DQN
以ALE AIRRAID环境为例

强化学习实验报告

张恒硕

2025 年 4 月 21 日

南開大學
Nankai University

目 录

图 片

表 格

1 实验目的

1. 分析学习AirRaid游戏环境，了解其状态、动作、奖励等设定。
2. 了解DQN原理，并理解各参数的意义与作用。
3. 编写DQN代码训练智能体进行AirRaid游戏，观察分析训练的过程和结果。
4. 检索阅读改进方法，分析其可能的优化结果。

2 实验原理

2.1 AirRaid游戏环境

以下环境介绍选取较为简单的默认环境。

作为一个模拟空战的游戏，其会在远端刷新向前移动的不同敌方飞机（可大致视为左中右三列），在近端操纵己方飞机左右移动和射击，躲避敌方子弹并命中敌方飞机来获取分数，己方飞机被击中后损失生命，生命耗尽后游戏结束。

状态空间 宽度为210像素，高度为160像素的图像窗口，可以选取RGB或灰度图模式，也可以直接观察内部信息RAM。在当前默认环境中，己方飞机只能在近端左右移动，敌方飞机只能从远端沿直线向前移动。在渲染时，窗口会在RGB和灰度图中切换显示。

动作空间

AirRaid提供了6种或18种可能的动作选择，鉴于任务已经比较复杂，选取前者：

- 0：NOOP（无操作）
- 1：FIRE（发射）
- 2：RIGHT（向右移动）
- 3：LEFT（向左移动）
- 4：RIGHTFIRE（向右移动并发射）
- 5：LEFTFIRE（向左移动并发射）

奖励与损失 击中不同敌方飞机可以得到不同分数，被击中将会在中点复活。一开始有三条命，得到1000分后可以获得额外一条命（鉴于很难达到2000或以上的分数，没有关注后续的加命规则）。

其他 环境还有很多可以设置的信息，比如折扣率、难度、模式等，这里不做介绍。

2.2 DQN

DQN，即Deep Q-Network（深度Q网络），是一种结合了强化学习与深度学习的技术，适用于高维空间的状态和动作问题，由DeepMind团队在2013年提出。其核心是用神经网络近似Q值函数 $Q(s, a, \theta)$，取代传统Q学习中的Q表，评估给定状态下采取某个动作的好坏程度。

以下介绍其改进版本（双q），其减少了过估计，提升了稳定性。

核心技术

1 经验回放（Experience Replay）：存储智能体的经验到固定大小的回放缓冲区，训练时随机抽取小批量样本，打破数据相关性，提高稳定性。其改进有优先经验回放。

2 目标网络（Target Network）：使用一个独立网络计算目标Q值，参数定期从主网络同步。分为硬更新和软更新，分别是直接复制和部分更新。

3 状态表示：将环境的状态作为输入传递给神经网络。在本次实验中，用游戏图片作为输入，并将多帧图像堆叠为一次输入。

步骤

- 相同初始化主网络和目标网络。
- 与环境交互，收集经验存入回放缓冲区。
- 随机采样小批量经验，计算目标Q值。
- 最小化损失函数，更新主网络。
- 每隔固定步数同步目标网络参数。

3 关键代码解析

帧堆叠

Listing 1: 帧堆叠

```
# 帧堆叠
class FrameStackWrapper(gym.Wrapper):
    def __init__(self, env, num_stack=4):
        super().__init__(env)
        self.num_stack = num_stack
        self._obs_shape = env.observation_space.shape  # 记录原始观测形状
        self.frames = np.zeros((self.num_stack, *self._obs_shape), dtype=np.uint8)  # 预分配固定内存数组
        self.observation_space = gym.spaces.Box(low=0, high=255, shape=(self.num_stack, *self._obs_shape), dtype=np.uint8)  # 调整观测空间

    def reset(self, **kwargs):
        obs, info = self.env.reset(**kwargs)
        self.frames[:] = obs  # 用初始观测填充所有帧
        return self.frames.copy(), info

    def step(self, action):
        obs, reward, terminated, truncated, info = self.env.step(action)
        # 滚动更新帧
        self.frames[:-1] = self.frames[1:]  # 左移旧帧
        self.frames[-1] = obs  # 插入新帧
        return self.frames.copy(), reward, terminated, truncated, info
```

