DQN for Unity Banana collection game environment

1. Introduction

Reinforcement learning is rooted in psychological and neurological perspective of animal behavior. In many cases the environment can be too complex that the number of states is not enumerable to a finite set. Animals and humans learn the complex environment around them through a harmonious combination of reinforcement learning and hierarchical sensory processing systems. DQN was one of the ground-breaking use of Deep neural networks in reinforcement learning to learn Atari games. The Deep Q-Network agent surpassed all previous algorithms and achieved levels comparable to professional human players.

This implementation to solve the Unit Environment Bananas, is based on the DQN algorithm used by Google DeepMind, to solve Atari games. The software agent (DQN) interacts with the environment thru a sequence of observations (37) actions (4) and rewards. The goal of the agent is to select an action that maximizes the cumulative future reward. This implementation uses a MLP (Multi-layer perceptron) network to approximate the optimal action-value function fig 1.

2. MDPs, Q-Learning & DQN:

Games with Discrete state action space is modeled using Markov Decision process. An MDP has a set of N states { s1, s2, s3…Sn} and for each state there is a set of M possible actions { a1, a2, …am}. Each action is associated with a reward (or penalty). The optimal policy selects the best action for each state so that the total reward is maximized. In most games, the total reward/the winner is decided only at the end of the game, even though, each intermediate action affects the final outcome. It is difficult to determine which action or subset of actions resulted in the winning a game. This is known as the credit-assignment problem.

Bellman found that the optimal action-value function obeys an important identity known as the Bellman equation. This is based on the intuition that, if at state s after taking an action a, the agent reaches a new state s' and if the agent knows that in at state s', action a' maximizes the final score. So the optimal value function is given by Eqn 1.

$$Q(s,a) = \underbrace{r}_{\text{immediate reward}} + \underbrace{\overset{\text{discount factor = 0.90}}{\gamma} max_{a'} Q(s',a')}_{\text{future reward}}$$

Eqn. 1

$r \rightarrow$ the reward for taking actions a in state s

$\gamma \rightarrow$ Discount factor for future returns

$s' \rightarrow$ Next state on taking action a.

$a' \rightarrow$ all actions from state s'

Most reinforcement learning algorithms use Bellman equation as an iterative update eqn. 2, to get the optimal value function. This is called the Q-value function.

$$Q_{t+1} = E[r + \gamma \max \ Q_t(s', a')\ ]$$

Eqn. 2

It converges as $i \rightarrow \infty$. This is impractical in most cases as the number of state action pairs can be extremely large. Instead a linear approximator is usually used to estimate the action value function $Q(s, a;\ \theta) = Q^*(s, a)$. In DQN and Deep Neural Network is used to find the optimal Q-Value function. A NN to with weights θ is referred to as Q-Network. The network is trained to optimize the parameters θ. This is done by adjusting the parameters at step up to reduce the mean square error in Bellman equation, where the optimal target values $r + \ \gamma \max Q^*(s', a')$ are substituted with approximate target values $r + \gamma \max Q(s', a', \theta^-)$ using parameters θ^-, from the previous iteration. The result is a sequence of loss functions Eqn 3.

$$L_i(\theta_i) = E_{s_{a_r}} = E\left[\left(y - Q(s, a;\ \theta_i)\right)^2\right] + E_{s_{a_r}}[V_s, [y]]$$

Eqn. 3

3. DQN Algorithm:

There are two modifications to the online Q-Learning other than using the Deep Neural Network, in DQN. One is the use of replay buffer and the other use of two networks.

1) ***Replay Buffer***
DQN uses experience replay to make the learning algorithm more stable. The agent's experience at each time step t is stored as a tuple e_i = (s, a, s', r) and these tuples are added to a buffer B = $\{ e_1, e_2, \dots \dots \dots e_n\}$. While learning (in the inner loop) data is sampled in mini batches from this buffer to train the network. The samples are picked at random. Correlation between consecutive samples affects the stability of the learning process. Randomizing reduces the correlation between consecutive samples thus reducing the variance of updates.
2) ***Two Networks***
DQN uses a target network and local network to improve the stability of the algorithm. This is done as follows. After every P updates the local network Q is cloned to create a target network Q'. Now use Q' to generate the next P Q-learning targets y_j to update the local network Q. This makes the algorithm more stable compared to online Q-learning.

