强化学习第二次实验报告

刘翎 1120202969 07112006

1 深度 Q 网络 (Deep Q Networks) [1]

1.1 DQN 概述

在第一次 CartPole-v0 环境的实验中，笔者使用 Q-learning 方法。而 CartPole 问题的状态集合是连续的，违背了 Q 表格构造需要离散的状态变量。因此，在使用 Q-learning 方法时，需要对连续状态进行离散化处理，即把状态的连续空间切分成多个范围，每个范围的状态集合为一个新的离散化状态，从而满足将环境反馈的 (State,Action) 映射到 Q 表格的特定位置。

实际上，离散化处理会造成很多问题，包括无法表现状态变量连续性的特点、状态离散化后集合过大导致内存难以维护 Q 表格等。因此，Q-learning 在面对状态连续的问题时显得捉襟见肘。于是，Deep Q Networks 出现了，其具体原理在后续小节详细展开。

1.2 价值函数近似表示

DQN 的提出受到当时流行的神经网络的影响，将 Q-learning 中无法表示连续状态的 Q 表格替换成近似表示的神经网络，具体公式如下

$$\hat{q}(s,a,\theta) \approx q_\pi(s,a) \tag{1.1}$$

其中 $\hat{q}$ 为由神经网络参数 θ 构成的动作价值函数，在接受状态 s 和动作 a 的输入后，能够近似计算得到动作价值。通过使用神经网络，将动作价值函数和价值函数看作待训练的黑盒子，对 Q-learning 的 Q 表格进行近似替换。

1.3 DQN 算法思路

Algorithm 1 DQN with Target Network

初始化大小为 N 的经验回放集合 D
随机初始化动作价值 Q 网络的参数 θ
初始化目标动作价值 $\hat{Q}$ 网络的参数 $\theta^- = \theta$
for episode = 1,M **do**
　初始化状态序列 $s_1 = \{x_1\}$，并获得其特征向量 $\phi_1 = \phi(s_1)$
　for t = 1,T **do**
　　使用 $\epsilon - greedy$ 的策略，以 ϵ 的概率随机选择动作 a_t；否则 $a_t = argmax_a Q(\phi(s_t), a; \theta)$
　　在环境中执行动作 a_t，得到奖励 r_t 和状态图片 x_{t+1}
　　更新 $s_{t+1} = s_t$，并得到新状态的特征向量 $\phi(s_{t+1})$
　　将四元组 $(\phi_t, a_t, r_t, \phi_{t+1})$ 存入 D 中
　　在经验回放集合 D 中随机抽样 $batch\_size = m$ 大小的样本 $(\phi_j, a_j, r_j, \phi_{j+1})$
　　根据 m 个样本，计算得到当前的目标 Q 值 y_j 为

$$y_j = \begin{cases} r_j & , for\ terminal\ \phi_{j+1} \\ r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & , for\ non-terminal \phi_{j+1} \end{cases}$$

　　根据 $loss = \frac{1}{m}\sum_{j=1}^{m}(y_j - Q(\phi_j, a_j; \theta))^2$，使用梯度反向传播更新 Q 网络中的参数
　　每 C 步更新目标动作价值网络 $\hat{Q} = Q$
　end for
end for

2 双深度 Q 网络 (Double Q-lerning) [2]

DQN 在计算目标 Q 值 y_j 时，通过取最大的 Q 值，虽然可以使 Q 值的估计向可能的优化目标接近，但是受到局部最优解的约束，会导致 y_j 过度估计，使得某个可能状态动作的 Q 值过大，而产生偏差。

因此，DDQN 通过解耦在计算目标 Q 值时，动作的选择和 Q 值计算两个步骤，使用两个 Q 网络分别负责，以消除过度估计的问题。具体的计算公式如下

$$y_t = r_t + \gamma \hat{Q}(\phi_{t+1}, argmax_a Q(\phi_{t+1}, a; \theta); \theta^-) \tag{2.1}$$

其中 Q 表示以 θ 为参数的动作价值网络，$\hat{Q}$ 表示以 θ^- 为参数的目标 Q 网络，除了目标 Q 值的计算公式不同以外，其余与 DQN 算法步骤相同。

3 优先级经验回放 (Prioritized Experience Replay) [3]

3.1 PER 概述

在 DQN 和 DDQN 中，采用建立 Replay Buffer 的方式，通过在经验池里随机采样，来计算更新目标 Q 值。实际上，样本的 TD 误差 =| 目标 Q 网络计算得到的目标 Q 值-当前 Q 网络计算得到的目标 Q 值 |，TD 误差越大，其对反向梯度的计算影响大，说明该样本对 Q 网络的更新帮助更大。

PER 算法根据上述思想，若以更大的概率采样 TD 误差较大的样本，则算法收敛速度更快，从而引入了经验回放的优先级策略。

3.2 带优先级的经验回放池

为了使得 TD 误差绝对值 $|\delta(t)|$ 越大的样本，其被选择的概率越大，建立了样本的优先级策略，即样本的优先级正比于 $|\delta(t)|$。

