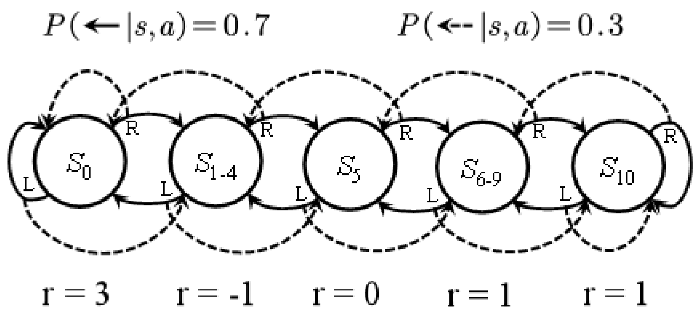
一、　实验内容

题目：11-State ChainWalk MDP



要求：计算并画出利用 bellman 最优算子以及优势学习算子情况下迭代策略的性能界 $||V^* - V^{\pi_k}||_\infty$ 以及动作间隔（action gap）的变化？

二、　实验环境定义与提示

环境特点：

1. 智能体在每个状态下执行“向左”或“向右”动作，按照动作指令转移一个状态的概率为 0.7，按动作指令相反方向转移一个状态的概率为 0.3；左（右）端点状态向左（右）转移状态时保持端点位置不变；
2. 奖励跟所处状态有关，中间状态 s5 奖励为 0，右半部分状态 s6-s10 奖励均为 1；左半部分状态，除左端点 s0 状态的奖励为 3，其余状态 s1-s4 奖励为-1；

提示：

1. 算子形式

Bellman最优算子：$\mathcal{T}^* Q(s,a)=r(s,a)+\gamma\mathbb{E}_{s'\sim P(\cdot|s,a)}[\max_{a'}Q(s',a')]$

优势学习算子：

$$\mathcal{T}_{\mathrm{AL}} Q(s,a)=r(s,a)+\alpha(Q(s,a)-\max_{\tilde{a}}Q(s,\tilde{a}))+\gamma\mathbb{E}_{s'\sim P(\cdot|s,a)}[\max_{a'}Q(s',a')]$$

取值$\alpha=0.99$和$\gamma=0.99$

2. 初始 Q 值随机生成，如 10 * np.random.random();

3. V^{π_k} 策略 π_k 的 V 值函数，而 π_k 是根据第 k 次迭代 Q 值诱导的贪婪策略

$\pi_k(s) = \text{argmax}_a Q_k(s,a)$；

真实最优策略为“在任何状态下都执行‘向左’的动作”，那么 Action gap 定义为

$mean_{s \in S}(Q(s, \text{'L'}) - Q(s, \text{'R'}))$

三、 分步解析与代码解析

（1） 定义状态，定义奖励函数$r(s,a)$，定义转移函数$transition(s,a)$

（2） 实现$Bellman$最优算子：$\mathcal{T}^*Q(s,a) = r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)}[\max_{a'} Q(s',a')]$，也就是当前的状态价值$Q(s,a)$为当前的实时奖励$r(s,a)$,加上后续的所有可能转移到状态$s'$的最大状态价值的期望。注意到这个，对于每一个$Q(s,a)$的更新，是对$transition(s,a)$返回的一个期望（$intended$、$opposites$）两个方向的加权和

```python
# Bellman 最优算子
def bellman_optimal_operator(Q):
    """
    Bellman 最优算子: 给定当前 Q，返回对所有 Q(s,a) 的更新结果。
    """
    new_Q = np.zeros((n_states, n_actions))
    for s in range(n_states):
        for a in [0, 1]:
            total = 0.0
            for (s_next, prob) in transition(s, a):
                r = reward(s_next)
                total += prob * (r + gamma * np.max(Q[s_next]))
            new_Q[s, a] = total
    return new_Q
```

