Samantha Wood
October 27, 2018

Udacity Deep Reinforcement Nanodegree
Project #1: Navigation

1. Learning algorithm

Neural network architecture. To train the agent, I used a Deep Q-Network (DQN) learning algorithm. The DQN algorithm uses a neural network to determine the optimal action-value function given an observation of the environment's state (more details in the DQN subsection below). The neural network model in this implementation takes the environment's state as input and applies four fully connected layers. Activation of the first three fully-connected layers is subjected to ReLU before submitting to the subsequent layer. The number of units in each fully connected layer is 48, 72, 96, and 4, respectively. My strategy for determining the number of units was to gradually expand the size of the hidden layers. The number of units in the final layer is equal to the number of discrete actions in the action space of the environment.

Note that this architecture differs substantially from the neural network implemented in Mnih et al. (2015). The neural network in Mnih et al. (2015) utilized three convolutional layers (with ReLU) followed by two fully connected layers. However, for the navigation environment, the state space has a singular dimension (an array of 37 features). Accordingly, there are no two-dimensional spatial relationships that can be exploited through convolution. While convolutional layers can be powerful, they are not appropriate for the navigation environment in which the state space is one-dimensional.

DQN learning algorithm. To train the neural network, I used a slightly modified DQN learning algorithm. The first notable feature of the DQN algorithm is *experience replay*. Experience replay has the advantage of making more efficient use of observed experiences. Without experience replay, some experiences will be under-observed because certain states of the environment may be rare and certain actions may be too costly to perform often. Another advantage of experience replay is that sampling the observed experiences at random helps to decorrelate actions and states, preventing the q-networks from oscillating or diverging catastrophically.

To use experience replay, I stored each experience tuple (state, action, reward, next state, done) in a replay buffer. During training, as the agent took a step, the corresponding experience was immediately stored in the replay buffer (up to a maximum of 10^5 experiences). The agent did not learn (i.e., update its q-networks) until the replay buffer was sufficiently large (at least 64 experiences stored), and once that threshold was reached, the agent only learned every four steps. During learning, the algorithm randomly selects a batch of experiences (the batch size is 64, which is why 64 experiences must be stored in the buffer prior to any learning) according to a uniform random distribution. Then, the q-networks' weights are updated based on this batch of experience tuples.

A second notable feature of the DQN algorithm is *fixed q-targets*. The technique of fixed q-targets involves maintaining two q-networks instead of one in order to decouple the state from the action for a more stable learning environment. The "local" q-network is updated with every learning step, while the "target" q-network is decoupled from the local q-network by being updated more slowly and infrequently.

The process for updating the weights of the neural network based on the experience tuples is a form of temporal difference learning called Q-learning. As in traditional neural network models, the weights are updated through backpropagation using an optimizer to change the weights in the direction of the gradient that minimizes the loss function. The learning algorithm used here is

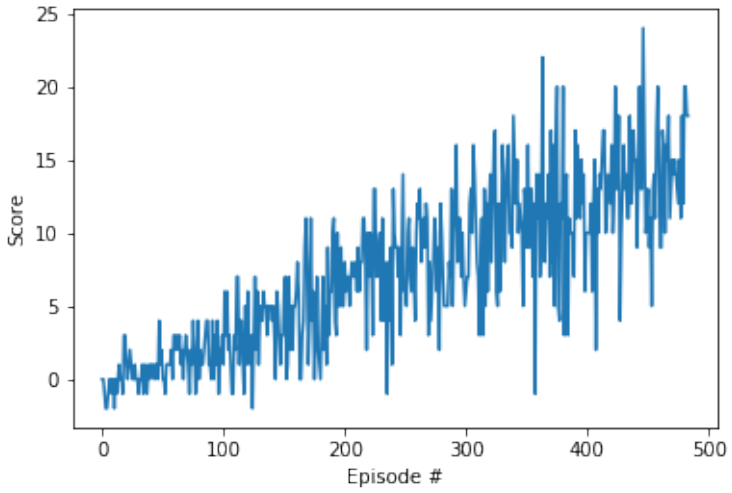
Adam, which is a variation of stochastic gradient descent that uses momentum (learning rate = 0.0005). The loss function is mean squared error.

