s_f, a)$ is never updated, but is set to the reward value r observed for state s_f. In most cases, $Q(s_f, a)$ can be taken to equal zero.

Learning Variables

1. **Learning Rate**

The learning rate or step size determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities).

2. **Discount Factor**

The discount factor γ determines the importance of future rewards. A factor of 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, i.e. r_t (in the update rule above), while a factor approaching 1 will make it strive for a long-term high reward. If the discount factor meets or exceeds 1, the action values may diverge.

3. **Initial Conditions**

Since Q-learning is an iterative algorithm, it implicitly assumes an initial condition before the first update occurs. High initial values, also known as "optimistic initial conditions", can encourage exploration: no matter what action is selected, the update rule will cause it to have lower values than the other alternative, thus increasing their choice probability. The first reward r can be used to reset the initial conditions.

Methods

1. **Monte Carlo Methods**

Monte Carlo methods - even though the underlying problem involves a great degree of randomness, we can infer useful information that we can trust just by collecting a lot of samples.

The *equiprobable random policy* is the stochastic policy where - from each state - the agent randomly selects from the set of available actions, and each action is selected with equal probability.

Algorithms that solve the prediction problem determine the value function v(or q) corresponding to a policy .

When working with finite MDPs, we can estimate the action-value function q corresponding to a policy in a table known as a Q-table. This table has one row for each state and one column for each action. The entry in the ss-th row and aa-th column contains the agent's estimate for expected return that is likely to follow, if the agent starts in state ss, selects action aa, and then henceforth follows the policy .

Each occurrence of the state-action pair s, a $(s \in \mathcal{S}, a \in \mathcal{A})$ in an episode is called a visit to s, a. There are two types of MC prediction methods (for estimating q):

First-visit MC estimates $q_\pi(s, a)$ as the average of the returns following only first visits to s,as,a (that is, it ignores returns that are associated to later visits).

Every-visit MC estimates $q_\pi(s, a)$ as the average of the returns following all visits to s,as,a.

Epsilon-Greedy Policies

A policy is ϵ-greedy with respect to an action-value function estimate Q if for every state sS, with probability 1, the agent selects the greedy action, and with probability , the agent selects an action uniformly at random from the set of available (non-greedy AND greedy) actions.

2. **Temporal-Difference Methods**

Whereas Monte Carlo (MC) prediction methods must wait until the end of an episode to update the value function estimate, temporal-difference (TD) methods update the value function after every time step.

TD Control

Sarsa(0) (or Sarsa) is an on-policy TD control method. It is guaranteed to converge to the optimal action-value function q_*, as long as the step-size parameter is sufficiently small and is chosen to satisfy the Greedy in the Limit with Infinite Exploration (GLIE) conditions.

Sarsamax (or Q-Learning) is an off-policy TD control method. It is guaranteed to converge to the optimal action value function q_*, under the same conditions that guarantee convergence of the Sarsa control algorithm.

Expected Sarsa is an on-policy TD control method. It is guaranteed to converge to the optimal action value function q_*, under the same conditions that guarantee convergence of Sarsa and Sarsamax.

Analyzing Performance

On-policy TD control methods (like Expected Sarsa and Sarsa) have better online performance than off-policy TD control methods (like Q-learning).

Expected Sarsa generally achieves better performance than Sarsa.

2.2. Using DQN to train the agent Q-learning can be combined with function approximation. This makes it possible to apply the algorithm to larger problems, even when the state space is continuous.

One solution is to use an (adapted) artificial neural network as a function approximator. Function approximation may speed up learning in finite problems, due to the fact that the algorithm can generalize earlier experiences to previously unseen states.

Deep Q-Network (DQN)

With Deep Q-Learning, a deep neural network is used to approximate the Q-function. Given a network F, finding an optimal policy is a matter of finding the best weights w such that F(s,a,w) Q(s,a).

the Deep Q-Learning algorithm represents the optimal action-value function q_* as a neural network (instead of a table).

Unfortunately, reinforcement learning is notoriously unstable when neural networks are used to represent the action values. In this lesson, you'll learn all about the Deep Q-Learning algorithm, which addressed these instabilities by using two key features:

Experience Replay

Fixed Q-Targets

Experienced replay

As for the network inputs, rather than feeding-in sequential batches of experience tuples, randomly sample from a history of experiences using an approach called Experience Replay.

When the agent interacts with the environment, the sequence of experience tuples can be highly correlated. The naive Q-learning algorithm that learns from each of these experience tuples in sequential order runs the risk of getting swayed by the effects of this correlation. By instead keeping track of a replay buffer and using experience replay to sample from the buffer at random, we can prevent action values from oscillating or diverging catastrophically.

