倒立摆强化学习算法研究

**课题成员: 赵为之、王英丞**

**指导教师: 谷多玉、叶金毅**

**摘 要：**倒立摆是验证理论算法的有效性的重要工具。本文研究了DQN（Deep Q-Learning）算法在倒立摆方面的表现，针对DQN的问题，提出了把Sarsaλ算法和DQN结合而成的新算法SDQN（Sarsaλ Deep Q-learning），从理论和实践两个角度证明了在倒立摆方面SDQN优于DQN：每一轮训练中SDQN的得分约为DQN的10倍，SDQN的稳定性约为DQN的4倍。

**关键词：**强化学习，倒立摆，DQN，Sarsaλ

Research on Reinforcement Learning Algorithms of Inverted Pendulum

**Author: Weizhi Zhao, Yingcheng Wang**

**Abstract**: Inverted pendulum is an important tool used to evaluate the performance of reinforcement learning algorithm. This essay focus on the performance of the DQN algorithm evaluated by the inverted pendulum. With regards to the limitations of DQN, a new algorithm, SDQN—a hybrid of DQN and Sarsaλ, is created. And have proven its superiority to DQN both theoretically and practically. DQN outperforms DQN by approximately 10 times and shows a stability 4 times of that of the DQN.

**Key word**: Reinforcement Learning, Inverted Pendulum, DQN, Sarsaλ

一 引言

倒立摆，Inverted Pendulum ，是典型的多变量、高阶次 ，非线性、强耦合、自然不稳定系统。倒立摆系统的稳定控制是控制理论中的典型问题 ，在倒立摆的控制过程中能有效反映控制理论中的许多关键问题 ，如非线性问题、鲁棒性问题、随动问题、镇定、跟踪问题等。因此倒立摆系统作为控制理论教学与科研中典型的物理模型 ，常被用来检验新的控制理论和算法的正确性及其在实际应用中的有效性。从 20 世纪 60 年代开始 ，各国的专家学者对倒立摆系统进行了不懈的研究和探索。

倒立摆的控制目标是：倒立摆的控制问题就是使摆杆尽快地达到一个平衡位置，并且使之没有大的振荡和过大的角度和速度。当摆杆到达期望的位置后，系统能克服随机扰动而保持稳定的位置。

传统的DQN（Deep Q-learning）算法通常用深度神经网络以及记忆回放算法替代Q-learning中Q表，深度神经网络可以用非常少的参数拟合非常复杂的函数，并且可以通过训练来高效的逼近目标函数，从而避免Q表训练时间过长、空间复杂度过高的问题。然而由于DQN算法只能单步更新，无法考虑长远利益的问题，难以通用；而且DQN算法有着诸多局限性，其在较高学习率下极易发生参数的震荡，并且DQN在许多问题中有易收敛到局部最优的缺点，限制了其适用范围。但目前针对该算法的神经网络已经提出了大量的优化方式，其神经网络的性能较为优秀。

PID算法（按偏差的比例（P）、积分（I）和微分（D）进行控制的控制器）具有原理简单，易于实现，适用面广，控制参数相互独立，参数的选定比较简单等优点，而且其数学原理可以严格证明。但是PID算法缺乏通用性，只要改变实验条件，就要重新设定参数。所以存在一些局限。

本文还发现另一种有趣的算法——Sarsaλ。研究表明该算法与DQN相比，更易收敛到局部最优，并且收敛过程相较于原始的DQN更加稳定，不易发生参数的震荡。由于Sarsaλ的更新机制属于回合制更新，此算法可以更好的考虑长远利益。例如，在进行走迷宫游戏时，Sarsaλ算法可以在成功走出迷宫后记录下走迷宫的所有步骤，并一次性更新这些步骤对应的奖励值。但是由于Sarsaλ使用Q表记录状态，存在训练时间长，空间复杂度高的问题。

很少有研究集中在改良DQN算法的更新机制。本课题以DQN算法基础，结合Sarsaλ的回合制更新机制，组合成了一种新算法，命名为SDQN（Sarsaλ Deep Q-learning）算法。该算法具有sarsaλ更易收敛到全局最优和收敛过程稳定的特性与DQN神经网络的优点。测试此算法在训练速度、训练效果、稳定性上是否相对DQN算法有所提升。

