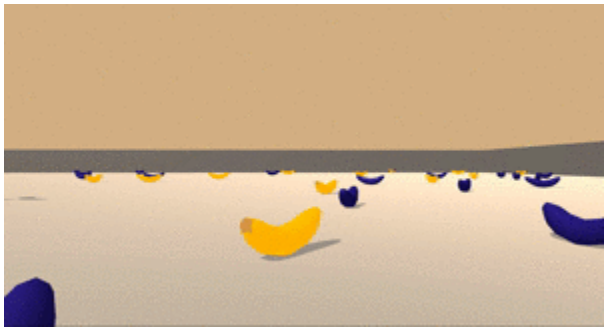
Project Report

DRLND Project 1: **Navigation**

Problem Description

The goal is to train an agent to navigate (and collect bananas!) in a large, square world.



The problem is framed as a rich game environment on the Unity platform. The environment has been designed by the Udacity Deep RL Nanodegree (DRLND) program team, and is provided as part of the starting materials. Interacting with the environment is through the use of a Python API exposed by the Unity Machine Learning Agents (ML-Agents) Toolkit.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to:

- "0" - move forward
- "1" - move backward
- "2" - turn left
- "3" - turn right

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of the agent is to collect as many yellow bananas as possible while avoiding blue bananas.

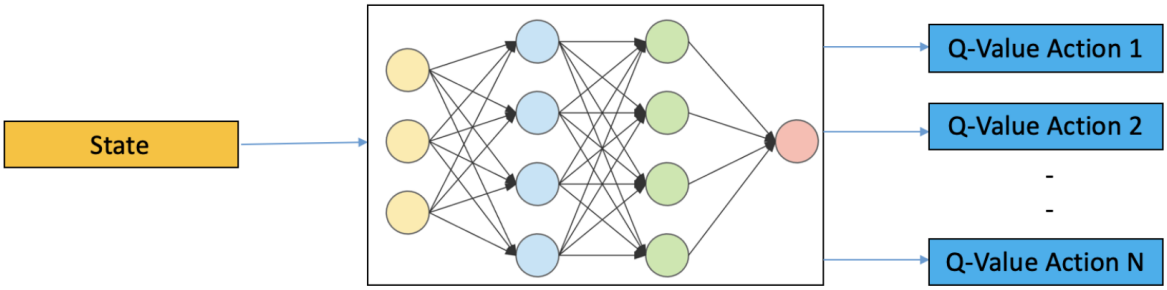
The task is episodic, and in order to solve the environment, your agent must get an average score of +13 over 100 consecutive episodes.

Learning Algorithm

To train an agent to navigate and achieve high rewards in the Bananas environment, we resort to Reinforcement Learning using a Deep Neural Network. The Deep Q-Learning (DQN) solution used by Mnih et al. [1] is well-suited for this task, given the Discrete nature of the action space used by the environment.

The observation space provides us with direct measurements of sensors in the environment such as agent's velocity and ray-based perception. Effectively, this does the work of the convolutional layers used in the DQN architecture, and we can treat the 37 dimensions of observations to directly feed the dense layers in our Neural Network.

We can, therefore, use a simplified Neural Network with just fully-connected layers:



Model

- The input to the neural network consists of a 37 feature vector provided by the environment.
- The first hidden layer is fully-connected and consists of 128 rectifier units.
- The second hidden layer is fully-connected and consists of 64 rectifier units.
- The output layer is a fully-connected linear layer with a single output for each valid action.

The agent selects and executes actions according to an epsilon-greedy policy based on Q. Reinforcement Learning, however, is notoriously unstable when neural networks are used to represent action values. We address these instabilities by using two key features:

- **Experience Replay**: a rolling history of past data via a re-play pool. Using the replay pool the behavior distribution is averaged out over many of its previous states smoothing out learning and avoiding oscillations. This has the advantage that each step update is potentially used in many weight updates.

- **Fixed Q-Targets**: use a separate network for generating the targets in the Q-learning update. More precisely, every C updates we clone the network Q to obtain a target network $\widehat{Q}$ and use $\widehat{Q}$ for generating the Q-learning targets for the following C updates to Q. This modification makes the algorithm more stable compared to standard online Q-learning.

The full algorithm for training the deep Q-network is as follows:

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Training Results

The implementation for solving this project used Python 3 and PyTorch. The code is shared on GitHub: https://github.com/aquanta-aagyaa/drlnd-p1-navigation

The important files are:

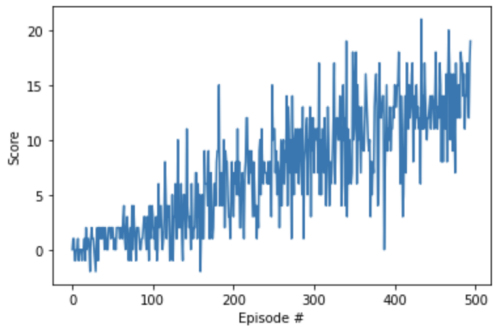
1. Navigation.ipynb - Jupyter notebook to walk through the training process
2. dqn_agent.py - The DQN agent and replay buffer implementation
3. model.py - implementation of the Q-network (DNN) class using PyTorch

Hyperparameters

The following hyperparameters allow the model to achieve the desired goal, of an average score of +13.0 over the last 100 episodes, within a reasonable amount of time -- 495 episodes!

- 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
- UPDATE_EVERY: 4 # how often to update the network

```
Episode 100     Average Score: 1.05
Episode 200     Average Score: 3.87
Episode 300     Average Score: 7.55
Episode 400     Average Score: 10.14
Episode 495     Average Score: 13.03
Environment solved in 495 episodes!     Average Score: 13.03
```



The model weights of the successful agent are saved to: **trained_model/model.pt**

Ideas for Future Work

Some ideas we can pursue to improve the performance of our agent:

Double Q-Learning
Conventional Q-learning is affected by an overestimation bias, and this can harm learning. Double Q-Learning requires using two separate function approximators that must agree on the best action. This simple modification keeps q-values in check, preventing them from exploding in early stages of learning, or fluctuating later on.

Prioritized Experience Replay
DQN samples uniformly from the replay buffer. We might want to sample more frequently those transitions from which there is much to learn.

Multi-step learning
Q-learning accumulates a single reward and then uses the greedy action at the next step to bootstrap. We might try to use forward-view *multi-step* targets, which often lead to faster learning.

References

1. Mnih, Volodymyr, et al. “Human-level control through deep reinforcement learning.” Nature518.7540 (2015): 529. https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf