RL 上机实验 2

1120200822 郑子帆

2022.11.09

目录

1 实验内容概述

用 Rainbow 完成雅达利小游戏：

- 深度 Q 网络
- 双深度 Q 网络
- 优先级经验回放
- 竞争双深度
- 分布式深度 Q 网络
- 噪声深度 Q 网络

2 实验环境搭建

2.1 实验硬件配置

在本次实验，我用自己的电脑进行了程序的编写和运行，具体配置如下：

- 硬件配置：Intel(R)Core(TM)i7-10750H CPU @ 60GHz，6 个内核
- 独立显卡：NVIDIA GeForce RTX 2080s
- 操作系统：Microsoft Windows 10 家庭中文版
- 代码编辑器：Microsoft Visual Studio Code 2020

2.2 gym、atari 环境搭建

由于在做实验 1 时，我安装的是 0.19.0 版本的 gym，通过上网查询资料后，了解到老版本中的 gym 导入的是 atari_py 包而新版本中导入的是 ale_py 包，所以在我最开始通过指令 *pip install gym[atari]* 安装相关 atari 依赖包后运行程序会有编译报错（找不到 ale_py）。于是我卸载了 gym 并通过指令 *pip install gym[atari,accept-rom-license]==0.21.0* 安装了新版的 gym 及 atari 相关依赖包，为了避免 RunTime Error 的问题，可以输入指令 *pip install autorom* 辅助导入游戏的 roms。通过 anaconda 的环境管理，可以看到最后的包的版本如下：

Package	Version
absl-py	1.0.0
alabaster	0.7.12
ale-py	0.7.5
altgraph	0.17.2
anaconda-client	1.9.0
anaconda-navigator	2.1.1
anaconda-project	0.10.1
gym	0.21.0
gym-notices	0.0.8

图 1: 相关安装包的版本

至此，实验环境搭建成功完成。

3 雅达利 (Atari) 游戏介绍

3.1 Atari2600

gym 官网的介绍文档网址为：https://www.gymlibrary.dev/environments/atari/ gym 中的雅达利小游戏是雅达利公司在上世纪 70-80 年代出品的一些经典小游戏，收录了 Atari2600 游戏机中的 61 个小游戏。

一些细节如下：

- 动作空间：完整的动作空间是 18 个离散值（0 17），简化的动作空间是 6 个离散值（0 5），但是在 v0-v4 中都是简化的动作空间，只有 v5 是完整的动作空间；
- 观测空间：返回的是一个 210 × 160 × 3 的 RGB 图像（一帧）；
- 环境版本：除了 v0-v5，还可以加参数 frameskip=True/False，若为 True 则不跳过每一帧图像；默认为 False，即智能体和环境交互时每 4 帧图像返回一次。