- 作为神经网络的输入，将游戏的4帧堆叠起来，而这4帧又是从原动画隔帧提取的，即每次输入的信息跨度为8帧。
- 采用预分配、初始填充、滚动更新等手段提高效率。

神经网络

Listing 2: 神经网络

```
# 初始化网络层（He初始化）
def layer_init(layer, bias_const=0.0):
    torch.nn.init.kaiming_normal_(layer.weight, nonlinearity='relu')
    torch.nn.init.constant_(layer.bias, bias_const)
    return layer

# DQN
class DQN(nn.Module):
    def __init__(self, n_actions):
        super(DQN, self).__init__()
        self.net = nn.Sequential(
            layer_init(nn.Conv2d(4, 32, 8, stride=4)),
            nn.ReLU(),
            layer_init(nn.Conv2d(32, 64, 4, stride=2)),
            nn.ReLU(),
            layer_init(nn.Conv2d(64, 64, 3, stride=1)),
            nn.ReLU(),
            nn.Flatten(),
            layer_init(nn.Linear(64 * 7 * 7, 512)),
            nn.ReLU(),
            layer_init(nn.Linear(512, n_actions))
        )
    def forward(self, x):
        x = x.float() / 255.0
        return self.net(x)
```

- 三个卷积层，之后展平接入两个全连接层，激活函数采用ReLU。
- 第一个卷积层输入为4，这是因为输入的是堆叠4帧的图像，其深度为4。
- 前向传播时，对数据进行归一化处理。
- 网络层的初始化采用He初始化。

经验回放区

Listing 3: 经验回放区

```
# 经验回放缓冲区
class ExperienceBuffer:
    def __init__(self, capacity=REPLAY_MEMORY):
        self.buffer = deque(maxlen=capacity)
    # 放入经验
    def add(self, experience):
        self.buffer.append(experience)
    # 采样
    def sample(self, batch_size):
        indices = np.random.choice(len(self.buffer), batch_size, replace=False)
        samples = [self.buffer[i] for i in indices]
        states, actions, rewards, next_states, dones = zip(*samples)
        return (
            torch.FloatTensor(np.array(states)).to(DEVICE),
            torch.LongTensor(np.array(actions)).to(DEVICE),
            torch.FloatTensor(np.array(rewards)).unsqueeze(-1).to(DEVICE),
            torch.FloatTensor(np.array(next_states)).to(DEVICE),
            torch.FloatTensor(np.array(dones)).unsqueeze(-1).to(DEVICE)
        )
```

- 在填满后，新经验入队，最老经验出队。

智能体

Listing 4: 智能体

```
# 智能体
class Agent:
    def __init__(self, policy_file=None):
        self.n_actions = gym.make("ALE/AirRaid-v5").action_space.n
        self.policy_net = DQN(self.n_actions).to(DEVICE)
        self.target_net = DQN(self.n_actions).to(DEVICE)
        self.steps = 0
        self.epsilon = INITIAL_EPSILON
        self.target_update = SAVE_INTERVAL  # 目标网络更新间隔
```

```
        self.loss_history = []  # 损失值记录
        self.steps_history = []  # 步数记录
        if policy_file:  # 加载策略
            self.load_policy(policy_file)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.optimizer = RMSprop(self.policy_net.parameters(), lr=0.00025, alpha=0.95,
            eps=0.01)  # RMSprop优化
        器
        self.buffer = ExperienceBuffer()
    # 加载策略（不会加载经验池）
    def load_policy(self, file_path):
        if os.path.exists(file_path):
            self.policy_net.load_state_dict(torch.load(file_path, map_location=DEVICE)
                )
            self.steps = int(file_path.split('_')[-1].split('.')[0])
            print(f"加载{file_path}")
        else:
            print(f"没有{file_path}")
    # epsilon衰减
    def decay_epsilon(self):
        if self.steps > OBSERVE:
            decay_steps = min(self.steps - OBSERVE, EXPLORE)
            self.epsilon = max(FINAL_EPSILON, INITIAL_EPSILON - (INITIAL_EPSILON -
                FINAL_EPSILON) * decay_steps / EXPLORE)
    # 更新网络
    def update_network(self):
        if len(self.buffer.buffer) < BATCH:
            return None
        states, actions, rewards, next_states, dones = self.buffer.sample(BATCH)
        # 双Q
        with torch.no_grad():
            next_actions = self.policy_net(next_states).argmax(1, keepdim=True)
            next_q = self.target_net(next_states).gather(1, next_actions)
            target_q = rewards + (1 - dones) * GAMMA * next_q
        current_q = self.policy_net(states).gather(1, actions.unsqueeze(-1))
        loss = F.smooth_l1_loss(current_q, target_q)  # Huber损失
        self.optimizer.zero_grad()
```

```
107         loss.backward()
108         nn.utils.clip_grad_norm_(self.policy_net.parameters(), 5)
109         self.optimizer.step()
110         # 定期更新目标网络
111         if self.steps % self.target_update == 0:
112             self.target_net.load_state_dict(self.policy_net.state_dict())
113         # 记录损失和步数
114         loss_value = loss.item()
115         self.loss_history.append(loss_value)
116         self.steps_history.append(self.steps)
117         return loss_value
118     # 保存网络
119     def save_model(self, progress_bar):
120         filepath = f"policy/policy_{self.steps}.pth"
121         torch.save(self.policy_net.state_dict(), filepath)
122         print(f"\n已保存{filepath}")
123         progress_bar.reset()
```

