作业4报告

xxx

(南京大学 xxx, 南京 210093)

# 强化学习的方法和过程

**1.1简介**

强化学习是一种通过智能体与环境的交互，不断试探并优化决策策略的方法。其核心思想是**通过试错学习最优策略**，以最大化累积奖励。在CartPole游戏中体现为维持杆子尽可能久的直立且不冲出两边的边界。我们从强化学习的基本结构出发来分析强化学习的方法和过程。

**1.2基本结构**

* **环境**: 指的是智能体交互的场景，提供状态、动作空间、奖励和下一状态。根据[官方文档](https://www.gymlibrary.dev/environments/classic_control/cart_pole/)，CartPole-v1的任务是控制小车向左或向右移动，以保持杆子直立。在本代码中：

（1）**动作空间**：离散的2个动作，0表示向左移动，1表示向右移动。

（2）**状态空间**： 4维连续值，包括Cart Position；Cart Velocity；Pole Angle；Pole Angular Velocity。

（3）**奖励**： 每次动作后，杆子保持直立则奖励为+1，回合结束时不再给奖励。一个episode结束的条件是：杆子的角度超出范围（超过±12°）；小车的位置超出轨道边界（±2.4米）；或者达到最大步数（500步）。

* **智能体**: 智能体是执行动作并学习的主体，由代码中的DQNAgent或DDQNAgent表示。 它根据策略policy选择动作并通过交互获得经验，以优化自身的策略。
* **策略：**策略定义了智能体如何选择动作。在代码中，策略是epsilon-greedy策略：智能体以epsilon概率随机选择动作（探索）；以1-epsilon概率选择预测Q值最大的动作（利用）。epsilon值随着回合数递减，探索与利用的权衡在训练初期倾向探索，后期趋向利用。
* **价值函数**： 智能体通过深度Q网络预测价值函数，即动作-价值函数（Q值）。它表示在某一状态下选择某一动作后的未来累积奖励。

**1.3 训练的方法和过程**

 初始化：

（1）创建环境env，初始化智能体（如DQNAgent）和经验缓冲区buffer。

（2）设置训练参数，如学习率（lr=2e-4）、折扣因子（gamma=0.99）、经验缓冲区大小（buffer\_size=10000）等。

**Main.py中三个变量的作用：**

学习率lr控制网络权重在每次更新中的调整幅度。学习率过高可能导致训练不稳定或不收敛。学习率过低可能导致训练速度过慢或停滞。在强化学习中，学习率影响神经网络对状态-动作价值函数（Q值）的逼近效果。

折扣因子gamma是一个范围在[0, 1]之间的超参数，用于衡量当前奖励与未来奖励的重要性。折扣因子如果gamma接近1，智能体更加关注长期奖励（例如规划更长远的策略）。如果gamma接近0，智能体更注重即时奖励（例如贪婪策略）。在强化学习中，gamma用于估计状态值或动作值时的加权求和公式：

缓冲区大小buffer\_size决定智能体在训练时可以存储多少条交互经验。缓冲区用于存储智能体与环境的交互数据（状态、动作、奖励、下一状态、终止信息）以及提供随机小批量经验样本，打破时间相关性，提升训练稳定性。缓冲区buffer\_size过小会导致训练样本不足，影响策略收敛效果。但同时缓冲区过大会增加存储和计算开销，减慢收敛速度。

 探索与利用：

在每个回合中，智能体调用act函数根据epsilon-greedy策略选择动作：初期高概率随机探索环境，避免陷入局部最优；随着训练进展，逐渐降低epsilon值，更倾向于利用当前最优策略。

**act函数和act\_no\_explore 函数两者区别在于是否采用epsilon-greedy策略**. 关键代码见下：

**act：**

1. **if** np.random.random() > eps:
2. **return** action\_values.argmax(dim=1).item()  # 利用策略
3. **else**:
4. **return** np.random.randint(self.action\_dim)  # 随机探索

**act\_no\_explore：**

1. **return** action\_values.argmax(dim=1).item()  # 总是利用策略

训练过程中采用 act\_no\_explore 函数会导致：（1）模型完全放弃探索，只会按照当前的策略行动。智能体可能陷入局部最优情景，无法发现更优的动作序列；（2）同时经验缓冲区收集的样本将高度相关，数据的多样性会不足，可能导致训练的Q网络过拟合。（3）也可能会使得智能体在训练初期就盲目地选择当前Q值最大的动作，而这些Q值可能是未正确估计的，从而影响策略优化过程。