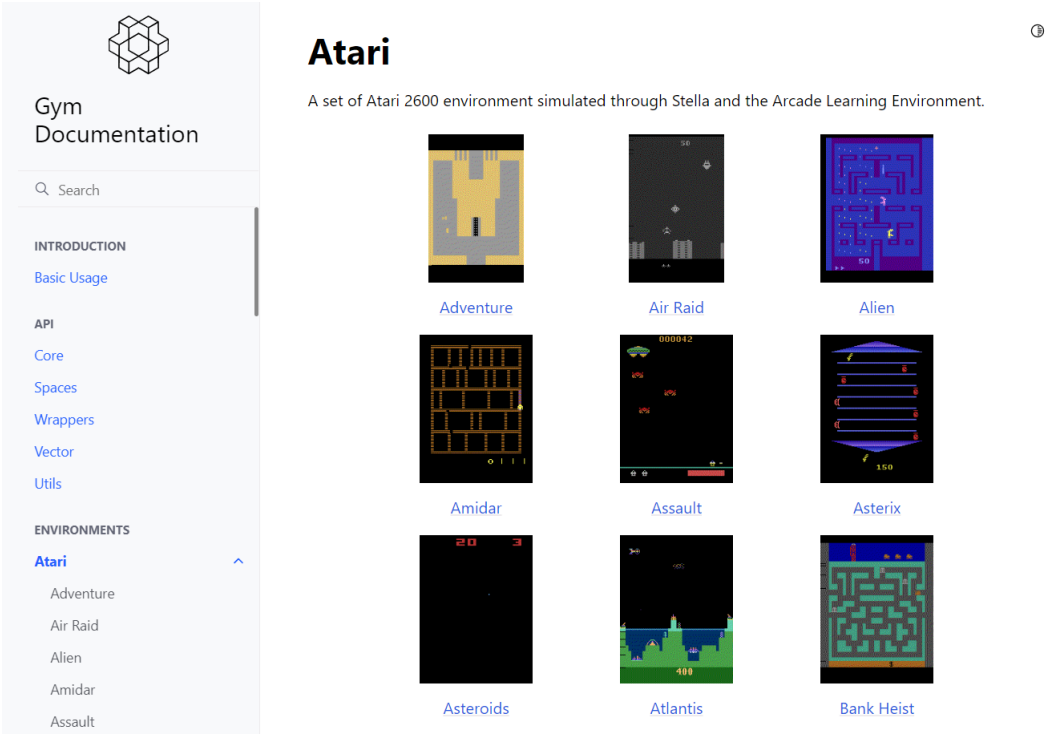


图 2: Atari 游戏合集

3.2 Pong 游戏

在本次实验中，我先后尝试了 breakout、Assault 等游戏，最后选择了 Pong 游戏。Pong 游戏是一个零和博弈游戏 (Zero-sum game)，两位玩家分别在屏幕的两侧操控一个拍子上下移动，其中会有一个小球在屏幕区域内直线运动，碰到上下边界或者两位玩家的即反弹。若有一方未能用拍子将小球打回，则对方得一分，先得到 21 分的玩家获胜。

在实际编写代码中，我最开始用的是 PongNoFrameskip-v4 环境，但训练速度实在太慢，故后全部改为 Pong-v4 环境，具体的设置如下：

- 状态：Pong 环境提供的状态默认是 Box(210, 160, 3), 也就是 3 通道的彩色图

- Pong-v0 和 Pong-V4 版本返回的动作都是 Discrete(6)，也就是离散的 6 个动作。其实 6 个动作中有用的只有 3 个，分别是左，右和不动（其中 Left 和 Leftfire 是一样的，Right 和 Rightfire 是一样的，Noop 和 Fire 是一样的）。
- 奖励：reward 的设计十分简单。自己得分则 reward=1，对手得分则 reward=-1，否则 reward=0。

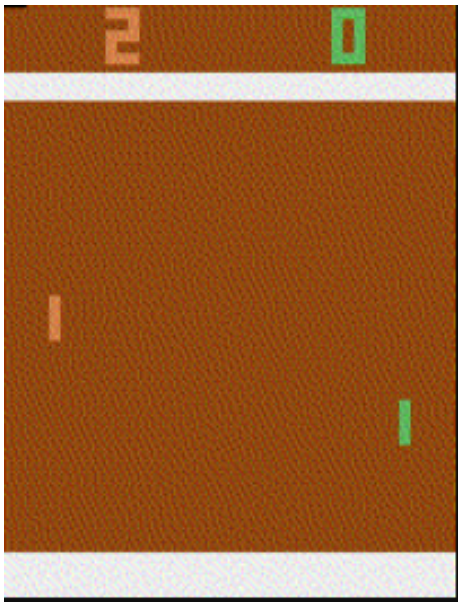


图 3: Pong 游戏截图

4 Rainbow 方法介绍

在《EasyRL》书中有 DQN 算法及其改进算法的讲解，网上也有很多资料。

总的来说，DDQN 是在目标值上做了一定的更改，用会更新参数的 Q 网络（下面简称为 current net）选出针对 s_{t+1} 的 Q 值最大的动作。PER 升级了经验回放池，dueling DQN 和 distributional DQN 改进了网络的输出结构，noisy DQN 通过为网络参数添加噪声实现随机尝试。

4.1 深度 Q 网络 (deep Q-network)

DQN 算法基于深度学习的 Q 学习算法：

1. DQN 中不再是 Q 表格，而是“真正的”Q 函数，它的 Q 函数值利用神经网络（比如最基本的多层感知机 MLP、对于图像特征提取可以用 CNN）进行逼近求解。
2. 在原来的 Q-learning 算法中，每一个数据只会用来更新一次值。DQN 算法采用了经验回放（experience replay）方法，具体做法为维护一个回放缓冲区 (replay buffer)，将每次从环境中采样得到的五元组数据（状态、动作、奖励、下一状态、done）存储到回放缓冲区中，训练 Q 网络的时候再从回放缓冲区中随机采样若干数据来进行训练。这样可以使样本满足独立假设，并且经验回放池中的数据通常并不只来自一个策略，性质较为多样，保证了智能体更多的探索性，学习效果往往会更好。
3. DQN 利用了卷积神经网络，其更新方法是 SGD，因此值函数更新从 Q-Learning 中的增量更新实际上变成了监督学习的一次梯度更新的过程。

网络训练的过程遵循以下公式：

$$Q_\pi\left(s_t, a_t\right)=r_t+Q_\pi\left(s_{t+1}, \pi\left(s_{t+1}\right)\right)$$

如下图，在训练过程中我们希望等式左侧的结果逐渐向右侧逼近，但是如果等式两侧的结果出自同一个模型，每次迭代过程等式两侧的结果都在不断变化，会导致学习的不稳定。所以可以先固定住右侧，只更新左侧网络的参数，在一定次数后再将右侧网络的参数更新为左侧的。此时右侧网络就称为目标网络。