- 智能体使用神经网络进行训练，在过程中不断减小ϵ（线性变化）来降低探索性。
- 使用Huber损失，由RMSprop优化器进行优化。
- 可以选择从头开始训练，也可以导入网络参数继续训练，但后者由于无法复现经验回放区，并不能完美继续。
- 网络部分使用双q，包含主网络和目标网络。

主函数

Listing 5: 主函数

```
124 # 参数配置
125 GAME = 'AirRaid'
126 GAMMA = 0.99  # 折扣因子
127 OBSERVE = 50000  # 观察步数，用于填充经验池
128 EXPLORE = 1000000  # 探索步数，用于逐步减少 ε
129 INITIAL_EPSILON = 1.0  # 初始 ε
```

```
FINAL_EPSILON = 0.01  # 最小 ε
REPLAY_MEMORY = 1000000  # 经验回放缓冲区大小
BATCH = 64  # 批量大小
FRAME_SKIP = 2  # 帧跳过数
SAVE_INTERVAL = 10000  # 每轮步数
POLICY_FILE = None  # "policy\policy_20000.pth"
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
if __name__ == "__main__":
    gym.register_envs(ale_py)
    env = gym.make("ALE/AirRaid-v5", render_mode=None, frameskip=1)
    env = AtariPreprocessing(env, frame_skip=FRAME_SKIP, screen_size=84, grayscale_obs
        =True, scale_obs=False)
    env = FrameStackWrapper(env, num_stack=4)  # 堆叠4帧
    # 初始化智能体
    agent = Agent(POLICY_FILE)
    state, _ = env.reset()
    current_episode_reward = 0
    progress_bar = tqdm(total=SAVE_INTERVAL, unit="steps")
    try:
        while True:
            #  ε -greedy
            if np.random.rand() < agent.epsilon:
                action = env.action_space.sample()
            else:
                with torch.no_grad():
                    state_t = torch.FloatTensor(state).unsqueeze(0).to(DEVICE)
                    q_values = agent.policy_net(state_t)
                    action = q_values.argmax().item()
            # 执行动作
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            # 存储经验
            agent.buffer.add((state, action, reward, next_state, done))
            state = next_state
            current_episode_reward += reward  # 累计当前回合奖励
            # 游戏结束时输出分数
```

```
            if done:
                print(f"\n本局得分: {current_episode_reward}")
                current_episode_reward = 0
                state, _ = env.reset()
            else:
                state = next_state
            agent.steps += 1
            progress_bar.update(1)
            # 开始训练
            if agent.steps > OBSERVE:
                agent.update_network()
                agent.decay_epsilon()
            # 保存模型
            if agent.steps % SAVE_INTERVAL == 0:
                agent.save_model(progress_bar)
                progress_bar.reset()
    except KeyboardInterrupt:
        print("中断")
        # 绘制损失曲线
        if len(agent.loss_history) > 0:
            plt.figure(figsize=(12, 6))
            plt.plot(agent.steps_history, agent.loss_history, alpha=0.6)
            plt.xlabel('步数')
            plt.ylabel('损失值')
            plt.title('DQN损失变化曲线')
            plt.grid(True)
            plt.savefig('training_loss_curve.png')
    finally:
        env.close()
        progress_bar.close()
```