（3） 实现优势学习算子：$\mathcal{T}_{\text{AL}}Q(s,a) = r(s,a) + \alpha(Q(s,a) - \max_{\tilde{a}} Q(s,\tilde{a})) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)}[\max_{a'} Q(s',a')]$，值得注意的是，就是在$Bellman$最优算子的基础上，增加了一个$\alpha(Q(s,a) - \max_{\tilde{a}} Q(s,\tilde{a}))$，那么这个项如何理解？当前$(s,a)$的$Q$与这个当前$(s,a)$的最优情况的一个差值，也就是一个优势项，当前动作是最优动作，那么就产生影响，否则，将会产生一个负数，降低次优动作的Q值。

```python
# 优势学习算子
def advantage_learning_operator(Q):
    """
    优势学习更新: 给定当前 Q, 返回对所有 Q(s,a) 的更新结果。
    按照公
式: T_AL Q(s,a) = r(s,a) + α(Q(s,a) - max_a Q(s,a)) + γE[max_a' Q(s',a')
]
    """
    new_Q = np.zeros((n_states, n_actions))
    for s in range(n_states):
        for a in [0,1]:
            # 计算 r(s,a) 项
            r_sa = 0.0
            for (s_next, prob) in transition(s, a):
                r_sa += prob * reward(s_next)
            # 计算 γE[max_a' Q(s',a')] 项
            expected_future = 0.0
            for (s_next, prob) in transition(s, a):
                expected_future += prob * np.max(Q[s_next])
            expected_future *= gamma
            # 计算 α(Q(s,a) - max_a Q(s,a)) 项
            advantage_term = alpha * (Q[s, a] - np.max(Q[s]))
            # 组合所有项
            new_Q[s, a] = r_sa + advantage_term + expected_future
    return new_Q
```

（4） 预处理最优策略下的v^*,由于已经知道最优策略是一直向左，也就是$(s,0)$。

```python
def compute_optimal_V(gamma=0.99, tol=1e-12):
    """
    对"永远向左"这一固定策略做策略评估，返回其状态价值 V^*(s)。
    在题目中已给出该策略就是最优策略。
    """
    V = np.zeros(n_states)
    while True:
        V_old = V.copy()
        for s in range(n_states):
            # 执行"向左"动作后，根据transition(s,0)计算下一步期望
            val = 0.0
            for (s_next, prob) in transition(s, 0):
                r = reward(s_next)
                val += prob * (r + gamma * V_old[s_next])
            V[s] = val
        # 判断收敛
```

```
        if np.max(np.abs(V - V_old)) < tol:
            break
    return V
```

（5） 策略评估，使用的是$Bellman$等式，$Q(s,a)=r(s,a)+\gamma\mathbb{E}_{s'\sim P(\cdot|s,a)}[Q(s',a')]$，专门用于评估固定策略的状态-动作价值$V^{\pi}(s,a)$，$Policy$就是$argmax(Q,axis=1)$的动作策略，我们需要使用$Bellman$等式对于这个固定的策略求解出收敛的策略的价值，而不是$\max\left(Q(s,a)\right)$作为对应的策略的价值，这个是不对的。

```
def evaluate_policy(policy, gamma=0.99, tol=1e-8, max_iter=1000):
    V = np.zeros(n_states)
    for _ in range(max_iter):
        V_old = V.copy()
        for s in range(n_states):
            a = policy[s]
            val = 0.0
            for (s_next, prob) in transition(s, a):
                r = reward(s_next)
                val += prob * (r + gamma * V_old[s_next])
            V[s] = val
        if np.max(np.abs(V - V_old)) < tol:
            break
    return V
```

