DRL Navigation Project Report

Madison Estabrook

MS; Facebook PyTorch Phase 3 Nanodegree Scholar

This report includes a detailed description of the learning algorithm used in the project and ideas for future projects. A plot of episodes versus scores is also included (see below [Figure 1]).

```
Episode 100	Average Score: -40.00
Episode 160	Average Score: 40.000
Environment solved in 60 episodes!	Average Score: 40.00
```

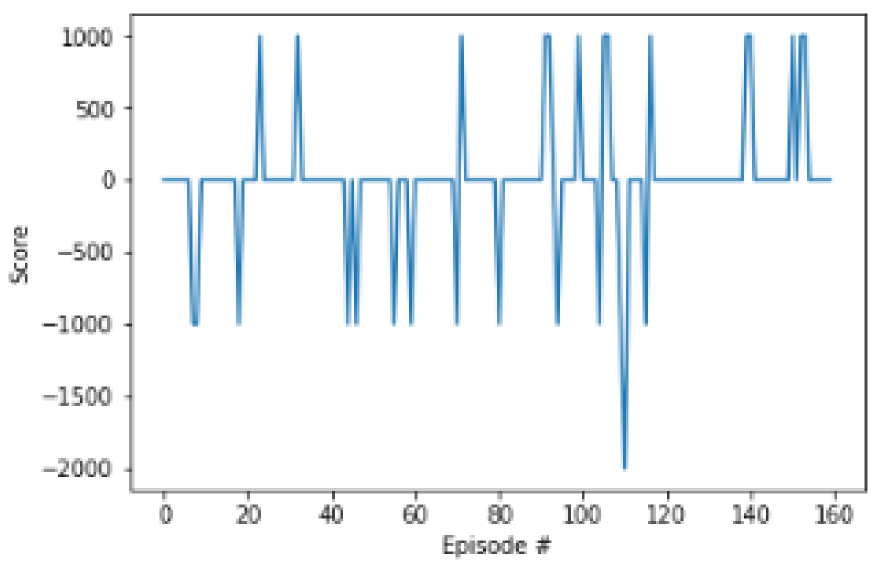

Figure 1: A plot of episodes versus scores

Learning Algorithm

The learning algorithm chosen was the Deep Q-Network (*DQN*). This algorithm was chosen for its success in other projects (example). This algorithm is as follows:

```
def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
    """Deep Q-Learning.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing epsilon
    """
```

```
12     scores = []
13     scores_window = deque(maxlen=100)
14     eps = eps_start
15     for i_episode in range(1, n_episodes+1):
16         env_info = env.reset(train_mode=True)[brain_name]
17         state = env_info.vector_observations[0]
18         score = 0
19         for t in range(max_t):
20             action = agent.act(state, eps)
21             next_state = env_info.vector_observations[0]
22             reward = env_info.rewards[0]
23             done = env_info.local_done[0]
24             agent.step(state, action, reward, next_state, done)
25             state = next_state
26             score += reward
27         scores_window.append(score)
28         scores.append(score)
29         eps = max(eps_end, eps_decay*eps)
30         print("\rEpisode {}\tAverage Score: {:.2f}"
31         .format(i_episode, np.mean(scores_window)), end="")
32         if i_episode % 100 == 0:
33             print("\rEpisode {}
34             \tAverage Score: {:.2f}".format(i_episode, np.mean(scores_window)))
35         if np.mean(scores_window)>=32.0:
36             print("\nEnvironment solved in {:d} episodes!\t
37             Average Score: {:.2f}".format(i_episode-100, np.mean(scores_window)))
38             torch.save(agent.qnetwork_local.state_dict(), "checkpoint.pth")
39             break
40     return scores
```

On line one, there are five hyperparameters:

1. `n_episodes` - this determines how many attempts the agent has at solving the environment. This is initialized to be equal to 2000 because previous work determined that 2000 was the minimum was episodes needed for that task. This value was not overwritten because (a) the learning algorithms self-closes (closes when it is done on its own) and (b) no additional attempts beyond the initial 2000 were needed.

2. `m_tax` - this determines how many timesteps the agent has when attempting to solve the environment. This value was not overwritten because success was achieved with the initial value, which was based on previous research.

3. `eps_start` - this determines the starting value for epsilon. This value was not overwritten because success was achieved with the initial value, which was based on previous research.

4. `eps_end` - this determines the minimum value for epsilon. This value was not overwritten because success was achieved with the initial value, which was based on previous research.

5. `eps_decay` - this determines how fast (or slowly) epsilon is decreased per episode (each attempt the agent has at solving the environment). This factor is multiplied by epsilon. This value was not overwritten because success was achieved with the initial value, which was based on previous research.

Lines 2-11 provide documentation for this learning algorithm. Line 12 sets the variable `scores` to be equal to an empty list. Line 13 sets the variable `scores_window` to be an instance of the deque data type with a maximum length of 100. Line 14 sets the variable `eps` to be equal to the hyperparameter eps_start. For each individual episode (i_eps) in the range of 1 and the hyperparameter n_episodes + 1:

1. Resets the environment
2. Retrieves the state
3. Sets score equal to 0
4. For each timestep in m_tax:
   (a) Sets the variable action to be equal what the agent did, which depends on the state and epsilon
   (b) Retrieves the next state
   (c) Retrieves the agent's reward
   (d) Determines if the agent is done
   (e) Causes the agency to step (learn from its previous actions)
   (f) Retrieves the next state
   (g) Adds the next reward to score
5. Add score to the list of scores
6. Sets eps to be greater (maximum) of either ens_end or eps_decay*eps
7. Prints training statistics
8. If the remainder (modulo) of the current episode number and 100 is 0:
   (a) Prints additional statistics
9. If the average of the list of scores is greater than or equal to 32.0:
   (a) Prints additional statistics
   (b) Saves the model
   (c) Breaks (stops) the loop
10. Return scores

The Model The neural network (model) uses PyTorch and is as follows:

```
class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
```

```python
    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

This model is a relatively simple neural network. This model has 2 methods - `__init__` and `forward`. The `__init__` method defines all the proprieties that this class has:

1. `seed` - this is random value that will be used later
2. `fc1` - this is the first of 3 fully-connected (*fc*) layers of the neural network. This layer is a linear layer from `state_size` and `fc1_units`
3. `fc2` - this is the second of 3 fc layers of the neural network. This layer is a liner layer from `fc1_units` and `gc2_units`
4. `fc3` - this is the third of 3 fc layers of the neural network. This layer is a linear layer from `fc2_units` to `action_size`

The `forward` method is the heart of the neural network. This method builds a network that maps `state` into action values using the proprieties that were previously defined. This method uses the ReLU function as the activation function. The ReLU function is defined to be:

$$ReLU(x) = max(0, x)$$

This means for any value of x, return the greater of x or 0. The value of x is the returned value from a fc layer.

Ideas for Future Work

Although success was achieved in the present project, there are methods through which the project could be improved. These include:

- Achieving success in the same environment in less than 60 episodes
- Comparing the following learning algorithms:
  - Double DQN
  - Dueling DQN
  - Prioritized experience replay
- Further documenting the agent's experience, such as through a .GIF or an online video