Deniz Aytemiz

Advanced Machine Learning - HW 5

Task 1: When using a neural network for supervised learning, there is an assumption of independent and identically distributed (i.d.d) data samples, which is not met when samples come from a sequential experience. The samples are highly correlated to each other which causes the learning to be unstable. By using memory replay, all sequential experiences are stored in a buffer which at each time step a different experience is chosen. This way somewhat independent samples are chosen at each time step which makes samples not as correlated to each other, though there might be some correlation due to the dynamics of the environment. This leads to a stable learning process.

Also, the memory replay proves to be more efficient since one experience has a chance of being used multiple times before it is discarded or forgotten. This makes the learning process use fewer samples than without a replay buffer.

Task 2:

```
Samples:  (array([[[18],
        [11],
        [17],
        [10]],

       [[19],
        [18],
        [13],
        [11]]]), (2, 2), (5, 9), array([[[24],
        [21],
        [25],
        [21]],

       [[25],
        [28],
        [21],
        [24]]]), (0, 1))
```

The experiences are stored in test_buffer. State And next state are arrays representing the current state and next state of the environment after action taken. Action and reward are integers representing the action taken and reward received. Done indicates whether episode is complete or not. In the output [18],[11],[17],[10] is one state and [19],[18],[13],[11] is the other state. actions are 2 and 2 for the 2 sampled states. rewards for both experiences are 5 and 9. the array([[[24],[21,...[24]]]) is the next state. (0,1) is the done indicating the episode is not done in the first experience and done in the second one.

Task 3:

```
# In this part, we are going to implement a quite basic neural network using pytorch.
# The dimension and profile of the Q-network is crucial to the final performance, therefore we will provide some specific suggestions:
# Suggestions: 2-5 fully connected layers, less than 512 neurons per layer
# The training time, or say computational complexity directly depend on the scale of the network, which means you may want to find a balance between performance and complexity
class DQN(nn.Module):                                                    # We build our neural network using nn.Module class in pytorch, you may find some examples at https://pyt
    def __init__(self, num_actions):                                     # 'num_action' is the number of all possible actions in the environment. Each action corresponds to one t
        super(DQN, self).__init__()
        ## student code here
        self.layers = nn.ModuleList([nn.Linear(4, 500), nn.Linear(500, 500), nn.Linear(500, 500), nn.Linear(500, 2)])            # In this part, you need to define your network layer
        # self.layers = nn.ModuleList([nn.Linear(4, 100), nn.Linear(100, 100), nn.Linear(100, 2)])
        ## student code end
    def forward(self, x):                                                # With the network defined, we have to define the forwarding method for the neural network, within which
        ## student code here
        for layer in self.layers:
          x = layer(x)
          x = nn.functional.relu(x)
                                        # define forward procedure of your network
        return  x           # Return the output of the network. It should be the same dimension with action space
        ## student code end
```

Figure 1: Task 3 code

```
[ ]  # Test Case of Task 3
     Test_DQN = DQN(10)
     state = env.reset()
     action_value = Test_DQN.forward(torch.FloatTensor(state))
     print('The Q value for each action:', action_value.data.numpy())
     # The output should be a vector consists of 2 Q values for action0 and action1. Below is an example:
     # The Q value for each action: [0.05737116 0.07451424]
```

```
The Q value for each action: [0.02950585 0.        ]
```

Figure 2: Test case results

Task 4:

```
]  # In this task, your are going to design epsilon-greedy policy.
   # The input of function the state and expected epsilon
   # Epsilon is the probability of taking actions randomly
   # Hint: you will compose to different strategies: taking action randomly & take action from Q-network; Every time you call this functi
   # Output: the output should be a integer, the index of the action in action space
   def choose_action(state, epsilon):
       state   = Variable(torch.FloatTensor(state).unsqueeze(0), volatile=True)
       ## student code here
       if random.uniform(0,1)>epsilon:                                                    # We can implement exploiting here
           q_value =  eval_model.forward(state)                          # Use .forward() method in DQN class to get the Q values of al
           action = torch.argmax(q_value).item()                 # Calculate the index of action with highest Q value. Hint: you can use
       else:                                                                              # We can implement exploring here
           action =  env.action_space.sample()                       # You can use built-in functions in Random or Numpy to generate a a
       ## student code end
       return action                                                                  # Return the index of action
```

