姓名：吕奇澹
学号：SA23006171

实验目的

1. 熟悉了解 DQN
2. 实现 DQN 算法
3. 实现 DQN 的改进算法 DoubleDQN, DuelingDQN 和 DuelingDoubleDQN

实验原理

实验网络：不采用表格，而使用神经网络拟合状态-动作价值函数 Q-Function

Experience Replay

由于样本是从游戏中的连续帧获取的，样本间的关联性较高。若无 experience replay（经验重放），算法在一段时间内可能会沿同一方向进行梯度下降。这种情况下，以同样的步长直接计算梯度可能导致不收敛。为解决此问题，experience replay 从一个内存池中随机选择一些经验样本，再基于这些样本计算梯度，从而避免了因样本连续性过强而引发的问题。
Experience Replay 的主要优势包括：

1. 高效的数据利用：每个样本可以被重复使用多次，提高了样本的使用效率。
2. 减少样本相关性：连续样本之间的高相关性可能导致参数更新时方差较大。通过采用该机制，可以有效降低样本间的相关性，从而减少更新过程中的方差。

Target Net

引入 Target Net 后，在一段时间里目标 Q 值使保持不变的，一定程度降低了当前 Q 值和目标 Q 值的相关性，提高了算法稳定性。用另一个 Target Net 产生 Target Q 值

实验内容

本文档首先构建了一个基础模型 BaseDQN，并定义了算法的基本流程。不同的 DQN 变体可以通过重写 BaseDQN 中的某些函数来实现。接下来，我们将详细介绍基础模型 BaseDQN 的结构和功能。之后，将分别探讨 DQN、DoubleDQN、DuelingDQN 和 DuelingDoubleDQN 这几种变种，并阐述它们相对于 BaseDQN 的具体改进和特点。

BaseDQN

```
class BaseDQN(metaclass=abc.ABCMeta):
    def __init__(self):
        super(BaseDQN, self).__init__()
        self.learn_step_counter = 0
        self.eval_net = None
        self.target_net = None
        self.optimizer = None
        self.loss_func = None

    def choose_action(self, state, EPSILON=1.0):
        state = torch.tensor(state, dtype=torch.float).to(device)
        if np.random.random() > EPSILON:  # random number
```

```
            # greedy policy
            with torch.no_grad():
                action_value = self.eval_net.forward(state)
            action = torch.argmax(action_value).item()
        else:
            # random policy
            action = np.random.randint(0, NUM_ACTIONS)  # int random number
        return action

    def model_step(self):
        self.learn_step_counter += 1
        if self.learn_step_counter % Q_NETWORK_ITERATION == 0:
            self.target_net.load_state_dict(self.eval_net.state_dict())
        if self.learn_step_counter % SAVING_IETRATION == 0:
            self.save_train_model(self.learn_step_counter)

    def update(self, preds: torch.Tensor, targets: torch.Tensor):
        loss = self.loss_func(preds, targets)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    @abc.abstractmethod
    def evaluating(self, samples: List[Data]) -> Tuple[torch.Tensor,
torch.Tensor]:
        raise NotImplementedError

    def learn(self, samples):
        # update the parameters
        self.model_step()
        preds, targets = self.evaluating(samples)
        self.update(preds, targets)

    def save_train_model(self, epoch):
        torch.save(self.eval_net.state_dict(), f"
{SAVE_PATH_PREFIX}/ckpt/{epoch}.pth")

    def load_net(self, file):
        self.eval_net.load_state_dict(torch.load(file))
        self.target_net.load_state_dict(torch.load(file))

    def name(self):
        return self.__class__.__name__
```

1. 核心功能之一是 **evaluating** 函数。不同的 DQN 变体将通过不同的方法来计算预测值（**preds**）和目标值（**targets**）。
2. **choose_action** 函数基于 ϵ-greedy 策略来选择动作，确保了探索和利用之间的平衡。
3. **model_step** 负责处理两个神经网络之间的同步问题，保证算法的稳定性。
4. **update** 函数根据 **preds** 和 **targets** 来更新评估网络（**eval_net**），是学习过程的关键部分。
5. **learn** 函数将 **model_step**、**update** 和 **evaluating** 整合在一起，构成了完整的学习流程，确保了模型的有效训练和优化。