The error is computed by taking the difference of the temporal difference target value and the current value assigned to the state in the neural network. This error is also referred to as the temporal difference error. The local q-network is the basic q-network that controls the agent. This q-network is updated with each learning step using backpropagation. The target q-network updates more slowly. At the end of each learning batch, a soft update function is applied to the target q-network. The soft update function computes a weighted average between the local (weight = 0.001) and target (weight = 0.999) q-network weights. Thus, the target q-network weights are very slowly moved closer to the local q-network weights.

The temporal difference target is equal to the observed reward plus the discount rate (0.99) times the maximum q-value of the next state, as computed by the *target* q-network. The current value is equal to the q-value of the current state as computed by the *local* q-network.

2. Plot of rewards

The number of episodes need to solve the task was 384. As illustrated in the plot of rewards below, the agent's performance steadily improved over the course of training.



3. Ideas for future work

Storing past actions in the state vector. One of the other techniques employed by Mnih et al. (2015) was storing past actions as part of the state vector. This technique allows the q-network to take advantage of temporal relationships in the environment. Temporal dynamics could be important in the navigation task because the agent's past moves impact the available bananas to collect, and the agent's perception of the objects around the forward direction changes with each step. In future work, I could adapt the code by saving the last state observation and appending it onto the next state observation. While the environment would provide an observation of the current state that has 37 features, the adapted code would append the prior state and provide the agent with a 74-feature state.

Double DQN. In the early stages of learning, the weights of the q-networks are still being adapted slowly away from their random initializations. When the learning algorithm estimates the temporal difference target, choosing the maximum values among a set of noisy numbers (from the relatively untrained target q-network) will lead to overestimating the temporal difference target

value. Double DQN addresses this issue by using the local q-network to select the best performing action and the target q-network to evaluate the action. The target q-network is updated much slower than the local q-network, so it can act as a separate function approximator. In order to get high temporal difference target values, the local and target q-networks must both estimate a high action value for the same action in the next state.

Prioritized experience replay. The baseline DQN algorithm randomly samples from the replay buffer; however, there are more efficient ways to sample the experiences than choosing blindly. Prioritized experience replay assigns priority to the experiences in the replay buffer according to the magnitude of the temporal difference error produced by the experience. Intuitively, experiences that have larger errors are experiences that the model handles poorly and could learn the most from. In order to employ prioritized experience replay, the temporal difference error needs to be stored as part of the experience tuples in the replay buffer (i.e., <state, action, reward, next state, done, *temporal difference error*>). When sampling from the buffer, the sampling probability for any given experience is the temporal difference error for that experience divided by the sum of all of the temporal difference errors in the buffer. Once an experience is picked, the agent learns from the experience, which produces a new temporal difference error. The experience is returned to the buffer with the updated error. To avoid overfitting to a subset that gets replayed repeatedly (i.e., certain experiences associated with higher error that continue to be selected for learning), the priority of the experiences can be raised to a power between zero (pure uniform random) and one (pure priority-based sampling) in order to skew the sampling distribution closer to uniform random.

An added complication when implementing prioritized experience replay is that the update rule must be adapted because prioritized experience replay biases the q-values that the model learns. The update rule should be adapted by multiplying the standard update rule by the importance sampling rate.

Dueling DQN. One disadvantage of the DQN algorithm is that the model estimates the value of any given state by estimating the full set of action values associated with the state. However, the value of most states is fairly consistent across actions, so it would generally be more accurate to estimate the value of the states directly. Dueling DQN addresses this weakness by using two dueling streams for the penultimate layer of the model. One stream estimates the state values, while the other stream estimates the advantage value of each action in the state. These two streams both connect to the final output layer to provide a q-value estimate.

Pixels to actions. Finally, biological organisms receive state information through their sensory organs. While the environment in this task provided the agent with a 37-feature representation of the environment state, a biological organism would receive a set of photoreceptor activations across retinal space. A more biologically-relevant model should use pixels (rather than a ray-based perception of objects around the forward direction) as inputs to select actions. The model used in Mnih et al. (2015) relied on pixels as input to the agent in order to learn how to play Atari games. If I adapted the navigation code to use pixels as input, I would also want to change the corresponding neural network architecture to include a series of convolutional layers that could take advantage of the spatial relationships in the environmental state. Besides being more biologically relevant, the pixels to action agent would also have the advantage of being applicable to a variety of tasks beyond the navigation task. For example, Mnih et al. (2015) used the same model architecture and learning algorithms to learn to play 49 different Atari games. Nearly any environment can be represented as a set of fluctuating pixel values.