



哈爾濱工業大學 (深圳)
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期: 2024 春季

课程名称: 人工智能 (实验)

实验名称: 实验三强化学习

实验性质: 综合设计型

实验学时: 2 地点: T2506

学生班级: 计科三班

授课教师: 郑海刚

实验与创新实践教育中心制

2024 年 5 月

一、实验环境

描述操作系统、开发环境（CPU\GPU）、使用的库等。

Windows、GPU: gymnasium, torch, matplotlib, collections, random, math, time 等

二、实验过程和结果分析

2.1 初始代码运行结果

运行 `reinforcement_q_learning.ipynb` 并将结果截图

2.2 优化

```
Episode 293, Cumulative Reward: 500
Complete
The training time is 316.858436822
Figure size 432x288 with 0 Axes>
```

代码及运行结果

可从神经网络结构的优化、超参数调优、优化经验回放区、奖励函数的设计、探索策略的设计等方面着手，挑选 4 个方向进行优化，并分析对比结果。

1. 优化一

(1) 优化代码描述

代码截图粘贴于此，并简单描述优化内容。

```

class DQN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.bn1 = nn.BatchNorm1d(128)
        self.layer2 = nn.Linear(128, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.layer3 = nn.Linear(128, 128)
        self.bn3 = nn.BatchNorm1d(128)
        self.layer4 = nn.Linear(128, n_actions)

    def forward(self, x):
        x = F.leaky_relu(self.bn1(self.layer1(x)))
        x = F.leaky_relu(self.bn2(self.layer2(x)))
        x = F.leaky_relu(self.bn3(self.layer3(x)))
        return self.layer4(x)

```

网络架构调整：适当减少层数或神经元数量以提高计算效率，同时不牺牲太多性能。

激活函数优化：使用 Leaky ReLU 或 ELU 等激活函数，避免 ReLU 可能引入的“神经元死亡”问题。

优化器调整：使用 Adam 优化器，并调整学习率。

Batch Normalization：在每层之后加入 Batch Normalization，稳定和加速训练。

(2) 运行结果截图

```

Complete
The training time is 294.63759756
<Figure size 432x288 with 0 Axes>

```

(3) 对比分析

与初始代码结果对比分析，从训练速度、收敛效果等方面进行分析。

训练速度：减少神经元数量从 256 到 128 可以减少计算量，从而提高训练速度。

使用 Batch Normalization 能加快模型的训练收敛速度。

Leaky ReLU 相较于标准 ReLU 减少了梯度消失的问题，进一步加速训练。

收敛效果:

Batch Normalization 在稳定训练过程中模型的输出分布，从而有助于更稳定的收敛。

Adam 优化器可以更好地调整学习率，使得训练更加高效和稳定。

Leaky ReLU 能更好地保持负数部分的梯度流动，从而避免神经元死亡，提高模型的学习能力。

2. 优化二

(1) 优化代码描述

截图代码粘贴于此，并简单描述优化内容。

```
class SumTree:
    def __init__(self, capacity):
        self.capacity = capacity
        self.tree = [0] * (2 * capacity - 1)
        self.data = [None] * capacity
        self.data_pointer = 0

    def add(self, priority, data):
        tree_idx = self.data_pointer + self.capacity - 1
        self.data[self.data_pointer] = data
        self.update(tree_idx, priority)

        self.data_pointer += 1
        if self.data_pointer >= self.capacity:
            self.data_pointer = 0

    def update(self, tree_idx, priority):
        change = priority - self.tree[tree_idx]
        self.tree[tree_idx] = priority
        while tree_idx != 0:
            tree_idx = (tree_idx - 1) // 2
            self.tree[tree_idx] += change

    def get_leaf(self, v):
        parent_idx = 0
        while True:
            left_child_idx = 2 * parent_idx + 1
            right_child_idx = left_child_idx + 1
            if left_child_idx >= len(self.tree):
                leaf_idx = parent_idx
                break
            else:
                if v <= self.tree[left_child_idx]:
                    parent_idx = left_child_idx
                else:
                    v -= self.tree[left_child_idx]
                    parent_idx = right_child_idx
        data_idx = leaf_idx - self.capacity + 1
        return leaf_idx, self.tree[leaf_idx], self.data[data_idx]

    def total_priority(self):
        return self.tree[0]
```

```

self.tree = SumTree(capacity)
self.alpha = alpha

def _get_priority(self, error):
    return (error + 1e-5) ** self.alpha

def push(self, error, *args):
    priority = self._get_priority(error)
    self.tree.add(priority, Transition(*args))

def sample(self, batch_size, beta=0.4):
    batch = []
    idxs = []
    segment = self.tree.total_priority() / batch_size
    priorities = []

    for i in range(batch_size):
        a = segment * i
        b = segment * (i + 1)
        s = random.uniform(a, b)
        idx, priority, data = self.tree.get_leaf(s)
        batch.append(data)
        idxs.append(idx)
        priorities.append(priority)

    sampling_probabilities = priorities / self.tree.total_priority()
    is_weight = np.power(self.tree.capacity * sampling_probabilities, -beta)
    is_weight /= is_weight.max()

    return batch, idxs, is_weight

def update_priorities(self, idxs, errors):
    for idx, error in zip(idxs, errors):
        priority = self._get_priority(error)
        self.tree.update(idx, priority)

