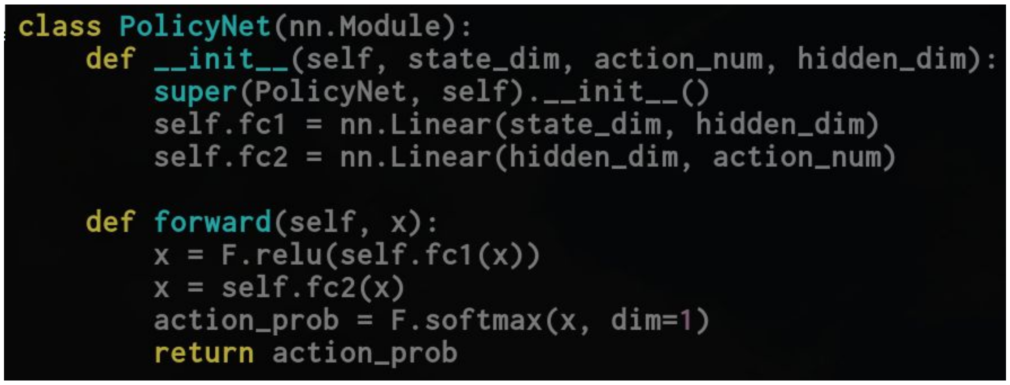
ADL 2019 HW3 Report

R07922001 陳佳佑

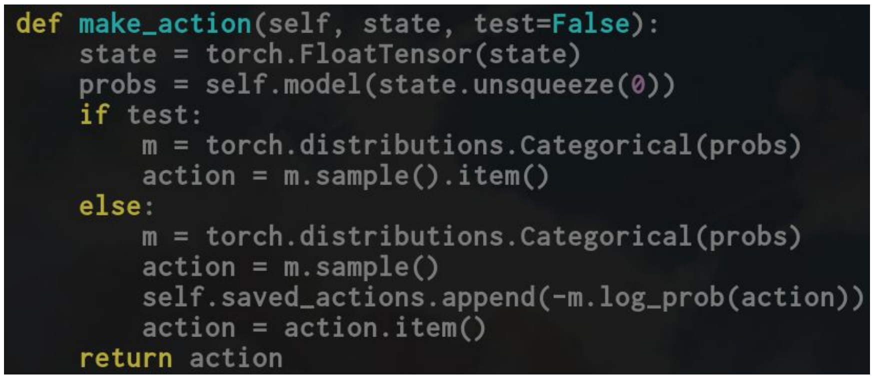
Problem 1: Basic Perfomance (Policy Gradient)

I just use the TA's model. The policy network is simply two fully connected layer.

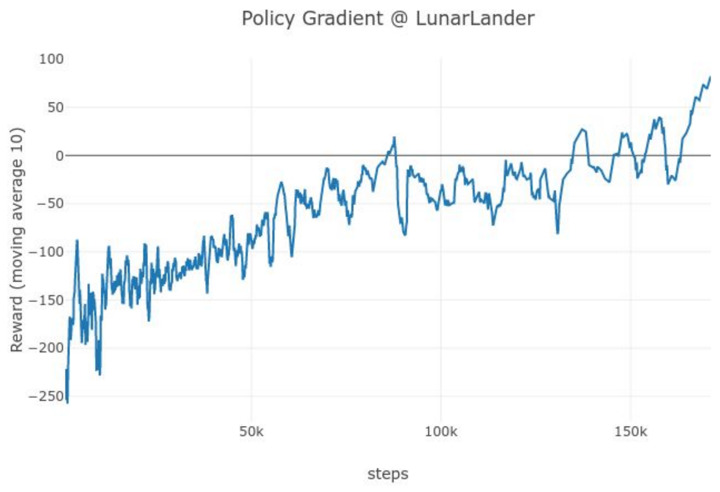


```
class PolicyNet(nn.Module):
    def __init__(self, state_dim, action_num, hidden_dim):
        super(PolicyNet, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, action_num)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        action_prob = F.softmax(x, dim=1)
        return action_prob
```

For updating, I will update after finishing an episode. And for make_action, I use sampling in both testing and training phase. However, it's quite weird that if I use argmax in testing phase the performance is super poor. Maybe the policy network hasn't really converged to a good enough phase.