四、 完整代码

```
import numpy as np
import matplotlib.pyplot as plt
import copy
# 环境参数
n_states = 11  # 一共有11 个状态
n_actions = 2  # 0: 左, 1: 右
gamma = 0.99  # 折扣因子
alpha = 0.99  # 优势学习算子中的学习率
# 设置随机种子，确保结果可重复
np.random.seed(0)
# 奖励函数
def reward(state):
    """到达状态 state 时获得的奖励。"""
    if state == 0:
```

```python
        return 3.0
    elif 1 <= state <= 4:
        return -1.0
    elif state == 5:
        return 0.0
    elif 6 <= state <= 10:
        return 1.0
    else:
        raise ValueError("Invalid state")
# 转移函数
def transition(state, action):
    # action: 0=left, 1=right
    # 如果向左走
    if action == 0:  # left
        # intended: 期望状态,opposite: 相反状态
        intended = state - 1
        opposite = state + 1
    # 如果向右走
    else:  # right
        intended = state + 1
        opposite = state - 1

    # 处理边界情况
    intended = max(0, min(n_states-1, intended))
    opposite = max(0, min(n_states-1, opposite))

    # return [(0.7, intended), (0.3, opposite)]
    return [(intended, 0.7), (opposite, 0.3)]
# 计算真实最优价值函数 V^*
def compute_optimal_V(gamma=0.99, tol=1e-12):
    """
    对"永远向左"这一固定策略做策略评估，返回其状态价值 V^*(s)。
    在题目中已给出该策略就是最优策略。
    """
    V = np.zeros(n_states)
    while True:
        V_old = V.copy()
        for s in range(n_states):
            # 执行"向左"动作后，根据transition(s,0)计算下一步期望
            val = 0.0
            for (s_next, prob) in transition(s, 0):
                r = reward(s_next)
                val += prob * (r + gamma * V_old[s_next])
            V[s] = val
```

```python
        # 判断收敛
        if np.max(np.abs(V - V_old)) < tol:
            break
    return V
# 计算真实最优 V 值
optimal_V = compute_optimal_V(gamma=gamma)
# Bellman 最优算子
def bellman_optimal_operator(Q):
    """
    Bellman 最优算子：给定当前 Q，返回对所有 Q(s,a) 的更新结果。
    """
    new_Q = np.zeros((n_states, n_actions))
    for s in range(n_states):
        for a in [0, 1]:
            total = 0.0
            for (s_next, prob) in transition(s, a):
                r = reward(s_next)
                total += prob * (r + gamma * np.max(Q[s_next]))
            new_Q[s, a] = total
    return new_Q
# 优势学习算子
def advantage_learning_operator(Q):
    """
    优势学习更新：给定当前 Q，返回对所有 Q(s,a) 的更新结果。
    按照公
式: T_AL Q(s,a) = r(s,a) + α(Q(s,a) - max_a Q(s,a)) + γE[max_a' Q(s',a')
]
    """
    new_Q = np.zeros((n_states, n_actions))
    for s in range(n_states):
        for a in [0,1]:
            # 计算 r(s,a) 项
            r_sa = 0.0
            for (s_next, prob) in transition(s, a):
                r_sa += prob * reward(s_next)

            # 计算 γE[max_a' Q(s',a')] 项
            expected_future = 0.0
            for (s_next, prob) in transition(s, a):
                expected_future += prob * np.max(Q[s_next])
            expected_future *= gamma

            # 计算 α(Q(s,a) - max_a Q(s,a)) 项
            advantage_term = alpha * (Q[s, a] - np.max(Q[s]))
```

```python
            # 组合所有项
            new_Q[s, a] = r_sa + advantage_term + expected_future
    return new_Q
def evaluate_policy(policy, gamma=0.99, tol=1e-8, max_iter=1000):
    V = np.zeros(n_states)
    for _ in range(max_iter):
        V_old = V.copy()
        for s in range(n_states):
            a = policy[s]
            val = 0.0
            for (s_next, prob) in transition(s, a):
                r = reward(s_next)
                val += prob * (r + gamma * V_old[s_next])
            V[s] = val
        if np.max(np.abs(V - V_old)) < tol:
            break
    return V
# 计算 V 值函数
def compute_V(Q):
    """计算当前诱导的 V^π(s) = max_a Q(s,a)"""
    return np.max(Q, axis=1)
# 计算性能界 ||V* - V^{π_k}||∞
def compute_performance_bound(optimal_V, current_V):
    """计算与 V^* 的无穷范数差"""
    return np.max(np.abs(optimal_V - current_V))
# 计算动作间隔
def compute_action_gap(Q):
    """计算动作间隔 (Action Gap): 每个状态下左右动作的 Q 值差异, 然后取平均
"""
    gaps = np.zeros(n_states)
    for s in range(n_states):
        gaps[s] = Q[s, 0] - Q[s, 1]  # 左(0) - 右(1)
    return np.mean(gaps)
# 主实验函数
def run_experiment(operator, n_iterations=400):
    # 初始化 Q 值
    np.random.seed(0)
    Q = 10 * np.random.rand(n_states, n_actions)

    performance_bounds = []
    action_gaps = []

    for _ in range(n_iterations):
```

```
        # 应用算子更新Q 值
        Q = operator(Q)

        # 获取当前贪婪策略
        current_policy = np.argmax(Q, axis=1)

        # 评估当前策略的真实价值函数
        current_V = evaluate_policy(current_policy)

        # 计算性能界
        bound = compute_performance_bound(optimal_V, current_V)
        performance_bounds.append(bound)

        # 计算动作间隔
        gap = compute_action_gap(Q)
        action_gaps.append(gap)

    return performance_bounds, action_gaps, Q
# 运行实验
n_iterations = 200
np.random.seed(0)  # 重置随机种子，确保两次实验使用相同的初始Q 值
bellman_bounds, bellman_gaps, Q_bellman = run_experiment(bellman_optima
l_operator, n_iterations)
np.random.seed(0)  # 重置随机种子，确保两次实验使用相同的初始Q 值
al_bounds, al_gaps, Q_adv = run_experiment(advantage_learning_operator,
 n_iterations)
fig, axs = plt.subplots(1, 2, figsize=(14, 5), sharey=False, sharex=Tru
e)
iters = np.arange(n_iterations)
# 左图: 性能界 ||V^* - V^π||∞
axs[0].plot(iters, bellman_bounds, label="Bellman Optimal", lw=2)
axs[0].plot(iters, al_bounds, label="Advantage Learning", lw=2)
axs[0].set_xlabel("Iterations")
axs[0].set_ylabel(r"Performance Bound ||V* - V^{π_k}||∞")
axs[0].set_title("Performance Bound over Iterations")
axs[0].grid(True)
axs[0].legend()
# 右图: 动作间隔 (Action Gap)
axs[1].semilogy(iters, np.abs(bellman_gaps), label="Bellman optimal")
# 使用对数坐标
axs[1].semilogy(iters, np.abs(al_gaps), label="Advantage Learning")  #
使用对数坐标
axs[1].set_xlabel("Iterations")
axs[1].set_ylabel("Action Gap")
```

```
axs[1].set_title("Action Gap over Iterations")
axs[1].grid(True)
axs[1].legend()
plt.tight_layout()
plt.show()
```