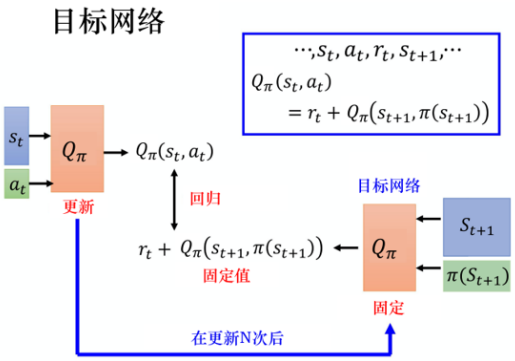


图 4: DQN 模型图

4.2 双深度 Q 网络 (double DQN)

本来是用目标网络 Q 来找使 Q 值最大的 a，现在改成用另外一个会更新的 Q 网络来找使 Q 值最大的 a。

4.3 优先级经验回放 (prioritized experience replay, PER)

假设有一些数据，我们之前采样过，发现这些数据的时序差分误差特别大（时序差分误差就是网络的输出与目标之间的差距），这代表我们在训练网络的时候，这些数据是比较不好训练的。既然比较不好训练，就应该给它们比较大的概率被采样到，即给它优先权（priority）。这样在训练的时候才会多考虑那些不好训练的数据，当然，这样也会改变数据的分布，通过调整 β 值可以调整分布偏差。

4.4 竞争双深度 Q 网络 (dueling DQN)

相较于原来的深度 Q 网络，它唯一的差别是改变了网络的架构。Q 网络输入状态，输出的是每一个动作的 Q 值。原来的深度 Q 网络直接输出 Q 值，竞争深度 Q 网络不直接输出 Q 值，而是分成两条路径运算。第一条路径会输出一个标量 V(s)，因为它与输入 s 是有关系的，所以称为 V (s)。第二条路径会输出一个向量 A(s, a)，它的每一个动作都有一个值。我们再把 V (s) 和 A(s, a) 加起来就可以得到 Q 值 Q(s, a)。

实现时，我们要给这个 A(s, a) 一个约束，通过归一化增强网络的更新。

4.5 彩虹 (rainbow)

将所有的 DQN 及其变种方法组合在一起。

5 Rainbow 完成 Cartpole 游戏

由于雅达利小游戏训练时用的是 CNN 结构的网络，所以训练速度、收敛速度较慢，不容易看出理想的结果。为了能够用 Rainbow 方法得到一个预想中的理想的结果，本实验在跑 Pong 之前先尝试了 Cartpole。

5.1 CartPole 游戏简介

在 gym 的官方文档中，可以找到 CartPole 游戏的介绍。

5.1.1 游戏介绍及目标

Cartpole，即车杆游戏，是一个经典的控制类问题。模型由一辆车和车上的一个竖直的杆在 2 维平面构成，杆通过一个车上安装的转轴与车相连，可以向平面内的任意方向进行摆动。

游戏目标为让智能体控制小车的左右移动，使得杆保持直立的时间尽可能长。

5.1.2 实验环境介绍

本实验使用 gym 库中的 Cartpole-v0 环境。其中状态共有四维，具体值如下。

数值	观测	最小值	最大值
0	车位置	-2.4	2.4
1	车速	-inf	inf
2	杆角度	-12 度	12 度
3	杆顶端的速度	-inf	inf

动作有两种，分别为向左和向右，用 0 和 1 表示。

小车每走一步，包括终止步骤，奖励均为 1。

游戏满足如下条件之一则结束。

1. 杆角度超出范围。
2. 小车的中心位置超出边界。
3. 小车累计步数大于 200。

5.2 7 个模型的 agent 实现

5.2.1 DQN

首先经验回放池的实现如下：

```python
from collections import deque

class ReplayBuffer(object):
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):    # 放入经验
        state      = np.expand_dims(state, 0)
        next_state = np.expand_dims(next_state, 0)

        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):          # 随机取batch_size个经验
        state, action, reward, next_state, done = zip(*random.sample(self.buffer, batch_size))
        return np.concatenate(state), action, reward, np.concatenate(next_state), done

    def __len__(self):  # 经验池长度
        return len(self.buffer)
```

图 5: Replay buffer

计算时序差分误差及梯度下降的实现如下：

```python
def get_td_loss(batch_size):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)

    state      = Variable(torch.FloatTensor(np.float32(state)))
    next_state = Variable(torch.FloatTensor(np.float32(next_state)), volatile=True)
    action     = Variable(torch.LongTensor(action))
    reward     = Variable(torch.FloatTensor(reward))
    done       = Variable(torch.FloatTensor(done))

    q_values      = model(state)        # q值
    next_q_values = model(next_state)   # 下一状态的q值

    q_value          = q_values.gather(1, action.unsqueeze(1)).squeeze(1)
    next_q_value     = next_q_values.max(1)[0]
    expected_q_value = reward + gamma * next_q_value * (1 - done)   # 期望的q值

    loss = (q_value - Variable(expected_q_value.data)).pow(2).mean()

    optimizer.zero_grad()
    loss.backward()     # 梯度下降
    optimizer.step()

    return loss
```