The replay buffer contains a collection of experience tuples (SS, AA, RR, S'S). The tuples are gradually added to the buffer as we are interacting with the environment.

The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

1.0.3 3. Training Process of the project

The neural network architecture used for this project can be found here in the model.py file of the source code.

The network architecture is defined using pytorch.

From all four test plots, it is observed that an average score of +13 over 100 consecutive episodes

3.1. Using different Learning variables In the following plots, you can see the effect of changing the epsilon decay rate on learning. It shows that by changing it from 0.995 to 0.95, it takes fewer episodes to get to the required score 13.

There are many other parameters to try:

n_episodes (int): maximum number of training episodes

max_t (int): maximum number of timesteps per episode

eps_start (float): starting value of epsilon, for epsilon-greedy action selection

eps_end (float): minimum value of epsilon

eps_decay (float): multiplicative factor (per episode) for decreasing epsilon

UFFER_SIZE = int(1e5):replay buffer size

BATCH_SIZE = 64:minibatch size

GAMMA = 0.99:discount factor

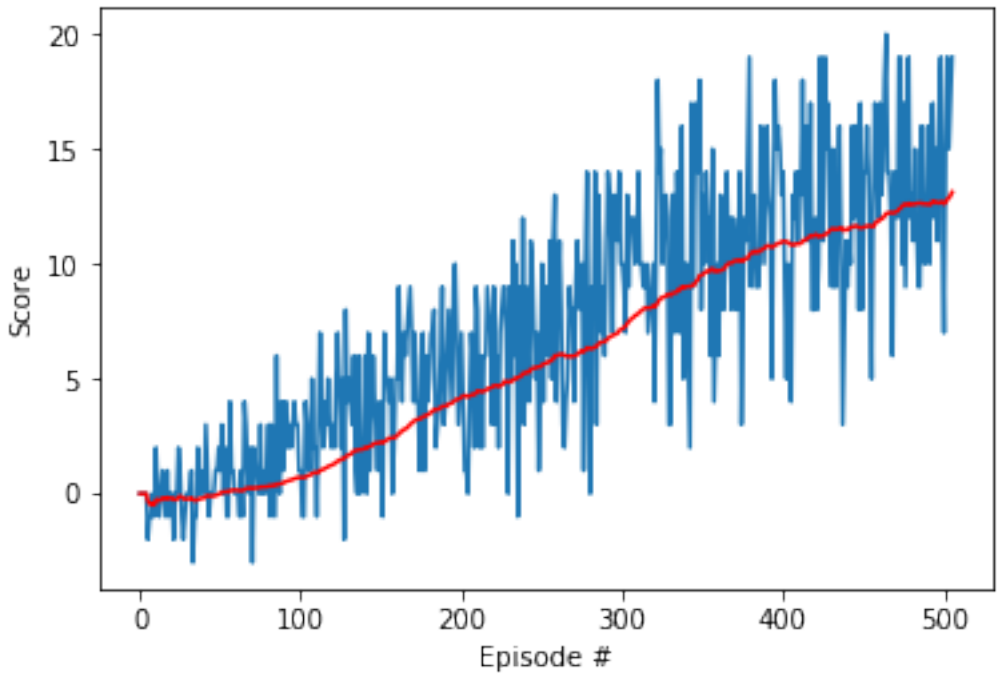
TAU = 1e-3:for soft update of target parameters