- 智能体每训练一定步数，即完成一轮，保存结果。
- 智能体每完成一局游戏，输出得分。
- 在终止训练时，输出损失随步数的变化曲线。

测试

Listing 6: 测试

```
# 一次实验
def run(queue, shared_policy_net_state, render):
    try:
        with torch.cuda.device(DEVICE):
            # 初始化环境
            gym.register_envs(ale_py)
            env = gym.make("ALE/AirRaid-v5", render_mode="human" if render else None,
                frameskip=1)
            env = AtariPreprocessing(env, frame_skip=FRAME_SKIP, screen_size=84,
                grayscale_obs=True, scale_obs=False)
            env = FrameStackWrapper(env, num_stack=4)
            # 加载模型
            n_actions = env.action_space.n
            policy_net = DQN(n_actions).to(DEVICE)
            policy_net.load_state_dict(shared_policy_net_state)
            policy_net.eval()
            # 运行测试
            state, _ = env.reset()
            total_reward = 0
            with torch.no_grad():
                while True:
                    state_t = torch.FloatTensor(np.array(state)).unsqueeze(0).to(
                        DEVICE)
                    q_values = policy_net(state_t)
                    action = q_values.argmax().item()
                    next_state, reward, terminated, truncated, _ = env.step(action)
                    total_reward += reward
                    state = next_state
                    if terminated or truncated:
                        break
            queue.put(total_reward)
    # 返回0分避免空队列
    except Exception as e:
```

```
225         print(f"进程出错:  {str(e)}")
226         queue.put(0)
227     finally:
228         if 'env' in locals():
229             env.close()
230         torch.cuda.empty_cache()
231 
232 if __name__ == "__main__":
233     policy = "policy/policy_200000.pth"
234     shared_policy = torch.load(policy, map_location='cpu')  # 统一加载到CPU,避免占
        用GPU
235     # 分批测试
236     rewards = []
237     for batch_start in range(0, n, CONCURRENT_PROCESSES):
238         batch_size = min(CONCURRENT_PROCESSES, n - batch_start)
239         print(f"正在执行批次{batch_start // CONCURRENT_PROCESSES + 1}/{(n - 1) //
            CONCURRENT_PROCESSES + 1}")
240         queue = Queue()
241         processes = []
242         for _ in range(batch_size):
243             p = Process(target=run, args=(queue, shared_policy, False))
244             p.start()
245             processes.append(p)
246         try:
247             for p in processes:
248                 p.join(timeout=120)
249             while not queue.empty():
250                 reward = queue.get()
251                 rewards.append(reward)
252         except KeyboardInterrupt:
253             print("中断")
254             break
255     # 结果处理
256     avg_reward = sum(rewards) / n
257     print(f"\n{n}次测试平均得分{avg_reward:.1f}")
258     # 演示运行
259     demo_queue = Queue()
```

```
260    Process(target=run, args=(demo_queue, shared_policy, True)).start()
```

