强化学习第二次作业——Fighting Game

May 16, 2023

Abstract

1 背景介绍

1.1 问题介绍

格斗游戏作为一种具有挑战性和战略性的游戏类型，对于玩家的反应速度、技巧和战术思维提出了高要求。基于强化学习的格斗游戏的目标是通过让游戏角色与环境进行交互来学习最佳的战斗策略。角色通过与其他角色或玩家进行对战来不断调整其行为，并根据反馈信号（例如奖励或惩罚）来优化其决策。通过大量的对战经验，强化学习算法可以学习到最佳的行动策略，使得游戏角色能够在战斗中更加智能和灵活。

1.2 问题建模

本次实验的格斗游戏基于 FightingICE 格斗游戏平台，每秒刷新 60 帧，要求 AI 在一帧内完成决策，并将动作输出到对战环境，对战环境进行状态的更新，并进行下一帧运算。

其中,状态空间包含 144 维,动作空间包含 40 维。Python 代码中的 gym_ai.py 中的 get_obs() 方法详细说明了状态空间的组成，通过对该方法进行分析得到状态空间 observation 的表示如下：

observation[0:9] 表示己方智能体的状态；

observation[9:65] 表示己方动作，用 one-hot 向量表示；

observation[65] 表示己方残余帧数；

observation[66:75] 表示对方智能体的状态；

observation[75:131] 表示对方动作，用 one-hot 向量表示；

observation[131] 表示对方残余帧数；

observation[132:138] 表示己方抛射物状态；

obsvertation[138:144] 表示对方抛射物状态。

本次实验采用多种强化学习方法对该问题进行求解,包含基于表格的 QLearning 算法、SARSA 算法以及 DQN 算法（Double Dueling DQN），下面将依次介绍。

2 Q-Learning 算法

2.1 算法原理

Q-Learning 算法是一种 value-based 的强化学习方法，Q 即 $Q(s,a)$，指在某一个时刻的 s 状态下，采取动作 a 能够获得收益的期望值。

Q-Learning 以贪心策略 $\pi(s) = \arg\max_a Q(s,a)$ 的 Q 值作为时间差分目标。贝尔曼期望方程为：

$$\begin{aligned} Q_\pi(s,a) &= \mathcal{R}_s^a + \gamma \sum_{s'} \mathcal{P}_{ss'}^a \sum_{a'} \pi\left(a' \mid s'\right) Q_\pi\left(s', a'\right) \\ &= \mathcal{R}_s^a + \gamma \sum_{s'} \mathcal{P}_{ss'}^a \max_{a'} Q_\pi\left(s', a'\right) \\ &\approx r_{t+1} + \gamma \max_{a'} Q\left(s_{t+1}, a'\right) \end{aligned}$$

Q-Learning 的参数更新过程为：

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha\left(r_{t+1} + \gamma \max_{a'} Q\left(s_{t+1}, a'\right) - Q\left(s_t, a_t\right)\right)$$

算法大致过程为：

Algorithm 1 Q-Learning 算法

```
function Q-LEARNING(N, alpha, gamma)
    initialization obs, Q
    while n < N do
        act = argmax(obs, Q)
        new-obs, reward, done, info = step(act)
        if not done then
            delta = reward + gamma * max(Q * new-obs) - Q(act) * obs
            Q(act) = Q(act) + alpha * delta * obs
            obs = new-obs
    return Q
```

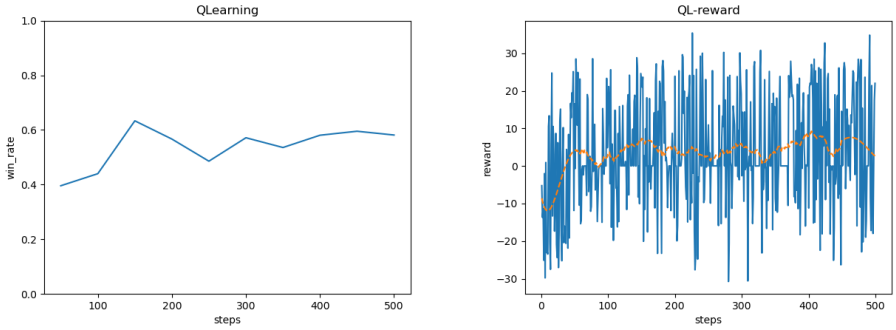
其中参数更新过程具体到 Python 中，可以参见以下代码：

```python
# QLearning
delta = reward + gamma * np.max(np.dot(weight, new_obs)) \
        - np.dot(obs, weight[act])
weight[act] = weight[act] + alpha * delta * obs
obs = new_obs
r = r + reward
```