在代码实现时，采用 SumTree 的二叉树结构来存储带优先级的经验回放池：所有的经验回放样本保存在叶子结点中，一个叶子结点保存一个样本的数据和优先级，父节点则保存儿子节点的优先级值之和。在采样时，针对 [0, 根结点优先级和点值] 范围内采样，则叶子结点中优先级越高的样本，其被采样的可能性越高。

同时，Q 网络的损失函数也考虑了样本优先级，具体公式如下

$$loss = \frac{1}{m} \sum_{j=1}^{m} \omega_j (y_j - Q(\phi_j, a_j; \theta))^2 \tag{3.1}$$

其中 ω_j 表示第 j 个样本的优先级权重，其通过 TD 误差归一化得到。

3.3 PER 算法思路

Algorithm 2 Double DQN with proportional prioritization

初始化数据结构为 sumTree 的带优先级的经验回放池 D，其 N 个叶子结点的优先级 $p_j = 1$
随机初始化动作价值 Q 网络的参数 θ
初始化目标动作价值 $\hat{Q}$ 网络的参数 $\theta^- = \theta$
for episode = 1,M **do**
　初始化状态序列 $s_1 = \{x_1\}$，并获得其特征向量 $\phi_1 = \phi(s_1)$
　for t = 1,T **do**
　　使用 $\epsilon - greedy$ 的策略，以 ϵ 的概率随机选择动作 a_t
　　否则 $a_t = argmax_a Q(\phi(s_t), a; \theta)$
　　在环境中执行动作 a_t，得到奖励 r_t 和状态图片 x_{t+1}
　　更新 $s_{t+1} = s_t$，并得到新状态的特征向量 $\phi(s_{t+1})$
　　将四元组 $(\phi_t, a_t, r_t, \phi_{t+1})$ 存入 D 中
　　在 sumTree D 中抽样 $batch\_size = m$ 个样本 $(\phi_j, a_j, r_j, \phi_{j+1})$
　　每个样本被采样概率 $P(j) = \frac{p_j}{\sum_i (p_j)}$，其优先级权重 $\omega_j = (N * P(j))^{-\beta} / max_i \omega_i$
　　根据 m 个样本，计算得到当前的目标 Q 值 y_j 为

$$y_j = \begin{cases} r_j & , for\ terminal\ \phi_{j+1} \\ r_j + \gamma \hat{Q}(\phi_{j+1}, argmax_a Q(\phi_{j+1}, a; \theta); \theta^-) & , for\ non-terminal \phi_{j+1} \end{cases}$$

　　根据 $loss = \frac{1}{m} \sum_{j=1}^{m} \omega_j (y_j - Q(\phi_j, a_j; \theta))^2$，使用梯度反向传播更新 Q 网络中的参数
　　更新 sumTree 中所有节点的优先级 $p_j = |y_j - Q(\phi_j, a_j; \theta)|$
　　每 C 步更新目标动作价值网络 $\hat{Q} = Q$
　end for
end for

3.4 关键代码展示

```
class PrioritizedReplayBuffer(ReplayBuffer):
    """
        带优先级带经验回放池
    """
    def __init__(self, size, alpha):
        super(PrioritizedReplayBuffer, self).__init__(size)
        assert alpha > 0
        self._alpha = alpha

        it_capacity = 1
        while it_capacity < size:
            it_capacity *= 2

        self._it_sum = SumSegmentTree(it_capacity)
        self._it_min = MinSegmentTree(it_capacity)
        self._max_priority = 1.0

    def push(self, *args, **kwargs):
        """See ReplayBuffer.store_effect"""
```

```
20          idx = self._next_idx
21          super(PrioritizedReplayBuffer, self).push(*args, **kwargs)
22          self._it_sum[idx] = self._max_priority ** self._alpha
23          self._it_min[idx] = self._max_priority ** self._alpha
24
25      def _sample_proportional(self, batch_size):
26          """
27              按照优先级进行sample
28              优先级越高的样本，被抽取的概率越高
29          """
30          res = []
31          for _ in range(batch_size):
32              mass = random.random() * self._it_sum.sum(0, len(
33                                      self._storage) - 1)
34              idx = self._it_sum.find_prefixsum_idx(mass)
35              res.append(idx)
36          return res
37
38      def sample(self, batch_size, beta):
39          assert beta > 0
40          # 按优先级sample样本
41          idxes = self._sample_proportional(batch_size)
42          # 更新 w_j = (p*N)^(-beta)/max(w)
43          weights = []
44          p_min = self._it_min.min() / self._it_sum.sum()
45          max_weight = (p_min * len(self._storage)) ** (-beta)
46
47          for idx in idxes:
48              p_sample = self._it_sum[idx] / self._it_sum.sum()
49              weight = (p_sample * len(self._storage)) ** (-beta)
50              weights.append(weight / max_weight)
51          weights = np.array(weights)
52          encoded_sample = self._encode_sample(idxes)
53          return tuple(list(encoded_sample) + [weights, idxes])
54
55      def update_priorities(self, idxes, priorities):
56          """
57              更新priorities = KL(p||q) ^ weight
58          """
59          assert len(idxes) == len(priorities)
60          for idx, priority in zip(idxes, priorities):
61              # assert priority > 0
62              assert 0 <= idx < len(self._storage)
```

```
63              self._it_sum[idx] = priority ** self._alpha
64              self._it_min[idx] = priority ** self._alpha
65
66              self._max_priority = max(self._max_priority, priority)
```

