My little document

August 8, 2016

Contents

Chapter 1

Progress History / 项目进度

1.1 Inspiration / 启发

1.1.1 DQN 算法

这是一个混合了深度学习 (deep learning) 中的卷积网络 (convolutional neural network) 和强化学习 (reinforcement learning) 中的 q learning 的算法.

Paper **http://arxiv.org/abs/1312.5602**

Code **https://github.com/spragunr/deep_q_rl**

特征

- 最大的特征是把 cnn 用在了强化学习中.
- 除此以外和一般 q learning 的差别是 dqn 会记录历史数据，并用历史数据进行训练. 而普通的 q learning 算法使用实时数据进行训练的.

1.1.2 小车避障视频

- **https://www.youtube.com/watch?v=zOgSC---rgM**

这个视频是驱动我做这个项目的主要因素. 在这个视频中，一辆小车在一个二维的空间中活动，场景中有障碍物存在. 当小车遇到障碍物的时候，会触发一个惩罚信号，然后游戏会被重置. 此外，小车可以获取每个时间点，前方不同角度存在的最近的障碍物到小车的距离的信息. 考虑到我们之后的研究，我们可以把这里的距离信息看做一个一维数组. 其背后的算法使得小车可以学到距离和惩罚信号之间的关系，在经过了足够的训练之后，小车就能学会避开障碍物了.

由此，我们通过这个视频知道了，强化学习算法，可以通过一维的前方距离信息以及撞击惩罚，来训练一个 agent/机器人/小车学会避开障碍物.

相关文献

之后我在文献中找到了一个相近的实现

引用 http://dl.acm.org/citation.cfm?id=1102426

Pdf http://machinelearning.wustl.edu/mlpapers/paper_files/icml2005_MichelsSN05.pdf

1.2 Reimplementation / 重现视频所实现的东西

由于我们没有视频中的程序的代码，我们要做的第一件事情，就是重现视频中的场景，训练一个 agent 在一个类似的二维空间中学会避障.

1.2.1 虚拟环境

由于我们的最终目标是训练 agent 学会在三维空间中避障. 因此，出于便利性的考虑，我们在这个阶段已经完成了三维虚拟环境 (Figure 1.1)，但是在算法中仍然将其视作一个二维空间，模拟成视频中的场景

具体的方法是，

1. 首先，根据场景内的三维模型生成深度信息 (https://en.wikipedia.org/wiki/Depth_map) (Figure 1.2).
2. 然后，截取地平线的部分 (Figure 1.3)，作为对视频中的一维距离信息的模拟，发送给 agent
3. 在场景中，当 agent 遇到障碍物的时候，场景会发送惩罚信号给 agent.

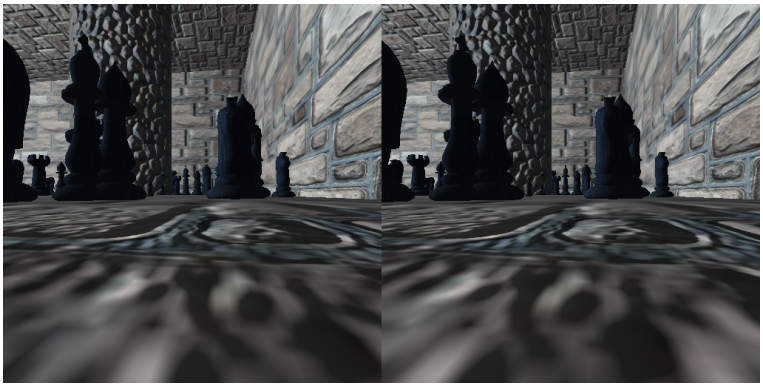

Figure 1.1: agent 视角的虚拟三维环境，通过三维引擎的立体视觉模式加入了左右两个视角

Figure 1.2: 通过三维引擎直接生成的深度信息

Figure 1.3: 截取地平线的部分生成一维的深度信息

1.2.2 连接虚拟环境和 DQN 算法

原本的 github 上的 DQN 程序是和 Atari 模拟器对接的. 在完成了虚拟环境之后，我们使用这个虚拟环境替换掉了 Atari 模拟器. 由于输入从 Atari 的二维画面转换成了一维的距离信息，我们也把 DQN 中的 cnn 替换成了普通的多层神经网络.

1.2.3 训练

随机行为

和一般的强化学习算法一样，最初阶段 agent 的行为是被设定为随机的，以此来积累周围环境的知识.

1.2.4 结果

在满是障碍物的地图上,agent 很明显的可以在较长的一段时间中避开障碍物. 但是并不能做到在所以情形下完全避开障碍物.

问题

我们缺乏有效的指标判断训练结果的有效性. 障碍物地图是随机生成的，有些情况下 agent 可能陷入无法避开障碍物的情形. 但是我们缺乏手段评估哪些障碍物环境是属于这一类别的.

在参数调整的后期，很难根据 agent 不碰撞的时间长度来判断算法的优劣.

1.3 三维环境

在能够实现视频中的东西之后，我们开始尝试将其从二维环境转换到三维环境.

1.3.1 深度信息 / Depth Map

由于在视频中,agent 接收的信息是一维的障碍物距离. 对应到三维环境中就是二维的障碍物距离，也就是 Depth Map. 由于我们已知通过一维的障碍物距离进行训练是一个有效的办法，类推用二维距离信息也可能是有效的.

TODO Finetune

有一些计算 Depth Map 的算法是基于神经网络的. 所以我们原本打算，如果使用这样的算法，可以在后期，将 depthmap 的神经网络加入到 DQN 中一起 finetune 优化，进一步提升性能.

通过双视角图像计算 Depth Map

存在着通过双视角和单视角计算 Depth Map 的方法 (文献)，其中双视角比单视角高效. 我们使用了 opencv 内置的算法，但是很可惜输出结果不够理想，没能很好的给出障碍物的距离.

直接通过三维引擎获取 Depth Map

在虚拟环境中，通过原本的三维模型，可以计算出最完美的 depth map. 这在现实中是做不到的的. 我们希望确认，至少通过最完美的 Depth Map，可以训练出较好的结果.

结果

训练结果基本接近于二维的情形. Figure 1.4

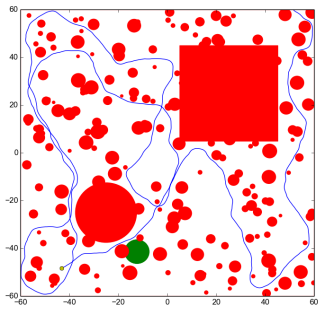

Figure 1.4: 这是 agent 一次随机行走记录，红色表示障碍物，黄色表示出发地，蓝色表示行走路线

1.3.2 允许 agent 在三维空间活动

因为对比文献中已有的结果来说，在二维的平面上活动的 agent 仅仅只需要地平线部分距离信息就足以学会避障了，而扩大输入到整个二维的 depth map 只是增加了不必要的输入. 因为不必要的新特征的存在，带来了 overfitting 的可能性.

相当于是为了证明使用二维的 depth map 的价值，我们尝试允许让 agent 在三维空间中活动，这样仅仅使用一维的地平线数据作为输入就不够帮助 agent 做出正确的避障行为判断了.

最后的训练结果是失败的，但是追查原因发现是因为我们使用的物理引擎在处理三维空间的时候的精度不够，在没有碰撞的时候给出碰撞信号，或是相反，因此 agent 提供了大量的错误训练数据.

物理引擎的精确性问题最后没能修正，所以我们只能放弃再三维空间中训练 agent 了. Figure 1.5

1.4 目标导航

1.4.1 深度信息 / Depth Map

结果

Figure 1.6

1.4.2 真实图像 / Raw Image

我记得把深度图替换成真是图像这个步骤我并没有在训练随机行为避障的时候实现，而是在实现了导航之后再做的. 因为最初没有预期这是能够做到的. 在确认了通过深度信息也能学会导航之后，我们开始测试，确认了，直接通过真实图像就能够做到让 agent 学会导航.