二 研究现状

2.1 Sarsaλ

Sarsaλ除了在更新时计算q值的动作是实际将要采用的动作之外，还有一个更新全路径的机制。其工作时的图。其本质是认为了每一步动作产生的影响不止于其所导致的下一状态，而是影响了其后的每一个状态，因此在进行完某步之后，应当更新之前路径上所有状态。考虑到更近和出现更多的状态有更大的影响，因此对每个状态-动作二元组用E来表达其对当前状态的影响程度，根据影响程度更新全路径。这种方法最大的优势便是若某个局部不优但在全局中十分优秀的动作被执行，即使在本回合不能立即获得奖励进而更新网络，却能在将来这个动作的效果显现时得到回报而更新。另外。λ和γ都是较大的常数，能够对即使在数十甚至数百个step前的动作做出有效的更新。这种方式较为有效地避免了智能体陷入局部最优的陷阱中。

算法运行流程：

Initialize Q(s,a) arbitrarily, for all ,

Repeat (for each episode):

, for all ,

Initialize S, A

Repeat (for each step of episode):

Take action A, observe R, S’

Choose A’ from S’ using policy derived from Q(e.g., ε-greedy)

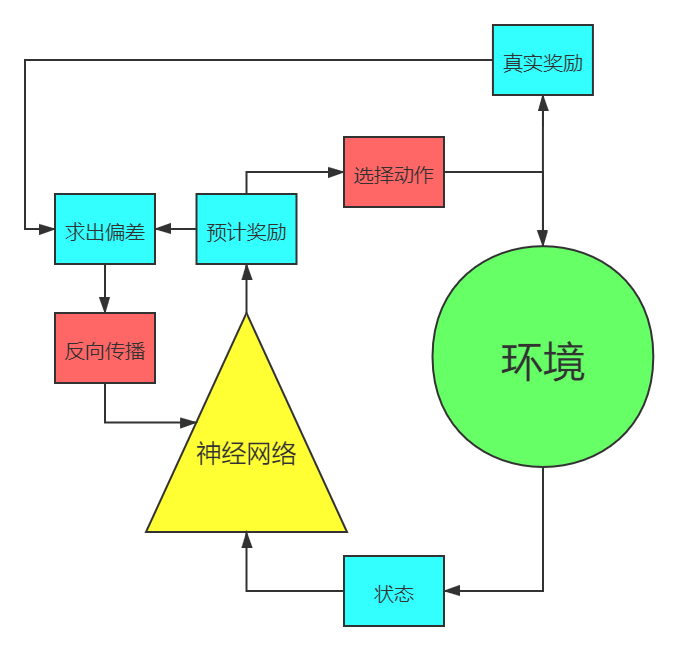
For all , :

Until S is terminal

2.2 DQN

DQN作为最基础的基于价值的强化学习策略，存在着诸多可以完善的点。首先，原始DQN认为整个episode（episode是一轮游戏的意思，没有官方的中文翻译，下文中的episode都是这个意思）本质上是多个状态——动作二元组的累加，而这些状态——动作二元组之间并无关联，就是说只有状态本身有意义而采用何种路径到达该状态没有影响，一个动作影响的方式只有由他导致的下一状态。这导致当获得了一个回报之后，这个回报向前传播到真正产生这个回报的状态-动作二元组的过程是缓慢而低效的。例如，考虑某动作的奖励在n个step后才能获得，则在此设计中需要n个episode才能使得这个奖励影响到产生它的状态-动作二元组，并且其再沿路径向前传播的过程中，每一步都要乘上一个极小的学习率，实质上导致了DQN要求某状态-动作二元组带来的收益必须很快发生而不能有大的滞后。这导致了DQN极易收敛到局部最优，过分追求某动作带来的，即时或者几乎即时的收益而忽略了长期的影响。这一缺点在保持立杆在中部的回报需要数百个step才能得到的平衡杆问题中是几乎致命的。但是，DQN这种将状态——动作二元组之间相互独立考虑的思路，为对DQN中的神经网络进行大量优化提供了条件。