其中，new_obs 和 reward 为观测量，通过一次实际的运行动作产生，$gamma = 0.95$，$alpha = 0.01$ 为初始固定设置，$weight$ 设置为 40×144 的权重矩阵，r 为待存储的 reward 信息。

2.2 实验结果

通过上述设置，对 Q-Learning 算法进行了 500 轮次的训练，训练过程以每 50 轮次的平均胜率进行展示。训练过程的胜率变化情况如 Figure 1(a) 所示。训练过程中的 reward 曲线如 Figure 1(b) 所示。



(a) 胜率变化曲线　　(b) reward 变化曲线

Figure 1: QLearning 训练过程

由 Figure 1可知，平均胜率和 reward 整体上呈现波动上升的趋势。训练结束后，在最佳训练权重上进行了 30 轮次的测试，测试胜率为 66.7%。

3 Sarsa 算法

3.1 算法原理

Sarsa 算法与 Q-Learning 算法类似，贝尔曼期望方程为：

$$
\begin{aligned}
Q_\pi(s,a) &= \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi\left(a' \mid s'\right) Q_\pi\left(s', a'\right) \\
&\approx r_{t+1} + \gamma Q\left(s_{t+1}, a_{t+1}\right)
\end{aligned}
$$

其参数更新过程为：

$$
Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha\left(r_{t+1} + \gamma Q\left(s_{t+1}, a_{t+1}\right) - Q\left(s_t, a_t\right)\right)
$$

算法大致过程为：

Algorithm 2 Sarsa 算法

```
function SARSA(N, alpha, gamma)
    initialization obs, Q
    while n < N do
        epsilon-贪心策略更新 act
        new-obs, reward, done, info = step(act)
        if not done then
            epsilon-贪心策略更新 explore-act
            delta = reward + gamma * (Q(explore-act) * new-obs) - Q(act) * obs
            Q(act) = Q(act) + alpha * delta * obs
            obs = new-obs
    return Q
```

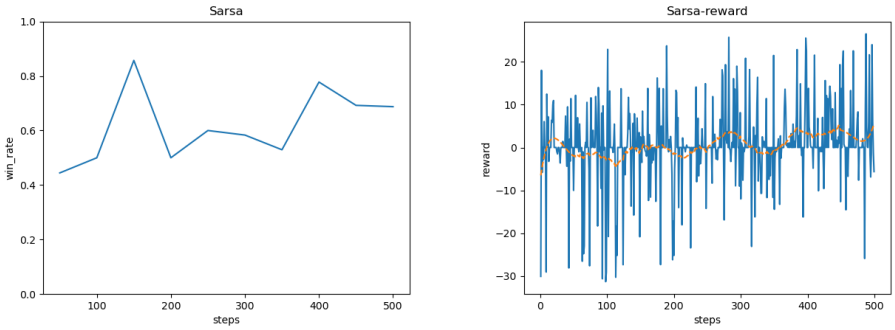
在 Python 中，以上更新过程的实现代码为：

```
# Sarsa
explore_act = np.argmax(np.dot(weight, new_obs))
if random.random() < epsilon:
    explore_act = random.randint(0, 39)
else:
    pass  # epsilon-贪心策略
delta = reward + gamma * np.dot(new_obs, weight[explore_act]) \
        - np.dot(obs,weight[act])
weight[act] = weight[act] + alpha * delta * obs
obs = new_obs
r = r + reward
```

其中，new_obs 和 reward 为观测量，通过一次实际的运行动作产生，$gamma = 0.95$，$alpha = 0.01$，$epsilon = 0.1$ 为初始固定设置，$weight$ 设置为 40×144 的权重矩阵，r 为待存储的 reward 信息。

3.2 实验结果

通过上述设置，对 Sarsa 算法进行了 500 轮次的训练，训练过程以每 50 轮次的平均胜率进行展示。训练过程的胜率变化如 Figure 2(a) 所示。训练过程中的 reward 曲线如 Figure 2(b) 所示。