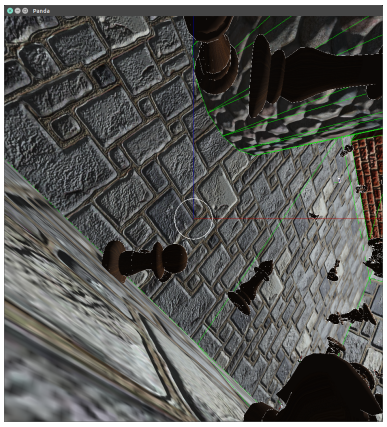

Figure 1.5: 三维空间中活动的 agent 视角的截图，物体周围的轮廓线表示了物理引擎实际进行碰撞检测计算的几何体.

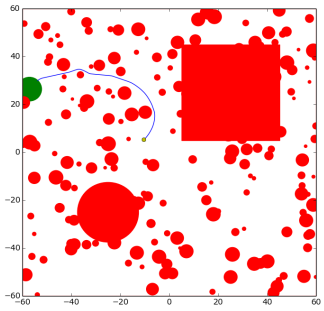

Figure 1.6: 使用 depth map 进行导航，绿色表示目标区域

结果

Figure 1.7

问题

我们无法确保 agent 学会的能力是泛用的，即不仅仅在训练其的虚拟环境中适用，也能在未知环境中适用. 推测在使用 Depth Map 的情况下，神经网络只需要收敛到一个能够判断哪个转向反向的障碍物更遥远的函数，这样的函数的适用性应该是比较广的. 但是在使用真实图像的时候，可能 k 就

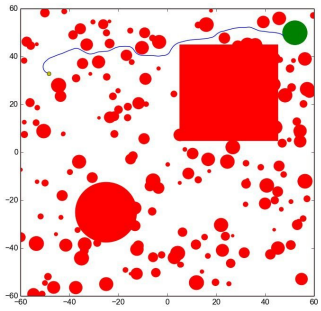

Figure 1.7: 使用真实图像进行导航

Chapter 2

Introduction

介绍下我们要处理的问题是什么，我们研究的是，在一个模拟的环境中，让 agent 学会在避开障碍物的情况下能够自行找到一个指定的目标。我们会使用 RL 算法来训练 agent. 找不到 paper，就用皮箱吧.

local navigation 对比 global navigation ，在绝大多数的 local navigation 中，目标的位置是给予的，关于导航成功的部分，给一个目标与起始距离和耗费时间的比例，

大概梳理一下，先描述问题，看看谁做过，目前看来主要是用非 raw image 的避障方案，而且我找不到根据 target 来做的方案. r1 之外自然是有的，大疆就是这么做的.r1 领域来说似乎导航不是一个问题?

我们假设 agent 在一个存在障碍物的环境中自由活动. 目标是让 agent 避开障碍物到达目标地点.

Chapter 3

Background

3.1 root

介绍下那些人处理过了最近的时候，dqn 发展很快
　关于避障问题，很早就有人用 RL 研究过，列举下他们的方法．绝大多数是对输入图像 encode 之后的结果．列举一下．导航发信器，

3.2 Deep Learning

3.3 Convolutional Neural Networks

3.4 Data Augmentation

3.5 Backpropagation

Chapter 4

Methods

4.1 root

我们做的分两部分，一部分是导航，一部分是避障，导航部分只有完成了设计。而且没有依据。避障部分完成了避障的部分发展自 dqn 的设计，首先我们设计了一个虚拟的三维环境，其中有很多障碍物。我们让 agent 在环境中走动，如果它撞到障碍物就会给予惩罚，然后结束这一轮模拟，重置环境。一个 RL 系统的定义。一个 RL 系统是一个研究如何在一个给定奖励规则的环境中，让一个 agent 找出获取最高奖励的办法，它受到了行为心理学的影响（ref）为了能够让一个 RL 系统可以训练出可以躲避障碍物的 agent，我们需要设定对应的奖励规则，我们要让 agent 学习到碰撞会带来惩罚，也就是说碰撞带来的奖励比不碰撞低，比如我们会设为负数

我们做的分两部分，一部分是导航，一部分是避障，导航部分只有完成了设计。而且没有依据。避障部分完成了避障的部分发展自 dqn 的设计，首先我们设计了一个虚拟的三维环境，其中有很多障碍物。我们让 agent 在环境中走动，如果它撞到障碍物就会给予惩罚，然后结束这一轮模拟，重置环境。一个 RL 系统的定义。一个 RL 系统是一个研究如何在一个给定奖励规则的环境中，让一个 agent 找出获取最高奖励的办法，它受到了行为心理学的影响（ref）为了能够让一个 RL 系统可以训练出可以躲避障碍物的 agent，我们需要设定对应的奖励规则，我们要让 agent 学习到碰撞会带来惩罚，也就是说碰撞带来的奖励比不碰撞低，比如我们会设为负数。

4.2 Virtual Environment / 虚拟环境

4.2.1 3D Engine

Chinese

由于我们在 ML 的部分主要是使用了同样是 python 环境中的 theano，考虑到虚拟环境和 theano 之间存在大量的数据交互，为了减少数据的流动量，我们认为最好的办法是，在虚拟环境中生成数据之后直接把数据的指针交个 theano，

而不是把这些数据再复制一遍。考虑到为了方便直接传递内存指针，我们选择了一个同样是 python 环境下的 3d 环境，Panda3D 游戏引擎。

English

We will build a virtual environment. And a large amount of data of images will be transferred between the virtual environment and the agent. Therefore, it would be much efficient if the two programs can share memory and transfer pointers of those images instead of themselves. Our Reinforcement Learning algorithms are built on Theano, a development environment in Python language. So we choose Panda3D, a 3D game development environment also in Python language. Then we can merge the two programs into one process.

4.2.2 Environment / 环境

我们使用的 model 来自于 padna3d 的 demo 障碍物主要包括象棋和墙壁，如图所示，As show in the Figure, models of obstacles include chess pieces and walls.

4.2.3 Other Plan / 另一个计划

最初的计划中，我们是希望设计两个不同的场景的，第一个是给予 agent 在地面活动的自由，就像遥控车。第二个是希望给予 agent 在三维空间活动的自由。但是由于物理引擎的原因，没有做到。最终，我们只设计了在 agent 在二维空间中活动的情景

4.2.4 Performance Problems / 性能问题

这里是一些由于技术不足导致的性能问题。There are some performance problems caused by technique inabilities, which make us unable to fully utilize the hardware's computation power.

Refresh Rate

受限于我们使用的三维引擎本身的设计，由于游戏用的引擎的设计是为了输出到 60hz 的屏幕刷新率的，它的渲染速度似乎是以 60hz 为上限的，因此我们使用三维引擎生成数据的时间成本比较大，相对于训练来说，First, a 3D game engine like Panda3D is designed to output video flow to a computer monitor. The refresh rate of a monitor is normally set to 60hz. The render speed of Panda3D is limited under this rate. Due to this design, the phase of generating training data consumes more time than the phase of training. 所以，我们会有重复使用历史数据的必要，这会带来一个问题，就是神经网络可能会 overfit 这些被重复使用的历史数据，而无法准确预测新的数据。

Data Transmission between Video RAM and Main RAM

另一个问题相对来说造成的影响比较小一些，这是上面提到的数据传递问题的衍生。即使按照上面的设计，我们可以直接将 Panda3D 生成的图像指针交给 theano，但是首先 Panda3D 需要从显存中把数据取到内存，在这之后，因为 theano 中的计算是靠 gpu 加速的，所以会再次将这些数据复制到显存中。如果说存在直接将显存中的数据指针交给 theano 的办法的化，这些复制操作就可以省去了。由于技术上的难度，我们没有深入去考虑这中方法。%Another problem is that, the training data is first generated by Panda3D in Video RAM,

Time Record / **时间记录**

我们记录了大致上在不同的环节会被消耗的时间：

4.3 Collision Detection

% 虚拟环境需要在 agent 碰撞的时候给予惩罚，所以我们需要一个碰撞检测程序 To release signals of penalty when collision happens, the virtual environment is required to determine collision between the agent and the obstacles. Collision detection is normally a part of a physics engine and there are several physics engines integrated within Panda3D.