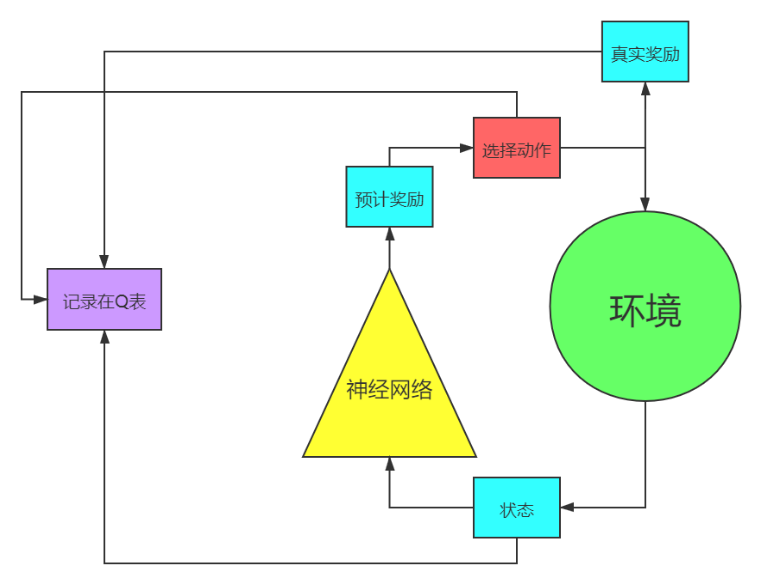
三 研究内容



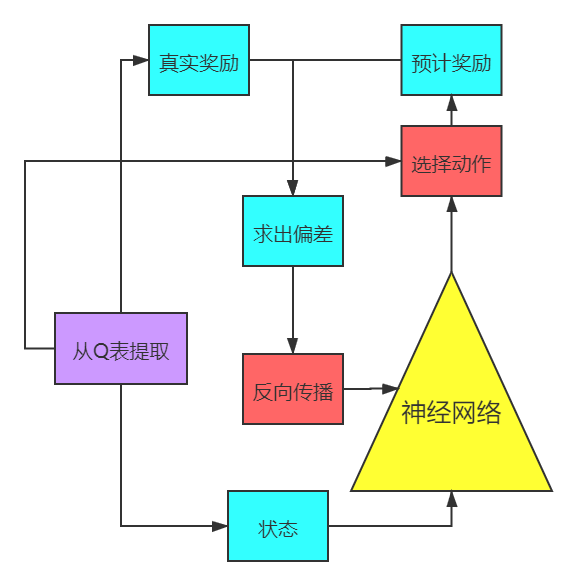
**图1 DQN运行步骤示意图**

这是DQN算法的运行流程图。如图所示，在神经网络根据现有的状态预测出奖励后，直接与真实奖励对比，再进行反向传播。所以DQN是一个单步更新的算法。这代表了DQN不能长远的预计多步之后的奖励。

本文对此问题提出了解决方法，通过结合Sarsaλ中的Q表，使得SDQN算法变为回合制更新的算法。具体运行流程如下图所示。



**图2 SDQN运行步骤示意图 - 回合中**



**图3 SDQN运行步骤示意图 - 回合后**

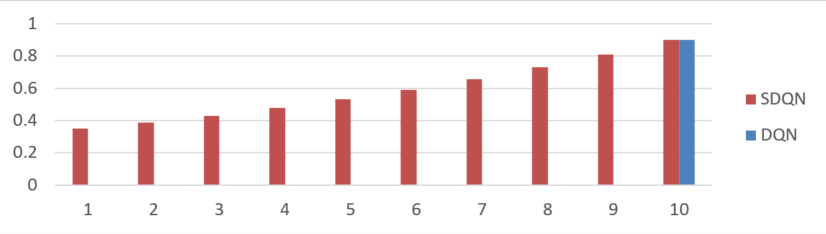
图2表示：在回合中，取消求偏差和反向传播的步骤，把状态、动作和真实奖励记录再Q表中。注意，Q表会根据下一步记录的奖励来更新已经记录的奖励。这样就让Q表中先后发生的动作产生的奖励有联系。神经网络是不能达成Q表的效果的，因为神经网络内部结构复杂，无法从直观上理解其意义，一但一个动作的训练完成，其奖励、训练效果就无法修改。而Q表是一个奖励与状态的对应表，结构简单，容易修改。只要将路径追踪的矩阵与Q表相乘，就可以批量修改奖励值，以达到让算法更有远见的效果。

