Deep reinforcement learning (11220CS565700)
HW2 report

林駿亨 110060062 EECS25

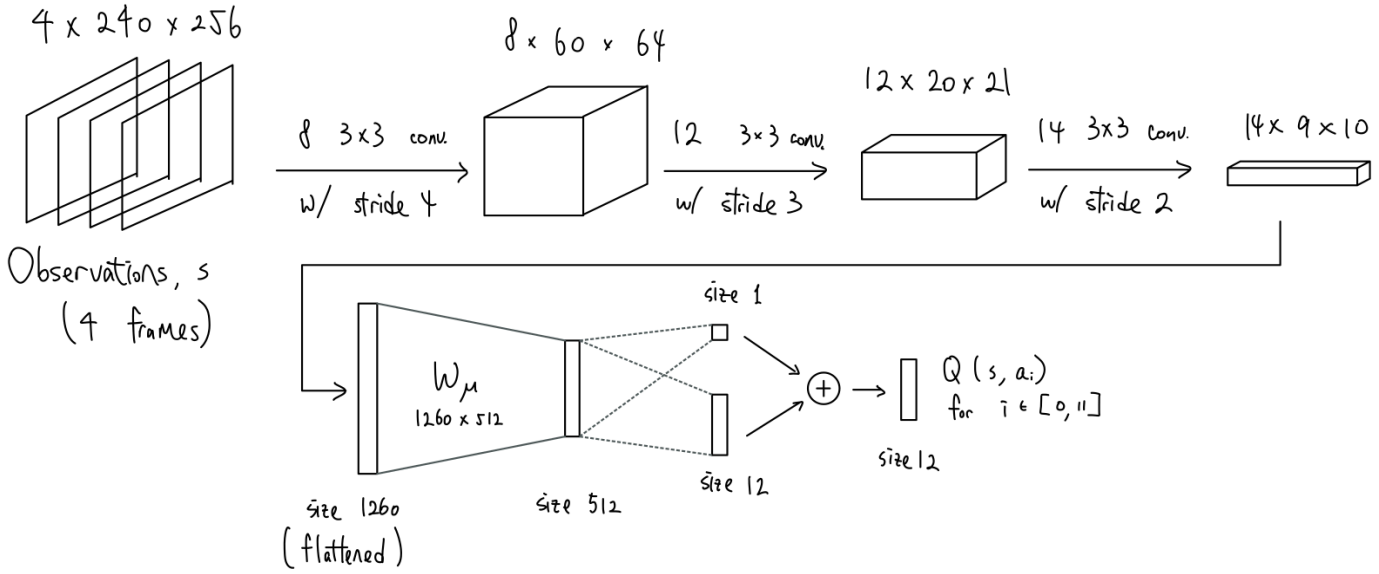
1. DQN implementation

In my implementation, I added:

- Dueling and noisy (only on linear layers like the original paper) design to the DQN architecture
- Prioritized experience replay (PER) for better experience sampling
- Double DQN learning target

Architecture

The whole architecture is as follows:



Firstly, whenever the agent receives 13 frames of game observation, it will choose the 0th, 4th, 8th and 12th frames of that frame stack (also known as frame skipping technique). The agent will then grayscales the chosen 4 frames and concatenates them together, resulting in a 4 x 240 x 256 tensor, and feeded into the model.

Convolution layers

The model architecture is modified from the original DQN paper, which includes 3 convolution (conv.) layers and 2 fully connected (FC) layers. The design of the 3 convolution layers are: 8 3x3 conv. filters with stride 4, 12 3x3 conv. filters with stride 3 and 14 3x3 conv. filters with stride 2 respectively.

Noisy Net design

The resulting 14 x 9 x 10 tensor is then flattened into a vector with size 1260, and feeded to the FC part. Transforming with a 1260 x 512 weight matrix, the vector becomes size 512. Note that this layer is incorporated with noisy net design, which is: During training, the weight is W = W_mu + W_sigma * W_epsilon, where

W_epsilon is sampled from standard Gaussian distribution (N(0, 1)) on every learning step. During testing, the weight is simply W_mu. As the weight is different on every learning step, the agent not only outputs random actions on the early learning stage (which encourages explorations), but also learns to overcome the randomness or noises (which encourages policy robustness) while pursuing the optimal policy on the later learning stage. Code snippet of noisy linear layer design:

```
def forward(self, input):
    if self.training:
        weight = self.weight + self.sigma_weight * self.epsilon_weight
        bias = self.bias + self.sigma_bias * self.epsilon_bias
    else:
        weight = self.weight
        bias = self.bias

    output = torch.nn.functional.linear(input, weight, bias)
    return output
```