4.3.1 Panda3D Built-In

% 我们使用了 Panda3D 中内置的物理引擎帮助做碰撞检测% 我们尝试了两种物理引擎，Panda3D 内置的引擎，可以在检测简单几何物体的碰撞，如果只是考虑设计一个 agent 可以在二维平面自由活动的场景的话，我们可以做到用简单几何体，圆柱和平面，来准确的描述 agent 可能 zhuangshang 撞上的障碍物，因为只需要考虑从顶视角看到这些物体的形状。% 但是在三维的情形中，用简单几何来描述准确描述模型的形状就很难做到了。

There are two ways to go about collision detection. One is to manually create simple collision geometries, like spheres and polygons, for the obstacles. Panda3D offers built-in collision detection that calculates the impacts between these geometries. It is fast, but unable to precisely depict collisions of complex models. When the agent is only allowed to move around on the 2D plane of our virtual room, this method works fine. Because all the models can be precisely depicted by circles and lines in a bird's-eye view.

4.3.2 Bullet

% 因此我们尝试了第二个物理引擎 bullet,bullet 可以通过三维模型数据直接生成对应的物理实体描述，% 避开了我们手动用简单结合去描述一个物体的问题。% 但是我们后来发现，在这种情况下 bullet 触发的碰撞信号不够准确，在

agent 实际上并没有碰撞的时候，% 就会发出碰撞信号。% 这中不准确性在实际训练 agent 的时候，带来了很大的干扰。% 因此，我们选择在只给予 agent 在二维平面活动自由的情况下，% 对比这两个不同的引擎，在我们固定所有其他参数，比如学习速度，% 在前者环境中的 agent 可以训练出有价值的结果，但是后者却不行。

% 为此，我们只能放弃给予 agent 在三维空间的活动自由。% 还有一种办法是抛弃模型复杂度，使用简单几何体构成的障碍物。但是我们担心这会让模型丢失太多泛性?

Another way is to create collision geometries for any models used for graphic rendering. Panda3D offers interface for the physics engine Bullet, and Bullet can generate these collision geometries. But it was found that Bullet cannot precisely detect collisions with these auto-generated geometries. Bullet might send signals of collisions when collisions actually didn't happen. We have applied the same machine learning algorithms on both engines in a scenario, which allows the agent to move on a 2D plane only. Comparing to the built-in engine of Panda3D, Bullet's collision detection lowered down the quality of the training samples seriously, and eventually prevented our attempts to train machine learning models with Bullet. And with the built-in engine, we have to restrict the agent's movement on the 2D plane to keep the complexity of collision detection at a low level, which can be handled as simple geometries.

4.4 Avoiding Obstacles

我们首先要考虑的问题是如何学习避障。我们使用一个和 dqn 类似的体系。和 dqn 面对的游戏环境不同的是，在避障问题中，唯一的事件信号是碰撞。除此以外,，虚拟环境不会给予 agent 任何信息告诉 agent 是否它所处的状态是我们希望的? 另一方面，当碰撞发生的时候，agent 前方的空间就被阻碍了。如果我们允许 agent 的行为只有向前前进，那么这个时候 agent 不管如何选择行为，都无法再移动了。这种情况下，我们选择停止模拟，所以当碰撞信号出现的时候，也就是马克夫链终结的时候。这说明在碰撞产生之前，所有的行为都影响到了这次碰撞，这是一条很长的马克夫链，是很难训练的。(重新表述一下)

然后，我们没有选择让 agent 能够选择倒退，因为 agent 在允许倒退的情况下，可能学到在原地循环前进后退，从而停留在一个区域不动的策略。这种策略下，agent 即使不识别到障碍物的存在，也可与做到躲避障碍物，但是这不是我们希望的结果。对于没有目的地的 agent 来说，停留原地不撞到障碍物是完美的策略。但是如果说给予 agent 一个目的地的话，对 agent 来说就会存在比停留原地能得到更多 reward 的策略了。在这种情景下，我们就可以加入后退作为 agent 的一个动作选项。如此的话，我们或许可以验证 agent 是否能够学会在撞到障碍物的时候，通过后退让自己从碰撞状态中恢复过来。这种情况下，我们可以验证在物理引擎模拟的碰撞物理效果下，agent 会如何应对。可惜我们加入导航功能是项目后期的考虑，所以没有做这个尝试。

4.5 Reward Rules

对于奖励的设置，由于碰撞是唯一的虚拟环境可以识别的信号，所以 reward 的设定主要在于存在碰撞的情况下，和不存在碰撞的情况下我们设定普通情况为 0，而有碰撞的情况下为 1

4.6 Depth Map

我们在训练中给予神经网络的输入是深度图像，而不是直接的图像输出。选择深度图像的原因，一是因为深度图提供了关于 zhijie 直接的关于各个方向障碍物的距离的信息，这应该降低 agent 的学习的收敛难度。我们在最初的时候，尝试了用 Panda3D 生成双眼视觉图像，然后使用了 opencv 中的算法将其转换为深度图像。可能是因为参数调节不合适，所以效果不是很好。之后，我们选择直接由 Panda3D 通过模型生成深度图像，这当然是最完美的深度图像了。但是如果，说我们要把虚拟系统换成现实，那么就不得不加入一个能够将现实图像转换为深度图像的算法。这些深度图像的精确度不会是完美的。不过，最近存在一些基于卷积网络的关于双眼视觉的研究， 如果我们使用一个 pretrained 的网络来为作为我们的神经网络的前置输入，那么在训练的后期，我们可以使用 RL 算法一并训练前置的生成深度图的网络，通过 finetune 来进一步提高网络的性能。

4.7 Histroy Data

我们使用和 dqn 中一样的历史记录，和 dqn 面对的问题不同的是，在 dqn 场景中的游戏中，agent 一般在游戏开始的时候，它的起始状态并不是完全随机的。在后续的行为选择中，如果 agent 偏向去选择一些会导致高 reward 的行为，而关于低 reward 的场景的行为的知识对于 agent 来说就不是必要的了。因此，在 dqn 的场景中，他们选择了在训练的起始阶段给予 agent 很高的概率去作出随机行为，扩展大体的知识，但是训练的后期就会降低这个概率，从而更多的使用那些可以让 agent 获得高 reward 的行为。

但是在我们的场景中，我们每次给予 agent 的起始状态是完全随机的。为此，让 agent 尽可能的学习到所有可能状态的知识

重述：也就是说，在 dqn 中，agent 一般会从非完全随机的初始状态出发，然后依靠 RL 算法，选择那些被认为会的到高 reward 的 action 和 state，这样的情况下，agent 在绝大多数情况下都是处在评估为高 reward 的 state 下，这样的 agent 如果总是遵从学习到的高 reward 的策略的话，那么环境中有些较为明显的导致低 reward 的情形，agent 可能会很少陷入进去，为此关于处于这样的情形下的 action value 就没有什么价值了。为此，省去这些 state 下的 action value 训练也不会影响到 agent 获取高 agent，但是在我们的情形中，由于我们设定让 agent 可以完全随机的从任何没有被撞击的 state 状态下起始，所有 agent 有必要在所有这些状态下给出准确的 action value 评估。为此，即使到了训练的后期，我们认为完全随机的探索行为得到的经验是必要的。而且，随机概率低的时候，少量的随机行为带来的效果会很快被 agent 选择的追求最大利益的行为抵消掉，宏观上来说，随机概率低的时候，agent 闯