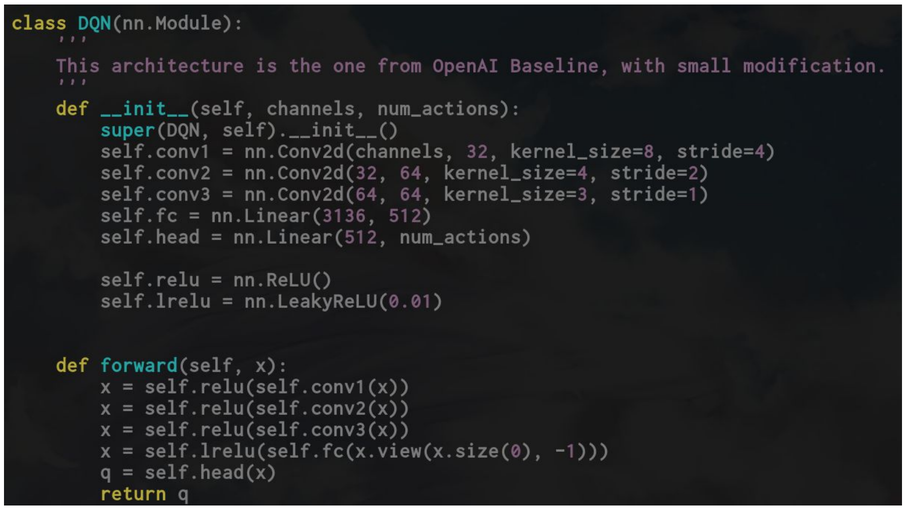


```
def make_action(self, state, test=False):
    state = torch.FloatTensor(state)
    probs = self.model(state.unsqueeze(0))
    if test:
        m = torch.distributions.Categorical(probs)
        action = m.sample().item()
    else:
        m = torch.distributions.Categorical(probs)
        action = m.sample()
        self.saved_actions.append(-m.log_prob(action))
        action = action.item()
    return action
```



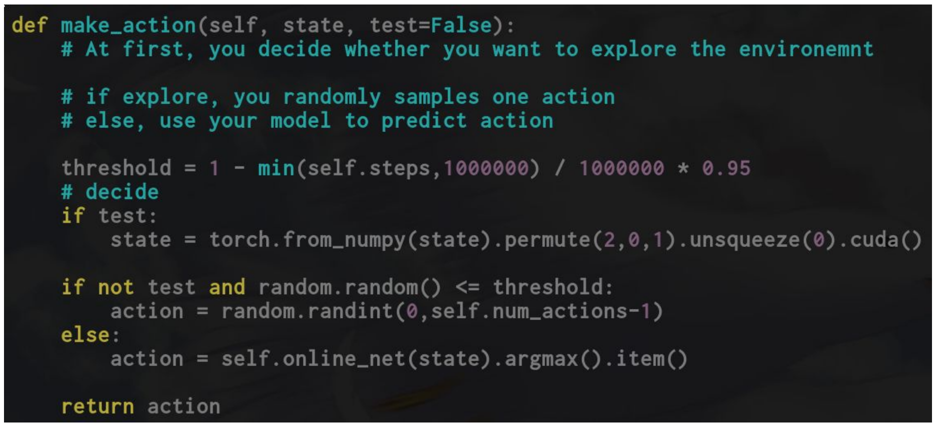
Problem 1: Basic Perfomance (DQN)

As same as the policy gradient stuff, I use the TA's code again. The Q network is composed of three conv layers and then two fully connected layers. It's quite reasonable to use convolution network here, since the input state is much more complicated than the environment LunarLander.

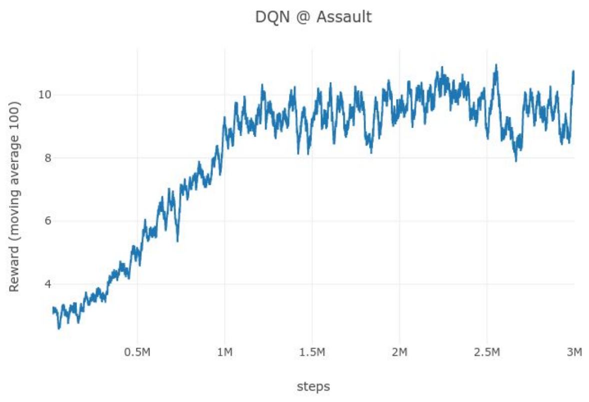


```python
class DQN(nn.Module):
    '''
    This architecture is the one from OpenAI Baseline, with small modification.
    '''
    def __init__(self, channels, num_actions):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(channels, 32, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.fc = nn.Linear(3136, 512)
        self.head = nn.Linear(512, num_actions)

        self.relu = nn.ReLU()
        self.lrelu = nn.LeakyReLU(0.01)


    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.lrelu(self.fc(x.view(x.size(0), -1)))
        q = self.head(x)
        return q
```