图 6: td loss

DQN agent 类的定义和实现如下：

```python
class DQN(nn.Module):
    def __init__(self, num_inputs, num_actions):
        super(DQN, self).__init__()

        self.layers = nn.Sequential(
            nn.Linear(env.observation_space.shape[0], 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, env.action_space.n)
        )

    def forward(self, x):
        return self.layers(x)

    def act(self, state, epsilon):
        if random.random() > epsilon:
            state   = Variable(torch.FloatTensor(state).unsqueeze(0), volatile=True)
            q_value = self.forward(state)
            action  = q_value.max(1)[1].data[0]
        else:
            action = random.randrange(env.action_space.n)
        return action
```

图 7: DQN class

经过每 1000 帧，将当前网络的参数更新到目标网络上，实现如上：

```python
def update_target(current_model, target_model):
    target_model.load_state_dict(current_model.state_dict())
```

图 8: upgrade

5.2.2 DDQN

对于 DDQN，改变了梯度更新时选下一状态对应的动作的方法，即用 current net 算得的 Q 值，改动不大，故此处不粘贴代码。

5.2.3 PER

对于 PER，最大的改变即将经验回放池升级成了优先级经验回放池。它的实现基于数据结构——线段树，它可以在 $O(log)$ 的时间内求出一段子区间内的和或者最大/小值，利用这一数据结构和给经验加优先级，类主要函数实现如下：

```
def push(self, *args, **kwargs):
    """See ReplayBuffer.store_effect"""
    idx = self._next_idx
    super(PrioritizedReplayBuffer, self).push(*args, **kwargs)
    self._it_sum[idx] = self._max_priority ** self._alpha
    self._it_min[idx] = self._max_priority ** self._alpha

def _sample_proportional(self, batch_size):
    res = []
    for _ in range(batch_size):
        # TODO(szymon): should we ensure no repeats?
        mass = random.random() * self._it_sum.sum(0, len(self._storage) - 1)
        idx = self._it_sum.find_prefixsum_idx(mass)
        res.append(idx)
    return res
```

图 9: prioritized replay buffer

5.2.4 dueling DQN

该算法改变了 Q 网络的输出，故网络结构有所改变，实现如下：

```
    self.feature = nn.Sequential(
        nn.Linear(num_inputs, 128),
        nn.ReLU()
    )

    self.advantage = nn.Sequential(
        nn.Linear(128, 128),
        nn.ReLU(),
        nn.Linear(128, num_outputs)
    )

    self.value = nn.Sequential(
        nn.Linear(128, 128),
        nn.ReLU(),
        nn.Linear(128, 1)
    )

def forward(self, x):
    x = self.feature(x)
    advantage = self.advantage(x)
    value     = self.value(x)
    return value + advantage  - advantage.mean()
```

图 10: 网络结构变化

5.2.5 distributional DQN

分布式主要将 Q 网络输出由值改成一个分布，实现如下：

```python
class DistributionalDQN(nn.Module):
    def __init__(self, num_inputs, num_actions, num_atoms, Vmin, Vmax):
        super(DistributionalDQN, self).__init__()

        self.num_inputs = num_inputs
        self.num_actions  = num_actions
        self.num_atoms    = num_atoms
        self.Vmin         = Vmin
        self.Vmax         = Vmax

        self.linear1 = nn.Linear(num_inputs, 128)
        self.linear2 = nn.Linear(128, 128)
        self.noisy1 = NoisyLinear(128, 512)
        self.noisy2 = NoisyLinear(512, self.num_actions * self.num_atoms)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = F.relu(self.noisy1(x))
        x = self.noisy2(x)
        x = F.softmax(x.view(-1, self.num_atoms)).view(-1, self.num_actions, self.num_atoms)
        return x
```

图 11: distributional class

5.2.6 noisy DQN

将 Q 网络中的参数添加高斯噪声，实现如下：

```python
class NoisyLinear(nn.Module):
    def __init__(self, in_features, out_features, std_init=0.4):
        super(NoisyLinear, self).__init__()

        self.in_features  = in_features
        self.out_features = out_features
        self.std_init     = std_init

        self.weight_mu    = nn.Parameter(torch.FloatTensor(out_features, in_features))
        self.weight_sigma = nn.Parameter(torch.FloatTensor(out_features, in_features))
        self.register_buffer('weight_epsilon', torch.FloatTensor(out_features, in_features))

        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))

        self.reset_parameters()
        self.reset_noise()

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

        return F.linear(x, weight, bias)
```