4 竞争双深度 (Dueling Networks) [4]

Dueling Networks 在 PER 的基础上，对 Q 网络结构进行了改进，将 Q 网络分为两个子网络部分：其中一个部分为价值函数部分 $V(s)$，仅与状态 s 有关，而与动作 a 无关；另一个部分则是优势函数部分 $A(s,a)$，同时与状态 s 和动作 a 有关。

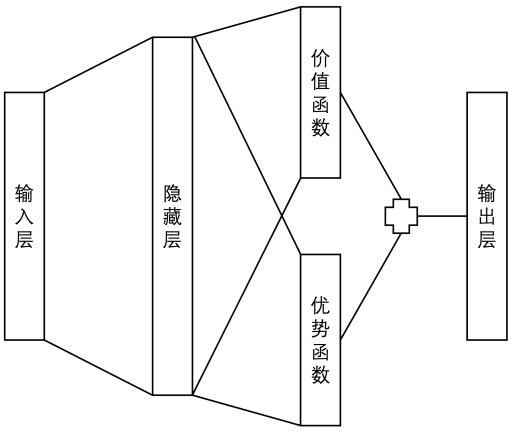

图 4-1 竞争双深度网络结构

则动作价值函数 Q 由这两部分可以表示为如下公式

$$Q(s,a,\omega,\alpha,\beta) = V(s,\omega,\alpha) + A(s,a,\omega,\beta) - \frac{\sum_{a'} A(s,a')}{N_{actions}} \tag{4.1}$$

其中，ω 是公共部分的网络参数，α,β 分别为价值函数和优势函数独占的网络参数，最后一部分是对优势函数进行中心化的处理，以增加 V 和 A 的作用辨识度。

4.1 关键代码展示

```
1  class DuelingCnnDQN(nn.Module):
2      def __init__(self, input_shape, num_outputs, env):
3          super(DuelingCnnDQN, self).__init__()
4                  # advantage和value网络定义部分
5          self.advantage = nn.Sequential(
6              nn.Linear(self.feature_size(), 512),
7              nn.ReLU(),
8              nn.Linear(512, num_outputs)
9          )
10
11         self.value = nn.Sequential(
12             nn.Linear(self.feature_size(), 512),
```

```
            nn.ReLU(),
            nn.Linear(512, 1)
        )

    def forward(self, x):
                # 前向传播
        x = self.features(x)
        x = x.view(x.size(0), -1)
        advantage = self.advantage(x)
        value = self.value(x)
        return value + advantage - advantage.mean()
```

5 分布式深度 Q 网络 (Distributional RL) [5]

分布式深度 Q 网络开创了一个新的改进方向，将动作价值 Q 的表示从值 (value function) 拓展到分布 (value distribution)。

5.1 分布式 Bellman 方程

相比于 DQN 中用分布的期望均值来描述 Q 网络，Distributional Bellman Equation 将分步特征完整表示出来，保留了更多的分布信息。对应的 Q 值分布的 Bellman 方程如下

$$Z(x,a) = R(x,a) + \gamma Z(x',a') \tag{5.1}$$

其中，DQN 中的 $Q(x,a)$ 和 Q 分布 $Z(x,a)$，满足以下关系

$$Q(x,a) = E[Z(x,a)] \tag{5.2}$$

5.2 C51 算法

在表示 Distributional DQN 的值分布时，为了符合现实中可能存在多个峰值的值分布，因此不能选择高斯分布的表示方法。在 Distributional RL 中提出了 C51 算法，即使用 51 个等间距的 atoms 来表示一个分布。