Dueling design

Modifying from dueling DQN design, the 512 vector is transformed into 2 streams: advantage stream A(s, a) (which has size 12, i.e. the number of available actions) and state value stream V(s) (which has size 1). The state-action value, Q(s, a), will be computed by summing up the 2 streams, i.e. Q(s, a) = A(s, a) + V(s). Since the Q-value is learned in the form of 2 streams, the training process is more efficient. For example, there is a bad state: Mario is almost dropped into a hole and is going to die no matter what action the agent chooses. In this case, the model can decrease V(s) directly, instead of decreasing each $Q(s, a_i)$ for i = 0 to 11 repeatedly. Code snippet of dueling architecture:

```
def forward(self, x):
    x = torch.relu(self.conv1(x))
    x = torch.relu(self.conv2(x))
    x = torch.relu(self.conv3(x))
    x = x.reshape(x.size(0), -1)
    x = torch.relu(self.fc1(x))
    value = self.value_stream(x)
    adv = self.advantage_stream(x)
    adv_average = torch.mean(adv, dim=1, keepdim=True)
    q_values = value + adv - adv_average
    return q_values
```

Additional designs

Besides the architecture, I also use double DQN the learning target and prioritized experience replay as replay buffer sampling method.

Learning with double DQN

The learning target of double DQN is:

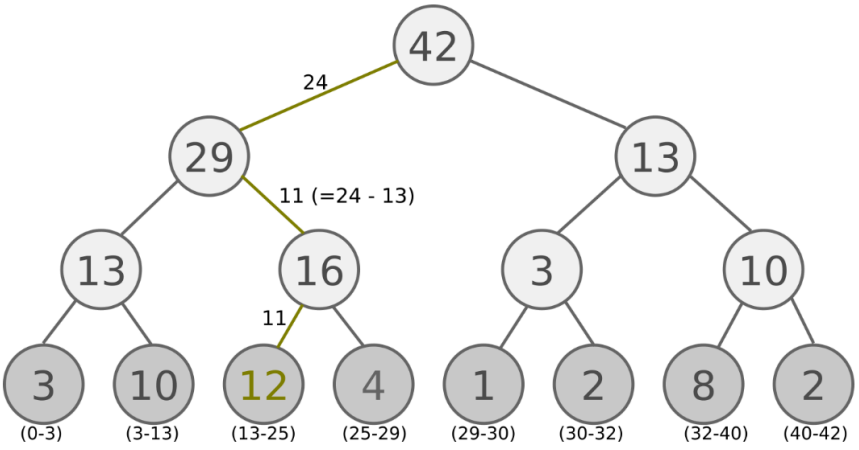
$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)$$

In my implementation, one transition to be stored to the replay buffer is (s_t, a, r, s_{t+1}, done), where s_t is a grayscale 4 skipped frame stack obtained by online network (parameterized with θ_t) taking action a at previous timestep t, r is the cumulative rewards of the 4 frames of taking that action, s_{t+1} is also a grayscale 4 skipped frame stack obtained by online network taking action a at current timestep t+1, and done indicates whether s_t is a terminal state. At last, the s_{t+1} is evaluated by target network (parameterized with θ^-_t). Using the above settings, the model is trained in double DQN style. Code snippet of the learning target calculation as follow:

```
q_values = self.model(states).gather(1, actions.unsqueeze(1)).squeeze(1)
next_q_values = self.target_model(next_states).max(1)[0].detach()
expected_q_values = (rewards + GAMMA * next_q_values * (1 - dones))
loss = (weights * torch.nn.MSELoss()(q_values, expected_q_values)).mean()
```

Sampling with prioritized experience replay (PER)

For PER, I chose the sum tree implementation style referencing this github repo. In the sum tree, each node contains 1 transition: (s_t, a, r, s_{t+1}, done), as well as its priority p. The priority of one node is the sum of its 2 children (if they exist), such that the root node has the highest priority to be sampled. Whenever a new node is going to be added to the tree, it will be assigned with a high priority, and thus new experiences are likely to be sampled. The following figure shows an example of a sum tree, where the number on each node indicates its priority:



In the training process, we will sample a batch (I chose batch size = 32) of nodes from the tree, and use their stored transitions to train our model. The loss is obtained as mean square error between the learning target and the Q-value computed by our model. However, before back propagating the loss into our model, each loss (from each node) is multiplied by a weight, which is inversely proportional to its priority (i.e.

Loss with higher priority will be decreased, vice versa). This setting can prevent the model from being biased to the experience with higher priority. At the end, we additionally compute the TD-error (the L1 error between learning target and computed Q-value), and use it to update the priorities of the sampled node. The higher the TD-error, the more the increase in priority. This setting encourages us to sample the experience that we performed badly, and thus improve the learning efficiency.