入一些可能会引起碰撞的环境，比如角落里的概率会小的多。为此，我们选择让随机概率从 0 到 1 之间周期性的改变，周期长度可以设定为历史记录的总长度，这样的话，历史记录中就会记录下等比例的随机行为和追求最大利益的行为了。

4.8 Overfitting and Data Augmentation

由于之前提到过的虚拟环境的性能问题的原因，生成新的图片的速度比使用这些图片进行训练的速度慢很多，这样如果我们分配大约相等的时间用于生成信的训练数据，以及用他们来训练模型的话，同样的图片会被使用多次，模型就有可能 overfit 这些历史数据，而不能够以同样的准确率去预测新的数据。

受启发于在同样使用深度卷积网络的图像分类中常用的 data aug 方法，我们尝试也引入这个方法，在历史数据的基础上，对其进行随机改动，生成的新的数据，改动的标准是，改动之后的数据本身也是在一定的情景之下有可能由虚拟环境自己生成的。包括平移和 flip 和 resize，其中 flip 要注意连 action 数据一起改动，crop 和 resize 需要注意同时改变一整个 phi 时间序列，按照同样的方式。flip 存在的问题是，如果没有使用 lstm 和或者较长的 phi，的话，可能会在无法从输入数据中找到碰撞的迹象的时候，就告诉 agent 碰撞产生了。这样的学习数据对于 agent 可能存在混淆的作用。resize 的问题是，被 resize 的一些列 phi，相当于把 agent 放入了一个所有的对象的缩小了的环境中，也包括距离在内，因此对于 agent 来说，这的驾驭，同样的 action 下，被 resize 的图像中 agent 前进的速度是不同的。这样的概率波动可能会让 agent 的学习收敛变困难。

4.9 Ensemble

ensemble 组合输出结果在需要提升 agent 性能的验证阶段，我们通过 data aug 重复多次（6 4）从网络中获取结果，然后去平均值

4.10 Short Term Memory?

我们没有使用 lstm 这样的记忆单元，phi 的长度也只有 2，由于我们没有允许 agent 后退，所以视野外的障碍物对 agent 的影响不是很大，因此没有短时记忆，造成的影响也不大

4.11 Network Architecture

在我们的问题中的一个问题是马克夫链太长了。我这里就先不多说了，具体书上查一下类似的描述撞击是所占比率相对较小的事件，在我们的实验中，撞击事件大概会占去所有学习数据的 0.5%，在其余情况中，决定一个 state 的 action value 的主要是其下一个 state，而 reward 都是 0 引起的问题是，在网络初始化的时候，如果对于某个 state 给予的 action value 的绝对值太大，

就会在使用其之前的一个 state 训练网络的时候，得到一个太过异常的参考值，同时留下一个过大的 loss value，在 bp 这种情况一般会导致网络进一步生成更大的 action value，最终导致 action value 和 loss value 趋向无穷大。

但是，由于我们给予撞击的 reward 是－１，而在其他情况下都是０，而撞击是一轮模拟终止的时候，因此，我们可以推测，对于任何 state，理想的 action value 是落在-1 到０之间的，所以我们可以在训练中，直接限定，如果某个 state 的评估超出了-1,0 的范围，我们在训练网络拟合其前一个状态的时候，就会把其当做-1，或者０

这样，我们就能是的网络度过训练最初的不稳定状态。

========== 我们用 dqn ××××××××××××××××

详细介绍下各个部分的缘由?

还有网络结构

4.12 Navigation

我们假设有一个目标对象，会不断发出信号，agent 可以由此判断目标地点的相对自己的方位。我们的目标是，让 agent 通过可以通过目标的方位信息来接近目标。这本身是一个很简单的问题，因为通过相对方位信息，我们可以知道目标是否在 agent 正前方。然后通过旋转 agent 的朝向，可以让 agent 的朝向指向目标。在这之后，只要 agent 直接向前，就会接近目标了。当时如果场景中存在障碍物在 agent 于目标之间，这个问题就不能用这种方法解决了。

由于我们现在已经可以让 agent 通过 RL 训练来避开障碍物了，我们打算从这个为什么选择极坐标

我们假设有一个目标函数

我们假设我们的神经网络从一片空白开始训练.

我们假设训练中，每一个样本可以起到改变其周围地带的补丁效果.

那么，极坐标需要的补丁数目可能会小于普通坐标

或者说，因为这个函数对方向很敏感?

最初的时候，我们设计允许 agent 自己选择停留，如果停留是将达到一定的次数，那么就假设 agent 会永久停留在某个位置，结束模拟。以此，我们可以让 agent 自行判断是否它已经没有进一步接近目标的能力了。这个方法似乎不好?与这个行为选择方案对应的是，我们使用的奖励函数是依据接近目标的程度判断的。因此 agent 在每一次行动之后都能得到一定的奖赏。

但是很可惜，agent 选择自己停止的情况并不多。所以这个机制起的效果或许不大，因此，我们开始选择，如果 agent 接近目标到一定程度的话，就给予其一定的奖赏，并停止模拟。

前期的训练结果很差。修炼结果在一定概率下会找到目的地，但在另一些情况下，会停留在远处。然后我们尝试了修改了 agent 的行为选项，我们发现，如果我们允许 agent 停留在原地转向，而不是一定要前进一定的距离的话和停留选项一样，原地转向本来在随机避障中是不存在的，因为如此的话 agent 总能存活下去了，我们逼迫 agent 不断前进，然后选择它认为撞击可能较小的方向，

然后在导航问题中，我们发现，如果允许的话，agent 几乎总是能找到目标，唯一的问题是它在途中花费很多时间转向。虽然我们不知道原因，但是我们这

和上面的问题是有关联的，因为在转向的同时必须要前进的情况下，agent 在找到正确方向的时候花费的大量转向行为，会把它带到一个和原位差距非常大的方向

以下是设计思考的内容，不一定适合记录下来然后为了避免转向花费过高的问题，这个问题要从算法上解决吗？给予停留惩罚？还是说给予一个较小的转向速度？停留惩罚没有太大意义，因为停留本身和选择正确方向前进之间已经有一个 reward 差值了，(除非你没有给予差值，而是选择以终点目标作为解答)

另一个问题是，如果我们不设定到达终点停止的话，平常ｒｅｗａ rd 是 0.05 到达终点会成为０．０２但是仍然高于完全停止的０，没有出现负面的惩罚预测，所以 agent 不会停止不过之后，agent 从 0,0 走到了-1,-24，然后停止了，不过更多情况下是，来回的走动。总体来说，除了长期停留意外，其他的问题不大，但是引入障碍物后，会成为问题把？

训练中的 terminate 对于我们的程序很重要，因为否则程序不会重置，这导致了学习到的结果，几乎完全是一次 reset 之上的，在那内部的随机行为。啊，不过，你看到了，上面这种停止也是有的，就是太快了，很少见。所以，问题是有些区域其实 agent 可能很少进入？比起重置的多样性来说，另一个问题是没有中止的话，就没有了给予模型定值的方式。总的来说这两者引入障碍物之后，其实都不会成问题了吧？唯一是问题的是长期停留。由于长期停留的结果是 0，而现在我们的 reward 均值是 0.05，如果加入障碍惩罚之后，变为负数的话，那么停留就会成为优势选项，所以，奖励均值必须要超过惩罚至于转向停留，怎么办呢？其实给予其一个长期停留的最终结果是无效的，因为因为网络无法分辨两次停留是否是长期，你没有 lstm，单单从输入数据来说，同样的输入数据，你却希望得到不同的 reward，(一次停留之后，前进，得到奖赏，另一次确是作为长期停留，被归零，这是个矛盾，当然会出现 loss 了。