对于输入的状态 s，经过 Q 网络，得到一个分布式矩阵 Z(x,a)，具体为 $N_{actions}$ 行、N 列（在 C51 中 $N = 51$）。则 atoms 便是将动作价值函数均分成 51 份的范围值，可表示成 $\{z_0, z_1, ..., z_{N-1}\}$。则输出矩阵中的每一行表示动作为 a 的一组 N 个范围的概率 $\{p_0^a, p_1^a, ..., p_{N-1}^a\}$。具体可表示成以下公式

$$Q(x,a) = E[Z(x,a)] \approx \sum_{i=0}^{N} z_i p_i^a \tag{5.3}$$

为了保证每行都是一组离散的分布，对每行进行 Softmax 操作，具体公式如下。

$$z_i = \frac{V_{max} - V_{min}}{N} * i \tag{5.4}$$

5.3 分布的损失函数

由于损失函数需要度量两个分布之间的距离，作者提出一种启发式的方法，即使用 KL 散度去衡量两个分布的距离。

首先，从经验回放池中采样 (s, a, r, s')，计算得到目标网络的分布结果 y，其具体计算过程如下

$$y = r + \gamma Z(s', a^*) \tag{5.5}$$

$$Q(s_{t+1}, a) = \sum_i z_i p_i(s_{t+1}, a) \tag{5.6}$$

$$a^* \leftarrow argmax_a Q(s_{t+1}, a) \tag{5.7}$$

其中，a^* 通过 $Q(s', a)$ 取最大值所决。

最后，将经过调整得到的动作价值分布 $y = r + \gamma Z(s', a^*)$ 对齐到 atoms 中，与 $Z(s, a)$ 计算 KL 散度，计算公式如下

$$D_{KL}(p||q) = \sum_{i=1}^{N} p(x_i)(log\frac{p(x_i)}{q(x_i)}) \tag{5.8}$$

其中，p 和 q 分别带入更新后的分布 y 和 $Z(s, a)$，计算得到对应的梯度反向传播进行网络的更新。

5.4 关键代码展示

5.4.1 计算目标网络的分布结果 y

```
def projection_distribution(self, next_state, rewards, dones):
        batch_size = next_state.size(0)
        # 每个atom的距离deltaz, 并获得support:size=atoms
        delta_z = float(self.Vmax - self.Vmin) / (
                self.num_atoms - 1)
        support = torch.linspace(
                self.Vmin, self.Vmax, self.num_atoms)
# (batch_size,actions,atoms)*(atoms) = (batch_size,actions,atoms)
        next_dist = self.target_model(
                next_state).data.cpu() * support
        # 选峰值作为next_action
        next_action = next_dist.sum(2).max(1)[1]
        next_action = next_action.unsqueeze(1).unsqueeze(1).expand(
                next_dist.size(0), 1, next_dist.size(2))
        next_dist = next_dist.gather(
                1, next_action).squeeze(1) # (batch_size,atoms)

        rewards = rewards.unsqueeze(1).expand_as(next_dist)
        dones = dones.unsqueeze(1).expand_as(next_dist)
```

```
        support = support.unsqueeze(0).expand_as(next_dist)
        # Tz=r+gamma*Z
        Tz = rewards.cpu() + (
                1 - dones.cpu()) * self.gamma * support.cpu()
        Tz = Tz.clamp(min=self.Vmin, max=self.Vmax)
        b = (Tz - self.Vmin) / delta_z
        l = b.floor().long()
        u = b.ceil().long()
        # 偏移
        offset = torch.linspace(0, (
                batch_size - 1) * self.num_atoms, batch_size)\
                .long().unsqueeze(1).expand(batch_size, self.num_atoms)
        # 将dist投影到对应的位置
        proj_dist = torch.zeros(next_dist.size())
        proj_dist.view(-1).index_add_(
                0, (l + offset).view(-1), (
                        next_dist * (u.float() - b)).view(-1))
        proj_dist.view(-1).index_add_(
                0, (u + offset).view(-1), (
                        next_dist * (b - l.float())).view(-1))

        return proj_dist
```

5.4.2 计算 loss

```
# 更新的distributional映射到对应的范围内 proj_dist=r+gamma*Z
proj_dist = self.projection_distribution(next_state, reward, done)
dist = self.current_model(state)
action = action.unsqueeze(1).unsqueeze(1).expand(
        self.batch_size, 1, self.num_atoms)
dist = dist.gather(1, action).squeeze(1)
dist.data.clamp_(0.01, 0.99)
# 使用KL散度计算loss = sum(p*log(p/q))
loss =  -((Variable(proj_dist) * dist.log()).sum(-1))
# 经验回放池的优先级同样使用KL散度进行更新
self.replay_buffer.update_priorities(
        indices, loss.detach().squeeze().abs().cpu().numpy().tolist())
loss = loss.mean()
```

6 噪声深度 Q 网络 (Noisy Nets) [6]

Noisy Nets 基于向输入中增加噪声，来促进模型的探索。

6.1 网络参数的噪声表示

假设网络参数为 θ，加入噪声的参数可以表示成如下公式

$$\theta = \mu + \sigma \odot \epsilon \tag{6.1}$$

其中，$\zeta = (\mu, \sigma)$ 为可学习的参数，ϵ 为维度相同以零为均值的随机噪声。

6.2 增加噪声的方法

(1) Independent Gaussian Noise

该方法是为每一个参数都添加一个独立的随机噪声

(2) Factorised Gaussian Noise

该方法为每一个神经元添加一个随机噪声，相比上一个方法节省计算量。

一般来说，针对 DQN 计算较为密集的算法，使用第二种方法；对于 A3C 可并行、更依赖于采样的算法，采用第一种方法。

6.3 损失函数的表示

$$\bar{L}(\zeta) = E[E_{(x,a,r,y)\sim D}[r + \gamma Q(y, b^*(y), \epsilon'; \zeta^-) - Q(x, a, \epsilon; \zeta)]^2] \tag{6.2}$$

