Deep Reinforcement Learning Expert Nanodegree

Report
Project 1: Navigation

Introduction

In this project a Deep Q-Network agent was trained with Deep Reinforcement Learning to walk across a large square world, while collecting yellow and avoiding blue bananas.

The project is composed by four files: *Navigation.ipynb* a Jupyter Notebook file containing the main code to initialize dependencies, environment and Agent; *dqn_agent.py*, a code that contains the characteristics of the agent and how it behaves through this projects exigencies; *model.py,* a model of the deep neural network(DNN) used by the agent to learn a policy and solve the environment; *checkpoint.pth* a file with the DNN's weights, that solved the environment, saved.

The problem trying to be solved is modeled as a Markov Decision Process, involving *mappings* from states to actions, in such way that this actions will maximize the total cumulative reward signal of the agent. States are any information that the environment provides the agent, excluding the reward signal. Actions are ways that an agent can interact with the environment, and rewards are signals that the agent receive after interacting with the environment.

The solution to the problem, on this project, is a mapping called *policy*, ϵ(epsilon)-greedy policy specifically, that is guided towards the maximum cumulative rewards by the *action-value functions (q(s,a)).* This function gives the agent a sense of how good or bad is to be in a given state. The actions are selected more probably for the ones with maximum *q(s,a)* in the state, while choosing less probably the other actions. After interacting many times with the environment, it's possible to find an *optimal policy* coming from an *optimal action-value function* $q_*(s,a)$. The sense of optimal here is a mapping that can accomplish a given goal, with certains restraints.

To approximate this function Deep Q-Networks are used. These are value based methods of Reinforcement Learning that make use of DNN's as *non-linear function approximators*. With a state as input, the network outputs *q(s,a)* for each possible action in that state. After a defined number of time steps, DNN's parameters are modified towards minimizing loss between a target action value function, and the currently one being used.

Implementation

Preparation

The goal is to train an agent able to get an average score of +13 over 100 consecutive episodes. The scores are distributed like follows: +1 each time the agent collects a yellow banana and -1 when it collects blue ones. The reinforcement learning system actions are directed towards maximizing cumulative scores thus, the agent ends up learning that, to achieve this, it's better to avoid the blue ones and collect more yellows, by changing Its actions on certain states.

At first, in the notebook file, the dependencies are installed, libraries are imported, the simulation environment is initialized.

The next step is to explore the State and Action Spaces. State space is a vector composed by 37 features including the agent's velocity, along with ray-based perception of objects around the agent's forward direction and, the action space, has a dimension of four (turn left and right, move forward and backward.

To learn how the Python API controls the agent and receives the feedbacks from the environment, a code cell is provided with a random action agent.

Learning Algorithm

The main implementation begins on fourth step: A Deep Q-Network agent is imported from *dqn_agent* code and initialized with proper parameters: The state and action space sizes. The *dqn* function is defined with it's parameters: The number of training episodes of *agent x environment* interaction, number of time steps for each episode, starting, final and decay values for the epsilon-greedy policy.

In each episode the environment is reseted and the agent is put in an initial state, while the number of timesteps is less than a maximum defined, at the *dqn function paramaters,* the following procedures are done:

The agent takes an action by evaluating the state it belongs, the action chosen is either the greedy action, the one that maximizes the action-value function, with probability equals to ϵ or, with 1-ϵ probability, a random action (including the greedy one). After selecting an action, and sending it to the environment, the agent receives it's *next state*, *reward signal* and a information that says if episode is over or not. With these informations the agent takes a step, these means that the following will happen: It will store the informations from the environment, called as one *experience*, action, reward and next state in a memory for future learning, this technique is called *Experience Replay* because it can provide multiple learning steps from a single experience. This procedure is made until memory's size is greater than *batch size* hyper parameter. When this happens the agent calls it's *learn* method, and a number of *experiences* is randomly sampled from the memory .

