Deep Reinforcement Learning Project 1 : Navigation

Rachel Schlossman

July 25, 2021

1 Learning Algorithm

The learning algorithm is an implementation of a deep-Q network (DQN) as described in [1]. Following this paper, the action-values, Q, in iteration i are updated using the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \tag{1}$$

where γ is the discount factor . The minibatches of (state, action reward, next state) experiences, $(s, a, r, s') \sim U(D)$ are sampled uniformly from a replay buffer. The variable θ_i^- are the target network parameters. The target network parameters are set to the local Q-network parameters, θ_i, every N timesteps. Eqn. 1 is implemented in the *learn* function using Python and Pytorch as follows:

```
def learn(self, experiences, gamma):
    """Update value parameters using given batch of experience tuples.

    Params
    ======
        experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', done) tuples
        gamma (float): discount factor
    """
    states, actions, rewards, next_states, dones = experiences

    ## TODO: compute and minimize the loss
    Q_target_next_states = self.qnetwork_target(next_states).detach().max(1)[0].
        unsqueeze(1)

    # target component of loss
    # Q_des = r + gamma * max_{a'} Q(s',a',w-)

    Q_des = rewards + (gamma * Q_target_next_states) * (1-dones)

    # Q_actual
    # Q(s,a,w)
    Q_act = self.qnetwork_local(states).gather(1,actions)

    # Compute loss
    # L = (r + gamma * max_{a'} Q(s',a',w-) - Q(s,a,w))^2
    loss = F.mse_loss(Q_des, Q_act)
    # Minimize loss
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    # update target network
    self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

1.1 Hyperparameters

The following heperparameters were used in the DQN implementation:

- $\gamma = 0.995$
- $N = 4$
- learning rate = 5e-4
- replay buffer size = 1e5
- minibatch size = 64
- time constant for soft update of target parameters, τ = 1e-3

2 Model Architecture

The input passes through two linear layers with relu activation followed by a linear output layer. The two hidden layers each are comprised of 64 nodes. The code implementation is shown below.

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

    def __init__(self, state_size, action_size, seed,
                 h1_units=64, h2_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
            h1_units (int) : width of 1st hidden layer
            h2_units (int) : width of 2nd hidden layer
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        # fc stands for "fully connected"
        self.fc1 = nn.Linear(state_size, h1_units)
        self.fc2 = nn.Linear(h1_units, h2_units)
        self.fc3 = nn.Linear(h2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        o1 = F.relu(self.fc1(state))
        o2 = F.relu(self.fc2(o1))
        o3 = self.fc3(o2)

        return o3
```

3 Results

The agent is able to receive an average reward over 100 episodes of +13 in 395 episodes, as shown in Fig. 1.

4 Future Work

The learning algorithm does not make use of Double Q-Learning, prioritized experience replay, or a dueling DQN architecture. Using one or more of these modifications could potentially improve the performance of the DQN agent.

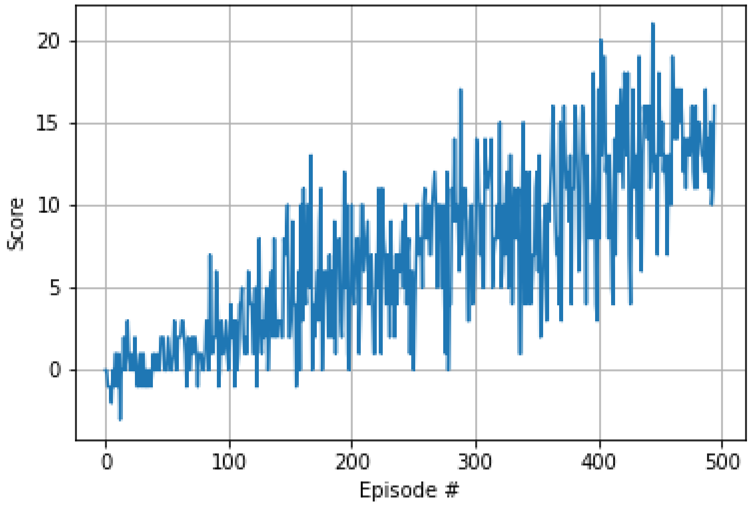

Figure 1: Average Score Plot

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.