3.1. ∈ − Greedy Learning

The agent uses and ε greedy policy to select and execute an action from the network Q. As we take the samples from the environment, it is not a good idea to just assign new values on top of existing values. This can create lot of instability. What is usually done in practice is to update Q(s,a) with approximations using a blend of old and new values using a learning rate. The improved equation is shown in Eqn 4.

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

Eqn. 4

The Network optimizes the action taken based on future rewards. This can be referenced back to Eqn 1 as below.

$$\underbrace{Q(s,a)}_{\text{TD target}} = r(s,a) + \gamma \underbrace{Q(s', \underbrace{argmax_a Q(s',a))}_{\text{DQN Network choose action for next state}}}_{\text{Target network calculates the Q value of taking that action at that state}}$$

Eqn. 5

3.2. The Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        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 $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

4. Architecture

The Architecture of the DQN implementation for Unity Environment : Bananas has five main components

1) Agent   2) The Network   3) Training 4) Saved Model

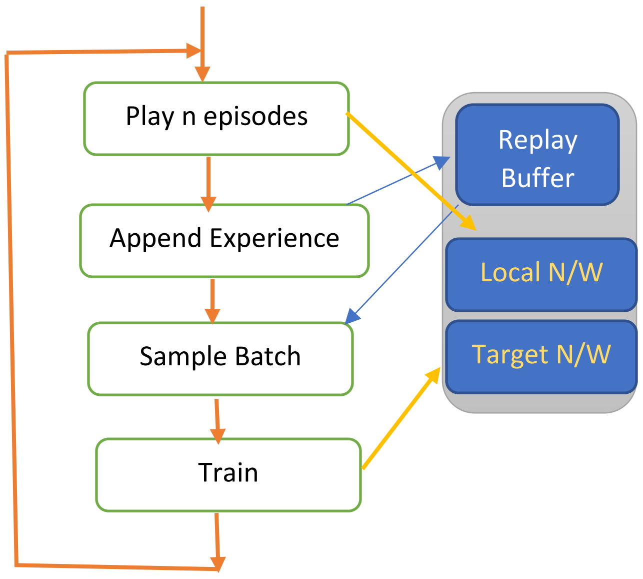

Fig. 1

2.1 Agent

The Agent has three components, two networks: local network to train and the other, target network to verify and a Replay buffer to save episodes of training. Data from the replay buffer is sampled to train the actor network while the target network is updated after specific time steps.

a. The Network

The Unity Banana environment has 37 observations for each state and 4 possible actions (see Environment below). The 37 observations form the input of a MLP network with 2 hidden layers and 4 output. The hidden layers have 128 neurons each. See Fig 2.