LR = 5e-4: learning rate

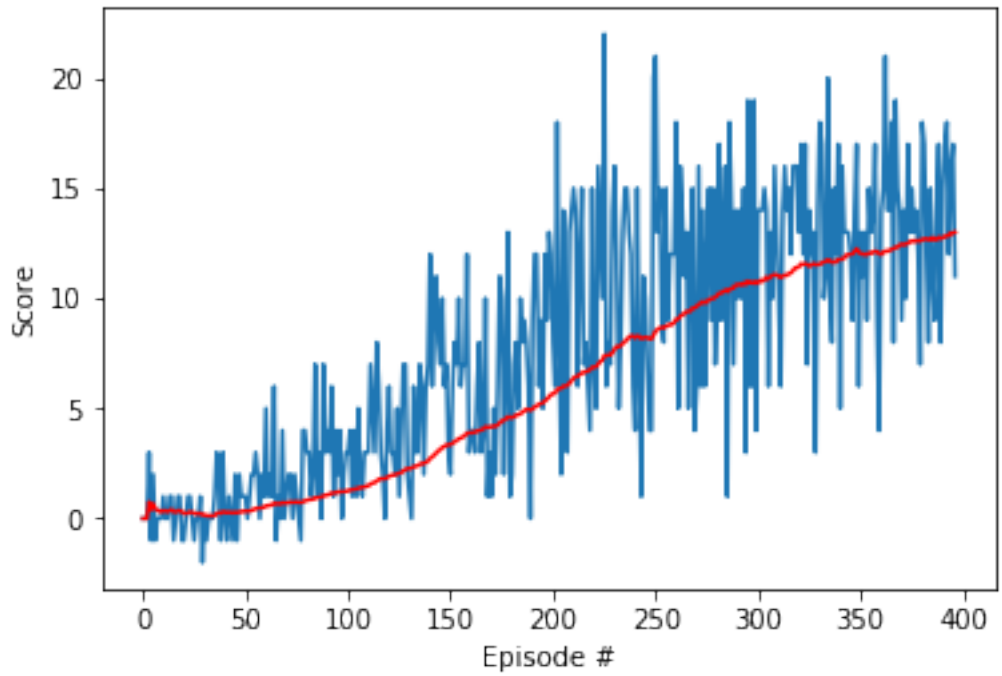
UPDATE_EVERY = 4:how often to update the network

```
In [34]: scores,avgs = dqn(n_episodes=600,eps_start=1.0, eps_end=0.01, eps_decay=0.995, ckpt_pat

Episode 100        Average Score: 0.70
Episode 200        Average Score: 4.14
```

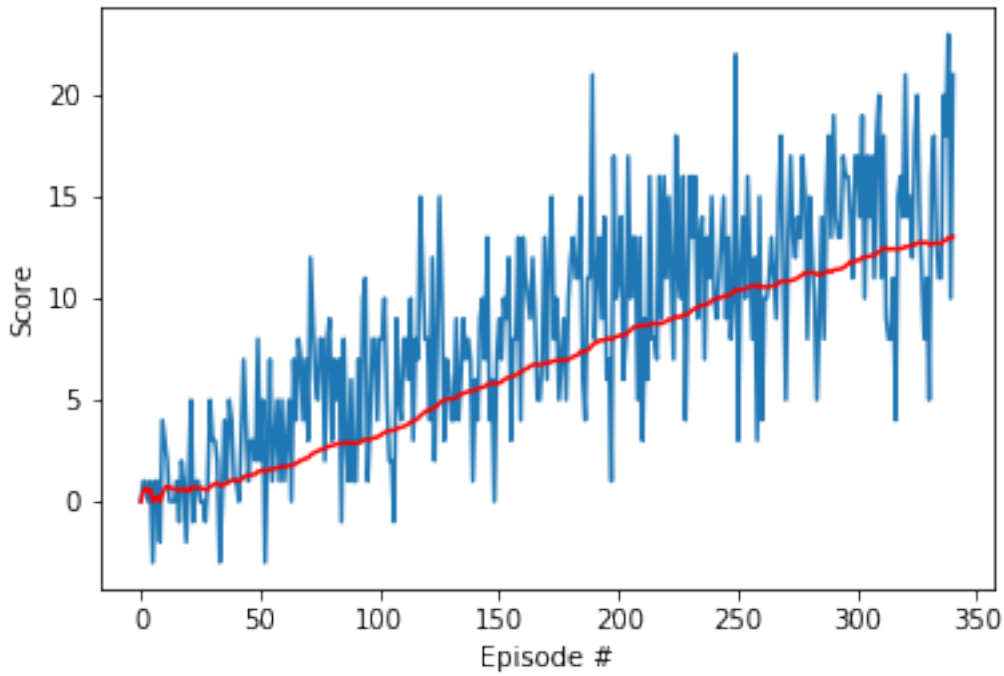
```
Episode 300        Average Score: 7.14
Episode 400        Average Score: 10.94
Episode 500        Average Score: 12.68
Episode 506        Average Score: 13.12
Environment solved in 406 episodes!        Average Score: 13.12

Total Training time = 10.7 min
```




```
In [32]: scores,avgs = dqn(n_episodes=600, eps_decay=0.98, eps_end=0.02, ckpt_path='pth_checkpoi

Episode 100        Average Score: 1.23
Episode 200        Average Score: 5.62
Episode 300        Average Score: 10.67
Episode 397        Average Score: 13.00
Environment solved in 297 episodes!        Average Score: 13.00

Total Training time = 8.3 min
```




```
In [35]: scores,avgs = dqn(n_episodes=600,max_t=1000, eps_start=1.0, eps_end=0.03,eps_decay=0.95

Episode 100        Average Score: 3.16
Episode 200        Average Score: 8.11
Episode 300        Average Score: 11.82
Episode 341        Average Score: 13.04
Environment solved in 241 episodes!        Average Score: 13.04

Total Training time = 7.2 min
```