DQN

```
def evaluating(self, samples: List[Data]) -> Tuple[torch.Tensor, torch.Tensor]:
    states, actions, rewards, next_states, non_ends = samples_to_tensors(samples)
    with torch.no_grad():
        target_next_q: torch.Tensor = self.target_net(next_states)
        next_qs = target_next_q.max(1)[0]
        targets = rewards + GAMMA * next_qs * non_ends
    qs = self.eval_net(states)
    preds = qs[torch.arange(BATCH_SIZE), actions]
    return preds, targets
```

DoubleDQN

采用 eval net 的动作而不是 target net 的收益最大动作计算 Qvalue。

```
def evaluating(self, samples: List[Data]) -> Tuple[Tensor, Tensor]:
    states, actions, rewards, next_states, non_ends = samples_to_tensors(samples)
    with torch.no_grad():
        eval_next_q: torch.Tensor = self.eval_net(next_states)
        next_actions = eval_next_q.max(1)[1]
        target_next_q: torch.Tensor = self.target_net(next_states)
        next_qs = target_next_q[torch.arange(BATCH_SIZE), next_actions]
        targets = rewards + GAMMA * next_qs * non_ends
    eval_q = self.eval_net(states)
    preds = eval_q[torch.arange(BATCH_SIZE), actions]
    return preds, targets
```

DuelingDQN

将 Vnet 和 Qnet 的计算结果综合起来，用以估算 Q 值。

```
def evaluating(self, samples: List[Data]) -> Tuple[Tensor, Tensor]:
    states, actions, rewards, next_states, non_ends = samples_to_tensors(samples)
    with torch.no_grad():
        target_next_v, target_next_q = self.target_net(next_states)
        next_qs = target_next_v + target_next_q - target_next_q.mean(dim=1,
keepdim=True)
        next_qs = next_qs.max(1)[0]
        targets = rewards + GAMMA * next_qs * non_ends
    eval_v, eval_q = self.eval_net(states)
    eval_q = eval_v + eval_q - eval_q.mean(dim=1, keepdim=True)
    preds = eval_q[torch.arange(BATCH_SIZE), actions]
    return preds, targets
```

特别指出，由于 Dueling 网络的结构与传统 DQN 存在差异，因此在 Dueling 模型中实现的 **choose_action** 函数需要进行适当的调整。这一改动是为了确保该函数能够适应 Dueling 网络独特的架构，并在此基础上有效地进行动作选择。

```
    def choose_action(self, state, EPSILON=1.0):
        state = torch.tensor(state, dtype=torch.float).to(device)
        if np.random.random() > EPSILON:  # random number
            # greedy policy
            with torch.no_grad():
                _, action_value = self.eval_net(state)
            action = torch.argmax(action_value).item()
        else:
            # random policy
            action = np.random.randint(0, NUM_ACTIONS)  # int random number
        return action
```

DuelingDoubleDQN

```
    def evaluating(self, samples: List[Data]) -> Tuple[Tensor, Tensor]:
        states, actions, rewards, next_states, non_ends =
samples_to_tensors(samples)

        with torch.no_grad():
            target_next_v, target_next_q = self.target_net(next_states)
            next_qs = target_next_v + target_next_q - target_next_q.mean(dim=1,
keepdim=True)
            next_qs = next_qs.max(1)[0]
            targets = rewards + GAMMA * next_qs * non_ends

        eval_v, eval_q = self.eval_net(states)
        eval_q = eval_v + eval_q - eval_q.mean(dim=1, keepdim=True)
        preds = eval_q[torch.arange(BATCH_SIZE), actions]
        return preds, targets
```