在训练中来说，长期停留的确是个问题，但是你通过加大 reward，可以在很多情况下避免，不过有些情况，则的确是陷入问题了。为此，我觉得办法是长期停留做 reset，而不是 terminate 可是问题是由于１０％其实真正的长期停留几乎是不会出现的，监控一下训练中 terminate 的长度吧。因为障碍的存在，长度不一定会很长，如果太长的话，我们就加入概率 reset

至于停留选项？如果网络能给出负数 reward 判断，那么就是停留，因为我们原本打算就是给予网络这样的就是希望它能选择停留时机。但是这个选项我在想或许不需要如此，因为停留是为了训练网络在任何输入中，都给予一个动作固定的 reward 0，那么我们直接这么设定好了，如果 reward 预测小于 0 就停留。或者其他常数，这个数可以训练吗？应该不行吧，这其实是你是否希望网络冒更大风险去追求利益？这里有个问题是，训练中，是否要按特定的数值选择在其之后停止？包括这部分知识，我觉得学习比较好，因为这样之后你才能去寻找那个风险常数的大体为止。多样性问题我们靠重置解决，避免停留，还有即使依赖随机 reward 常数问题 terminate 问题，我们靠撞击解决，给予足够的 terminate

因此，我们现在消除停留选项，并且加入重置机制，最初最好关闭重置，然后根据平时的一般游戏长度来决定重置概率或者长度，但是，没有了停留，在没有撞击的情况下，就缺乏 reward 常数了。或许 discount 能解决这个问题？还是说我们直接引入障碍？

現在遇到一個問題，就是我們允許 agent 作出原地轉向的情況下，沒有辦法單獨訓練避障吧？你需要輸出一下，看看當前的存活平均時間，這個問題之後可

能需要觀察下，如果存活时间太长就是问题了现在的问题是，明明 rewards 为负数，为什么训练结果是正数? 我们做些极端的方法测试下? 比如 clip? clip 可以强制出负数，不过，如果要加入导航功能的话，就不太好办了，我们要搞清楚 yuanyin 原因嗯,clip 之后很快起效果了，我想我们或许可以先训练避障，在网络稳定之后消去 clip

---

现在要先找出问题原因，我们最好是把 action 改成前进的，这种情况下 penalty rate 会比较高．生存率明显下降了，毕竟原地旋转指令，不会带来碰撞啊．但是不停留的话，导航那边又会有问题

还有一点是，好像避障对 lr 的要求很高，而导航则需要较高的 lr，

结论: 好像 clip+ 高 penalty 的情况下，至少我们可以让 q mean 回归负数.clip 到还好，我觉得可以在后期网络稳定后移除，问题是 penalty 呢? 我的想法是，我们至少训练 layer3 之前的部分．固定下来，然后再改变环境到 navigation 问题上，然后训练 layer3 以后的部分

似乎避障中的 loss 在 0.08 左右，比导航中的 0.008 高，因此如果想说要一起训练的话，明显导航问题会受影响，看来导航问题的值要提升 10 倍才能不被影响

—- 我们打算通过 success 5 来训练一份参数出来用

由于改变了很多东西，我们现在好像无法重现 suc5，除了直接使用它的代码．如果有心的话，逐步的检查 suc5 有的，而现在的代码没有的成分，应该可以找出原因，然后对现在的代码修改．可是这要花很多时间，所以我决定，直接用 suc5 生成前三层 layer 的数据拿过来用

虽然 loss 精度不一样，但是 suc5 如果消除 (-1,0) 的限制，也会偏向正数．这因此变成了一个一开始就存在的问题．

主意: 我们做定时重置的话，一定程度上起了随机化的效果，因此可以消去随机参数了，但是对于碰撞来说依旧是问题，碰撞率依然会减少吧? 会吗? 会陷入无法避免碰撞的情形吗? 很困扰呢，为了优化，我们是要减少碰撞的，但是为了训练，碰撞不能太少

关于 0 以上这个问题记录下吧

发现，对于 suc5 来说 (-1,0) 是必要的，虽然，即使没有，我们可以用 sgd 0.02 和 1000updates 让其收敛到 0.046，这个时候 lr 是 0.002，但是表现结果却不好 (偏向左转)，偏向问题很有可能，是因为用了 1000updates 导致马克夫链的传播速度太慢。但是如果不用它的话，我们似乎无法确保收敛不会发散还有，用 1000updates 的时候的收敛速度比不用的时候慢很多除非用了 (-1,0) 限制我们已知 rmsprop 0.0004 是有效的，再高的 lr 会扩散。但是 sgd 和 momentum 在 0.02 不会轻易扩散，然后 momentum 的速度似乎比 sgd 快一些? sgd check 结果为 0.14,0.13，0.12 其实 1000updates 起到了 <0 条件类似的作用，updates 的减慢导致不会很快出现超过 0 的数值，但是僵化很严重，但是奇怪的是，服务器上 q mean 是-0.14，可是在我本地 q 是 0.10 该死啊，本地的验证程序有问题啊。validate 的时候忘了加载 nip 了，因此以上的解答全不做数啊。不过加载了之后，还是不行，没有转向问题了，但是实际上辨识能力并不好，基本是走直线，诡异的是 q val 是-1.5，但是训练 mean—0.15 std=0.17 好吧，结论是效果不错，我们是搞错验证手段了。唯一

的问题是容易进死角。另外，为了在每次 save 的时候做足够的 train，我们加大了每次 view 的训练频率，加大了 88 倍。如果不用 sgd 而使用更高效的收敛算法或许可以放慢频率? 在 88 频率的情况下，momentum 第一轮 check 可以达到 0.069，至于 rms 好像不能在这个训练 rate 下收敛好奇怪，我修改了频率，但是 log 中看不出来，第一次 check 花的时间是差不多的，嗯，的确是差不多的，因为 check 不是依据 view 还是依据 train 定下来的，这说明 view 的成本不是那么高啊，因此 momentum 的速度似乎和 rms 差不多。—- 那么这个结论有什么用? 墙角问题似乎是因为随机起步点太靠中央了。因此使用 1000updates 或许并不真的带来很大问题，使用它和 momentum 配合的话，或许我们可以消除 (-1,0) 限制，但是 1000updates 实际上有可能是依靠把数值维持在 0 以内做到了这一点，因此，如果我们加入 reward 的话，1000updates 或许也无法做到维持 0 以内了，那该怎么办呢?

因此，当前的实验是，测试 momentum 0.02，无 1000updates，(-1,0)，60 随机起始下的训练结果，loss 估计可以达到 0.046，如果通过，基本上通过了，很有效，loss0.054 的表现就很好了，训练中 penalty 是 0.01 左右下一步是 momentum0.02,，(-1,1)，60 随机起始下的训练结果。已知 sgd 下是可以通过的，就是训练速度好慢。不选择 rms，一是因为 r1 限定太小，二是因为每次 updates 的时候，都有崩溃的可能 loss 第四轮 check0.074，并且 q mean 0.18，但是 vaidate 结果很好。所以虽然 q mean 不准确，但却是有效果的。因此 1000updates 或许不用了下一步，我们测试 nav 系统，消除 reward，使用 nav 的取样策略，pena1 会比较低，(-1,0)，momentum 0.02 哦对了，我们忘了对深度图预处理了。到底是 x 适合学习还是 1/(1-x) 适合学习呢? 没有后者的情况下，第二轮 check 是 0.091 对比有后者的 suc5 (-1,1)，第二轮是 0.074。第三轮是 0.089 对 0.068 从 mean 和 std 上，很难看出到底 suc5 更依赖哪一个，因为 mean，std 都和初始状态一样，只好加上去试试看了。因为我们现在不知道 nav，loss 将不下来的原因，明明 penalty rate 两者都差不多. 不过加了也没什么效果。那么到底是什么原因呢? 难道我们只能用 suc5 作为基础来添加 reward 吗?