(a) 胜率变化曲线　　(b) reward 变化曲线

Figure 2: Sarsa 训练过程

由 Figure 2可知，平均胜率和 reward 整体上呈现一个波动上升的趋势。胜率值在 150 轮次处出现较大的波动，这可能是因为在训练过程中多次出现 py2j 超时的问题，影响了训练过程中的测试效果。在第 101-150 轮次中，出现了 43 次 py2j 超时的报错，这导致实际上这些轮次里只进行了 7 次测试，因此误差较大。训练结束后，在最佳训练权重上进行了 30 轮次的测试，测试胜率为 60%。

4 DQN 算法

4.1 算法原理

DQN(Deep Q-Network) 算法 [1] 常常用于状态和动作都十分复杂的情况。具体而言，它采用一个深度神经网络作为值函数近似器，来估计每个状态动作对的价值。它的核心机制包含经验回放和目标网络。

经验回放指的是 DQN 将智能体在过去交互过程中获得的经验存储在经验回放池中，每次训练时，从中随机抽样固定数目的样本来构建训练批次。这种做法可以大大降低样本之间的时间相关性，提高训练的效率和稳定性。

目标网络是为了解耦 Q 值估计和 Q 目标值估计的问题。目标网络是一个与主网络（行动网络）相互独立的网络，用于估计 Q 目标值。它的参数更新频率比较低，每隔一段时间将行动网络的参数更新给目标网络，为一段时间内的训练提供一致的目标值。如果不引入目标网络，由于目标 Q 值也会随着训练过程中策略网络参数的更新而变化，导致梯度难以传播，训练困难。

DQN 的损失函数如下：

$$loss = \|Q_\theta(s,a) - (r + \gamma \max_{a'} Q_{\theta'}(s',a'))\|_2$$

上式中 Q_θ 表示主网络，$Q_{\theta'}$ 表示目标网络。

格斗游戏中，当智能体处于不同位置时，决策的权重也不同。例如，当两个智能体相距较近时，应当以攻击为主；当二者相距较远时，应该以移动为主。此外，尽管格斗游戏的状态维度大，但是并非每个状态都需要估计每个动作的值。针对这种情况，采用 Dueling DQN 算法 [2]，如图3所示。该算法是将 Q 函数解耦为 V 函数和优势函数 A 两部分，优势函数定义为

$$A(s,a) = Q(s,a) - V(s)$$

Dueling DQN 网络输入状态 s，输出状态价值 V(s) 和优势函数 A(s,a)。最终需要的 Q(s,a) 可以用如下公式计算

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{\|\mathcal{A}\|} \sum_{a_i \in \mathcal{A}} A(s,a_i)$$

本次作业采用的 Q 网络采用多层感知机的方式实现，具体实现如下：

```
class DuelQNetwork(nn.Cell):
  def __init__(self, params):
        super().__init__()
        self.base = nn.SequentialCell([
          nn.Dense(
          params['state_space_dim'],
          256,
          weight_init="XavierUniform",
          activation="relu").to_float(ms.float32),
          nn.Dense(
          256,
          256,
          weight_init="XavierUniform",
          activation="relu").to_float(ms.float32),
          nn.Dense(
          256,
          128,
          weight_init="XavierUniform",
          activation="relu").to_float(ms.float32),
        ])

        self.value_net = nn.Dense(128, 1,
                                  weight_init="XavierUniform"
                                  ).to_float(ms.float32)
        self.action_net = nn.Dense(128, params['action_space_dim'],
                                   weight_init="XavierUniform"
                                   ).to_float(ms.float32)
        def construct(self, x):
          f = self.base(x)
          state_value = self.value_net(f)
          advantage = self.action_net(f)
          q_value = state_value +
                    (advantage -
                    P.ReduceMean(keep_dims=True)(advantage, 1))
          return q_value
```