- 在测试训练得到的网络参数结果时，并行（分两批）进行十次试验，取平均分，另外展示一次游戏过程。

4 实验结果与分析

4.1 实验过程

参数选取

如下表1和2：

表 1: 参数表1

γ	ϵ	观察步数（填充经验池）	探索步数（逐步减少ϵ）
0.99	1.0 -> 0.01	50000	1000000

表 2: 参数表2

经验回放缓冲区大小	批量大小	帧堆叠数量
1000000	64	2

训练过程

经过900000步训练（90轮，每轮10000步，耗时7.5个小时，下图1给出了训练过程中的最高分：

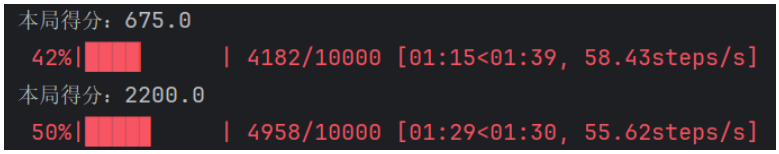


图 1: 训练过程最高分

4.2 纵向比较

得分

由于检索网络信息并未得到该游戏的人类水平或深度强化学习训练水平，很难进行横向比较，因此就训练过程的得分变化做纵向比较，如下图2。

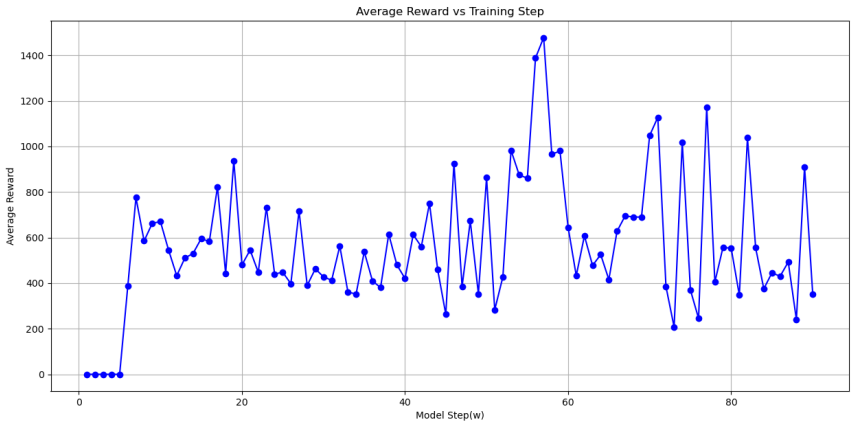


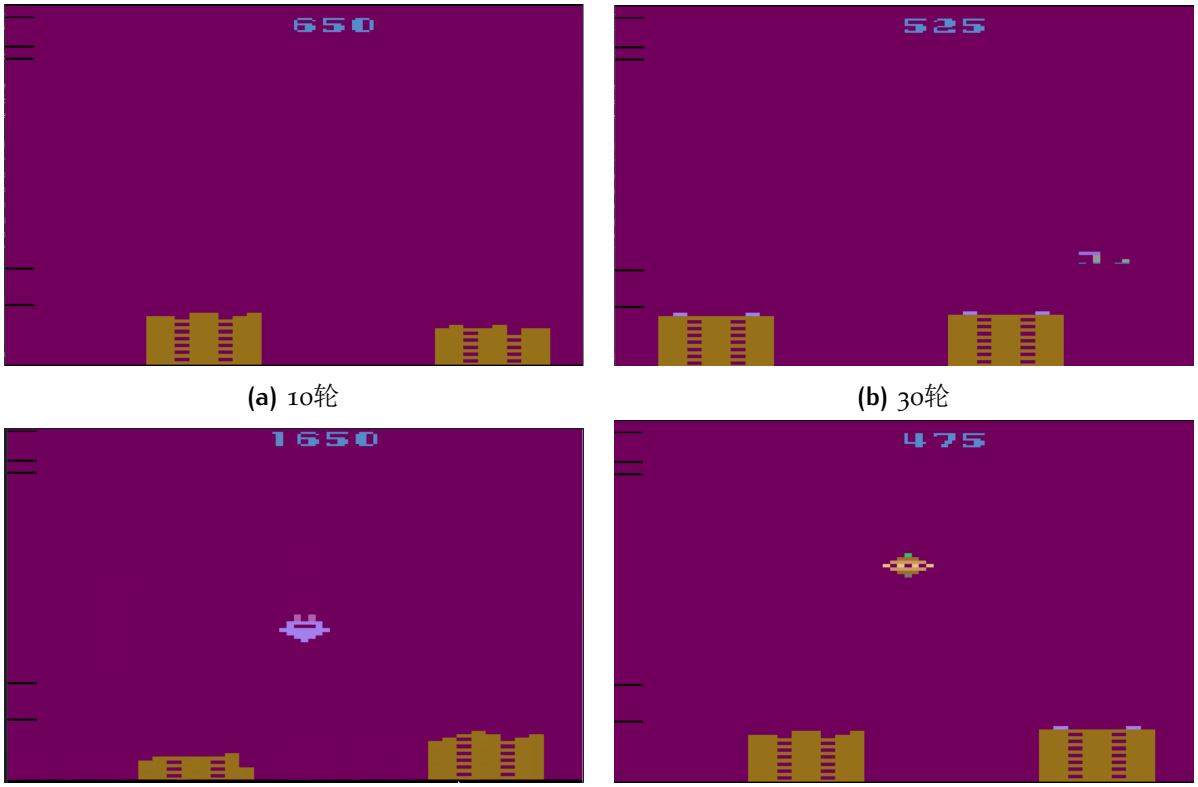
图 2: 训练过程得分变化

可以看到，整个过程中，智能体的性能改善不太明显，甚至在60轮左右后大幅跳水。

策略

观察训练不同阶段的智能体行为，结合以下图组3 on the following page给出的相应得分，做出以下总结：

- 10轮：智能体开始采用龟缩策略（左列），但是仍有大幅移动的出击行为，可能会到达右侧进行攻击。智能体的命中率和闪避性能较差。
- 30轮：智能体采用龟缩策略（中列），命中率和闪避性能都没有得到提升。
- 56轮：智能体得分峰值，龟缩策略（左列）大成，不会超出最左侧，命中率和闪避性能较强。但比较致命的是，智能体在复活后回到左侧的途中，可能出现再次被击中的情况，导致连续死亡。
- 80轮：智能体不再采取龟缩策略，出击范围扩展到两列（中右），说明智能体可能已经开始了真正的出击策略的学习。