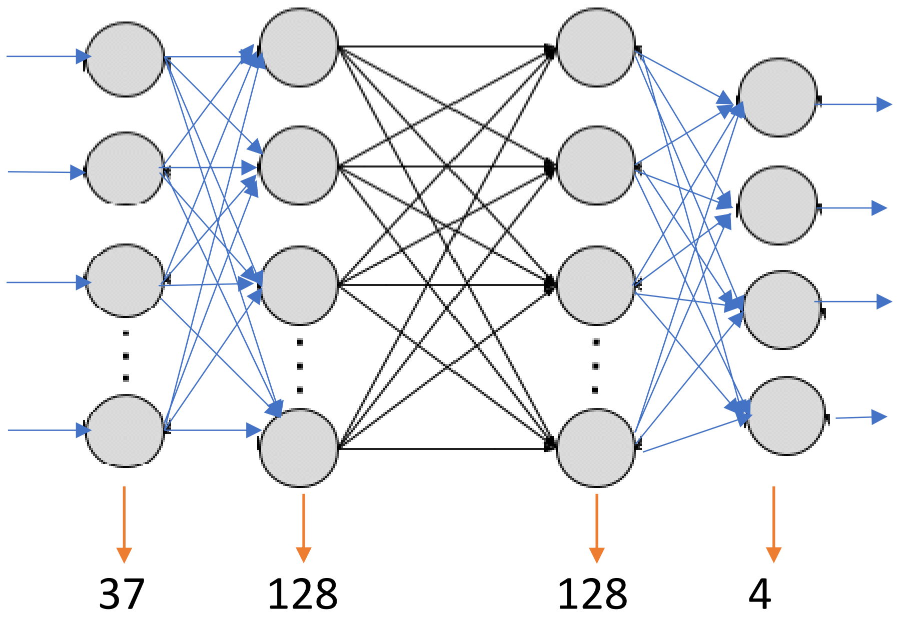

Fig 2

b. Training & Testing

The Agent is trained for 1800 episodes. With this network an average score of 13+ over 100 episodes was achieved within 600 episodes consistently. The training can be run by executing. The parameter used for the model are

Episode count :

Maximum : 1800

Epsilon :

```
starting value of epsilon, for epsilon-greedy action selection : 1.0
minimum value : 0.01
 multiplicative factor (per episode) for decreasing epsilon
```

- Python Bananas.py

c. Saved Model execution.

The model is saved after 1800 episodes. This may not consistently perform and show a score of 13+ each time. But will gain a reasonable score in most runs and 13+ in few cases. To run saved model

- Python run_model.py

The code runs the game 3 times using saved model

5. Environment

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 your agent is to collect as many yellow bananas as possible while avoiding blue bananas. 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.

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

6. Code

There are 4 files

1) Bananas.py
   This file has the code to train the networks. To run training execute this script.
2) dqn_agent.py

The Agent and replay buffer is defined in this file. The Agent class instantiates two networks: local and target. The target network is the network against which the actions of local network is compared. Replay buffer is used to store episodes of game play. Data is randomly retrieved from the replay buffer to train the networks.

3) model.py
   Model.py has the MLP definition: both forward and backward pass.
4) run_model.py
   run_model.py if executed run the saved model. Make sure the Banana application path is valid before running this.

7. Improvements

There are several improvements to the original algorithm that is discussed in literature a few examples that are well suited for simple incremental improvements are discussed below.

1. Prioritized Experience Replay
   Prioritized experience replay is based on the idea that the agent can learn more effectively from some episodes of the game is of more value than other. So a priority is assigned to data in the replay buffer for retrieval and learning.

   The central component of prioritized replay is the criterion by which the priority of each transition is measured. One criterion would be the amount the RL agent can learn from a transition in its current state (expected learning progress). While this measure is not directly accessible, a reasonable proxy is the magnitude of a transition's TD error δ, which indicates how 'surprising' or unexpected the transition is: specifically, how far the value is from its next-step bootstrap estimate.

   The algorithm stores the last encountered TD error along with each transition in the replay memory. The transition with the largest absolute TD error is replayed from the memory. A Q-learning update is applied to this transition, which updates the weights in proportion to the TD error. New transitions arrive without a known TD-error, so we put them at maximal priority in order to guarantee that all experience is seen at least once.

   Since there are thousands of transitions an optimal data structure is needed to store and retrieve those with higher priority. A heap implemented on an array can be used to get samples with highest priority faster.

   So tan improvement would be to add TD error estimated along with transitions, make the queue a priority queue and make sampling guided by priority.

2. Double DQN.
   This improves estimation of action value. The max operator in standard Q-learning and DQN, uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent

this, Double DQN decouples the selection from the evaluation. This is the idea behind Double Q-learning

8. References

1. Human-level control through deep reinforcement learning. The nature Volodymyr Mnih1 et. al
2. Reinforcement Learning An Introduction Richard S. Sutton & Andrew G. Barto
3. Deep reinforcement Learning hand-On. Maxim Lapan.
4. Hierarchical Object detection with Reinforcement Learning
5. Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed Q–targets
6. Artificial Intelligence Leonardo Araujo dos Santos
7. Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms. Nikhil Buduma et. al