For exploration, I adopt the epsilon-greedy method. The threshold is linearly decayed to 0.05 in 1M steps. And for testing, I'll use the maximum greedy policy. If the epsilon greedy is still adopted, the performance is pretty poor.

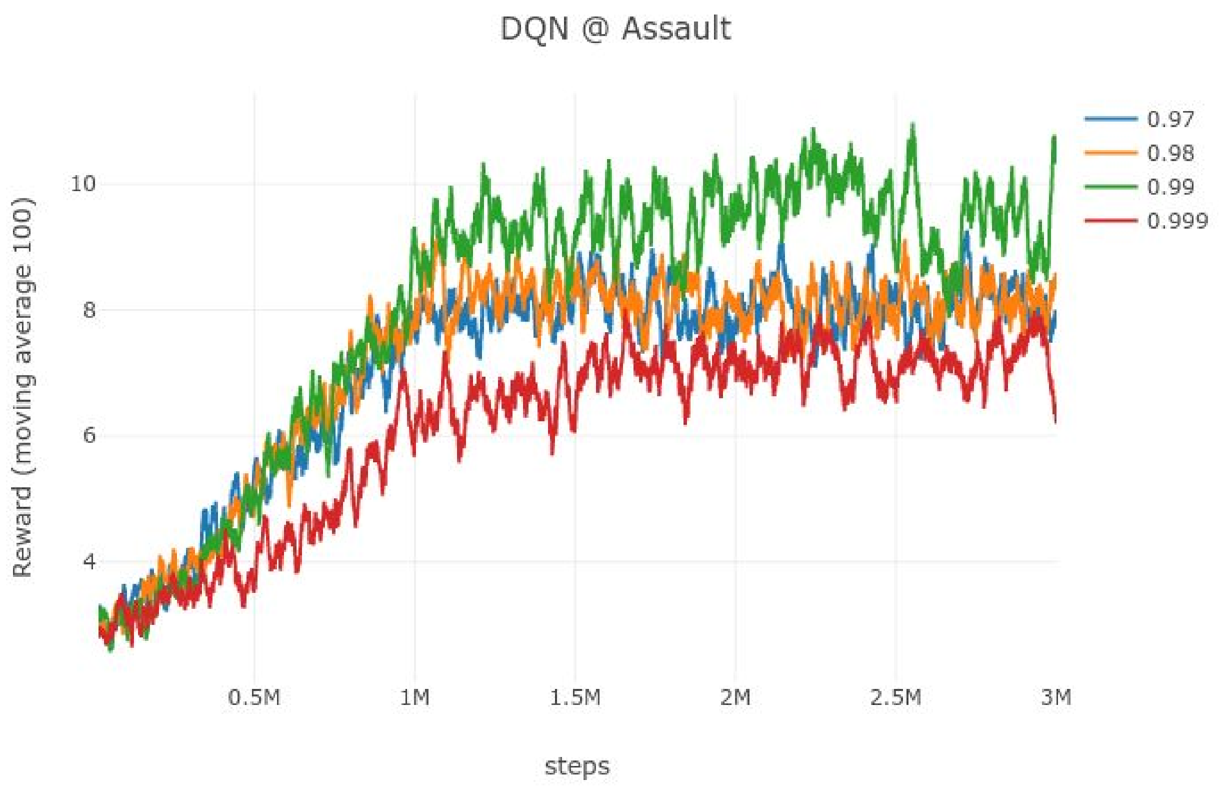


```python
def make_action(self, state, test=False):
    # At first, you decide whether you want to explore the environemnt

    # if explore, you randomly samples one action
    # else, use your model to predict action

    threshold = 1 - min(self.steps,1000000) / 1000000 * 0.95
    # decide
    if test:
        state = torch.from_numpy(state).permute(2,0,1).unsqueeze(0).cuda()

    if not test and random.random() <= threshold:
        action = random.randint(0,self.num_actions-1)
    else:
        action = self.online_net(state).argmax().item()

    return action
```