其中，$b^*(y) = argmax_{b\in A} Q(y, b(y), \epsilon''; \zeta)$，在选择 a^* 时又采样了一次随机噪声。

6.4 关键代码展示

```
class NoisyLinear(nn.Module):
    """
        添加噪声的网络层
    """
    def __init__(self, in_features, out_features, use_cuda, std_init=0.4):
        """
            初始化NoisyLinear (mu+sigma*epsilon)
            weight_mu
            weight_sigma
            weight_epsilon
            bias_mu
            bias_sigma
            bias_epsilon
        """
        super(NoisyLinear, self).__init__()

        self.use_cuda    = use_cuda
        self.in_features  = in_features
        self.out_features = out_features
```

```
        self.std_init      = std_init
        # 噪声层中可学习的部分(mu,sigma)
        self.weight_mu     = nn.Parameter(
                             torch.FloatTensor(out_features, in_features))
        self.weight_sigma  = nn.Parameter(
                             torch.FloatTensor(out_features, in_features))
        # 噪声层中噪声部分(epsilon), 不可学习
        self.register_buffer('weight_epsilon', torch.FloatTensor(
                             out_features, in_features))
        # 每个变量的bias
        self.bias_mu       = nn.Parameter(torch.FloatTensor(
                             out_features))
        self.bias_sigma    = nn.Parameter(torch.FloatTensor(
                             out_features))
        self.register_buffer('bias_epsilon', torch.FloatTensor(
                             out_features))
        # reset
        self.reset_parameters()
        self.reset_noise()

    def forward(self, x):
        """
            前向传播
        """
        if self.use_cuda:
            weight_epsilon = self.weight_epsilon.cuda()
            bias_epsilon   = self.bias_epsilon.cuda()
        else:
            weight_epsilon = self.weight_epsilon
            bias_epsilon   = self.bias_epsilon

        if self.training:
            weight = self.weight_mu + self.weight_sigma.mul(
                                Variable(weight_epsilon))
            bias   = self.bias_mu   + self.bias_sigma.mul(
                                Variable(bias_epsilon))
        else:
            weight = self.weight_mu
            bias   = self.bias_mu

        return F.linear(x, weight, bias)

    def reset_parameters(self):
```

```
        """
            randomize parameters:(mu, sigma)
        """
        mu_range = 1 / math.sqrt(self.weight_mu.size(1))

        self.weight_mu.data.uniform_(-mu_range, mu_range)
        self.weight_sigma.data.fill_(self.std_init / math.sqrt(
                        self.weight_sigma.size(1)))

        self.bias_mu.data.uniform_(-mu_range, mu_range)
        self.bias_sigma.data.fill_(self.std_init / math.sqrt(
                        self.bias_sigma.size(0)))

    def reset_noise(self):
        """
            randomize noise:epsilon
        """
        epsilon_in  = self._scale_noise(self.in_features)
        epsilon_out = self._scale_noise(self.out_features)

        self.weight_epsilon.copy_(epsilon_out.ger(epsilon_in))
        self.bias_epsilon.copy_(self._scale_noise(self.out_features))

    def _scale_noise(self, size):
        """
            随机初始化大小为size的噪声
        """
        x = torch.randn(size)
        x = x.sign().mul(x.abs().sqrt())
        return x
```

7 Rainbow [7]

Rainbow 是由 DeepMind 提出，在 DQN 网络基础上，融合了以下 6 种改进的强化学习方法，具有 model-free、off-policy、value-based、discrete 的特点。

- Double Q-lerning
- Prioritized Replay
- Dueling Networks
- Multi-step Learning
- Distributed RL
- Noisy Nets

本次实验在 Atari 小游戏-PongNoFrameskip-v4 环境下，实现并训练了 Rainbow 和以上除了 Multi-step Learning 的深度学习方法，将在第 8 节详细分析实验结果。

7.1 Rainbow 模型构造介绍

首先，将 TD 的单步自举换成 n 步自举，则目的分布可以表示成

$$d_t^{(n)} = (r_t^{(n)} + \gamma_t^{(n)} z, p_{th\bar{e}ta}(s_t + n, a_{t+n}^*)) \tag{7.1}$$