```

(2) 运行结果截图

```

Complete
The training time is 316.85843682
<Figure size 432x288 with 0 Axes>

```

(3) 对比分析

与初始代码结果对比分析，从训练速度、收敛效果等方面进行分析。

每个训练步骤的计算时间相对较长，因为需要计算 TD 误差并更新 SumTree 结构。采样过程中需要在 SumTree 中查找对应的叶节点，增加了额外的计算开销。

由于对 TD 误差大的经验进行优先采样，重要经验被更频繁地利用，从而加快了学习进程。

在相同的训练步骤内，通常可以更快达到一定的奖励阈值，表明模型收敛速度更快。

样本效率高，能更好地利用每次训练步骤，使得最终表现更好；加权重要性采样减少了偏差，提高了策略的稳定性和鲁棒性。

3. 优化三

(1) 优化代码描述

代码截图粘贴于此，并简单描述优化内容。

加入重要性采样权重 (is_weights): 在计算损失时考虑采样的权重，减少优先经验回放引入的偏差。

批量损失加权: 使用 SmoothL1Loss 时，结合 is_weights 对每个样本的损失

失进行加权平均，提高稳定性。

更新优先级：在每次优化后，根据计算出的损失更新经验回放中对对应样本的优先级，确保重要样本在未来被更频繁地采样。

```
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return

    transitions, idxs, is_weights = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))

    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    state_action_values = policy_net(state_batch).gather(1, action_batch)

    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    non_final_next_actions = policy_net(non_final_next_states).argmax(1).unsqueeze(1)
    next_state_values[non_final_mask] = target_net(non_final_next_states).gather(1, non_final_next_actions).squeeze()

    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

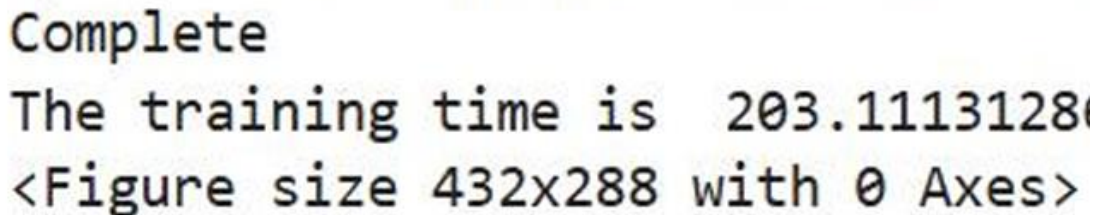
    # Convert is_weights to tensor
    is_weights = torch.tensor(is_weights, device=device, dtype=torch.float32).unsqueeze(1)

    # Apply weights to loss
    criterion = nn.SmoothL1Loss(reduction='none')
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))
    weighted_loss = (loss * is_weights).mean()

    optimizer.zero_grad()
    weighted_loss.backward()
    torch.nn.utils.clip_grad_norm_(policy_net.parameters(), 1.0)
    optimizer.step()

    # Update priorities
    memory.update_priorities(idxs, loss.detach().cpu().numpy())
```

(2) 运行结果截图



Complete
The training time is 203.1113128
<Figure size 432x288 with 0 Axes>

(3) 对比分析

与初始代码结果对比分析，从训练速度、收敛效果等方面进行分析。

引入优先经验回放和重要性采样权重，使每个优化步骤稍微复杂，但由于样本利用率的提高，整体训练速度加快。虽然单步时间增加，但总体训练时间减少。

样本效率高，能更好地利用每次训练步骤，使得最终表现更好。加权重要性采样减少了偏差，提高了策略的稳定性和鲁棒性。通过优先经验回放，使得重要的经验被更频繁地使用，显著提高了收敛速度。加权重要性采样减少了偏差，使模型更稳定。

4. 优化四

(1) 优化代码描述

代码截图粘贴于此，并简单描述优化内容。

修改 batch_size=256;

```
# BATCH_SIZE is the number of transitions sampled from the replay buffer
# GAMMA is the discount factor as mentioned in the previous section
# EPS_START is the starting value of epsilon
# EPS_END is the final value of epsilon
# EPS_DECAY controls the rate of exponential decay of epsilon, higher means a slower decay
# TAU is the update rate of the target network
# LR is the learning rate of the ``AdamW`` optimizer
BATCH_SIZE = 256
GAMMA = 0.99
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 1000
TAU = 0.005
LR = 1e-4
```

(2) 运行结果截图

```
Episode 256, Cumulative Reward: 500.0
Complete
The training time is 113.70201158523
<Figure size 432x288 with 0 Axes>
```

(3) 对比分析

与初始代码结果对比分析，从训练速度、收敛效果等方面进行分析。

虽然单步训练时间增加，但由于每次更新的样本数量增加，总体收敛所需的训练步骤可能减少，整体训练时间减少。

由于训练过程更稳定，噪声更小，最终模型的表现更好，平均奖励更高，收敛效率也更好。