Navigation Project Report

Description of implementation:

For this project, a computerized “agent” is trained to play the Unity Bananas game using the Deep Q-Network concept. The trainable parameters are contained within a 3-layer neural network, which is implemented in pytorch. This network serves as a function approximator, returning action values for each action, given the input state. The states and actions are provided by the Unity environment, this being the Bananas game environment in this case. After training, the agent should be able to maximize its score by selecting actions with the highest value, as determined by the trained network. In this way, the network becomes part of the implementation for the optimal policy, or at least, a close approximation to the optimal policy. The agent employs two important techniques that were recently developed in the field of deep reinforcement learning, these being “Experience Replay” and “Fixed Q Targets”. Together, these techniques help to stabilize the training process.

The object of the Banana game is to acquire as many yellow bananas as possible while avoiding blue bananas. Generally, the agent figures to maximize its score by selecting the action with the highest value for any given input state. However, early in the training the action values will not be accurate. The trainable parameters, or weights, are initialized randomly, so the agent must learn “from scratch” how to solve the game. Since the agent knows very little at the start of training, it benefits greatly from exploration. Exploration is implemented using the “epsilon greedy” strategy, in which, with probability epsilon, the agent selects an action at random, or selects the action with the highest value with probability 1-epsilon. As the agent learns more about the environment it is able to make better decisions based on its knowledge, so exploration is less advantageous and “exploitation” become the right thing to do. Thus, epsilon is set to one at the beginning of training, meaning the agent always selects actions at random. As training proceeds, epsilon “decays”, meaning that it gradually decreases in value so that the agent is much more inclined to select the action with the highest value, which by this time should be close to the optimal policy. Epsilon does not necessarily have to decay to zero, a minimum value for epsilon can be set (eps_end) to allow for some exploration even in a highly trained agent.

Hyperparameters for Navigation Notebook:

n_episodes = 2000 (maximum number of episodes)

max_t = 1000 (maximum time steps)

eps_start = 1.0 (starting epsilon value)

eps_end = 0.0 (ending epsilon value)

eps_decay = 0.995 (controls epsilon decay rate)

Hyperparameters for dqn_agent:

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)

Hyperparameters for model (pytorch network):

action_size = 4 (action size is fixed by the environment)
state_size = 37 (state size is fixed by the environment)
Number of nodes in first layer set to 64
Number of nodes in second layer set to 32
Number of nodes in output layer set to 4 (must match action_size)

For this implementation the hyperparameters for the Navigation Notebook and dqn_agent were set at the same values used in the Deep Q-Networks lesson workspace for OpenAI Gym's LunarLander-v2 environment. These values worked very well for the Unity Banana game. For the pytorch model, some experimentation with the hyperparameters was done. This involved changing the number of nodes in the first and second fully connected layers (the number of output nodes is fixed by the number of actions). Experimental values of 32, 64 and 120 were tried for each layer. Changing these values made only a slight difference in performance of the learning algorithm, the variance being in the number of episodes required to “solve” the game. The game is considered solved when the agent is able to achieve and average score of +13 over 100 consecutive episodes. However, for this testing a slighting higher average score of 14 was required. The best results were obtained using 64 nodes for the first layer and 32 for the second layer. With this configuration, the game was solved in about 540 episodes.

The algorithm worked very well as is, meeting the higher standard of 14 for average score on 100 consecutive episodes rather easily. However, it might be possible to achieve even higher scores with some improvements. Two such improvements were suggested in the lessons, these being prioritized experience replay and dueling DQN. The use of a multitude of such techniques (the “rainbow” strategy) would also figure to improve the scores.