图 12: noisy linear

5.2.7 rainbow

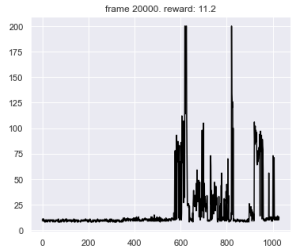
将上面结合起来即为 Rainbow。

5.3 7 个模型的实验结果

对于 CartPole 游戏，我在自己的本机进行了运行。参数值设定如下（部分参数只在特定模型中有）：

超参数	含义	设定值
epsilon_start	ϵ 初始值	1.0
epsilon_final	ϵ 最终值	0.01
epsilon_decay	ϵ 衰减率	500
epsilon_by_frame	当前帧 ϵ 值	-
num_frames	总训练帧数	20000
batch_size	训练批量大小	32
gamma	折扣率	0.99
lr	学习率	0.005
alpha	经验优先率	0.6
beta_start	重要性采样偏差初始值	0.4
beta	重要性采样偏差	-
num_atoms	Q 函数值域划分区间数	51
Vmin	限定值下限	-10
Vmax	限定值上限	10

分别运行代码，可以得到如下各模型训练出来的 agent 随每轮训练的得分 (横坐标:episodes, 纵坐标:scores) 和每帧的 MSE(均方误差) 值:



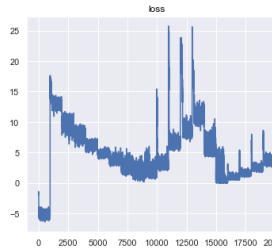
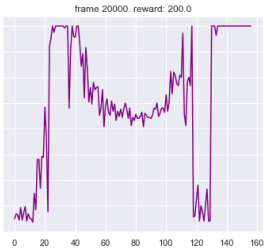


图 13: DQN 训练结果



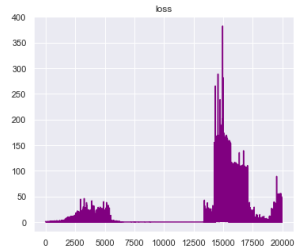


图 14: DDQN 训练结果

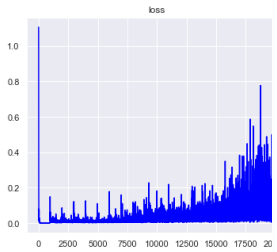
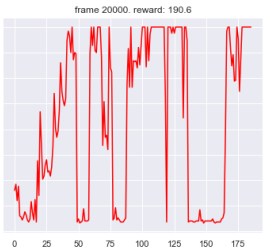


图 15: PER 训练结果



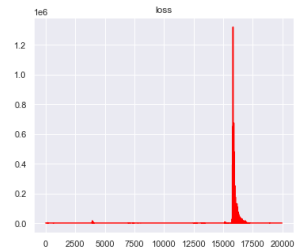
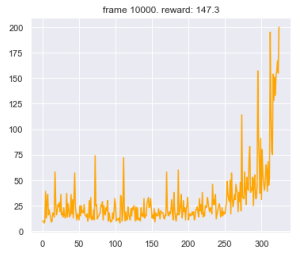


图 16: dueling DQN 训练结果



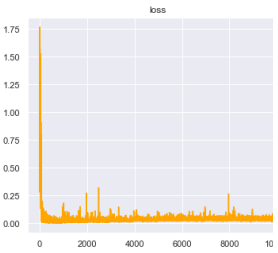
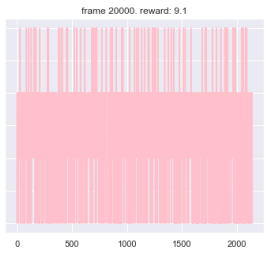


图 17: distributional DQN 训练结果



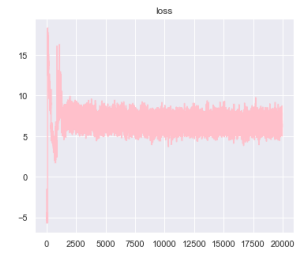


图 18: noisy DQN 训练结果

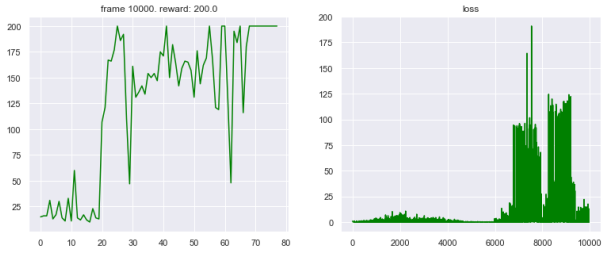


图 19: Rainbow 训练结果

对比各图，可以发现，DDQN、PER、dueling DQN 和 Rainbow 方法的训练效果很好，但是 DQN 和 distributional DQN 的训练效果并不好。合成彩虹图如下：

图 20: 彩虹图-折线

可以看到，上面这个折线图上下波动太大，影响直观感受，故我们考虑用相邻两项取平均和平滑曲线 (上一得分 *0.8+ 当前得分 *0.2) 的方式修正此图，结果展示如下：

