Project 1 Report

Solution

This project solved the Unity ML-Agents Banana.exe/app environment using a deep reinforcement learning model.

DQN creates a buffer of prior experience tuples (state, action, reward, next_state, done) to be used during training. These prior experience examples are sampled randomly when learning updates are performed.

The DQN used in this implementation is the simple DQN with two networks: one is local, and one is target. The first one to retrieve Q values while the second one includes all updates in the training. After updates then synchronize these two. The purpose is to fix the Q-value targets temporarily so we don't have a moving target to chase.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i)\right)^2$$

Where

D is the dataset containing the experience buffer,
r is the reward for the experience example selected,
gamma is the discount rate,
θ i represents the weights of the "local" network and timestep i
θ i− represents the weights from the target network at timestep i.
This target network gets updated more smoothly/slowly and is therefore a more stable estimate of action-state values.
s' means next state and a' means next action

The learning algorithm trained here uses an epsilon-greedy method for selecting new actions while being trained. The epsilon represents the probability of choosing an action at random instead of following what is currently expected to be the "best" action in the given state

Implementation

- Navigation.ipynb - the main code to run which loops through episodes and updates the epsilon value.
- qmodel.py - defines the Q networks that are used by the learning agent
- qagent.py - defines how the learning agent acts when asked to provide an action, learn from a time step, etc.

The QNetwork class defined in model.py allows the user to create 2 fully-connected hidden layers.

Hyperparameters

DQN Hyper Parameters

- n_episodes (int): maximum number of training epi5sodes
- eps_start (float): starting value of epsilon, for epsilon-greedy action selection
- eps_end (float): minimum value of epsilon
- eps_decay_val (float): multiplicative factor (per episode) for decreasing epsilon
- eps_ratio (float): control the exponentially rate of decay

The values used are

scores = dqn(n_episodes=1000, eps_start=1.0, eps_end=0.005, eps_decay_val=0.995, eps_ratio_power=0.5)

DQN Agent Hyper Parameters

- 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 local and target networks
- WD = 0.00 # Weight Decay/Regularization for neural network

Neural Network Architecture

The Q network architecture must have 37 inputs (state space per Unity program) and 4 outputs (values for each available action in the environment). More than two hidden layers can also have good result which I did not try many times. With 2 hidden layers should be minmium to represent an arbitrary decision boundary. Using too few neurons in the hidden layers will result in underfitting, Using too many neurons in the hidden layers may result in overfitting. The selection of layers and neurons in between the input and output layers is part of the exploration with trial and error to find a solution to this problem.

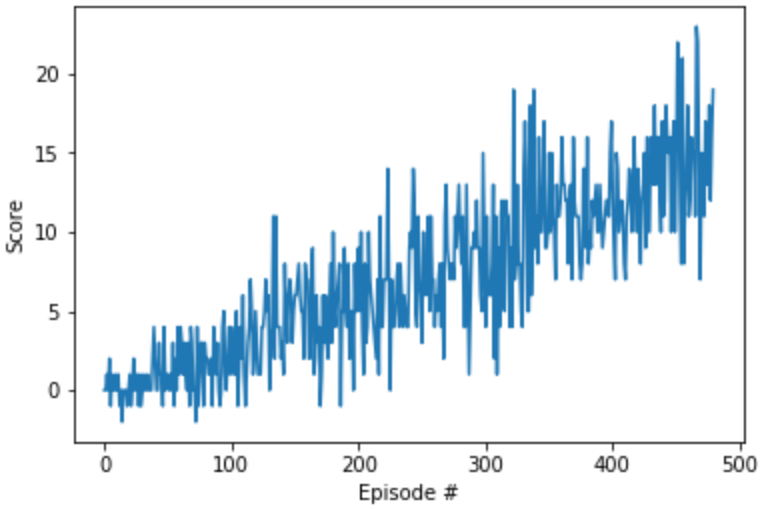
The QNetwork class defined in model.py create 2 fully connected hidden layers layers. The neural networks has the following structure.

- Input: 37 dimension vector
- Hidden Layer 1
    - Linear with 150 nodes
    - Relu
- Hidden Layer 2
    - Linear with 75 nodes
    - Relu
- Output Layer
    - Linear with 4 nodes
- Optimization algorithm
    - Adam
- Learning rate
    - 5e-4

Final Results

This implementation which solved the environment (gaining a score of >13 averaged over 100 consecutive episodes) after 480 episodes. The console output and a plot of the scores are displayed below.

Environment solved in 476 episodes! Average Score: 13.05



Future Improvements

Here's a list of optimizations that can be applied to the project:

Prioritized Experience Replay

This has not been implemented but could have a significant benefit in reducing the number of episodes required to solve this environment.

In this project, the navigation agent samples uniformly at random from D when performing updates. This approach is limited because the memory buffer does not differentiate important transitions and always overwrites with recent transitions owing to the finite memory size N. Similarly, the uniform sampling gives equal importance to all transitions in the replay memory. Using a clever prioritization scheme to label the experiences in replay memory, learning can be carried out much faster and more effectively.

Dueling architecture

The advantage of the dueling architecture lies partly in its ability to learn the state-value function efficiently. With every update of the Q values in the dueling architecture, the value stream V is updated – this contrasts with the updates in a single-stream architecture where only the value for one of the actions is updated, the values for all other actions remain untouched. This more frequent updating of the value stream in our approach allocates more resources to V ,thus allows for better approximation of the state values