Mid-term Assignments: DQN

课程名称：高级算法	任课老师：陈旭
学号：18340199	姓名：余傲泰
完成日期：2020/11/18	邮箱：1733157506@qq.com

一、算法原理

1.1 Deep Q-learning Network(DQN)

DQN 是 Q-learning 的改进版本，二者都是基于值迭代的算法，但是在普通的 Q-learning 中，当状态和动作空间是离散且维数不高时可使用 Q-Table 储存每个状态动作对的 Q 值：

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R' + \lambda \underset{a}{max}\, Q(s',a) - Q(s,a))$$

而当状态和动作空间是高维连续时，使用 Q-Table 就十分困难（在本例 Atari: Breakout game 中状态多达 $256^{210\times160}$ 种）。所以可以把 Q-table 更新转化为一函数拟合问题，通过拟合一个函数来代替 Q-table 产生 Q 值，使得相近的状态得到相近的输出动作，这种方法称为价值函数近似（Value Function Approximation）。在 DQN 中使用神经网络作为估计函数。

1.2 Loss Function

上文提到 DNQ 使用的神经网络具体为卷积神经网络（CNN），CNN 可以将高维的数据映射到较低维的矩阵中，很好地实现了数据降维，减小了训练的计算量。神经网络的训练是一个最优化问题，是对损失函数 Loss Function 的最优化。

$$L(w) = \mathrm{E}[(r + \gamma \underset{a'}{max}\, Q(s',a',w) - Q(s,a,w))^2]$$

这个公式表面上看起来很复杂，实际上很好理解，它就是一个残差模型，和我们平常见的最小二乘法很类似，真实值与预测值之间的差的平方。为了使损失函数最小化，我们需要大量的样本进行训练，通过反向传播使用梯度下降算法来更新卷积神经网络的参数。所以，我们利用 Q-learning 算法为 Q 网络提供有标签的样本，让每个状态的 Q 值去逼近目标的 Q 值。

1.3 Experience Replay

在 DQN 中，提出了经验池的概念，功能主要是解决相关性及非静态分布问题。具体做法是把每个时间步 agent 与环境交互得到的转移样本，形式为：

$$(S_t,\ A_t,\ R_{t+1},\ S_{t+1})$$

分别表示当前状态，采取的行为，获得的收益，下一个状态，储存到回放记忆单元，然后在经验池中随机采样，以 min-batch 读取训练网络结构，以解决相关性的问题。使用 Experience Replay 的原因及优点：

- 深度神经网络作为有监督学习模型，要求数据满足独立同分布。
- 通过存储-采样的方法打破了 Q-learning 算法得到的样本之间的关联性。
- 数据利用率高，因为一个样本被多次使用。
- 连续样本的相关性会使得参数更新的方差比较大，该方法可以减少这种相关性。

二、算法流程

NIPS 2013 版 DQN 算法伪代码[1]：

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

三、代码分析

常量定义，见注释：

```
GAMMA = 0.99                    # 衰减因子
GLOBAL_SEED = 0                 # 随机数种子
MEM_SIZE = 100_000              # 经验池大小
STACK_SIZE = 4

EPS_START = 1.
EPS_END = 0.1
EPS_DECAY = 1000000

BATCH_SIZE = 32
POLICY_UPDATE = 4               # policy Q-function 更新 step
TARGET_UPDATE = 10_000          # target Q-function 更新 step
WARM_STEPS = 50_000
MAX_STEPS = 50_000_000          # step 总数
EVALUATE_FREQ = 100_000         # 评估 step
```

环境定义，主要是配置 CPU 或者 GPU：

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
env = MyEnv(device)
agent = Agent(
    env.get_action_dim(),
    device,
    GAMMA,
    new_seed(),
    EPS_START,
    EPS_END,
    EPS_DECAY,
)
memory = ReplayMemory(STACK_SIZE + 1, MEM_SIZE, device)
```

训练过程：