图3表示：在回合后，取消环境，把真实奖励、动作和状态从Q表中提取出来，重新输入进神经网络中，求出偏差并进行反向传播，训练神经网络。此时，Q表中的奖励值已经经过修改，每一步的奖励之间具有联系。这时，用Q表代替环境，来训练神经网络，其网络就会倾向于把某一动作之后的长远利益也预计到。在回合结束后，清空Q表以释放空间。

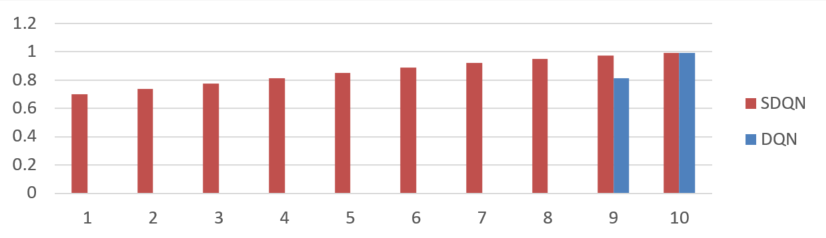
神经网络拥有表达大量状态的能力，但是缺乏可修改性。Q表的空间复杂度极大，但是可以修改，可以做路径追踪。SDQN算法把DQN的神经网络和Sarsaλ的Q表结合，既让算法有神经网络的预测能力和训练能力，又拥有Q表的长远眼光。每回合清空Q表释放了大量空间，降低了空间复杂度。对DQN算法做出了客可观的优化。

四 研究结果

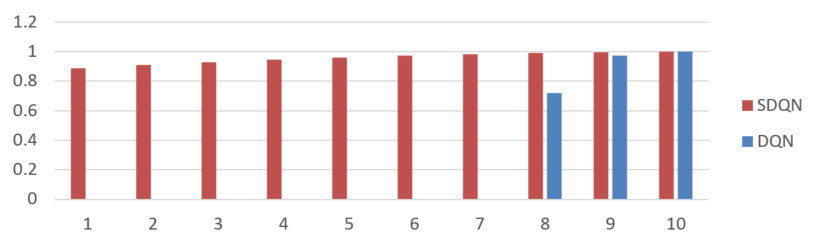
4.1 速度分析



**图4 SDQN和DQN奖励值传播示意图（第一轮）**



**图5 SDQN和DQN奖励值传播示意图（第二轮）**



**图6 SDQN和DQN奖励值传播示意图（第三轮）**

首先，本文从理论上证明了SDQN算法相比DQN算法，更有考虑长远利益的眼光。

假设一次训练共有n步，第n步有奖励为1，其余步奖励为0。

每一步获得的奖励记为。

算法认为，在某一步的预期奖励记为，初始为0。

SDQN和DQN的学习率记为l，SDQN的路径衰减记为d。

本文认为如果某一算法在第一步就能考虑到第n步的奖励，则认为这种算法有长远眼光。

对于DQN算法：(episode是一轮游戏的意思，表示第一轮游戏)