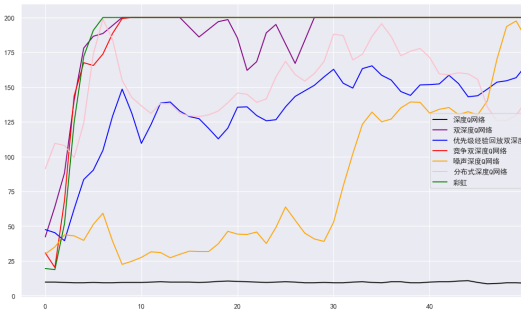


图 21: 彩虹图-相邻平均值

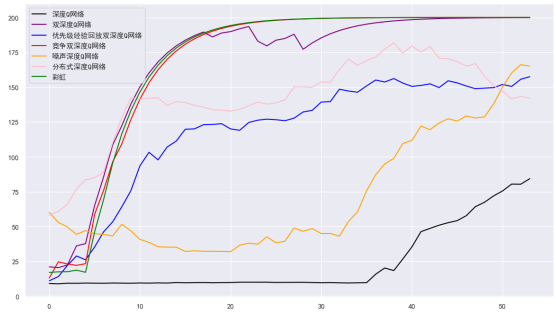


图 22: 彩虹图-平滑曲线

上面两图可以直观地看出 rainbow 方法十分优秀，当然，DDQN 和 dueling DQN 表现也同样出色。

6 Rainbow 完成雅达利

6.1 7 个模型的 agent 实现

在 5.2 节中，讲解了 7 个模型的主要优化方法，在雅达利游戏中，因为观察到的是 RGB3 色图像，故我们将网络架构中的全连接层换成卷积核大小分别为 8、4、3 的 2D 卷积网络，并在后

面增添两个全连接层。其他地方基本没有改动。

```
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()
)
```

图 23: CNN 架构

6.2 7 个模型的实验结果

对于雅达利游戏，因为训练比较慢，故我在智星云上租了一个云主机，搭载 windows10，cuda11，配有 2 个 cpu 内核和一个 RTX 2080Ti 的 GPU，用它进行模型训练。经过长达一周的调试和删改，最终参数值设定如下（部分参数只在特定模型中有）：

超参数	含义	设定值
epsilon_start	ϵ 初始值	1.0
epsilon_final	ϵ 最终值	0.01
epsilon_decay	ϵ 衰减率	30000
epsilon_by_frame	当前帧 ϵ 值	-
num_frames	总训练帧数	600000
batch_size	训练批量大小	32
gamma	折扣率	0.99
lr	学习率	0.001
alpha	经验优先率	0.6
beta_start	重要性采样偏差初始值	0.4
beta	重要性采样偏差	-
num_atoms	Q 函数值域划分区间数	51
Vmin	限定值下限	-10
Vmax	限定值上限	10