Figure 3: Task 4 code

```
# Test Case for Task 4
test_state = env.reset()
for i in range(10):                                    # We generate 1
    test_action = choose_action(test_state, 0.5)
    print('action ', i, 'is: ', test_action)

# The result of returned action should be an index in the range of action
# action  0 is:  1
# action  1 is:  0
# action  2 is:  1
# action  3 is:  0
# action  4 is:  1
# action  5 is:  1
# action  6 is:  1
# action  7 is:  1
# action  8 is:  0
# action  9 is:  1
```

```
action  0 is:  1
action  1 is:  1
action  2 is:  0
action  3 is:  1
action  4 is:  0
action  5 is:  1
action  6 is:  1
action  7 is:  0
action  8 is:  0
action  9 is:  1
```

Figure 4: Test code output

Task 5:

```
[ ]  # In this part, you will need to design a function that returns a
     # The input is the current training step. The bigger it is, the s
     # The output is the current epsilon of type 'float'
     epsilon_initial = 1.0                # The epsilon should be 1 at
     ## student code here
     def epsilon_by_frame(frame_idx):       # Design your function her
       return np.exp(-frame_idx*0.0013)
     ## student code end
```

Figure 5: Task 5 code

```
# test case for task 5
plt.plot([epsilon_by_frame(i) for i in range(10000)])
# print(epsilon_by_frame(1000))
# print(epsilon_by_frame(4000))
# Check if your epsilon is decreasing along the training steps. As a suggestion, the epsilon value should be
```

```
[<matplotlib.lines.Line2D at 0x7f769ebb7340>]
```

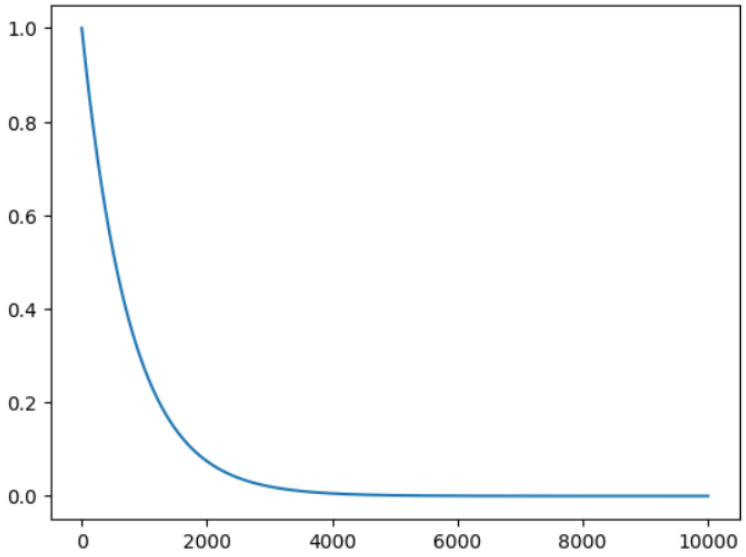


Figure 6: Test case output

Task 6:

```
def compute_td_loss(batch_size, gamma):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)  # Sa

    state      = Variable(torch.FloatTensor(np.float32(state)))
    next_state = Variable(torch.FloatTensor(np.float32(next_state)))
    action     = Variable(torch.LongTensor(action))
    reward     = Variable(torch.FloatTensor(reward))
    done       = Variable(torch.FloatTensor(done))
    ## student code here (Note that we have provided some comments based on our impl

     # Calculate the expected Q value using the target network
    next_q_state_values = target_model.forward(next_state)
    next_q_values = torch.max(next_q_state_values, dim=1)[0]
    expected_q_values = reward + gamma * next_q_values * (1 - done)

    # Calculate the Q value using the evaluation network
    q_values = eval_model.forward(state)
    q_value = q_values.gather(1, action.unsqueeze(1)).squeeze(1)

    # Compute the loss using MSE
    loss = F.mse_loss(q_value, expected_q_values.detach())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    return loss
```