Problem 2: Experiment with DQN hyperparameters

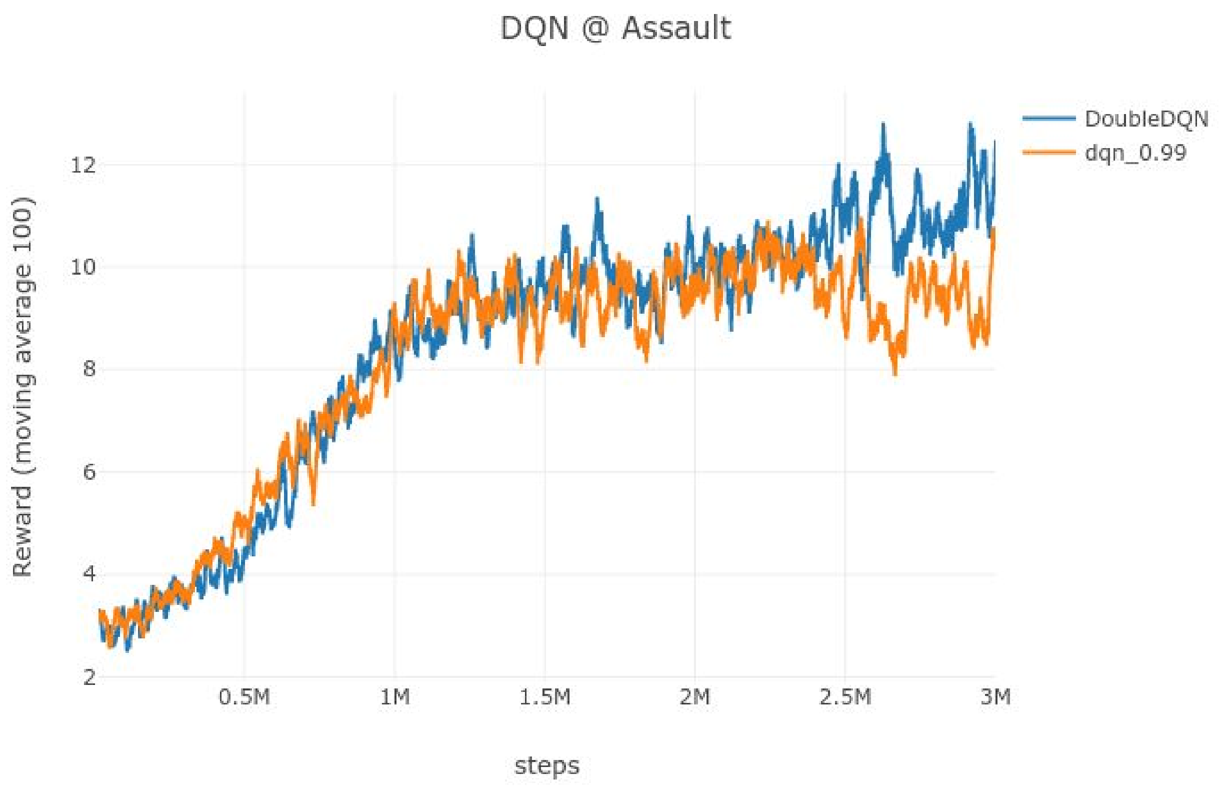
In general reinforcement setting, we would adopt a cumulative reward to let the algorithm sense for stuff happen quite long time after. And in most of the cases, 0.99 for gamma is adopted. However, in some sparse reward tasks, the reward may happen long time after. Is the 0.99 applicable? What's more, each game has different reward distribution. It may be possible that some other GAMMA works better.



However, by the result, the traditional GAMMA=0.99 beat all other candidates. It's quite non-trivial that 0.999 works worse than 0.98 and 0.97, since I have done the same study in Breakout and the result shows that 0.999 is the best. The reason that 0.999 works such bad may due to the fact that the reward is quite dense in 'Assault'. In 'Breakout', we would get a reward only if we break a brick. As a result, the reward may be more sparse than 'Assault' and 0.999 outperform others.

