Deep Reinforcement Learning - Navigation

Project Overview

The environment for this project is a square world filled with yellow and blue bananas. The goal of the agent is to collect yellow bananas while avoiding blue bananas. The environment thus gives the agent a reward of +1 whenever a yellow banana is collected, and -1 for a blue banana.

The state space is continuous and has 37 dimensions, which include the agent's velocity and some representation of the objects in front of the agent. The action space is discrete and has 4 possible values: forward, backward, left and right.

The environment is considered solved when the agent gets an average reward of at least +13 over 100 consecutive episodes.

Learning Algorithm

At a high-level, the learning algorithm which trains the agent consists of two components, a Q-Learning agent and a neural network which is used as a function approximator for the Q-value function.

Q-Learning Agent

The agent is defined in the *Agent* class inside Submission.ipynb. At a high-level, the agent trains the same way any Q-Learning agent does:

- Plays multiple episodes for gathering information.
- At the beginning of each episode, starts with an initial state obtained by resetting the environment.
- For a fixed number of iterations (or until the end of the episode):
    - Obtains an action to execute using an epsilon-greedy policy for the current state.
    - Executes the action on the environment to obtain the next state and reward.
    - Learns in some way from the next state and reward obtained by executing the current action in the current state.
    - Sets the current state to be the next state.

The above learning loop is implemented within the *dqn*() function inside the notebook.

There are some points to be noted about the above learning algorithm:

- The logic which obtains the action to execute in the current state (the *act*() function inside the *Agent* class) uses the neural network for obtaining the Q-values for every possible action in the current state, and then chooses the best action with probability (1 - epsilon).
- The learning from a (state, action, reward, next_state) tuple is also performed by using this information to update the neural network weights (the *learn*() function inside the *Agent* class). For this, we need to construct a loss function from the given information:
    - Since the neural network outputs real values (Q-values), the loss is a simple squared error loss: (target - prediction)**2.
    - The prediction is simply the output of the network with the given state as input, indexed by the given action.
    - The target, whose true value is unknown, is approximated as (reward + gamma*next_state_value).
    - next_state_value is the best value which can be obtained starting from next_state, and is approximated by passing next_state through the same neural network, and then choosing the maximum value from among all the outputs.
- For breaking correlations between sequences of consecutive tuples, the learning is not performed the moment a (state, action, reward, next_state) tuple is generated. Instead, the tuple is stored inside a buffer known as a replay buffer (the *ReplayBuffer* class inside the notebook). Every few steps, a batch of tuples is randomly sampled from the buffer and learning is performed over the batch. This logic is in the *step*() function inside the *Agent* class.
- Since the neural network is also used for computing the value of the next state (to be used within the target of the loss function), a separate copy of the neural network is maintained specifically for this purpose. The weights of this target network are a lagging copy of the live network weights, which means they are kept frozen for a few steps and then synchronized periodically with the live network weights (the *soft_update*() function inside the *Agent* class). This helps in keeping the target values stable, which in turn stabilizes the training process.

Neural Network

The neural network is used as a function approximator for the Q-value function. Instead of taking in a (state, action) pair and emitting a single scalar Q-value, the network in this case takes in the state as input and emits Q-values for every possible action in the state.

The input for the network is thus a numerical vector representing the state. Since the environment in this case itself returns a 37-dimensional vector in the continuous space, it can be used directly as the network input.

The output layer of the network has 4 units, one for each of the possible actions. No activation is applied on the outputs since this is a real-valued regression problem.

The network also has two hidden layers, with 100 and 200 units respectively. A relu activation is applied to the outputs of the hidden layers.

The code for the network is inside the *QNetwork* class within the notebook.

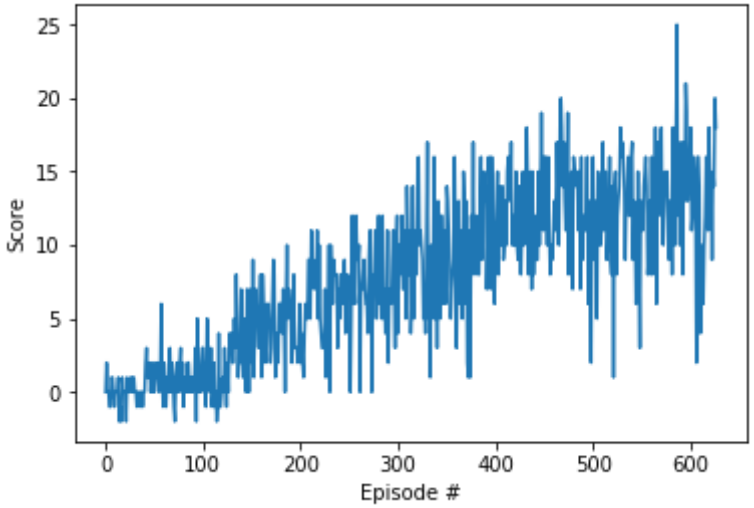
Hyperparameters

The learning was performed using the following hyperparameters:

- A replay buffer size of 100,000.
- A batch size of 64 while sampling from the buffer.
- A gamma (discount value) of 0.99
- A tau of 0.001. This is used for updating the target network weights, by interpolating the existing values with the live network weights.
- A learning rate of 0.0005 for the Adam optimizer used for training.
- Batches were sampled from the replay buffer every 4 steps.
- 2000 episodes were played by the agent.
- A maximum of 1000 steps were executed within each episode.
- Epsilon (used by the agent for computing epsilon-greedy policies) started at 1.0 and decayed by a factor of 0.995 with every episode, until a threshold of 0.01 was hit.

Results

The agent was able to get an average return of 13 (over 100 episodes) after playing 527 episodes. The following is a plot of the returns over time as training progressed:



Improvements

The following are a couple of ideas for improving the agent's performance:

- Use double DQN for addressing the fact that a DQN tends to overestimate Q-values due to its greedy nature (it computes the target by using a next state value which is the maximum of all possible values emitted by the Q-Network).

- Use prioritized experience replay for sampling from the replay buffer using weights. The weights for the entries in the buffer can be proportional to their TD-errors (a large TD-error indicates that the agent still has scope to learn from the entry).