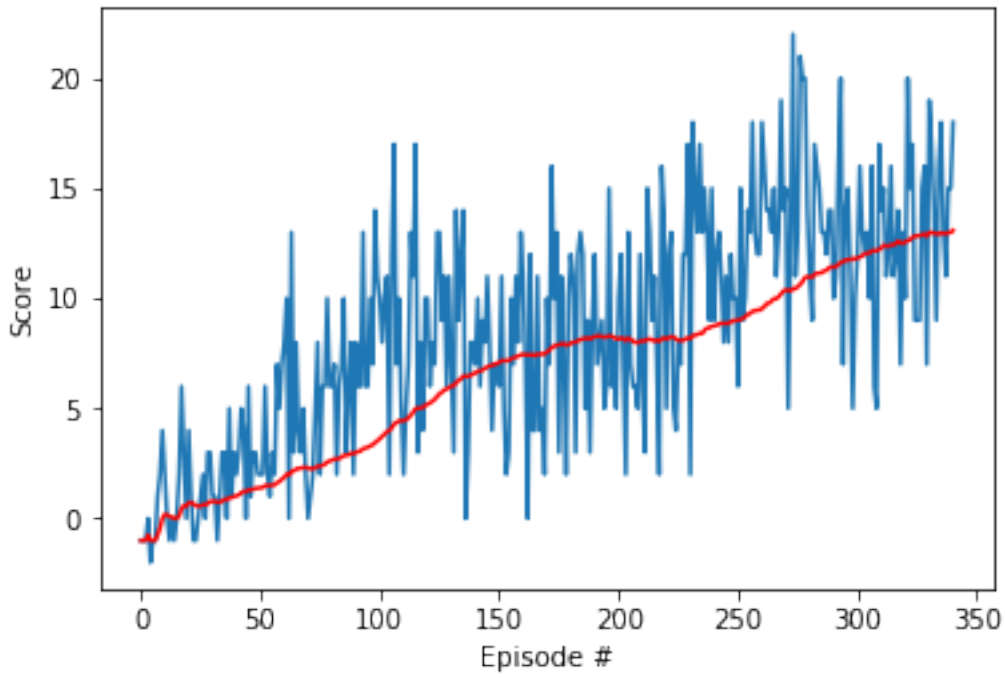


3.2. Change the model parameters Can change neural network architecture to see the effect too:

model_ori.py: The network contains three fully connected layers with 64, 64, and 4 nodes respectively. model.py: The network contains three fully connected layers with 64, 64,64 and 4 nodes respectively.

Testing of bigger networks (more nodes) and deeper networks (more layers) did not produce much improved results. (test 3 and 4 contains comparison of 2 layer NN and 3 layer NN results)

```
In [12]: scores,avgs = dqn(n_episodes=600,max_t=1000, eps_start=1.0, eps_end=0.03,eps_decay=0.95

Episode 100        Average Score: 3.52
Episode 200        Average Score: 8.16
Episode 300        Average Score: 11.81
Episode 341        Average Score: 13.08
Environment solved in 241 episodes!        Average Score: 13.08

Total Training time = 7.6 min
```



It seems Adding one layer doesn't improve the number of episodes to get to score =13, and it takes slightly longer

1.0.4 4. Ideas for Future Improvement

Deep Q-Learning Improvements

Several improvements to the original Deep Q-Learning algorithm have been suggested. Over the next several videos, we'll look at three of the more prominent ones.

1. **Double DQN** Deep Q-Learning tends to overestimate action values. Double Q-Learning has been shown to work well in practice to help with this.
2. **Prioritized Experience Replay** Deep Q-Learning samples experience transitions uniformly from a replay memory. Prioritized experienced replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability.
3. **Dueling DQN** Currently, in order to determine which states are (or are not) valuable, we have to estimate the corresponding action values for each action. However, by replacing the traditional Deep Q-Network (DQN) architecture with a dueling architecture, we can assess the value of each state, without having to learn the effect of each action.

Learning from Pixels

In the project, the agent learned from information such as its velocity, along with ray-based perception of objects around its forward direction. A more challenging task would be to learn directly from pixels!

This environment is almost identical to the project environment, where the only difference is that the state is an 84 x 84 RGB image, corresponding to the agent's first-person view of the environment.

The Deep NN model for MLP (multi-layer perception) can be changed to pixel-based approach using CNN (convolutional neural network), where the input is 84 x 84 RGB images.