Problem 3: Improvement to DQN (Double DQN)

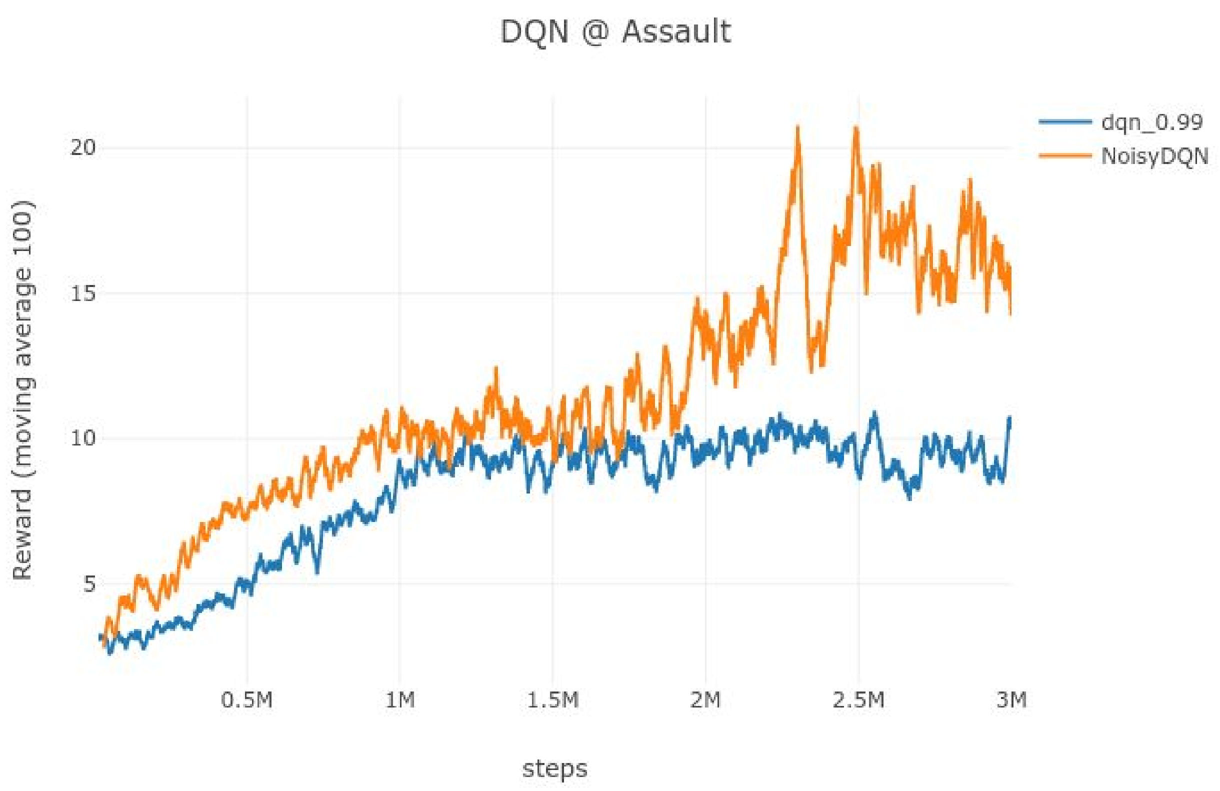
In the origin update formula of DQN, the online_net would be overestimated to the target_net, since the Q(next_state) of target_net is took as maximum. To reduce the bias, we could select the the action of Q_target(next_state) by argmax Q_online(next_state).



From the result, DoubleDQN really improves the performance. We could see that DoubleDQN performs like vanilla DQN in the first 2M steps. However, after the decay finished, DoubleDQN still improve for a little moment. This may be the effect of not over-estimation.

Problem 3: Improvement to DQN (Noisy DQN)

In "Parameter Space Noise for Exploration" and "Noisy Networks for Exploration", they try to find a better exploration method instead of epsilon greedy. In general, epsilon greedy method has a problem that the noise isn't state dependent. However, in real world setting, the effect of a noise is often sequentially. As a result, both paper propose method to add noise to parameter space to produce a state dependent noise. In this homework, I implement the latter one "Noisy Networks for Exploration" (Noisy DQN). The implementation is quite easy. All the w in origin network would be replaced as w_t * noise + b_t. In optimal condition, the noise should be different for each parameter. However, random sampling for such large amount may waste a lot of time. As a result, the implementation version would sync all the noise in the current network. The noise would be resampled in each epoch.



By the graph above, we could see that NoisyDQN outperforms the vanilaDQN. This implies that the noise in parameter space really help exploration. However, the performance of NoisyDQN has large variance. It may be due to the original std for sampling the noise.