```
for step in progressive:
    if done:
        observations, _, _ = env.reset()
        for obs in observations:
            obs_queue.append(obs)

    # 如果经验池足够大，则开始训练，随机选取动作
    training = len(memory) > WARM_STEPS
    state = env.make_state(obs_queue).to(device).float()
    action = agent.run(state, training)

    # 计算 reward
    obs, reward, done = env.step(action)
    obs_queue.append(obs)
    memory.push(env.make_folded_state(obs_queue), action, reward, done)

    # 对 policy Q-function 的训练过程
    if step % POLICY_UPDATE == 0 and training:
        agent.learn(memory, BATCH_SIZE)

    # 对 target Q-function 进行更新
    if step % TARGET_UPDATE == 0:
        agent.sync()

    # 写入文件
    if step % EVALUATE_FREQ == 0:
        avg_reward, frames = env.evaluate(obs_queue, agent, render=RENDER)
        with open("rewards.txt", "a") as fp:
            fp.write(f"{step//EVALUATE_FREQ:3d} {step:8d} {avg_reward:.1f}\n")
```

损失函数以及梯度下降算法：

```
def learn(self, memory: ReplayMemory, batch_size: int) -> float:
    """learn trains the value network via TD-learning."""
    # 随机选取动作
    state_batch, action_batch, reward_batch, next_batch, done_batch = \
        memory.sample(batch_size)

    # 计算损失函数值
    values = self.__policy(state_batch.float()).gather(1, action_batch)
    values_next = self.__target(next_batch.float()).max(1).values.detach()
    expected = (self.__gamma * values_next.unsqueeze(1)) * \
        (1. - done_batch) + reward_batch
    loss = F.smooth_l1_loss(values, expected)

    # 反向传播梯度下降
    self.__optimizer.zero_grad()
    loss.backward()
    for param in self.__policy.parameters():
        param.grad.data.clamp_(-1, 1)
    self.__optimizer.step()

    return loss.item()
```

四、实验结果

样例代码中迭代 5×10^7 次所需要的时间为 $100+$ 小时，消耗的计算资源较多，所以将迭代次数减少到 5×10^6，训练一次的时间大约为 10 小时，结果如下：

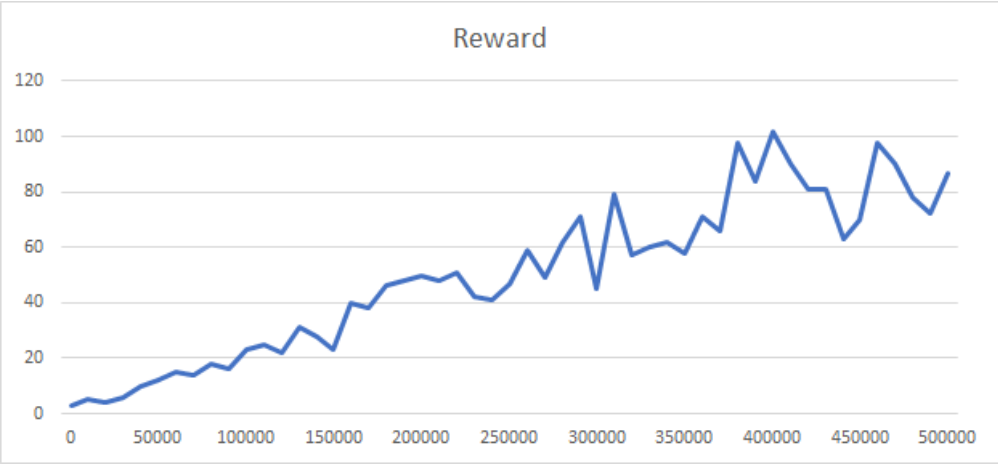

图 1 DQN-Reward

五、Dueling DQN——DQN 的改进

Dueling DQN 是 DQN 的一个改进版本，它通过优化神经网络的结构来优化学习算法。其最重要的一点就是改进了 DQN 中的网络结构。Dueling DQN 考虑将 Q 网络分成两部分，第一部分仅与状态 S 有关，与具体要采用的动作 A 无关，这部分我们叫做价值函数，记做 $V(S, w, \alpha)$，第二部分同时与状态 S 和动作 A 有关，这部分叫做优势函数（Advantage Function），记为 $A(S, A, w, \beta)$，那么最终我们的价值函数可以重新表示为：

$$Q(S, A, w, \alpha, \beta) = V(s, w, \alpha) + A(S, A, w, \beta)$$

其中，w 是公共部分的网络参数，而 α 是价值函数独有的网络参数，而 β 是优势函数独有的网络参数。这种方法能够有效地对 Q 值进行更新，因为每一次 V 值更新之后，都要加在 A 函数的所有维度上（相当于一个 bias），相当于其他动作的值也同时被更新了。

模型修改：