五、 评价指标

（1） 评价指标：优势性能，衡量的是当前策略与最优策略的价值函数接近程度，如收敛于 0，说明策略已经收敛至最优，下降速度放映的是算子逼近最优策略的效率

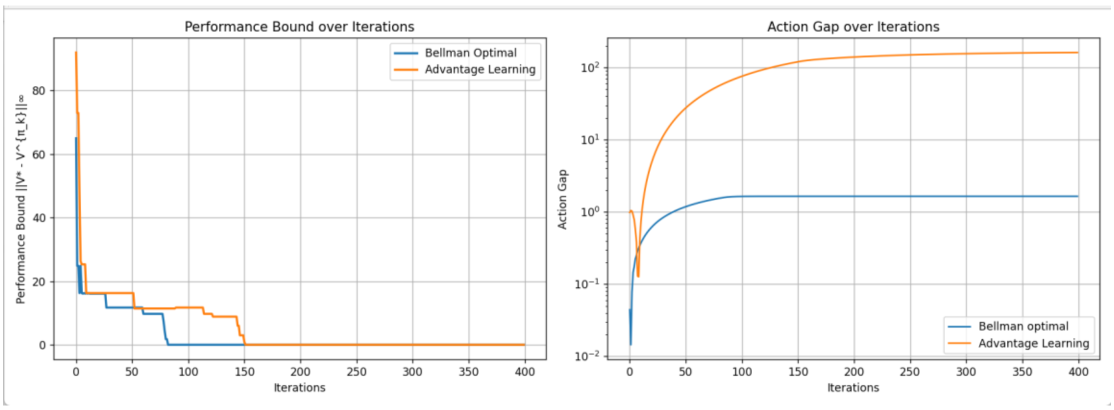
（2） 动作间隔：衡量策略的动作选择正确性，间隔越大，策略越倾向于固定动作，间隔大的策略对环境扰动更不敏感。