---

nav 0.076 的结果，validate 很差。因此我们可以认为 nav 的代码有根本性的错误了。比起对比 nav 和 suc5，我觉得我们只好以 suc5 为基础，重新添加 nav 了，我们假设问题在于 1auncher，所以仅仅用 suc5 的 1auncer 替换 nav 的对应代码改动中才发现，原来 suc5 的 phi 是 6，不过好像 nav 没法简单改到 6，只好用 suc5 该了兼并似乎还行，唯一的问题是，position 那边用了假数据后变成 NAN 了，所以屏蔽了它们如果问题不大，我们下面尝试加入真 position 数据，如果再没有问题，我们加入 reward? 关于经验策略怎么弄呢? 兼并后竟然遇到了性能问题。不过可能单纯是因为 nav 的消耗增大了检查发现，消耗出现在 train 原先是 4 秒，现在是 7 秒，但是实际上 train 是 3 秒，其余时间是收尾操作，或许原本两者的消耗都可以减半，但是现在的话，只有降低收尾记录操作的概率了。不过也可能是现在的收尾操作和之前不一样确认整个程序的消耗都在 train，就是不知道 nav 新添加的参数的转换，对于 gpu 来说负荷有多大

我去，之前竟然是锁死了 nav 的前三层在训练

我们总算调好 nav 了，那么下一步，先用 (-1,0) 训练前三层，因为这个训练

方法最有效。然后，我们固定前三层，看看它在（-1,1）的表现如何，希望尽可能接近前者。然后现在的问题是，我们的 discount 设定不同，怎么办，是否哪一方可以妥协？我觉得避障那里妥协余地会大些。我希望知道 nav 在不同的 discount 下的表现。但是 nav 不适合独立训练，（因为取样问题），因此我们大概要训练两份避障，95 和 97

下一步，关于 agent 单步跨距怎么设定？转弯的前进度设多少？1/10？因为原本我们就知道 agent 会花费很多时间去转弯。还有停留怎么设定？转圈怎么办？当前来说，避障中不存在这个问题，10 % 的随机率，似乎可以让任何 agent 撞墙。最多 1000 步。可是如果我们降低转圈率的步进的话，很明显的停留会成为优势，而步进会很快带来死亡，但是不允许转圈的话，agent 要找到目标很难啊

这真的很成问题，我觉得，我们必须要能独立于避障解决 nav 问题，然后再连接它们，如果你不确定，单纯只有 nav 的情况下，能否解决他，那么连接之后自然更难。也就是说，比起调整 reward 数值之前，你要确保，即使真的调高了 reward 也不会无法训练出结果。比较矛盾的地方在于转弯步进，nav 需要一个较小的转弯，以确保它能在原地停留决定方向，但是如果允许原地停留，避障就会选择它避免撞击。不过至少我们还有 10 % 随机行为，

出人意料的一次训练成功。虽然 loss 比较大，在 0.10 以上，q 在 0.30 左右，就是说，转弯步进对于 nav 训练的影响其实没有那么大这是一次基于固定的前三层的训练。loss 从 0.08 开始上涨。但是结果竟然不错。由于前三层固定了，所以后续我们可以放松那部分其他还有什么优化的想法吗？我们没法用数据说明训练结果，因为有些初始环境太严苛。不过你可以和没有训练的作对比，没有训练的特征是，基本找不到目标，并且虽然训练过和没训练过都很容易在初始情况下撞墙，但是没有训练过的即使经过了初始环境也会撞墙，这是个分布差别。本来我们是要调整转弯步进，还有 reward 来调整训练结果的，似乎没有必要了？但是你可以对比下，不做 pretrain 固定 13 的情况下结果有多少差别。不过要做对比的话，首先我们要写出可以作对比的程序来。顺便：这一轮训练长度是学习速度从 0.02 到 0.0002

先看看没有 pretrain 的结果吧，怎么说呢，我觉得 finetune 是很难说的一件事情，如果没有 pretrain 就可以训练好的话，那么自然是不需要 finetune 的结果好像是如此，没有 pretrain 似乎 loss 收敛到 0.10 以下了？不知道长期结果如何。q mean 波动很大，路径是 0，-0.3,0.3,0 第一轮 check，loss 0.95 第二轮 0.073，不过这和 fix 的情况是差不多的，那边前几轮的 loss 也很小，真是很奇怪。大概是因为 r1 固有的问题吧。要学习 0.97 下的 reward，花费的时间会比较长？我们之后试试看 0.95 和 0.90？随着 loss 一起上升的是 q mean，显示出我们的想法大体是对的。

画一下平面图，看看走过的路径？总的来说现在一个明显的问题是，当 agent 从目标旁走过的时候，它会错过很多，走到远处而不会立即回头怎么解决呢？加入目标地奖励怎么样？本来担心的是使用目标地点奖励，这个事件出现的概率太小，但是在步进奖励的辅助下，这个事件的概率会增大？到什么程度？

我们先把目的地奖励加进去，看看频率有多高。如果这个办法不行，那么调整

discount, discount 的调整对短期奖励，比如步进奖励有利，对于碰撞惩罚这样的长期奖励的分辨不利，所以结果上来说，有可能增大碰撞的可能性。不过，也难说，因为要从转弯中获得步进奖励，嗯，勉强是有一定的区分可能性。同时还有步进奖励本身的大小，也是直接和碰撞问题竞争的，但问题是，碰撞本来就是个风险问题，你希望 agent 多冒险得到奖励还是如何呢？当然冒险过头导致碰撞的话，自然奖励也没了。因此，我们或许可以把碰撞惩罚消除，然后直接用奖励吸引 agent？当然，碰撞是个终点，所以你至少要给个奖励 0，但是这样的话，逆行 agent 的奖励就是负数了。比碰撞还低，agent 就宁愿碰撞了。可是如果你不给负数奖励的话，那么就是告诉 agent 活下去就有奖励，但是奖励额度不同，那么是要小数额的长期奖励还是大数额的短期奖励呢？到达目标这个奖励是绝对要的。但可惜是这种事件太小概率了。如果碰撞，就得不到这个奖励了，所以如果只有这个奖励的话，它本身就能促进避免碰撞。步进奖励的问题是，它不是单一的，它是由幅度的。调整幅度上的不同数值，那么 agent 的冒险倾向就会改变。我们假设在这里引入碰撞惩罚。那么大体上来说，就是，只要活着就有奖励，如果步进，奖励会增大，但是碰撞了，就什么都没了。如果反向步进的奖励小于碰撞，那么 agent 一定回去选择碰撞的。避障和导航，这是两个不同的目标，融合的最佳方式是如果避障失败，那么就无法导航了。可是问题是，导航事件的几率实在太小了。难道我们只能通过缩小随机分布范围来增大导航概率吗？为了解决导航的小概率问题，我们原本的办法是使用步进奖励．但问题是，终点奖励可以一次性被碰撞抹消，但是步进奖励会留下来．这构成的问题是，这部分的步进奖励会构成一种冒险竞争因素，即使冒险碰撞失败，但是 agent 仍然获得了奖励．