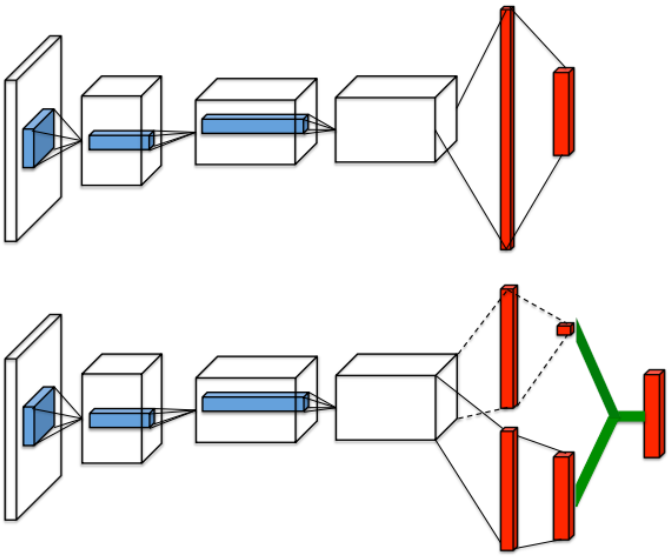


Figure 3: DQN（上）和 Dueling DQN（下）示意图

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

Figure 4: DQN 算法详细流程

此外，为了缓解 Q 函数过估计的问题，引入 Double DQN 算法 [3]，即利用主网络选择动作，利用目标网络估计目标值。具体而言，其损失函数的计算如下：

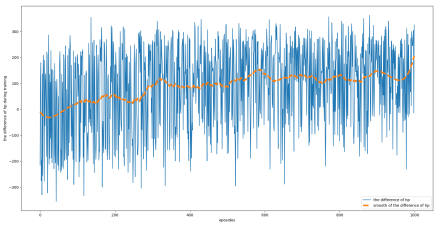
$$loss = \|Q_\theta(s,a) - (r + \gamma Q_{\theta'}(s', \arg\max_{a'} Q_\theta(s', a')))\|_2$$

4.2 实验结果

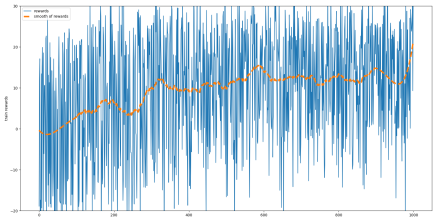
DQN 算法采用 mindspore_rl 框架实现。

4.2.1 实验参数配置

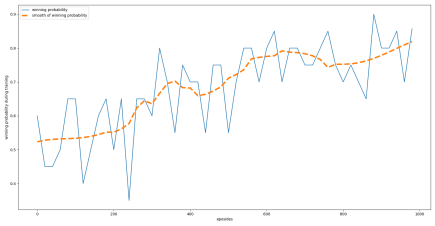
DQN 的经验回放池容量设置为 20000，存储的数据格式为 $[s, a, r, s']$，采样大小 sample_size 设置为 32，γ 值设为 0.9，学习率设为 0.001。总共训练 1000 个 episode，其中，每隔 10 个 eposide 进行 3 个 eposide 的测试。



(a) 己方相比对方血量差异的变化



(b) 奖励的变化



(c) 胜率的变化

Figure 5: DQN 训练时训练阶段的曲线变化

初始化时，智能体的行动策略采用随机策略，当经验回放池大小达到一个 sample_size 时，初始化结束。

训练时，智能体的行动策略为 $\epsilon-$ 贪心策略，初始时 $\epsilon = 0.9$，接着按照如下公式进行衰减。

$$\epsilon = (\epsilon_h - \epsilon_l)\exp(-\frac{episode}{decay}) + \epsilon_l$$

其中，$episode$ 表示当前训练的轮次，$decay$ 是衰减参数，ϵ_h 表示初始时的 ϵ 值，ϵ_l 表示所容许的最小的 ϵ 值，本次实验中 $\epsilon_h = 0.9$，$\epsilon_l = 0.05$，$decay = 200$。

测试时，采用贪心策略，保证可以学到的 Q 网络权值可以被完全利用。

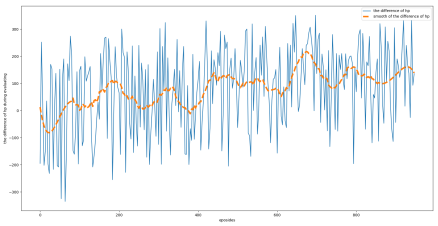
优化器采用均方根传播（RMSProp）优化器，它使用指数加权移动平均的方法计算梯度的平方均值，并根据该平均值来调整学习率。RMSProp 可以自适应地调整不同参数的学习速率，对于具有大梯度的参数使用较小的学习率，对于具有小梯度的参数使用较大的学习率，使得学习过程更加平稳。