六、 结果分析

（1） $case$ 1:参数设置

$gamma = 0.99, alpha = 0.99, n_{iterations} = 200$时，结果如下图：

- 图中，优势学习（橙色）后期优势性能值更低，最终策略更接近理论最优，$Bellman$（蓝色）初期下降更快，说明它短期收敛的效率更高；优势学习（橙色）间隔显著更高，因为$alpha = 0.99$显著放大了最优动作优势，$Bellman$（蓝色）间隔平稳增长，因为依赖环境反馈自然形成差异。
- 对应来说，如果需要快速原型开发，优先选择$Bellman$最优算子，因为初期收敛快，如果需要部署高可靠性策略，选择优势学习算子，因为最终精度高，动作确定性更强。

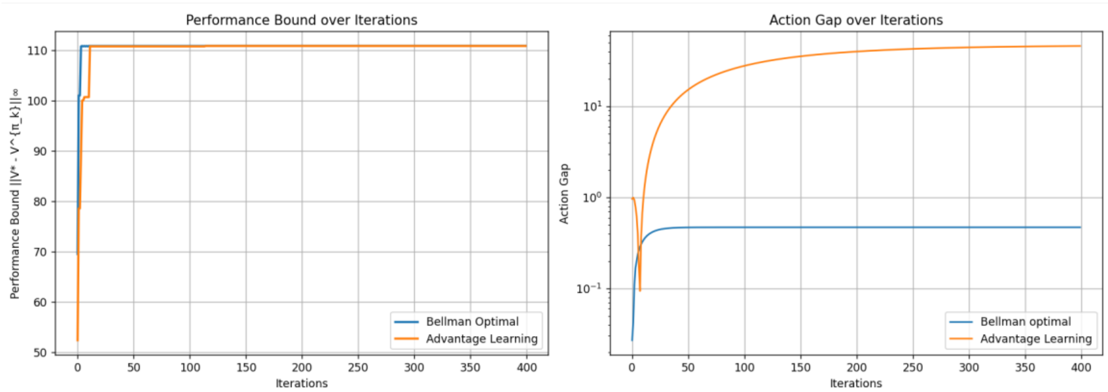


（2） 超参数敏感性测试：探索$gamma$、$alpha$的值对性能界和动作间隔的影响？

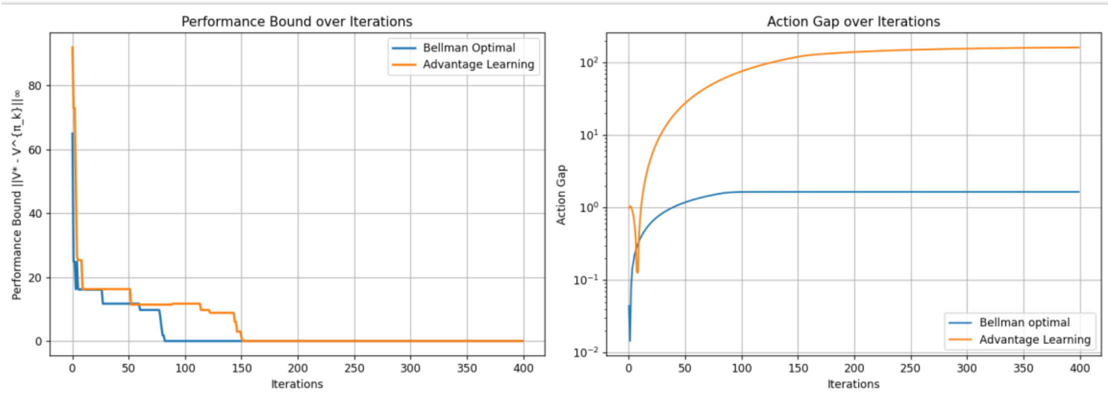
$gamma$：影响的公式部分项$\gamma\mathbb{E}_{s'\sim P(\cdot|s,a)}[\max_{a'} Q(s',a')]$，影响的解释：控制下一个状态的最大$Q$值对当前$Q$值的更新的贡献程度！