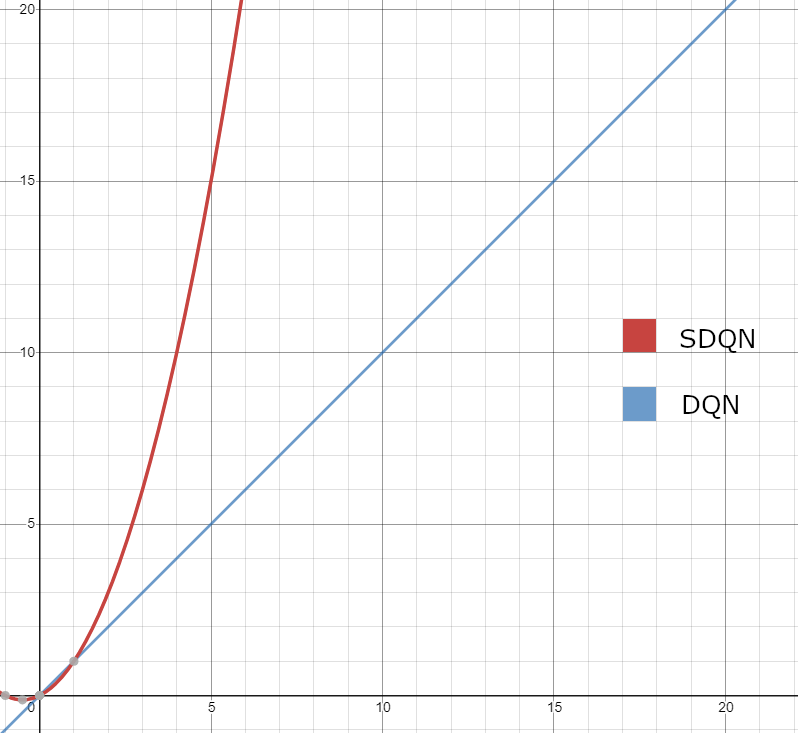
在第n次训练时，才能调整。

对于SDQN算法：

在第一次训练时，就能调整。

SDQN每一次训练时要对Q表进行次矩阵乘法操作，但由于此操作与深度神经网络运算量在同一个数量级，所以几乎不增加时间复杂度，只增加空间复杂度。

从以上计算可以看出，DQN算法在第n次训练才能更新，而SDQN算法在第一次训练就能更新。并且更新值为和，在一般情况下。所以SDQN的大于DQN的，所以SDQN算法更有长远眼光。