Memory

```
class Memory:
    def __init__(self, capacity: int):
        self.buffer: collections.deque[Data] = collections.deque(maxlen=capacity)

    def __len__(self):
        return len(self.buffer)

    def size(self):
        return len(self.buffer)

    def set(self, data: Data):
        # TODO
        self.buffer.append(data)

    def get(self, batch_size: int) -> List[Data]:
        # TODO
        assert batch_size < len(self.buffer), "no enough samples to sample from"
        return random.sample(self.buffer, batch_size)

    def remove_last(self):
        """删除最后加入的 episode 的数据"""
```

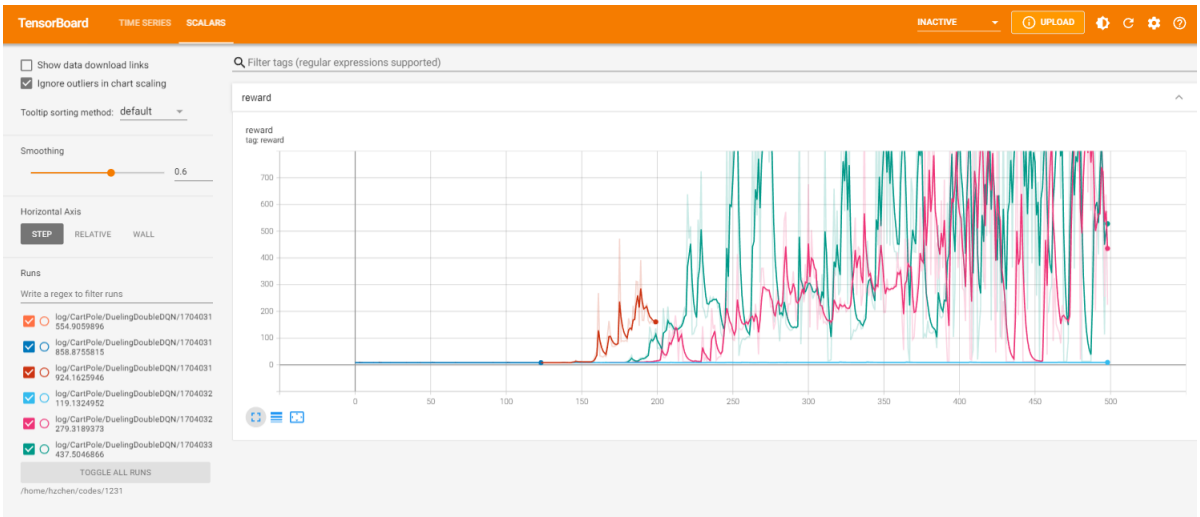
```
        episode_last = self.buffer[-1].episode
        while self.buffer[-1].episode == episode_last and len(self.buffer) >=
MIN_CAPACITY:
            self.buffer.pop()
```

在实验过程中，我们观察到当某些 episode 过长时，会造成 memory 中的数据大量集中于这个特定 episode，从而导致数据之间的相关性过高。这种高度相关的数据会使神经网络难以收敛，从而影响训练效果。为了解决这个问题，我们在训练过程中引入了 episode 长度的限制，将其最大长度设定为 memory 容量的十分之一。这样的限制有助于减少数据的相关性，促进神经网络的有效学习和收敛。

```
# 后一个条件是避免同一个episode的把memory的数据占满，导致不是iid的数据，从而训练效果糟糕
while not done and (num_step <= 0.1 * MEMORY_CAPACITY or TEST):
    num_step += 1
    # choose best action
    action = model.choose_action(state=state, EPSILON=EPSILON if not TEST else 0)
    # observe next state and reward
    next_state, reward, done, truncated, info = env.step(action)
    ep_reward += reward
    ...
    state = next_state
...
print(f"episode: {i} , the episode reward is {round(ep_reward, 3)}")
```