4.2.2 实验结果

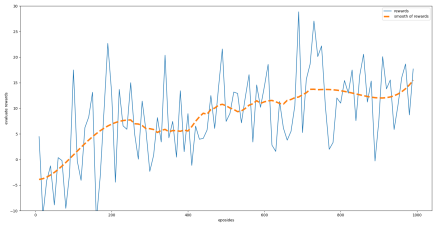
训练过程的曲线变化 图5所示是训练阶段的曲线变化。从曲线变化可以看出，随着训练轮次的增加，游戏结束己方相比于对方的血量差异、游戏结束时获得的奖励以及胜率都是在波动上升的。这表明，智能体学习到了合适的策略。

图6所示是训练时每隔 10 个 eposide 进行 3 次验证的结果的曲线变化。从曲线变化可以看出，由于采用了完全贪心策略，该图中各个曲线相比于训练阶段的曲线的变化趋势更加明显。

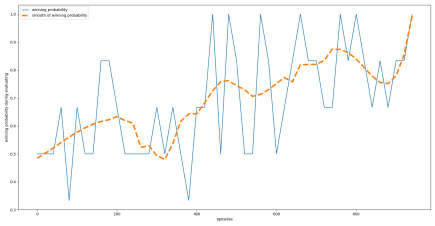
测试 1000eposide 的结果 为了更加详细的评估智能体学习到的策略是否有效，加载最后一次训练保存的权重文件，采用测试模式运行 1000 个 eposide，得到的结果如图7所示。该图表明，测



(a) 己方相比对方血量差异的变化



(b) 奖励的变化



(c) 胜率的变化

Figure 6: DQN 训练时验证阶段的曲线变化

试时的胜率达到了 94.11%，并且大部分情况下，己方可以以较大优势 (血量差值大于 100) 赢得比赛。

4.3 失败案例

在1.2中介绍了状态空间的组成，从中可以看出环境返回的状态中也包含了对方状态和对方选择的动作，猜想能不能在训练初期同时学习对方的策略，即每次加入经验池的不只是己方的经验 (s_1, a_1, r_1, s_1')，还包括对方的经验 (s_2, a_2, r_2, s_2')，期望在初期策略较差时可以通过对方策略快速学习到合适的策略。

因此，对于环境返回的 obsvervation，取 observation[0:9,65:75,131:144] 作为己方状态 s_1，从 observation[9:65] 中提取出己方动作 a_1'；取 observation[66:75,131,0:8,65,138:144,132:138] 作为对

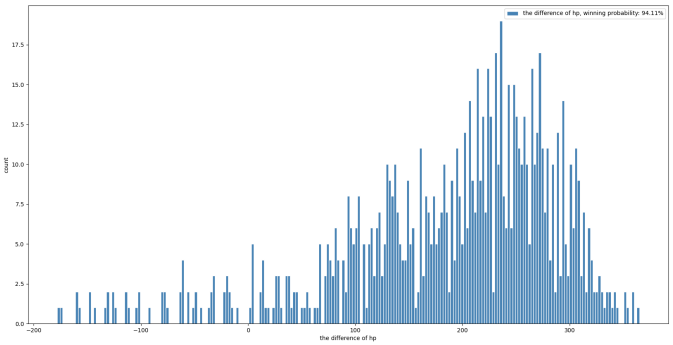


Figure 7: DQN 测试 1000eposide 己方相比于对方血量差异的分布直方图

方状态 s_2，从 observation[75:131] 提取出对方动作。将该问题视为一个二人零和博弈问题，取 $r_2 = -r_1$。将得到的 (s_1, a_1, r_1, s_1') 和 (s_2, a_2, r_2, s_2') 加入经验回放池。

此外，由于环境返回的动作是在 java 代码中定义的动作，这些动作有 56 个且序数和 python 代码中定义的动作序数不同。java 代码中动作的定义在 enumerate.Action 中，python 代码中动作的定义位于 myenv.fightingice_env._actions 中，通过对这些动作进行分析，可以将 obsvertation 中取出的动作转化为符合 python 代码定义的动作。