设置四组实验，保持$alpha = 0.99, n_iterations = 400$,开始设置四组$gamma = [0.5,0.8,0.9,0.99]$,查看结果：

$gamma = 0.9$,没有收敛于 0，收敛于 110 左右，$action\ gap$ 差值为 45.48



$gamma = 0.99$，同时记录收敛时，$action\ gap$的差值，此时为 159.52

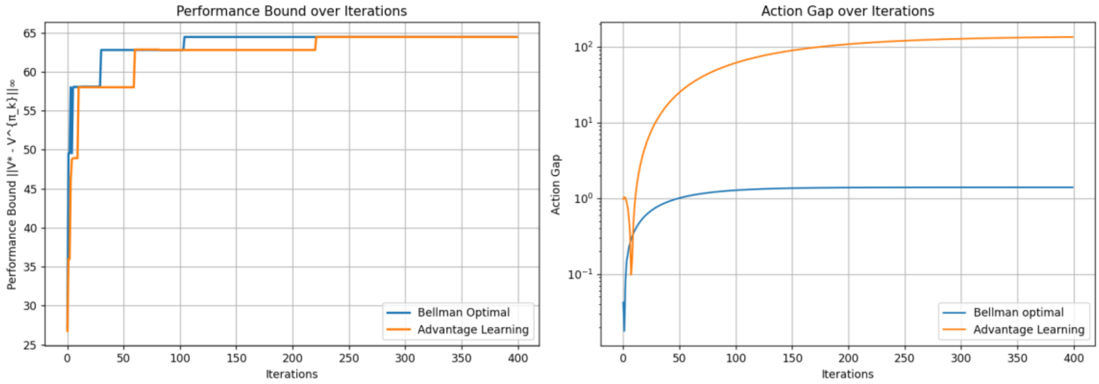


发现$gamma$较小的时候，性能界是没有正确收敛于 0 的，所以$gamma = [0.5,0.8]$的情况图片由于和$gamma = 0.9$的类似，将不展示，反思为什么会出现这种情况？

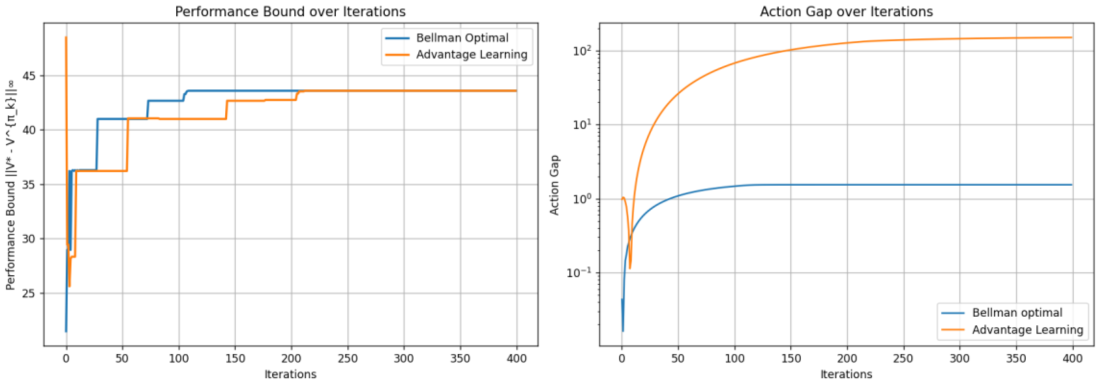
$gamma$较小，性能边界收敛值大，因为长期回报信息被压缩，动作间隔小，因为未来奖励的惩罚力度不足，因为$gamma$和$alpha$的值是需要协同调节的。

