# Report

May 27, 2020

## 1 Navigation

### 1.0.1 Introduction

Unity ML-agent library has a lot of simulation evnrionments to test intelligent agent models. This notebook solves the navigation task in an environment with bananas. The task is to move an agent to gain as many scores as possible by obtaining yellow bananas and avoiding purple ones. The agent uses a convention model(DQN) to learn an intelligent behavior to meet the objective. All the code is self contained in a notebook for a potential modification of any logic. Each component is separated to its own cell, so it can be easily written to a separate file using "%writefile" if necessary

---

### 1.0.2 Environment

#### Requirements

- Linux based OS
- Python (dependency is listed in requirements.txt)
- Linux unity banana environment

The following code will import modules and packages required to train and evaluate a DQN model

#### The environment spec

- Number of agent: 1
- State space: 37 dimensional vector
- Action space: 4 discrete numbers (0: forward, 1: backward, 2: left, 3: right)
- Reward: +1: yellow banana, -1: purple banana

### 1.0.3 Implementation

#### Model

- **Description**
  - Standard fully-connected NN with a Batchnorm layer and ReLU activation. This model is used to evaluate Q values given a state.
- **Architecture**

    - 2 Hidden layers (128, 64 nodes)
    - 1 Output layer (4 nodes)
    - Each hidden layer uses ReLU and Batchnorm1d
    - Final output layer is a single linear layer without any activation
- **Discussion**
    - The initial hidden layer expanded the number of nodes for exploring rich feature space
    - ReLU and Batchnorm layer are used for efficient training
    - Final output layer doesn't use any activation because only the relative scale matters to select an action

### Buffer

- **Description**
    - Replay buffer used in a DQN algorithm. It is a standard deque with a sampling function
- **Interface**
    - Initialization
        - sz: Replay buffer size
        - pf: Probability function for sampling
    - put(obj): Stored obj in a replay buffer
    - sample(n): Sample n objects from a replay buffer
    - update_pf(pf): Update the probability function pf
- **Discussion**
    - pf can be used for prioritized experience replay
    - It was used to explore the potential benefit of the skew in experience sampling, but was default to uniform sampling

### Training

- **Description**
    - Standard DQN training algorithm
- **Detail**
    1. Initialize a model and hyperparameters
    2. The model (agent) interacts with the banana Unity environment to collect tuple (state, action, reward, next-state)
    3. The collected tuples (experiences) are stored in a replay buffer
    4. The experience is randomly batch-sampled every 4 steps("train_step") and used to train the NN model
    5. 2-4 are repeated until the average score(return) reaches the objective score (13)
- **Hyperparameters**

| name | value | description |
|---|---|---|
| n_episodes | 3000 | Max number of episodes to explore |
| max_t | 3000 | Max number of time steps of each episode |
| bn | 64 | Training batch size |
| gamma | 0.99 | Return discounting factor |

| name | value | description |
|---|---|---|
| eps, eps_decay, eps_min | 1.0, 0.999, 0.05 | Random action exploration parameters |
| tau, tau_decay, tau_min | 0.001, 0.9999, 0.001 | Target network soft-update parameters |
| train_step | 4 | Frequency of the training |