Figure 7: Task 6 code

```
# Test Case for Task 6:
test_state = np.array([0.27,1.95,-0.16,-2.29])
test_next_state = x=np.array([0.32,2.14,-0.21,-2.63])
test_action = 1
test_done = False
test_reward = 1.
for i in range(100):
    replay_buffer.push(test_state, test_action, test_reward, test_next_state, test_done)
test_loss = compute_td_loss(8, 0.9)

print('The test loss is: ', test_loss)
# The output should be a tensor less than 10 when you first run this cell
# I would suggest you to rerun previous "Evaluate Network and Target Network" cell to initial
```

```
The test loss is:  tensor(1.0678, grad_fn=<MseLossBackward0>)
```

Figure 8 : Test code for Task 6

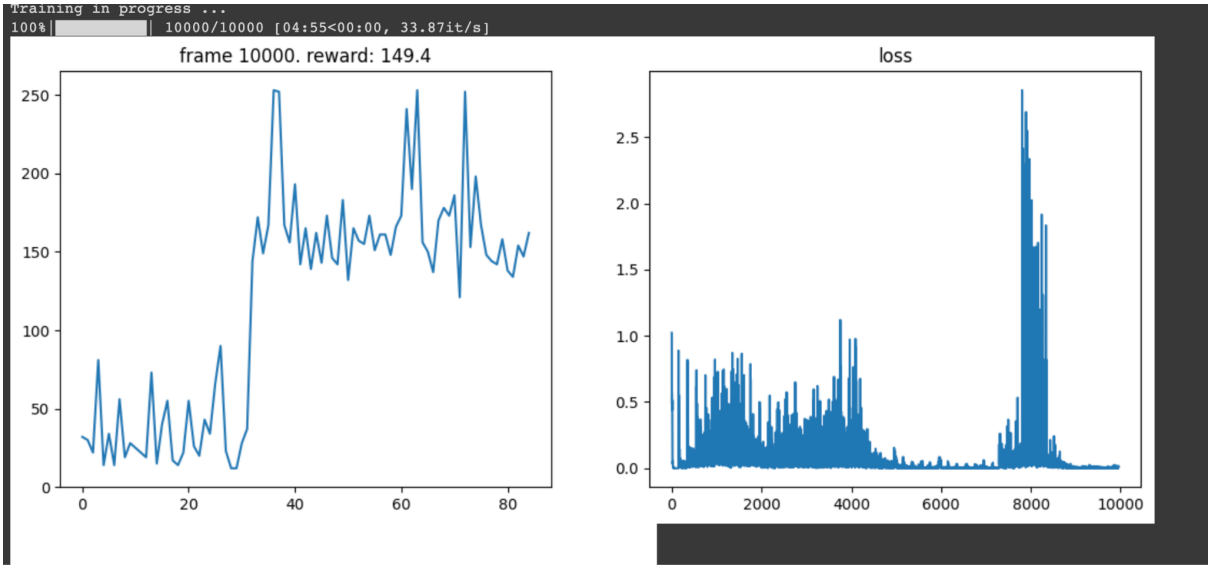
Task 7:

```
[ ]  # In this part, we will update target network using the parameters from evaluate network
     # The input is the evaluate network and target network
     def update_target(eval_net, target_net):
         global eval_model
         global target_model
         ## student code here
         target_net.load_state_dict(eval_net.state_dict())
         # for i in eval_net.state_dict():
         #   target_net.state_dict()[i] = i
          # Hint: You may use nn.module.load_state_dict() to update parameters. More info:https://pytorch.org/tutorials/beginner/saving_loading_models.html
         ## student code end
```

```
[ ]  # This function is defined to plot reward and loss curves
     def plot(frame_idx, rewards, losses):
         # clear_output(True)
         plt.figure(figsize=(20,5))
         plt.subplot(131)
         plt.title('frame %s. reward: %s' % (frame_idx, np.mean(rewards[-10:])))
         plt.plot(rewards)
         plt.subplot(132)
         plt.title('loss')
         plt.plot(losses)
         plt.show()
```

Task 7: Code for Task 7

Task 8:



Task 8: Results after training

Batch size bigger than 512 did not work that well in the training results. Layers greater than 500 and multiples of 500 did not work that well either. The network update period did not do that much of a difference but overall there is a decreasing pattern in loss.