As previously explained the objective is to approximate a target *action value function*. This is done by first, as the agent is initialized, creating two different Q-Networks, target and local, this approach is named *Fixed Q-Targets*, standing that one will remain unchanged while the other will move towards it. The learning takes place in a numerical optimization using stochastic gradient descent, with some tweaks, called *Adam*. The optimization objective is to find DNN's weights that minimize the *mean squared error* between target's Q-Network predicted action value for the next state and the local's network predicted action value for the current state. Each optimizer step is taken in the direction of minimizing this loss with respect to the weights. At least a soft update is made in the target network parameters, this update is controlled by a parameter *TAU*, that defines a proportion of the local networks weights on the new target network.

Another implementation used is *Double Q-Learning*, instead of evaluating the target Q-network's values using it's own best actions predictions, that would be highly biased, the model uses the local network's best actions for predicting the target next state values.

After this *agent step* the state becomes the next state, giving continuation to the episode, reward is added to *scores* variable and environment checks if the episode is over, otherwise, another time step is executed and the agent selects an action.

Neural Network Architecture

The artificial neural network used is very simple and small. Composed of three hidden layers with 256,128 and 64 nodes respectively. Each hidden layer is followed by a ReLU activation function. The network take as input the state and outputs the action value function for each possible action in that state.

Rewards per Episode

Next, an image of the Agent's rewards obtained in each episode until solving the environment.

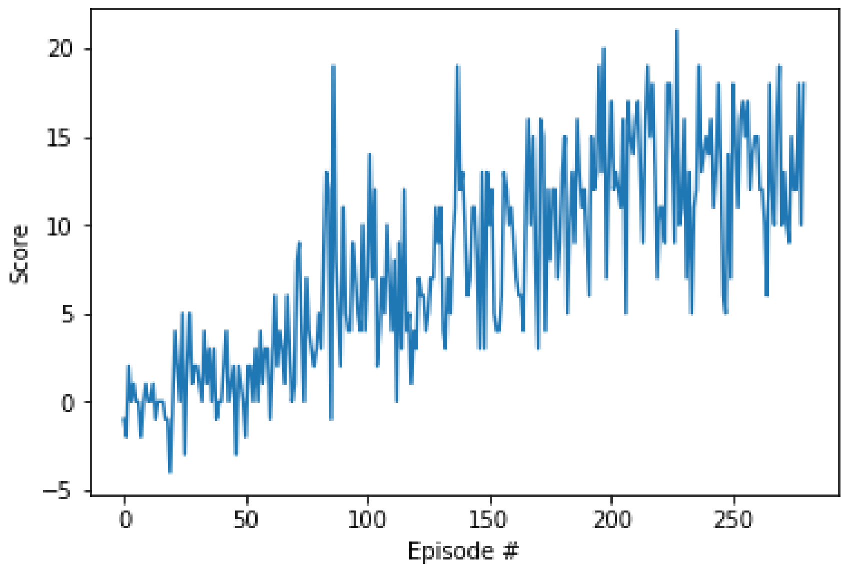

Figure 1: Rewards per episode plot of the solving agent.

Results

The maximum number of episodes for solving episodes was 1800. The agent with Double Q-Learning first solved it with 600. After tweaking some parameters the below results were obtained:

Episode Interval	Average Score
0~100	2.53
100~200	8.90
180~280	13.00

Table 1: Average scores in their following episode interval.

A table withmodified hyperparameters is included below:

Hyperparameter	**Value**
eps_start	1
eps_end	0.025
eps_decay	0.85
Learning Rate	$1e^{-3}$
TAU	$2e^{-3}$
BATCH_SIZE	64

Table 1: Hyperparameters used by the best solving agent.

Ideas for Future Work

Deep Reinforcement Learning nature is very volatile, as Figure 1 shows, the ups and downs are very present during the whole training. To help in these issues features like *Experience Replay, Fixed Q-Targets* and *Double Q-Learning* were used. Still, there are more techniques that can and should be used: *Prioritized Experience Replay, Dueling Networks, Multi-Step Learning* and *Distributional RL.* Combining these with the *Double Q-Learning* compose a named *Rainbow Agent*. It's shown that these improvements combined outperform the traditional approach used in this project.

Also use *rainbow agent* with convolutional neural network and solve the "Pixel Navigation" optional project, that takes as input the sequences of frames from the environment, this is a must do for really training smart agents that can interact with the real world.