对应的损失函数由 KL 散度表示为

$$loss = D_{KL}(\Phi_z d_t^{(n)} || d_t) \tag{7.2}$$

其中，Φ_z 表示将 $d_t^{(n)}$ 对齐到 d_t 相同的分布中。

接着，经验回放池对应的优先级 TD 误差也引入 KL 散度进行计算

$$p_t \propto (D_{KL}(\Phi_z d_t^{(n)} || d_t))^{\omega} \tag{7.3}$$

最后，将 Dueling Network 和 Distributional RL 的网络结构结合起来。

假设卷积层输出的状态特征为 ϕ，价值函数部分网络参数为 η，优势函数部分网络参数为 ψ，设 atoms 个数为 N。

则价值函数部分网络 v_η 输出为一个 N 维的向量；优势函数部分网络 a_ψ 为 $N \times |N_{actions}|$ 的矩阵，则每个 atoms 的概率表示如下

$$p_\theta^i(s,a) = \frac{exp(v_\eta^i(\phi) + a_\psi^i(\phi,a) - \bar{a}_\psi^i(s))}{\sum_j exp(v_\eta^j(\phi) + a_\psi^j(\phi,a) - \bar{a}_\psi^j(s))} \tag{7.4}$$

7.2 关键代码展示

7.2.1 神经网络设置

```
# 定义网络
# State: 需要经过卷积层, 提取feature特征
self.features = nn.Sequential(
        nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
        nn.ReLU(),
        nn.Conv2d(32, 64, kernel_size=4, stride=2),
        nn.ReLU(),
        nn.Conv2d(64, 64, kernel_size=3, stride=1),
        nn.ReLU()
)
# Noisy层NoisyLinear
# DuelingDQN = value+advantage
self.noisy_value1 = NoisyLinear(
        self.feature_size(), 512, use_cuda=USE_CUDA)
self.noisy_value2 = NoisyLinear(
        512, self.num_atoms, use_cuda=USE_CUDA)
```

```
self.noisy_advantage1 = NoisyLinear(
        self.feature_size(), 512, use_cuda=USE_CUDA)
self.noisy_advantage2 = NoisyLinear(
        512, self.num_atoms * self.num_actions, use_cuda=USE_CUDA)
```

7.3 前向传播

```
def forward(self, x):
        """
                前向传播
        """
        if type(x) == np.ndarray:
                x = torch.from_numpy(x)
        batch_size = x.size(0)
        x = x / 255.
        x = self.features(x)
        x = x.view(batch_size, -1) # reshape成batch_size个行
        # 求得价值函数部分value
        value = F.relu(self.noisy_value1(x))
        value = self.noisy_value2(value)
        # 求得优势函数部分advantage
        advantage = F.relu(self.noisy_advantage1(x))
        advantage = self.noisy_advantage2(advantage)
        # reshape
        value = value.view(batch_size, 1, self.num_atoms)
        advantage = advantage.view(
                batch_size, self.num_actions, self.num_atoms)
        # Q=V+A-A.mean()
        x = value + advantage - advantage.mean(1, keepdim=True)
        # 对计算出的分布计算softmax, 得到大小为action*atoms的矩阵
        x = F.softmax(x.view(-1, self.num_atoms)).view(
                -1, self.num_actions, self.num_atoms)

        return x
```

7.3.1 依据 epsilon-greedy 策略获得动作

```
def act(self, state, epsilon):
        """
                输入state, 获得当前最优的action
        """
        if random.random() > epsilon:
```

```
                state = Variable(torch.FloatTensor(
                        np.float32(state)).unsqueeze(0), volatile=True)
                # 计算动作价值函数的分布(1,num_actions,num_atoms)
                dist = self.forward(state).data.cpu()
                dist = dist * torch.linspace(
                        self.Vmin, self.Vmax, self.num_atoms)
                # 取峰值作为action
                action = dist[0].sum(1).max(0)[1].numpy()
        else:
                action = random.randrange(self.num_actions)
        return action
```

7.3.2 loss 计算和梯度反向传播

```
def compute_td_loss(self, batch_size, beta):
        # sample一组batch_size大小的样本
        state, action, reward, next_state, done, weights,\
        indices = self.replay_buffer.sample(batch_size, beta)

        # 更新的distributional映射到对应的范围内 proj_dist=r+gamma*Z
        proj_dist = self.projection_distribution(
                next_state, reward, done)
        dist = self.current_model(state)
        action = action.unsqueeze(1).unsqueeze(1).expand(
                self.batch_size, 1, self.num_atoms)
        dist = dist.gather(1, action).squeeze(1)
        dist.data.clamp_(0.01, 0.99)
        # 使用KL散度计算loss = sum(p*log(p/q))
        loss =  -((Variable(proj_dist) * dist.log()).sum(-1))
        # 经验回放池的优先级同样使用KL散度进行更新
        self.replay_buffer.update_priorities(
                indices, loss.detach().squeeze().abs().cpu().numpy().tolist())
        loss = loss.mean()
        # 梯度反向传播
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        # reset noise
        self.current_model.reset_noise()
        self.target_model.reset_noise()

        return loss
```