(a) 10轮 (b) 30轮

(c) 56轮 (d) 80轮

图 3: 不同阶段的得分

给出的56.gif文件是采用56轮的网络参数的演示过程，其一定程度上代表了已有策略的最高水平。

损失

下图4 on the next page展示了训练过程中的损失变化。

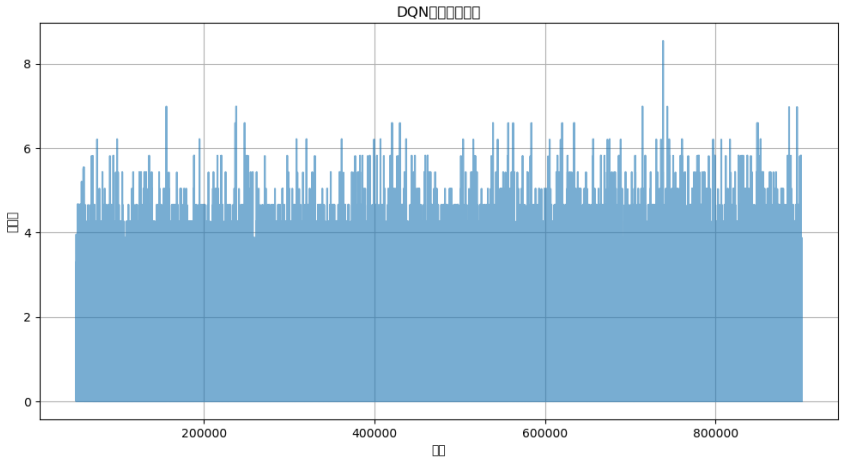


图 4: 训练过程损失变化

这个损失变化曲线是极差的，其说明智能体并没有找到优秀的策略进行收敛，而是还在探索有效策略，这与上述策略变化的过程是一致的。

4.3 分析

保障基础得分 更换观察角度，观察训练过程中每局游戏的结果（终端输出，这里不进行展示），发现虽然上限提高不多，但是下限有了显著提高，即能稳定获得不错的成绩。这说明，虽然智能体没有充分学习到如何在长时间内获得更高的分数，但是学习到了在游戏初始阶段保证基本得分的能力。

效果差的原因

- 训练轮数不足，智能体没有充分学习到如何在长时间内获得更高的分数，没有得到一个有效的策略。
- 环境的随机性较大，敌人的位置、型号、攻击间隔都带来极大扰动，使智能体在不同局中表现差异较大，即使长期训练也无法获得大量有意义数据。
- 在已有的训练过程中，智能体的策略经过了多次转变，概括来说，是从龟缩策略的长期得分向真正的出击得分转变。该变化导致每种策略都没有得到充分优化。

稠密奖励 本次实验选取的AirRaid是一个奖励较稠密的游戏，因为每次击中敌方飞机都可以获得奖励，但是采取一些龟缩策略也可能永远无法获得奖励。为避免后者情况，可以鼓励更多的探索。

4.4 改进思路

（大语言模型提供）

1. 优先经验回放（Prioritized Experience Replay）：根据TD误差优先级采样，能更高效地学习关键经验，加速收敛，在稀疏奖励任务中效果显著。
2. 多步TD目标（n-step Learning）：平衡偏差与方差，加速策略传播，在需要长期策略的游戏中可更快学到关键序列动作。
3. 奖励裁剪（Reward Clipping）：将奖励限制在特定小范围内，避免大奖励幅度波动，提升训练稳定性，适用于奖励范围差异大的环境。更激进的是使用符号函数，但是明显不适用于当前游戏。

5 实验总结

本次实验尝试对AirRaid游戏训练一个智能体，采用DQN，效果不佳，但分析训练过程得到了启示。

深度强化学习极大地拓展了强化学习的应用范围，但是复杂度（时间和空间）是其最大的制约因素之一。