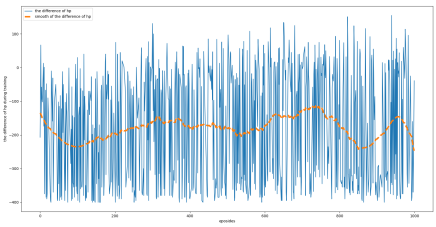
一般来说，经验池中的经验和己方当前策略越接近，训练效果越好，且到训练后期，己方已经学习到了一个相比对方更有优势的策略，此时再将对方的经验加入经验回放池将不利于己方学习到更好的策略。基于此，设置一个阈值 $thre$，每次和环境交互时产生一个符合 $U(0,1)$ 分布的随机数 n，若 $n > thre$，将对方经验 (s_2, a_2, r_2, s_2') 加入经验回放池，否则不加入经验回放池。阈值的更新公式如下

$$thre = \frac{cur\_eposide}{max\_eposide} * \frac{4}{3}$$

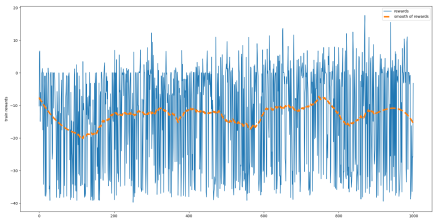
除了上面的设置外，该算法的其余设置都和成功算法相同。

遗憾的是，这种方法并没有正确学习到一个好的策略。如图8和图9所示。训练阶段，己方相比对方血量差异大部分情况下小于 0，大部分奖励小于 0；测试阶段己方血量全小于对方血量，这表明己方智能体完全没学到一个合适的策略。

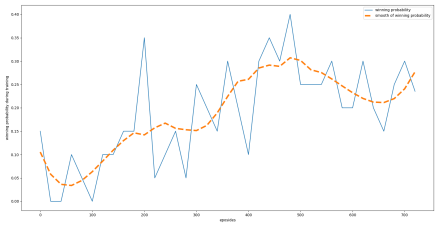
推测原因可能是因为状态空间维度减小后导致智能体难以学习到合适的策略。另外，训练过程中通信失败的次数过多 (1000 个 eposide 训练和 300 个 eposide 验证中有 409 个 eposide 失败) 也可能是一大原因。此外，由于 mindspore 调试困难，不排除是本算法的代码实现存在一些问题。



(a) 己方相比对方血量差异的变化

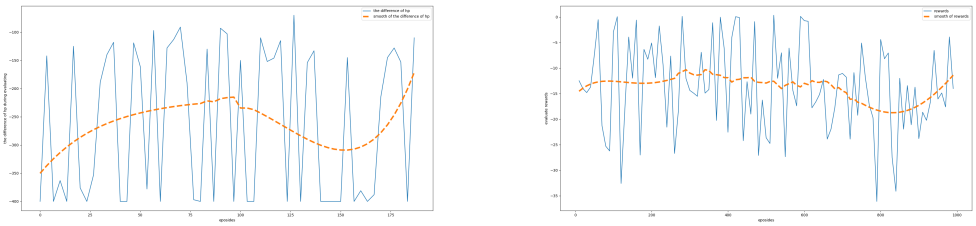


(b) 奖励的变化



(c) 胜率的变化

Figure 8: 失败的 DQN 算法训练时训练阶段的曲线变化



(a) 己方相比对方血量差异的变化　　(b) 奖励的变化

Figure 9: 失败的 DQN 算法训练时验证阶段的曲线变化

5 小结

5.1 遇到的问题

第一个问题是训练过程中经常出现和环境通信失败的情况，经过检查源代码，发现是发送给 java 环境的动作未能在 60s 内获得反馈，猜测这种问题可能是服务器本身配置较低导致的。

第二个问题是 mindspore 库调试较为困难。尽管 mindspore 的代码风格和 Pytorch 较为相似，但是 Pytorch 的计算图是运动时动态构建的，可以逐行调试，本次实验中基于 mindspore 的代码则是基于静态图的方式，计算图在计算前编译，调试本身比较困难。mindspore 官网给出了一键动静转化的方式，但是经过运行发现未能奏效。由于 mindspore 库使用人数较少，网上教程较少，且官网文档较为简略，导致代码运行过程中遇到的问题较难解决。

5.2 结论

本文针对格斗游戏，采用 Q-Learning、SARSA 算法和 DQN 算法进行实验。实验结果表明，Double Dueling DQN 算法的结果比基于表格的 Q-Learning 算法和 SARSA 算法好得多。这表明，在状态空间和动作空间维度较大时，由于神经网络强大的表示能力，DQN 算法具有显著优势。

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602 (2013).

[2] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, N. Freitas, Dueling network architectures for deep reinforcement learning, in: International conference on machine learning, PMLR, 2016, pp. 1995–2003.

[3] H. Van Hasselt, A. Guez, D. Silver, Deep reinforcement learning with double q-learning, in: Proceedings of the AAAI conference on artificial intelligence, Vol. 30, 2016.