实验结果

CartPole



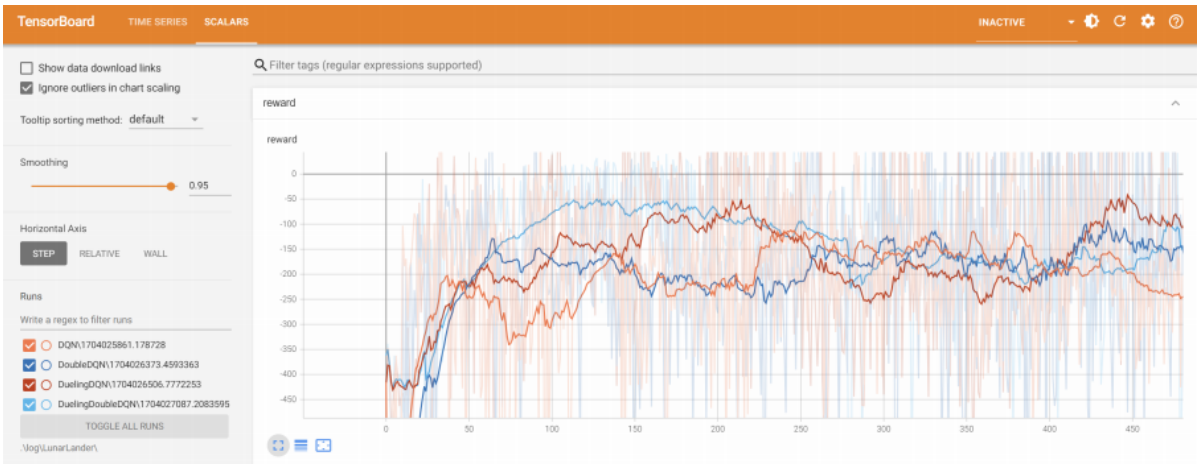
从图表中可以观察到，未采用 Dueling 结构的 DQN 和 DoubleDQN 在训练的后期阶段均出现了收敛问题。相比之下，采用 Dueling 结构的方案均显示出正常的收敛行为，表明 Dueling 架构在稳定性方面具有优势。

就收敛速度而言，传统 DQN 显示出了最快的收敛速度，而其他三种方法（DoubleDQN、DuelingDQN 和 DuelingDoubleDQN）在这方面的差异不大。

在最优性能方面，这四种方法表现出了相同的结果，都达到了每个 episode 的最大步数限制，此例中为 1000 步。这表明尽管它们在收敛速度和稳定性方面有所差异，但在最终性能上能够达到相似的水平。

从图中可以观察到，采用 Dueling 架构的 DuelingDQN 和 DuelingDoubleDQN 在后期都表现出了较为明显的波动。相比之下，DoubleDQN 也显示了一定程度的波动，但 DQN 本身展现了更好的稳定性。
在收敛速度方面，DuelingDQN 的收敛速度是最慢的。而其他三种方法（DQN、DoubleDQN、DuelingDoubleDQN）在收敛速度上的差异不大。
就最优效果而言，这四种方法的表现是一致的，都达到了最大奖励 -100。这表明尽管他们在波动性和收敛速度上有所不同，但在最终效果上仍然相同。

LunarLander



由于奖励值的波动较大，图表中对数据进行了显著的平滑处理。
就最优性能而言，DuelingDQN 和 DuelingDoubleDQN 两者的效果相近，与此同时，DQN 和 DoubleDQN 的表现也大致相同。总体来看，DuelingDQN 和 DuelingDoubleDQN 的表现优于 DQN 和 DoubleDQN。
关于这四种方法的收敛速度和稳定性，它们之间的差异并不显著，表现出相似的趋势。

实验收获

1. 对 DQN（Deep Q-Network）算法进行了深入的理解和熟悉，掌握了其核心原理和应用场景。
2. 成功实现了 DQN 算法，通过实际编程操作加深了对算法结构和功能的理解。
3. 实现了 DQN 算法的几种改进版本，包括 DoubleDQN、DuelingDQN 和 DuelingDoubleDQN，这一过程不仅增强了对这些高级技术的理解，还提高了解决复杂问题的能力。