下面继续给出$gamma = 0.98$和$gamma = 0.985$的情况：

$gamma = 0.98$,时，$action\ gap$的差距值为 134.12



$gamma = 0.985$,时，$action\ gap$的差距值为 149.51

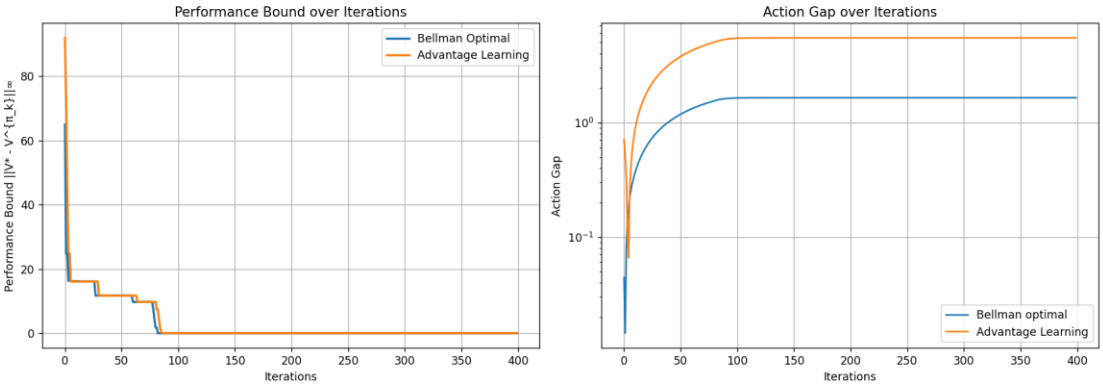


可以发现，随着$gamma$的值越大，$Bellma$算子和优势学习算子更加注重未来的状态的情况，而不是局限于当前的状态的情况，同时随着这个$gamma$值变大，动作间隔也在变大，说明在充分考虑这个未来的情况之后，动作的确定性也在变大!

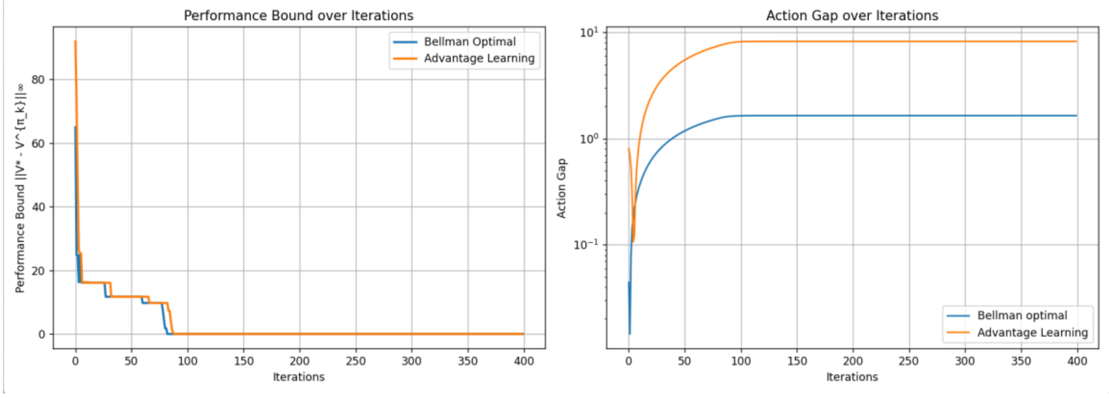
$alpha$:影响部分：影响优势学习算子中的$\alpha(Q(s,a) - \max_{\tilde{a}} Q(s,\tilde{a}))$，也就是对于当前动作与最优动作的修正，当当前的动作就是最优的动作的时候，没有影响，否则将产生一个负数项，会削弱当前状态的Q值。