分别运行代码，可以得到如下各模型训练出来的 agent 随每轮训练的得分 (横坐标:episodes, 纵坐标:scores)：

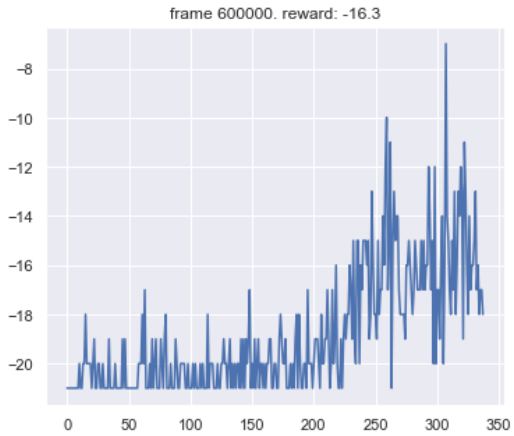


图 24: DQN 训练结果

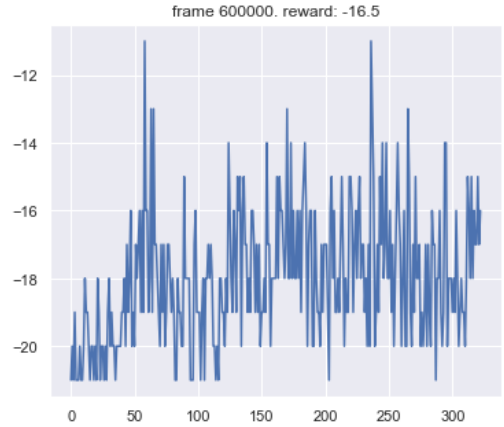


图 25: DDQN 训练结果

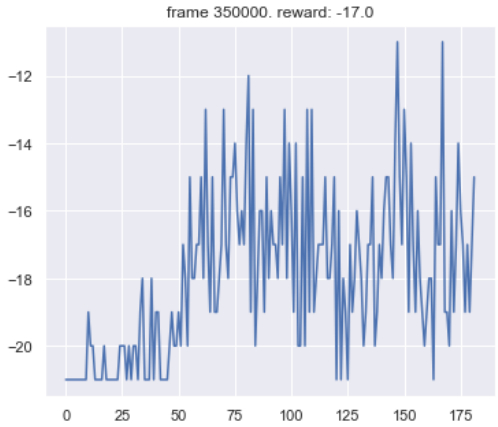


图 26: PER 训练结果

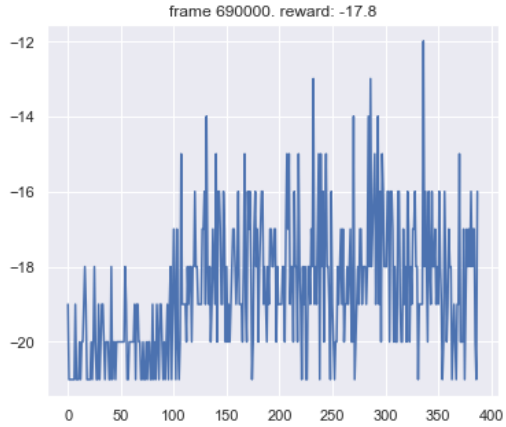


图 27: dueling DQN 训练结果

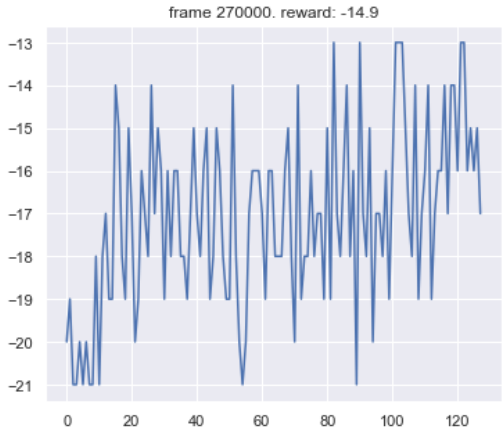


图 28: distributional DQN 训练结果

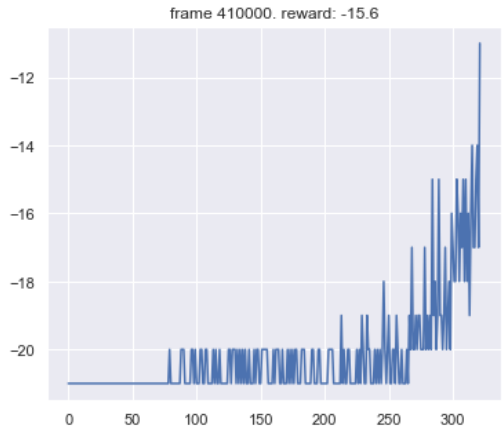


图 29: noisy DQN 训练结果

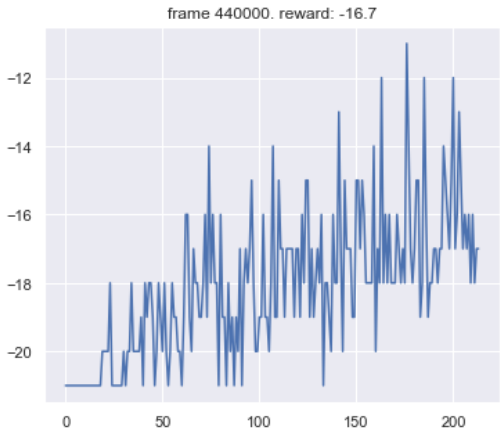


图 30: Rainbow 训练结果

对比各图，可以发现，在都运行 15h 左右的情况下，DQN、DDQN 和 dueling DQN 训练速度较快，而 noisy DQN 较慢；对于训练效果，DQN 得分随训练轮数上升较为稳定，noisy DQN 和 Rainbow 训练的效果相比其他的训练效果稍好。但是由于训练轮数实在偏小，所以结果并没有很强的说服性，在这里也没能看出 Rainbow 相比其他几种方法的明显优势。