**图7 SDQN与DQN单次训练更新次数对比图**

然后，本文再从更新次数的角度，通过计算证明SDQN算法相比DQN算法更新更快，训练速度更快。

设第x步训练会有次更新

本文认为，在学习率l相同的情况下，更新次数越多，说明更新更快，训练速度更快。

对于DQN

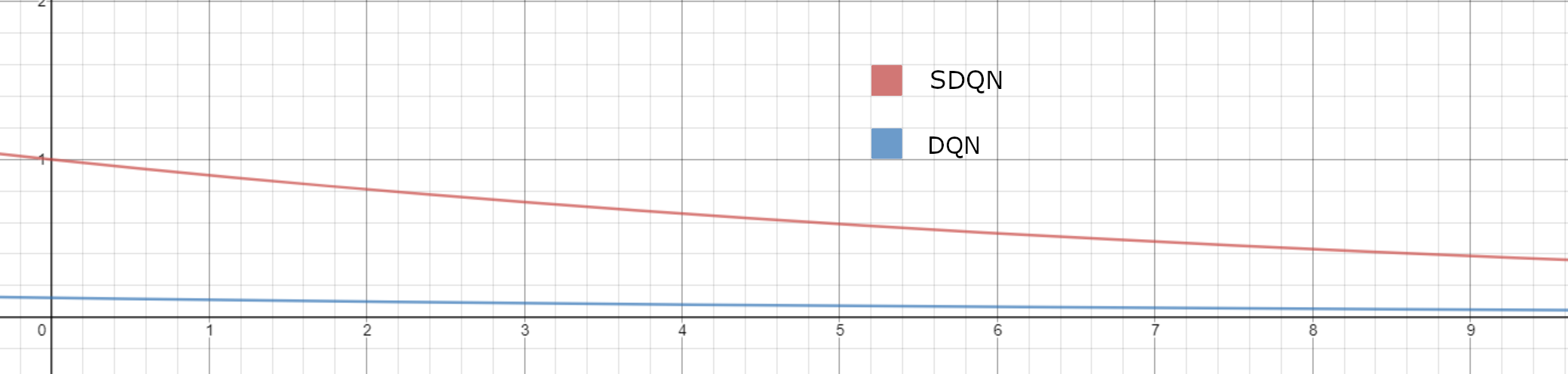
一次训练共更新n次。

对于SDQN

一次训练共更新次

比较这二算法更新次数大小，可以用做差法。

当n>1时，-n>0，这说明每次训练SDQN更新次数大于DQN更新次数。SDQN更新和训练更快，能更快的优化算法。



**图8 SDQN与DQN对n步之前的状态的更新幅度对比**

接下来，本文从更新奖励幅度幅度的角度，通过计算证明SDQN算法相比DQN算法更新更快，训练速度更快。

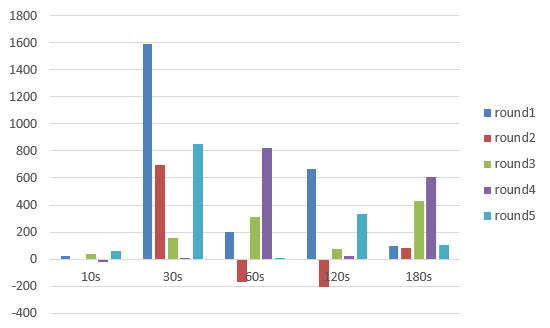
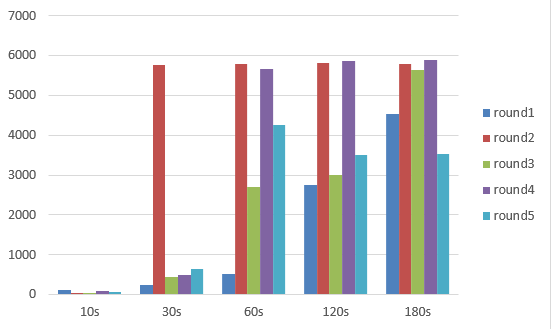
考虑状态s1时采用动作a1，会稳定导致一个长为n的路径，在路径的尽头取得大小为r的额外（未被估计的）奖励。

对于DQN：每个episode中，这个奖励只能传到到路径中的上一个状态，每次传递时的系数为：

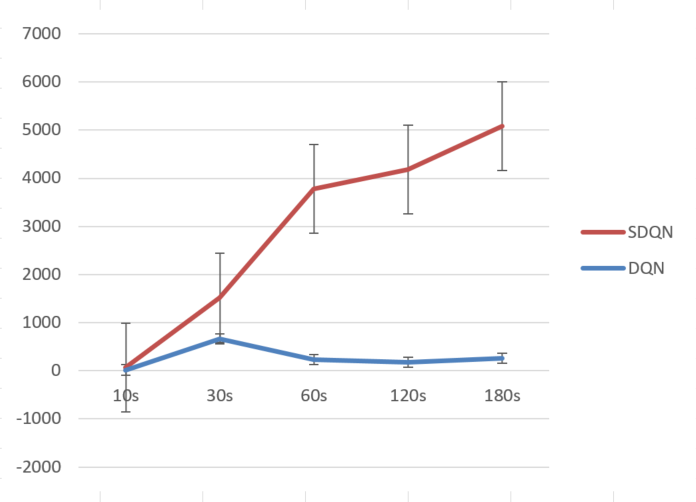
最终经历n个episode之后，的q值第一次被更新，更新幅度为:

对于SDQN算法：当获得奖励之后，立刻对之前路径上所有点进行更新，对于，更新幅度为:

对比：考虑到在倒立摆中，回归中间的操作至少在20个step后才可能产生有意义的回报，故取，结合（事实上在原始DQN中，α的取值会比在该算法中低得多，大约是当前α的3%），，，计算得出第一次修改的幅度差异大约为40个数量级，因而长期的回报在该算法中对q值的影响大得多，更不易陷入局部最优。

上文已经用数学计算证明了SDQN比DQN更有长远眼光，训练效率更高。下面本文用实践来测试DQN与SDQN的训练的分随时间的变化，以此分析DQN与SDQN的训练效率与训练效果。分别用SDQN和DQN进行了5次训练，用程序记录了其在10s、30s、60s、120s、180s时的得分，并取平均，得到了下面三张图表。

**图9、10 DQN（左）与SDQN（右）五轮奖励随时间变化表**



**图11 SDQN算法与DQN算法的奖励平均值变化表**

从图中本文可以看出，SDQN的得分明显高于DQN，并且SDQN的得分总是在增长，而DQN算法不稳定，有增长也有下跌。

总之，可以得出结论，SDQN算法，明显比原版DQN快。

结合数学计算与SDQN算法在训练过程中的表现，本文进行以下分析。

观察了原版DQN控制倒立摆的演示，可以发现其实倒立摆很少倒下，但是经常以非常慢的速度向两边偏移，最终因为出界而游戏结束。结合奖励机制思考，神经网络不知道有一种操作方法就是先使倒立摆倾斜，再把它推回中央，最后平衡住倒立摆。DQN是单步更新的算法，每训练一次只能学会预测某一帧的情况，然而一秒钟有30 帧画面，所以凭借DQN做出先倾斜再回推这种高难度动作是很难的。把倒立摆故意向一端倾斜是一个扣分动作，DQN不会主动做出这个动作，因为它不知道通过这个动作可以获得更长远的利益。