这个方法总的来说是很有效果的，它的确能帮助 agent 接近目标那我们只能这么办了，用这个方法构建第一个体系，然后用这个体系生成能够达到终点的样本案例

还有一点，就是，我们甚至可以不这么做，因为步进奖励由于太直接了，基本上就是相当于直接把奖励加到了不同的 action 上．因此你可以期待，在普通的避障体系上，在危险系数小的情况下 (reward 较高)，我们可以直接算出对于导航最有利的 action 由此，就可以引导 agent 到达目的地了．这是个想法，但是既然我们有现成可用的辅助系统在，那么这个不可靠的想法就放边上吧．虽然这个实现起来会比较简单，但是可行度很难说呢

基本上来说，在普通的情况下,suc rate 的概率太小了，最高只有 0.001．在我们调整步进奖励到 0.02 之后,loss 倒是减小到了 0.05，但是我们搞不清楚这个结果有多少价值．所有这些参数，包括 discount 的调整都取决于你愿意牺牲哪一个特性．其中大概有一个平衡点，可以获得最佳结果，但是你不知道通过什么标准来评定最佳结果．所以说，我们看不到继续这部分实验的意义．找出一个特定的参数吗？更有价值的还是加入辅助系统．之后，靠奖励解决问题．奖励包括两部分，一个是撞击，一个是到达目的，因为这两者都是终点，所以都需要一个奖励设定．但是其实存在第三个奖励设定．实际上就是存活奖励．正是由于我们给的存活奖励是 0，按照 discount 削减后，最后得到避障的 q 范围是 (-1,0) 因此我们需要设定三个奖励数值，但是

自己根据 nav 有利的行为添加 reward 并不好，这还包括你到底希望在一个特定的情形下,agent 如何行动？完全背向的时候的确要转身，但是方向贴近的时

候改怎么转身? 这其中涉及了很多策略，还不容用现成训练好的.

关于步进奖励 0.02 情况下添加终点奖励的训练结果：不好，完全不识别方向. 所以 0.05 是必要的. 终点达成率太低，所以训练中基本上没起作用，我们倒是可以把 0.02 换成 0.05，这样就可以提高终点达成率了，你可以因此观察下，在终点达成率高的情况下，终点奖励能起到什么作用. 能解决 agent 从目标旁走过一去不复返的问题嘛? 毕竟这是我们当前唯有的问题. 其他的问题，虽然理论上 0.05 这个数值的确定很随便. 但是实际用起来却很好.

测试结果不太理想,0.05 的情况下，我们已知结果是可以引导进入区域的，虽然这个引导的稳定性不足. 但是似乎加了 10 % 概率之后，在训练中似乎时做不到引导进入区域的.

还有个问题，我们根据 check 来调节 lr，但是你注意到了，调节过程中，可能因为步进奖励没有稳定下来，所以 loss 是一直在上涨的，但是这个时候 lr 已经相当低了. 老实说，其实大家固定 lr 的方法和我们设定 0.05 的情况类似，是很随便的，最新的实验：关闭 lr 啊，关了 lr 之后，竟然会在 330 轮产生发散. 只不过少了两次 lr 递减而已

画图之后发现，结果大部分时候还行，但是有些时候，会撞墙，我们猜测原因是因为 agent 追求短期的步进 reward 而不顾未来的利益造成的. 所以我们希望用这个模型作为辅导者，然后重新训练

现在已经有画图总结成功率了，下一步我们可以对比下,fixed 13 和没有 fix，并且 lr 最小 0.002 的情况下的训练结果. 哪边成功率更高?

现在的问题是，我们，如果我们需要选择一个指导者，那么当然希望他是最好的，那么你就需要评估各个不同的指导者. 可选择的范围包括，是否固定 13,lr 怎么选择,discount 设多少. 通过这些对比，我们能找出一个最好的指导者. 然后使用这个指导者，我们可以训练一个全新的. 要点在于，新训练出来的要比旧的好再评估方面，我们需要看: 1. 成功率 2. 失败时候的步数分布，因为步数分布越小，说明这只是一个起步失败. 3. 成功的步数分布，这个分布也要越小越好，说明没有绕路. 眼下，根据我们随机选择的结果来看 fix 13 小 lr 的结果比没有这么做的要好. 原因是什么呢? 是 lr 的问题嘛? 这个实验很耗时间，所以我们似乎只能一边写论文一边去做了. 嗯，大体就是这样，训练一个网络，怎么说也该花一天吧?

我的猜测，虽然 loss 一直在上涨，不过 lr 似乎是必须要降低的，或许就是因为 lr 降低了,loss 才上涨，因为降低了，才找到了路径.100 回合其实够了吧，所以 lr 这个参数我们就不改了，还是去改 discount 吧，从今天到明天，训练一组 fix 下的 discount 0.95 看看

还有实验中加一组 random 进去吧，有很多东西可以对比，比如失败的情况下的步数分布. 成功反正大概是不可能了. 嗯，大体情况就是如此，那么下一组训练: discount 0.95,nofix,lr 无最小限制然后等明天的训练结果，顺便把手头的 model 做完测试，然后对比. 找出一个好的指导员，然后下一步测试指导方案. 实验就是这些内容了. exp todo mark 不过 lr0.02 在 300 轮之后竟然会发散，我们选了 100 轮作为基数还真是运气好.

然后关于 1000updates，周期太大的话，收敛会很慢，周期太小的话，可能会发散，选一个不会发散的平衡点

有机会的话，我们让 agent 通过双眼视觉训练一次吧，说不定会得到意外的结果．毕竟你很多参数都找好了．对比一下就知道了． 但是我觉得或许结果会很好，但是比较让人担忧的一点是过耦合，也就是说在当前虚拟环境没问题，但是做一个新环境就会有问题．这个问题或许可以扔到 future 里面去 STEREO 的训练在 500 轮发散了，第三轮 check 是 0.088，我在想发散是否是因为 overfit 的原因? 我们把训练速度放慢 4 倍，多生成 4 倍的历史数据看看结果． 看来这是一个难题啊

结果不好,loss 停留在 0.10，比起最好结果的 0.04 差太多,validation 中勉强可以避障，但是失误频率明显的很大．我们换用乘法吧，在好多 paper 中都看到了比如最近的 Stereo Matching by Training a Convolutional Neural Network to Compare Image Patches 中的 dot product，和你之前看到的那篇里面都是乘法不过上面这篇中有两个体系，第二体系不是乘法，第二个体系是在 conv 层次以前，分开计算 left,right，其实类似于 phi 的用法，在 conv 以前平等对待 left,right，之后到了 full con 之后再开始连接他们，所以我们要改动也简单，把 left right 转换成 phi 就行了 — 由于无法训练出来的原因，我们做了大改动．改动内容如下： 首先，之前训练的时候，由于没有打开游戏窗体，结果 stereo 图像的生成错误．所以那部分训练，相当于是使用了一个宽体的输入图像第二，由于稳定性的原因，我们把图像输出修正到-0.5-0.5 的范围，结果原来这个范围内的激活率太小了，需要修改初始 std 来加强网络活跃度，而且不能让他发散． 第三，我们加入了乘法门，作为左右图像的匹配性依据．依靠调整各层网络激活率，这个乘法的结果应该不会太过分的小第四，由于图像增大和 channel 的增多，历史记录长度缩小到了 6000，图片生成速度减低，所以我们加倍了训练频率，此外 batch 大幅缩小，一方面为了省资源，一方面怕 overfit 第五,check 周期从 100 改成了 400，也是怕过早降低会 overfit 吧，第一轮 check 就是 800 轮,loss0.06，其实前 400 轮也是 006，最为对比，之前失败的情况下都是 0.10，还有成功的则是 0.045