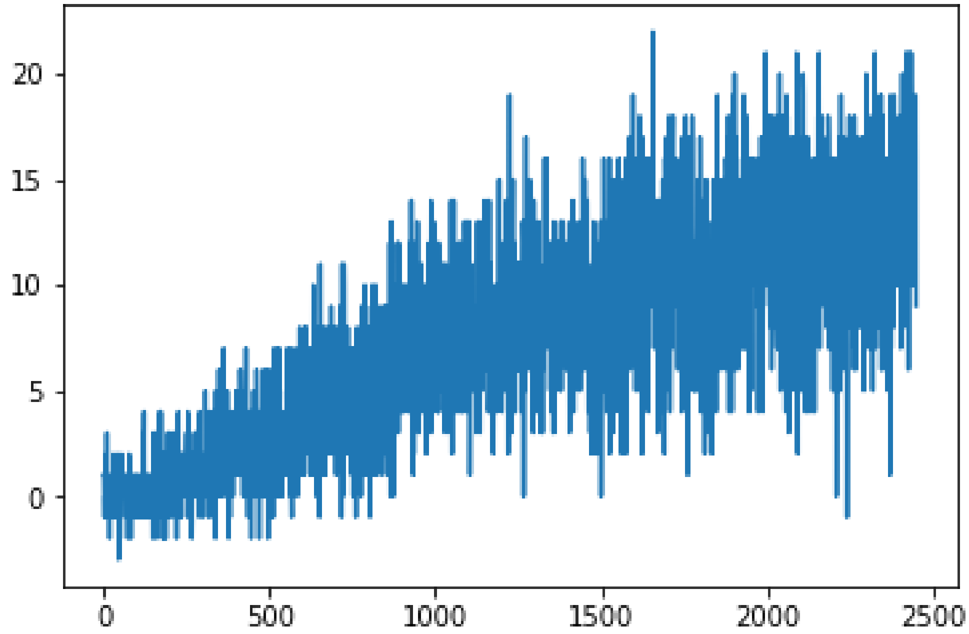
#### 1.0.4 Results

The following 2 charts show the raw and average scores of the agent as it trains. (X: # episodes, Y: Score of each episode)

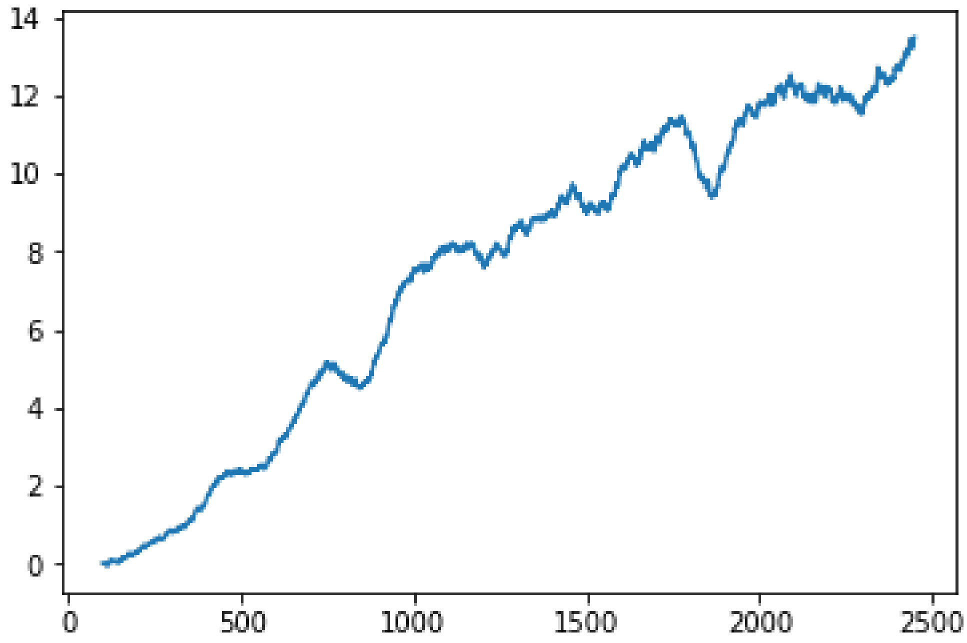
A few interesting observations are

- It required ~2500 episodes to achieve the average score of 13 over 100 last episodes
- The score variance increases as it trains. This means that the agent is specialised and not robust against different conditions
- The average score generally increases, but the improvement slope decreases. This is natural because the agent can learn a few simple tricks to get some scores, but it is difficult to learn a long-running strategy to get high scores.

```
[15]: [<matplotlib.lines.Line2D at 0x7fa6de208240>]
```



```
[16]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa6de228898>
```



### 1.0.5 Future work

1. Regardless of the shown scores, the actual agent's actual behavior can be different from ideal. The model and training can be debugged to find out typical behavioral issues and can be improved by tweaking an algorithm. The following general principles can be applied for the improvement
   - Reward reshaping to give more guidance to the agent
   - Encode characteristics of observation into the model architecture to better utilize it
2. General improvements of RL model can be tried and compared against the simple DQN algorithm used in this notebook
3. Replay buffer can be improved with a careful design of dynamic sampling pdf. Some of the information to be used are
   - Difference of return expections from current to next step (Typical Experience Prioritisation)
   - Experience sample's distance from the learning signal(reward)