使倒立摆倾斜一个小角度来回到中心这一动作，在刚刚做出时是几乎没有收益的，其收益需要数十step之后才开始显现，且该收益本身较小。该收益在DQN中沿路径传播时，每到上一step都会被乘上学习率，考虑到为了稳定性，学习率learning\_rate取值为0.01，该收益在传播数次后就接近于0，更为严重的是这种回溯往往在一个episode中发生次数较少，甚至需要数十个episode后最初的回报的影响才能到达产生它的状态-动作二元组。因此这个动作的q值会往往小于局部最优的保持直立而不回归中心，且q值需要大量时间才能传到真正产生了回报的状态-动作二元组。q值的传播微弱而缓慢。dqn无法在短时间内收敛到全局最优。

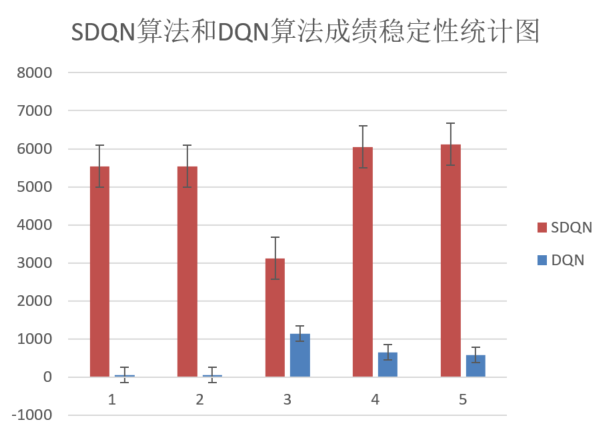
然是通过把DQN和Sarsaλ结合，每个step都能影响到其中所有动作，所以算法就有了更好的计算长远利益的能力。观察了新算法的训练过程。再两三次出界之后，SDQN算法就学会了使倒立摆倾斜从而推回中央的技巧，到了第60、70轮，倒立摆就像静止了一样，几乎看不到不稳定的情况了。

这是利用了衍生于sarsaλ的更新方式，考虑路径上每个状态-动作二元组对其后路径的影响。这时由回到中部这一动作产生的回报较为高效地在每个episode中传播，事实上，回到中部这一动作带来的收益在获得后，立刻传播到了产生它的状态-动作二元组，并且每向前传播一次，q值衰减极小（lambda=0.988，gamma=0.9,因而只衰减到约90%）。q值高效而快速地传播，故而智能体快速收敛到全局最优。

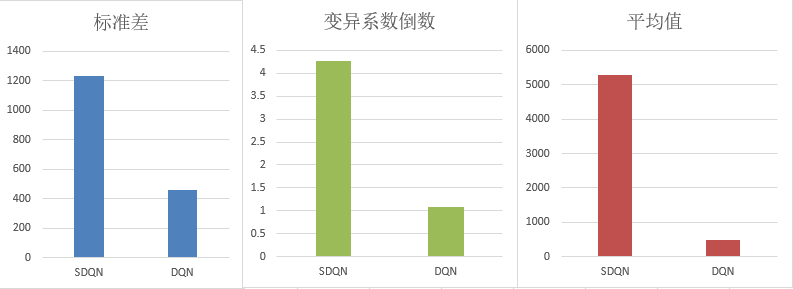
4.2 可靠性分析

因为电脑硬件性能限制，本文把每一轮的步数限制在5000步，达到5000步后强制结束，运行5000步的满分为6250。

分别用两种算法训练5次，每次2分钟，记录在2分钟时的得分，比较得分是否出现了较大的波动。并求出标准差、平均数、变异系数。



**图12 SDQN算法与DQN算法的成绩稳定性统计表**

****

**图13 SDQN算法与DQN算法的标准差（越小越好）平均值（越大越好）变异系数倒数（越大越好）**

如图所示，SDQN算法相比DQN算法要稳定。虽然新算法的成绩标准差为1233，大于老算法的标准差457。但是因为新算法的平均分更高，所以新算法的变异系数为0.23，远小于老算法的0.92。变异系数是标准差除以平均数，用来衡量数据的离散程度。通过变异系数可以看出，SDQN算法比DQN算法要稳定很多。