同样地，我们汇总得到了一个彩虹图如下：

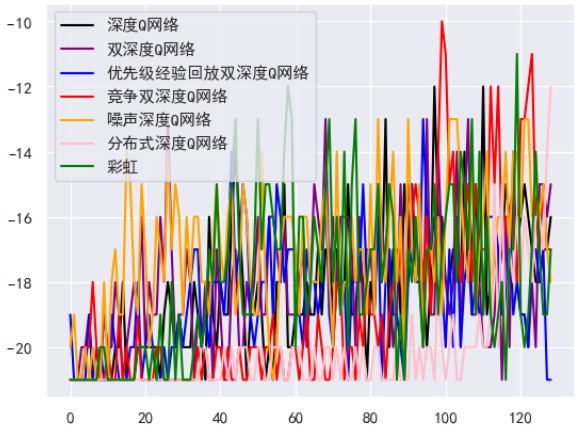


图 31: 彩虹图-折线

还是由于纵坐标范围较小，折线波动较大，这张图视觉效果并不好，故我采用了和上面 Cartpole 一样的方法绘出了平滑曲线，结果如下：

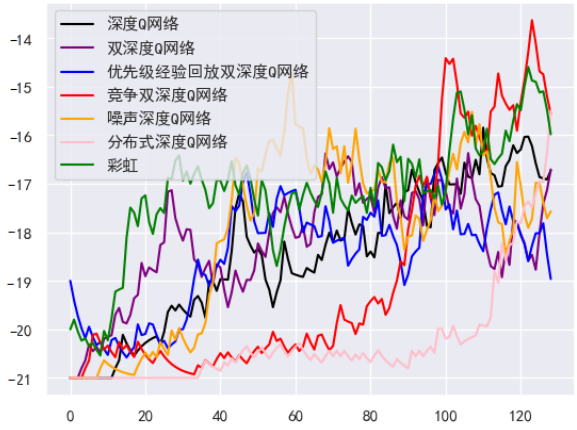


图 32: 彩虹图-平滑曲线

由上图还是可以看到，虽然 rainbow 没有足够的优势，但是训练效果还是很不错的。另外，和 Cartpole 有相同的问题，distributional DQN 的训练效果依然不好，可能还是跟参数设置有一定的关系，而且可能经过大量轮数才能体现出它的优势。

7 写在最后

7.1 说明与总结

由于时间原因和生活费有限，本人仅租了一台 gpa 主机并耗时 10 天分别跑了 7 个算法的 Pong 游戏和 CartPole 游戏。在《EasyRL》书中，各个算法分别跑了 2 亿帧，这个数量对于我来说非常庞大，跑出来不太现实，故选择了 60 万帧，大概一个 agent 需要跑 20h 左右。在和张远方同学的交流过程中，了解到如果把 agent 网络结构由 CNN 换成 MLP（就像上面跑 CartPole 那样），我们可以大大加速训练效率，究其原因，大概因为 MLP 的参数量远远小于 CNN 的参数量，

所以在 optimizer 更新的时候速度会大幅提升。若用 MLP 网络结构，对于一个 agent，大概 3-4h 就可以完成 1500 轮左右的训练，且从得分曲线来看训练效果并没有和 CNN 网络结构有明显差异。不过因为轮数不够，可以看到曲线还没有收敛，所以收敛后也许 MLP 的效果不如 CNN 好。

图 33: 跑一次至少要 13-14h

另外，由于我还处于 RL 的初级学习阶段，这次上机实验的代码实现对我来说有很大的难度，尤其是分布式 DQN 的编写和 Rainbow 组合的实现。所以本人实现的代码参考了 Github 网站上的开源代码（网址：`https://github.com/Kaixhin/Rainbow`）和 DeepMind 最初发表的论文：A Distributional Perspective on Reinforcement Learning和Rainbow: Combining Improvements in Deep Reinforcement Learning，其中 SegmentTree 数据结构类的代码直接用了 Openai 的开源代码。同时感谢黄昱欣、张远方、王九铮等同学在本次上机实验中给予我的帮助和相互之间的探讨讨论，这些都让我对于 DQN 及其改进算法有了更深的认识，促进我这次上机实验更好地完成和 RL 的学习。

当然，这次实验还有一些遗憾，比如自己由于算力、时间、自身能力等问题没能完成消融实验 (Ablation Experiment)，没能进一步分析出不同改进对于 agent 训练速度、训练效果的影响强弱，在以后的学习中可以尝试完成一下。

8 提交文件说明

1120200822-郑子帆-上机实验 2.pdf: 为本文件，实验报告
代码:
|———1.DQN.ipynb: DQN 的代码
|———2.DDQN.ipynb: DDQN 的代码
|———3.PER.ipynb: PER DQN 的代码
|———4.dueling DQN.ipynb: dueling DQN 的代码
|———5.distributional DQN.ipynb: distributional DQN 的代码
|———6.noisy DQN.ipynb: noisy DQN 的代码
|———7.rainbow.ipynb: rainbow 方法的代码
|———1.csv: DQN 跑 CartPole/Pong 的实验结果
|———2.csv: DDQN 跑 CartPole/Pong 的实验结果
|———3.csv: PER 跑 CartPole/Pong 的实验结果
|———4.csv: dueling DQN 跑 CartPole/Pong 的实验结果
|———5.csv: distributional DQN 跑 CartPole/Pong 的实验结果
|———6.csv: noisy DQN 跑 CartPole/Pong 的实验结果

|———7.csv: rainbow 跑 Cartpole/Pong 的实验结果

|———layers.py: NoisyLinear 类的实现

|———replay_buffer.py: 经验回放池和优先级经验回放池类的实现

|———plot_in_rainbow.py: 根据 csv 存放的实验结果画图