参数设置与探讨：固定$gamma = 0.99, n_iteration = 400$,探索$alpha = [0.7,0.8,0.9,0.99]$的情况

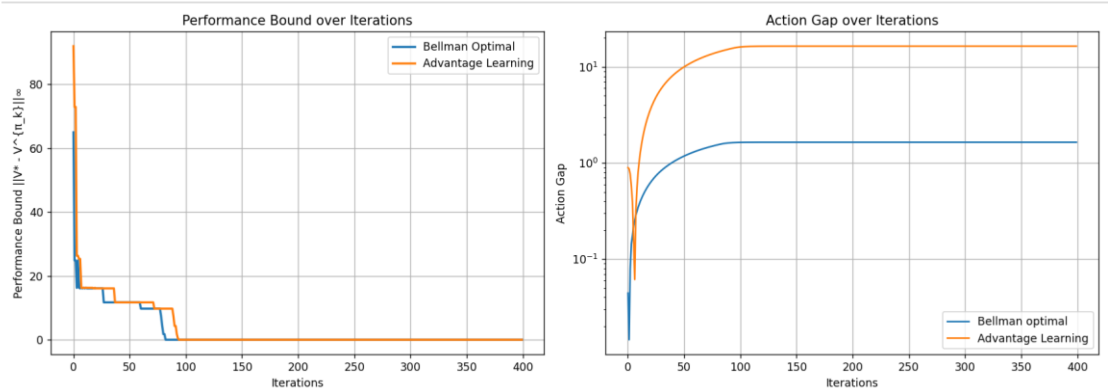
$alpha = 0.7$,时，$action\ gap$的差值为 3.84



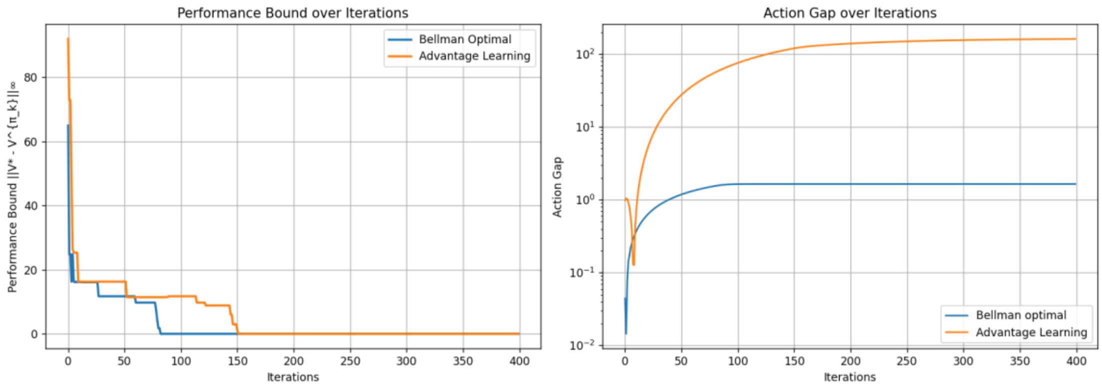
$alpha = 0,8$时，$action\ gap$的差值为 6.58



$alpha = 0.9$时，$action\ gap$的差值为 14.81



$alpha = 0.99$时，$action\ gap$的差值为 159.52



调整$alpha$的时候，影响的是优势学习算子，可以看到，收敛之后的$action\ gap$的差值随着$alpha$的变大而逐渐变大，说明优势学习算子随着对于错误动作的惩罚力度的增加，会让决策过程中，动作趋向正确动作的确定性也增加！

七、 小结

$Bellman$算子适合用于需要短期收敛较快的工作，但是综合性能来说，由于$Bellman$算子相较于优势学习算子，缺少了对于当前动作的惩罚，而导致这个$action\ gap$的值会较小，也就是对于正确动作的确定性会较小，相比之下，优势学习算子的$action\ gap$会随着$alpha$的值变大，最后呈现一种指数级别的增加，也就是对错误动作的惩罚力度增大，会让策略动作更加确定的选择最优动作。