五 结论和展望

SDQN在本项目中被证明是更优于DQN的一种算法，它收敛更加快速和稳定。

DQN在本次测试中，较快速地收敛到了局部最优，能够保持直立却不能稳定在中心，此现象在超过四千个episode后仍然未改变，推断DQN在本任务中陷入了局部最优。

SDQN快速地收敛，能长时间地保持在中心直立，并且在后续的训练中，成绩稳定而不发生突然的下降。认为SDQN有效地收敛到了全局最优。

这些对比说明了SDQN能够在较高学习率下稳定快速收敛，且更不易陷入局部最优，这主要是由于结合了sarsaλ的算法通过对路径上的每个状态记录重要程度，然后整体性地按回合更新，避免了DQN每步直接更新导致的陷入局部最优和收敛过程中参数的震荡。

目前对神经网络的考察缺乏有效的方法，只能给出猜测，然后通过实验验证。例如在本项目中，对于SDQN优于DQN的原因只给出了猜想，并简单的通过数学进行定性的证明。在将来的研究中，除了不断提出，改进新算法，也有必要加强神经网络的可解释性。做到知其然更知其所以然，相信当本文对神经网络有了深刻的理解之后，本文目前遇到的大量问题（过拟合，参数震荡，超参调节，对网络结构的选择）等问题给出详细的解释，使得对神经网络的研究摆脱经验化的局面。

六 参考文献

[1]高阳.强化学习研究综述[D].自动化学报.cnki.com.cn,2004-1

[2]刘全,翟建伟,章宗长,钟珊,周倩,章鹏,徐进.深度强化学习综述[J].计算机学报,2018,41(01):1-27.

[3]唐振韬,邵坤,赵冬斌,朱圆恒.深度强化学习进展:从AlphaGo到AlphaGo Zero[J].控制理论与应用,2017,34(12):1529-1546.

[4]马朋委. Q\_learning强化学习算法的改进及应用研究[D].安徽理工大学,2016.

[5]赵婷婷,孔乐,韩雅杰,任德华,陈亚瑞.模型化强化学习研究综述[J/OL].计算机科学与探索:1-11[2020-05-22].http://kns.cnki.net/kcms/detail/11.5602.tp.20200331.1819.002.html.

[6]王欣,王芳.基于强化学习的动态定价策略研究综述[J].计算机应用与软件,2019,36(12):1-6+18.

[7]张汝波.强化学习理论\_算法及应用[EB/OL].cnki.com.cn,2000-10.

附录

算法改进：

在原DQN算法中加入Q表和路径追踪表

self.q\_table = pd.DataFrame(columns=[0, 1], dtype=np.float64)

self.eligibility\_trace = self.q\_table.copy()

加入检查状态函数，如果某个状态不存在，就会在Q表中加入这一种状态。不用一次把所有状态加入，来节约空间。

def check\_state\_exist(self, state):

    if state not in self.q\_table.index:

        self.q\_table = self.q\_table.append(

            pd.Series(

                [0] \* 2,

                index = self.q\_table.columns,

                name = state,

            )

        )

        self.eligibility\_trace = self.eligibility\_trace.append(

            pd.Series(

                [0] \* 2,

                index = self.eligibility\_trace.columns,

                name = state,

            )

        )

        self.match\_table = self.match\_table.append(

            pd.Series(

                [0],

                index = self.match\_table.columns,

                name = state,

            )

        )

加入remember函数，当到达一种新状态，就把这种状态和其奖励记录在Q表里，并根据路径追踪衰减值修改Q表中的奖励。

def remember(self, s, a, r, s\_):

    self.check\_state\_exist(self.to\_str(s))

    self.eligibility\_trace.loc[self.to\_str(s), a] += 1

    self.q\_table += self.eligibility\_trace \* r

    self.eligibility\_trace \*= 0.988

    self.match\_table.loc[self.to\_str(s), 0] = s\_

当一轮游戏结束后，使用restore函数，把Q表中的数据储存进DQN的memory。然后再清零Q表。

def restore(self):

    for index, row in self.q\_table.iterrows():

        if row[0] == 0:

            action = 1

        else:

            action = 0

    self.store\_transition(np.array(self.to\_list(index)), action, row[action], np.array(self.match\_table.loc[index, 0]))

    self.q\_table = self.zero\_table.copy()

    self.eligibility\_trace = self.zero\_table.copy()

    self.match\_table = self.zero\_match\_table.copy()