 状态转移：

智能体在当前状态执行动作，与环境交互后获得：next\_state（下一状态）；reward（动作的奖励）；done（回合是否结束）。

 经验回放：

1. 交互数据存储到缓冲区buffer中，之后从中随机采样小批量数据用于训练：

（2）通过随机采样打破时间相关性，提升模型泛化能力。

 策略更新：

智能体通过深度学习算法更新网络权重：

* 1. 使用目标Q值：Q(s, a) = r + γ \* max Q(s', a')；
  2. 利用均方误差（MSE）计算损失；
  3. 最后通过梯度下降优化模型。

 训练结果可视化反馈（自己增添的基础设施模块）：

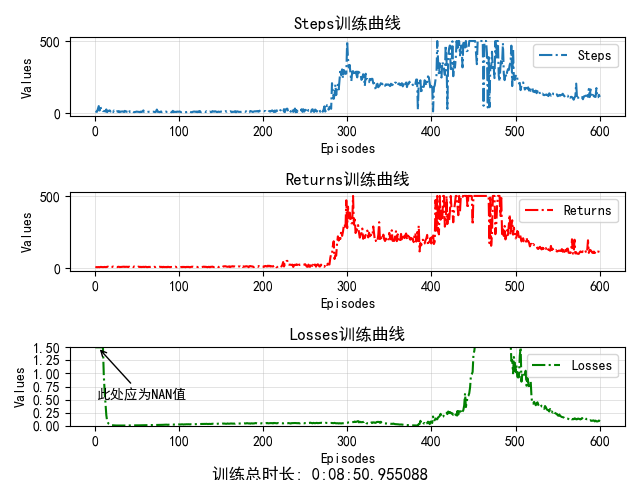
我在main.py中使用matplotlib.pyplot包记录了所有episode所对应的steps、return、loss值并且绘制成图片保存在train log文件夹中，关键代码见下：

1. \_, axes = plt.subplots(3, 1)
2. axes[0].plot(episodes, steps, label="Steps", linestyle="-.")
3. axes[1].plot(episodes, returns, label="Returns", linestyle="-.", color="red")
4. axes[2].plot(episodes, los, label="Losses", linestyle="-.", color="green")
6. axes[0].set\_title("Steps训练曲线")
7. axes[1].set\_title("Returns训练曲线")
8. axes[2].set\_title("Losses训练曲线")
9. axes[2].set\_ylim(0, 1.5)
10. axes[2].set\_yticks([0, 0.25, 0.5, 0.75, 1, 1.25, 1.5])
11. axes[2].annotate("此处应为NAN值", xy=(4, 1.5), xytext=(2, 0.50), arrowprops=dict(arrowstyle='->'))
13. **for** ax **in** axes:
14. ax.grid(alpha=0.5, linewidth=0.5)
15. ax.legend()
16. ax.set\_xlabel("Episodes")
17. ax.set\_ylabel("Values")
18. plt.tight\_layout()
19. plt.savefig(f"./train log/training process {current\_time}.png")
20. plt.show()
21. **print**(f"训练过程图像已保存至./train log/training process {current\_time}.png")

# 修改参数，改善性能

由于有了之前的训练结果可视化反馈，我们可以更加直观地看到改进前后的学习效果。后面我又修改了绘图部分的代码，使之能够在图片最下方显示训练时间。

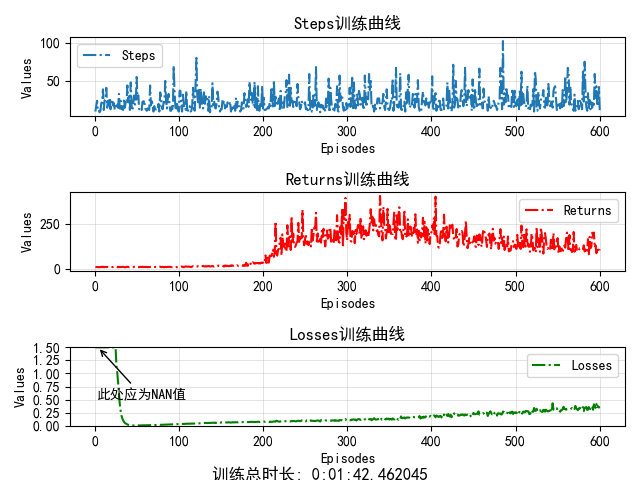
### 2.1改进之前：



agent = DQN，lr = 2e-4，gamma = 0.99，buffer\_size = 10000，batch\_size = 256，（episilon\_start = 0.9，episilon\_end = 0.05，episilon\_decay\_rate = 0.99）。