7.3.3 具体训练过程

```
# 进度条：循环遍历num_frames步
pbar = tqdm(range(1, self.num_frames+1))
for frame_idx in pbar:
        # 更新epsilon
        epsilon = epsilon_by_frame(frame_idx)
        # 依据epsilon-greedy策略选择动作
        action = self.current_model.act(state, epsilon)
        # 与环境交互一步
        next_state, reward, done, _ = self.env.step(action)
        # 将新的信息放入经验回放池
        self.replay_buffer.push(state, action, reward, next_state, done)
        # 更新状态
        state = next_state
        episode_reward += reward
        # 一轮episode结束
        if done:
                state = self.env.reset()
                all_rewards.append(episode_reward)
                episode_reward = 0
                episode += 1

        # 计算beta和loss
        if len(self.replay_buffer) > self.batch_size:
                beta = beta_by_frame(frame_idx)
                loss = self.compute_td_loss(self.batch_size, beta)
                losses.append(loss.item())

        # 更新target_Q = current_Q
        if frame_idx % 1000 == 0:
                self.update_target(self.current_model, self.target_model)
```

8 Atari 实验结果对比

8.1 实验环境

Anaconda 建立虚拟环境，选择 python=3.9 的版本，需要 pip 安装以下第三方库即可运行代码

- Pytorch
- torchvision
- torchaudio
- pytorch-cuda=11.6
- matplotlib

- IPython
- opencv-python
- pyglet
- ale-py
- gym==0.19.0
- atari_py==0.2.6

本实验选择的 Atari 小游戏，env_id="PongNoFrameskip-v4"，是乒乓球小游戏中的一个版本，跳帧为 1，重复动作的概率为 0。在游戏中，玩家可以垂直移动挡板，通过使用挡板来回击球，来与另一侧的玩家竞争。目标是先于对手达到 11 分；当一方未能将球还给另一方时，将获得 1 个积分。由于先得到 21 积分的一方获得胜利，因此该游戏的 reward 范围为 $r \in [-21, 21]$。

其余游戏的相关信息如下表所示

名称	数值
动作空间	Discrete(18)
观察空间	(210, 160, 3)
奖励范围	[-21,21]

8.2 7 种强化学习方法的实验结果

本实验针对要求的 7 种模型，在"PongNoFrameskip-v4" 环境下进行训练，得到 eposide-rewards 图像结果如下。

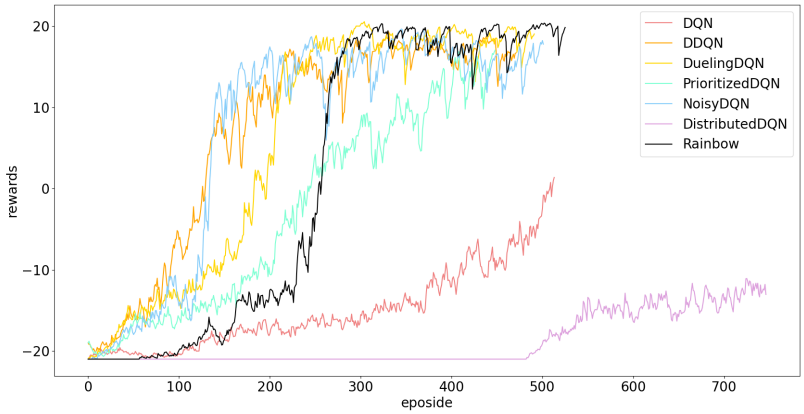

图 8-1 7 种强化学习网络模型实验结果

由上图可以看出，在该游戏环境下，Noisy Nets 表现最佳，Dueling DQN 和 Double DQN 也表现不错，基本都能在相对较少的 eposide 下收敛到游戏最优的结果。相比之下，DQN 和 Distributional DQN 表现较差，训练相同的 epoch 的情况下仍未收敛到最优的 reward。Rainbow 则在该游戏中表现一般，虽然能够最终收敛到最优，但其收敛速度较慢。

为了分析该实验结果，打印 DQN、DDQN、DuelingDQN、PER、NoisyDQN、DistributionalDQN 训练过程中的 loss 值，具体如下图所示。