```
class DQN(nn.Module):
    def __init__(self, num_actions, device):
        super(DQN, self).__init__()
        self.num_actions = num_actions
        self.conv1 = nn.Sequential(
            nn.BatchNorm2d(4),
            nn.Conv2d(in_channels=4, out_channels=32,
                      kernel_size=8, stride=4, bias=False),
            nn.BatchNorm2d(32))
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64,
                      kernel_size=4, stride=2, bias=False),
            nn.BatchNorm2d(64))
        self.conv3 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=64,
                      kernel_size=3, stride=1, bias=False),
            nn.BatchNorm2d(64))

        self.fc1_adv = nn.Linear(in_features=7*7*64, out_features=512)
        self.fc1_val = nn.Linear(in_features=7*7*64, out_features=512)
        self.fc2_adv = nn.Linear(in_features=512, out_features=num_actions)
        self.fc2_val = nn.Linear(in_features=512, out_features=1)
        self.relu = nn.ReLU()
        self.__device = device
```

实验结果：

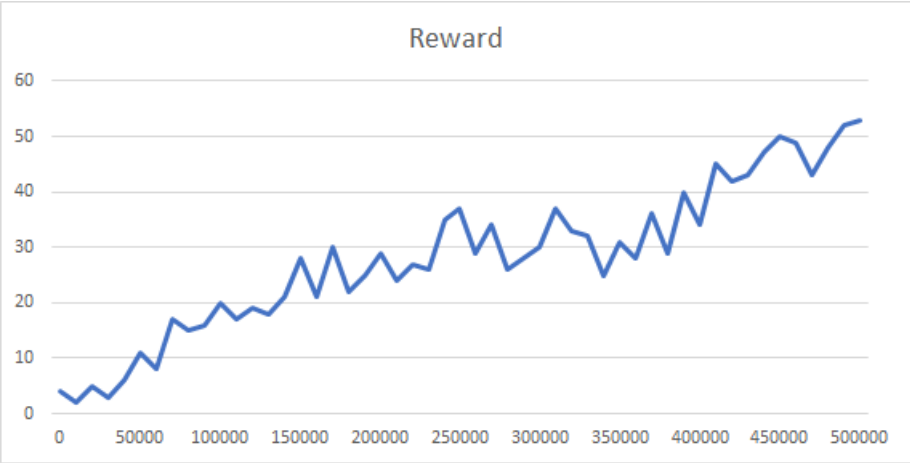

图 2 Dueling DQN-Reward

可见，Dueling DQN 的 Reward 反而降低了，不过震荡幅度减小了，在游戏中的表现也有所提升，可预见的是，有足够多的训练次数的话，效果会比 DQN 好。

六、开源代码链接

七、参考资料

[1] DQN算法分析：https://blog.csdn.net/yeqiang19910412/article/details/76468407

[2] Implementing the Deep Q-Network 翻译：https://zhuanlan.zhihu.com/p/31374784

[3] Deep Reinforcement Learning 基础知识：https://blog.csdn.net/songrotek/article/details/50580904

[4] 强化学习(十二) Dueling DQN：https://www.cnblogs.com/pinard/p/9923859.html