可以看到损失呈现先增大后减少的趋势，而且Returns在逐渐能达到500步之后，开始不稳定并且衰退到200步左右，说明这个参数对应的效果并不是很好。

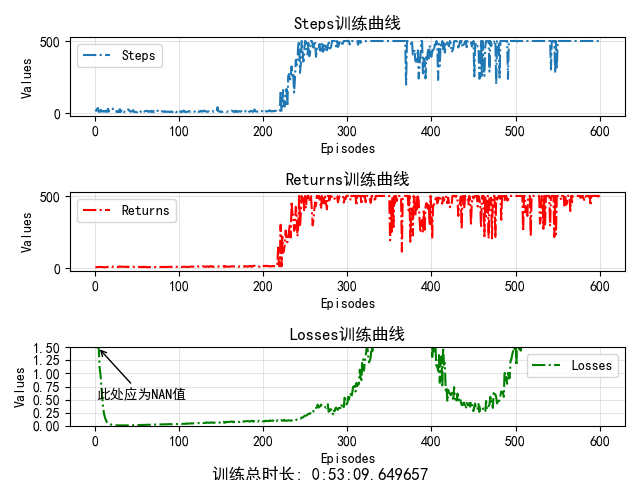
### 2.2尝试改进： 先举出一个训练失败的例子：我们先尝试使用时间更久但是效果理论上更优的参数，为实现这一目的，我们可以降低lr，略增大gamma，增大buffer\_size，增大batch\_size（训练理论上会更平稳），增加探索力度（增大episilon\_start和episilon\_decay\_rate）。训练结果如下：



agent = DQN，lr = 1e-4，gamma = 0.99，buffer\_size = 50000，batch\_size = 512，（episilon\_start = 0.9，episilon\_end = 0.05，episilon\_decay\_rate = 0.999）。

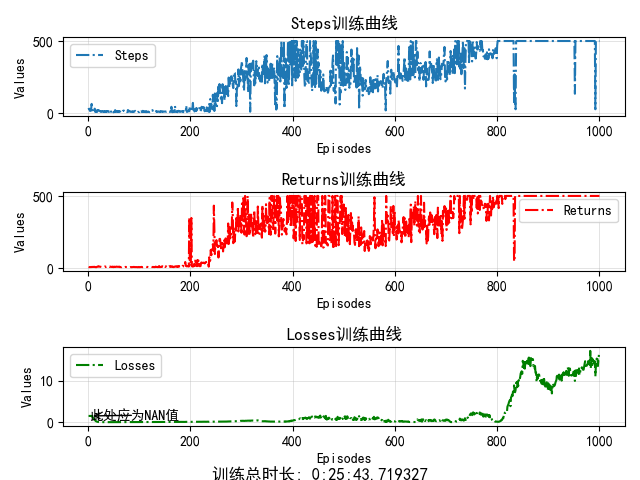
然而出乎意料，这个代码的效果下降了很多，初步猜测是episilon\_start过大而episilon的衰减过慢（参考：，），导致模型探索的占比过大；同时经验缓冲区过大，导致模型对刚开始的错误的Q值函数记忆过于深刻；还有单次训练的批次数量太大导致Q值函数过拟合，以及训练样本数的相对减少。

根据上述的猜测，我们试着使用另一组参数，训练结果如下：



agent = DQN，lr = 1e-4，gamma = 0.99，buffer\_size = 20000，batch\_size = 128，（episilon\_start = 0.9，episilon\_end = 0.05，episilon\_decay\_rate = 0.991）。

这个效果就相当好了，在episode为290和500左右基本上都实现了Steps稳定在500，说明我们之前的猜想基本正确（参考：）。下一步我们增加测试轮数来进一步验证我们的猜想：



agent = DQN，episodes=1000，lr = 1e-4，gamma = 0.99，buffer\_size = 20000，batch\_size = 256，（episilon\_start = 0.9，episilon\_end = 0.05，episilon\_decay\_rate = 0.995）。