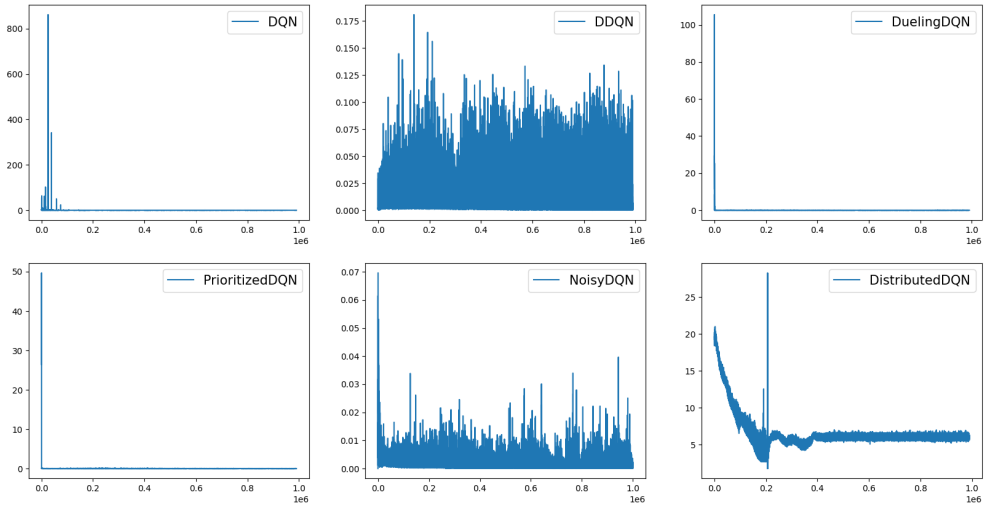

图 8-2 6 种强化学习方法的 loss 变化

课上展示的 Rainbow 在多个 Atari 游戏上表现比其他强化学习算法优秀许多，与笔者自己的实验结果有较大的差别，为此笔者总结了自己的一些思考，具体如下：

（1）distributional 的主要在多峰分布上占有优势，而由于 pong 的游戏环境较为单一且简单，实际上需要采取的最优策略可能符合高斯分布，即单峰。因此，distributional 的分布计算反而增加了计算消耗。

（2）Rainbow 文章中的结果是在多个 Atari 小游戏上的平均 reward 的归一化结果，本文仅在其中一个简单的小游戏上进行训练，该游戏不能充分展现和发挥出 Rainbow 的优势。

（3）相比之下，可以看出 DDQN、NoisyDQN 小体量的模型，不仅收敛速度快，且结果表现良好。所以并不是越复杂的模型，其表现愈加出色。

8.3 消融实验

为了探究 Rainbow 各个部分起到的作用程度，本文在相同的实验环境下对 rainbow 进行了消融实验，具体结果如下。

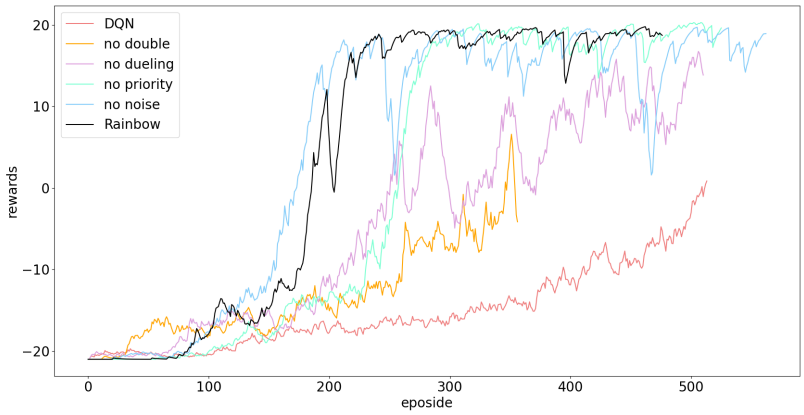

图 8-3 Rainbow 消融实验结果

由图可以看出，Double DQN 和 Dueling DQN 的部分对 Rainbow 的影响最大，其对于网络结构的改进帮助较大。相比之下，去掉 priority 后只是模型收敛速度减慢，验证了带优先级的经验回放池能够 sample 出更有利于网络更新的样本，从而加快收敛速度；而去掉 noise 后模型收敛速度反而增快，但是其游戏能力浮动较大，这可能是由于该游戏环境较为简单，噪声对于模型的鲁棒性帮助不大。

参考文献

[1] Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning[J]. nature, 2015, 518(7540): 529-533.

[2] Van Hasselt H, Guez A, Silver D. Deep reinforcement learning with double q-learning[C]//Proceedings of the AAAI conference on artificial intelligence. 2016, 30(1).

[3] Schaul T, Quan J, Antonoglou I, et al. Prioritized experience replay[J]. arXiv preprint arXiv:1511.05952, 2015.

[4] Wang Z, Schaul T, Hessel M, et al. Dueling network architectures for deep reinforcement learning[C]//International conference on machine learning. PMLR, 2016: 1995-2003.

[5] Bellemare M G, Dabney W, Munos R. A distributional perspective on reinforcement learning[C]//International Conference on Machine Learning. PMLR, 2017: 449-458.

[6] Fortunato M, Azar M G, Piot B, et al. Noisy networks for exploration[J]. arXiv preprint arXiv:1706.10295, 2017.

[7] Hessel M, Modayil J, Van Hasselt H, et al. Rainbow: Combining improvements in deep reinforcement learning[C]//Thirty-second AAAI conference on artificial intelligence. 2018.