可以看到上述参数在1000轮的训练中仍然有较好的性质，说明我们的判断正确。

当然，我们可以通过更改默认的device选项代码来使用GPU。但是作业包里的pytorch版本是不支持cuda的。

# DQN模型和DDQN模型对比

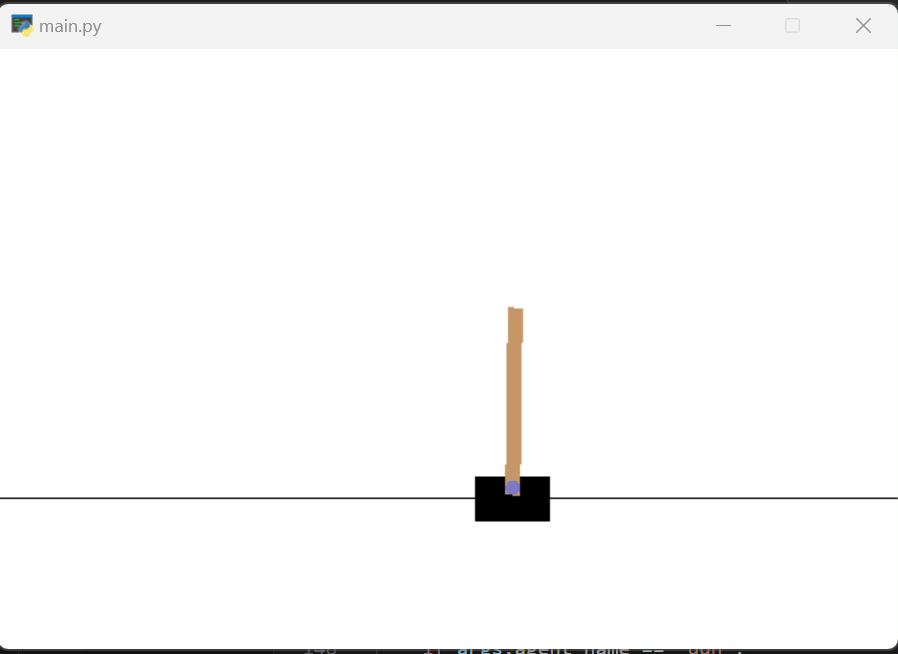
### 3.1代码区别

两者在目标 Q 值的迭代更新上有所不同：

在 DQN 中，目标 Q 值直接从目标网络中选择最大 Q 值：从目标网络中对每个状态选择 Q 值最大的动作，并直接用该值作为目标。这其中潜在的问题是目标 Q 值是通过目标网络直接计算的最大 Q 值，目标网络用于同时选择和评估最大值，可能会高估 Q 值，尤其在复杂或高噪声环境中。

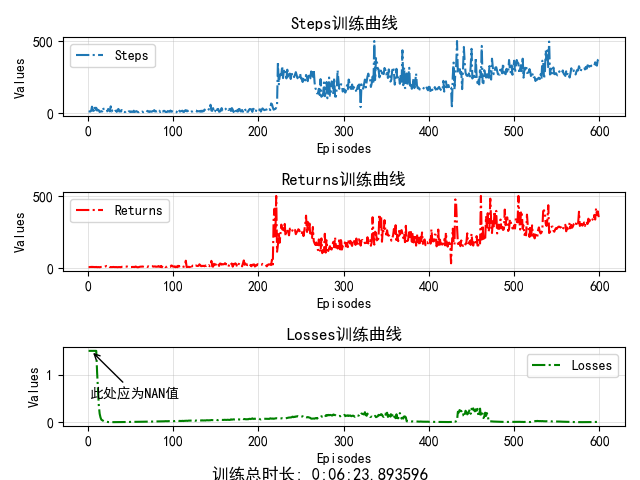
1. Q\_targets\_next = self.qnetwork\_target(next\_states).detach().max(1)[0].unsqueeze(1)
2. Q\_targets = rewards + (gamma \* Q\_targets\_next \* (1 - dones))

而DDQN 使用两个网络分离动作选择和 Q 值评估，其中qnetwork\_local选择下一步的最优动作，而qnetwork\_target 评估由在线网络选择的动作的 Q 值。这样子设计能够避免单一网络同时选择和评估动作所带来的高估问题，更准确地估计动作价值，同时对环境的复杂性有更强的适应性，学习更稳定。相关关键代码见下：

1. action\_max = torch.argmax(self.qnetwork\_local(next\_states), dim=1).unsqueeze(1)
2. Q\_targets = rewards + (gamma \* torch.gather(self.qnetwork\_target(next\_states).detach(), 1, action\_max) \* (1 - dones))

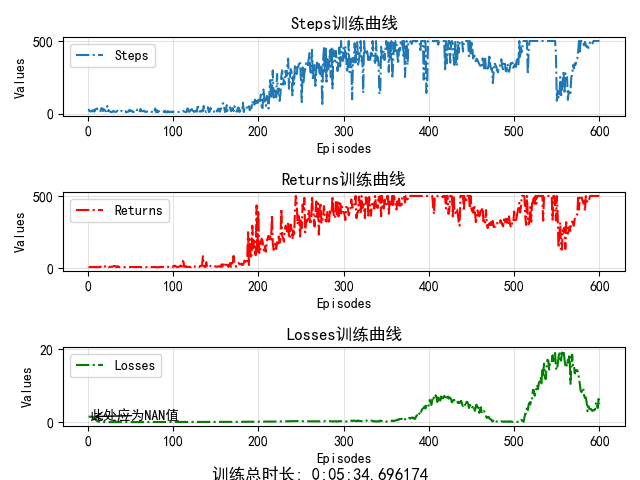
### 3.2效果区别

如下是DDNQ的训练示意图：

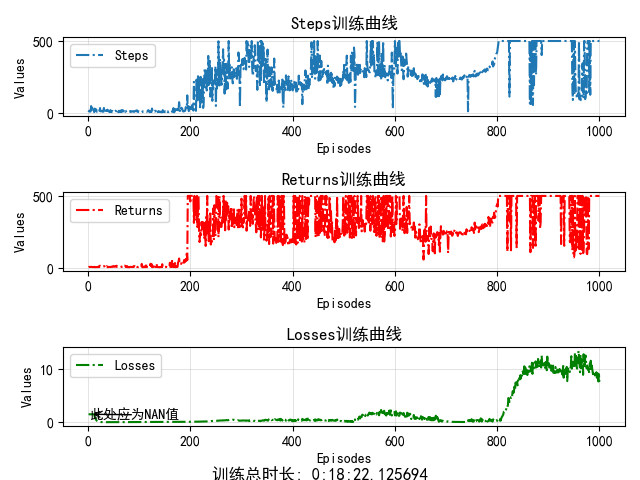


agent = DDQN，lr = 2e-4，gamma = 0.99，buffer\_size = 10000，batch\_size = 256，（episilon\_start = 0.9，episilon\_end = 0.05，episilon\_decay\_rate = 0.99）。

同样的，原始参数训练的效果并不是很好,但是我们可以看到DDQN网络区别于原来DQN网络的地方在于DDQN在episode为500-600的区间内呈现上涨趋势且平均Returns在300左右。我们祭出时间更久但是效果更好的参数来试试，由于DDNQ的网络信息更加丰富，buffer\_size的大小不宜设置过大，我们维持原来的大小10000。



agent = DDQN，lr = 1e-4，gamma = 0.99，buffer\_size = 10000，batch\_size = 128，（episilon\_start = 0.9，episilon\_end = 0.05，episilon\_decay\_rate = 0.995）（参考：）



agent = DDQN，episodes=1000，lr = 1e-4，gamma = 0.99，buffer\_size = 10000，batch\_size = 128，（episilon\_start = 0.9，episilon\_end = 0.05，episilon\_decay\_rate = 0.995）

### ****3.3实验结论****

多次实验比对结论如下：

| **指标** | **DQN** | **DDQN** |
| --- | --- | --- |
| **收敛速度** | 收敛较快 | 稍慢 |
| **累计奖励表现** | 高，但有较大波动 | 稳定，波动较小 |
| **损失波动性** | 较大，训练过程中波动显著 | 较小，更平滑 |
| **稳定性** | 收敛后偶尔回退 | 更稳定，回退现象较少 |

从结果来看，相对于DQN网络而言，DDQN不仅加速了计算时间，其稳定性也更高，并且使得出现训练效果回退的频率更低，因此强化学习的效果也越好。就优化层面而言，我们可以考虑调整软更新里的参数，使得每一次参数变化更加平滑地过渡，减少训练过程中的不稳定性，并且减少过高地Q值估计。

References:

[1] [Cart Pole - Gym Documentation](https://www.gymlibrary.dev/environments/classic_control/cart_pole/)

[2] [强化学习（二）：贪心策略（ε-greedy & UCB）\_蔚-greedy-CSDN博客](https://blog.csdn.net/qq_36426650/article/details/104767998)

[3] [强化学习笔记（二）：DQN与DDQN - 知乎](https://zhuanlan.zhihu.com/p/576414326)

[4] [强化学习入门笔记——Q​-learning从理论到实践 - 知乎](https://zhuanlan.zhihu.com/p/365814943)