训练结果，可以明显看到接近物体的时候,q 会降低，但是似乎 q 的精度不够因此无法判断清楚左右，所以生存率不高． 这种距离判断是泛型的吗? 还是仅仅 overfit 了 chessbox? 我们无法知道，反正没有第二个场景可用． 那么精度要如何提升呢? 首先，我们的 loss 很高，所以我觉得,overfit 的可能性或许还是比较小的（还是说每次新增的图片都带来了 loss?），降低 loss 一是和 lr 有关，第二是和 batch size 有关． 第三，甚至可能和网络复杂度有关．可以尝试的方向：把乘法门关了，之前的几次实验都是错误的，包括在 11，同时让网络接触左右，以及 13 以后再合并左右．不在 11 做其实是合理的，因为左右图片存在对称性质．在 11 让网络接触左右，不是会产生大量冗余? 还是说我们的大量左右对称数据最终可以消除他们? 啊啊啊，我突然想到了，你做左右 flip 的时候，难道不是没有换左右眼吗? 幸好对于乘法门来说左右对称，所以大概影响不算大．但是呢，在 11 就让网络接触左右的优势是可以进行细节分析． 所以我们的乘法门，其实是接近于为了这样的要求弄出来的．

另外，我们要减低图片精度吗? 其中的问题在于，其实要声称 84 的 depth，需要用到 256 的 stereo 吧? 因为对于远处的物体来说，要后者辨识其中的差距是比较难的，所以明显要用高精度的图片才能生成 84 的 depth． 因此，总的来说，一方面，我们想知道如果去除乘法门，是否可以得到类似的结果? 另

一方面，要提升精度，似乎只有增加 batch 这个方法? 对比来说，现在也就是 batch 降低了. 但是这种和 lr 相关的问题，难道不能通过训练时间来解决吗? 如果我们训练的时间足够长，是否 loss 可以降低呢? 不过或许 batch 真的是个问题，因为你明显看到 log 中的 loss 波动从来没平息过.

---

mu184 的结果似乎不好,400 轮的时候 mu1256 已经可以进入 0.061 了，但是 mu184 只达到了 0.075，看来图像精度是必要的，不过因为图像大小不同，影响了网络结构，所以网络深度是不同的. 这说明，小图片看来不够用了. 不过 256 和 84 差了至少 3 倍，按面积是 10 倍

还有个问题，我们现在用的 13 mu1 是不分左右的. 这样可以吗? 说起来，如果不限制 (-1,0) 的话，原本的 depth 的 loss 也有 0.06 左右，但是却不像 mu1256 那么容易碰撞，是因为后者 overfit 了，还是，解除正数限制仅仅会影响 loss 但是对一般行为没影响? 大概只是没影响，因为这种情况下，其实拉大了 q 的差距范围，反而更容易对比找出优势选择了.

那么我们还有什么办法呢? 调整初始参数可以帮助网络进入活跃状态，后续的问题有两个,256 所提供的信息是否足够了? 对于人来说其实好像差不多了. 那么就是信息提取还不足，比如说,3x3 的挖掘我们没有试过. 但是这很吃计算量，所以很难确定核心数目

做一下 mu1 的深入分析. 大致来说,conv 在一个区域内，根据一定的特征会被激活，共有特征越多，说明左右图像越是想像，但是越是相像，说明物体越远，而远处的物体，是不用担心撞击的. 所以越是找不到共有特征，说明越有可能撞击，因此应该会的到较低的 q，眼瞎 mu1256 的 q 的宏观评估还是比较准确的，死亡之前的 q 基本都在-0.5 以下. 只要这个图片共有特征越少，越是说明障碍物多，可能会撞击. 但是要判别左右的话，需要什么信息呢? 需要发现，左侧或者右侧障碍物多，也就是共有特征少. 而不是全图性的共有特征数目. 其实对于 depthmap 来说,depth 也就是障碍物的存在，早就已经检测过了. 因此只要大体统计 depth 左半右半的 depth 深度就可以搞清应该选择的方向了. 但是 mu1 的情况也差不多，不过就我们所见,mu1 检测全图 q 还算行，但是似乎没有检测左右的能力? 其实也可以说是很模糊吧，其实 mu1 的训练结果倾向直走撞墙.

说下 mu190 的结果，虽然 loss0.070，但是结果还行，确切说，似乎是，碰到简单些的障碍基本上似乎是没问题的，但是对于复杂障碍的处理能力，大概是比 depthmap 差很多的. 有种感觉 mu190 的走动比 mu1256 还自然些.

其实 suc5 也是 400 轮进入 0.058，最后是可以进入 0.041 的. 不过这是最初那个时候，随机范围比较小的情况下. 但是有个问题，就是 suc5 就算那个状态下，碰撞其实发生就很小了吧? 还是进入 0.041 以后的结果呢? 看来我们是没有选择了，只有用 mu1256 了. 不过有没有优化的余地呢? 为了避免 overfit，有没有办法，增大 history?，把图片存到硬盘的话. 大体的机制应该是，在训练中，逐步的存旧图片到硬盘，取出一些图片. 其实图片生成的时候就可以直接去硬盘了,history 则是循环载入那些图片. 我们的预期是循环加载的速度至少要比生成新图片快一些.jpg 还是 npy 好呢? 前者要解压，后者要读取数据量. 还有多线程怎么处理呢?

这个机制生成之后，其实可以变成一个在不同 mode1 之间交互经验的办法. 由一个 mode1 生成的大量经验就可以很快被另一个用了. 至于单一 mode1，我觉得可以走两个进程，一个进程训练，一个进程和 panda 交互，然后训练进

程定时的保存 model，交互那边则读取，交互那边则是生成数据，由定时这边读取. model 的存取明显是可以定时的,io 不会很大，数据最好是定时批量读取，因为有目录状态检测问题，所以生成或许也是如此. 那么同步问题呢? 文件系统是有锁的，所以我们大体上只要在读取的时候，丢弃尾巴，应该不会有太大问题. 速度同步问题，现在明显生成是比计算慢的，所以最好在计算那边 sleep，等待生成了足够的新数据再计算? 还是说因为历史足够长所以没关系呢? 这个政策等会想吧. 那么我们是否跨进程，用文件系统来锁定，还是跨线程做内部锁定呢? 外部锁定或许比较好，一方面，可以方便我们去删除数据，还有就是可以使用不同的 model，用不同的 py 程序去运行. 其实更有趣的是，数据甚至可以通过网络同步. 不过真是好大的工作量啊，这样真的有用吗?

---

mu1256 的训练速度似乎不是 400 轮 0.06，因为当时出过一些 bug 导致过重启好奇怪啊，这次 mu1256 好像收敛不了? 我们搞错了哪里的参数了吗? 好奇怪，真的没有收敛，停留在了 0.066，我明明拿到过 0.055 的难道是因为之前的训练频率太高导致的 overfit? 我们来列举一下第一次 mu1256 的 log 记录. 第 400 轮的时候是 0.096,600 轮 0.087 700 轮 0.069,800 轮 0.061,1500 轮 0.055. 对比来看，最诡异的就是 700 轮和 800 轮的大幅变动的 loss 了，本来明明是高过其他地方的. 时间上来看，的确,700 轮恰好是唯一的一次程序崩溃后的 reset. 看来是一个训练好的网络在 reset 后轻易的 overfit 了新的数据. 结果上来看这个结果并不比 mu190 好原因看来可能同时来自高频训练和程序崩溃，同时 his 又过小. 至于验证，那么我们现在就改掉频率,reset 训练看看，结果似乎真的如此，第 100 轮,loss 是 0.048，往后也维持了.

那么没办法了，看来之后的优化只能选择 mu190 了吧? 顺便分析下 mu1256 的结果，还算可以，感觉问题似乎主要在于近视. 在物体接近的时候才会作出反应，一般这样似乎会太晚. 其实，老实说，对于有地面作为参照物的情况下，人是根本不需要 stereo 的，参照地面就完全足够判断距离了. stereo 在日常生活中实在是很难说有什么意义. 问题大概就是在于近视: 近处的障碍物能够回避，但是远处的不行，无法回避远处的话，就会陷入近处无路可走的问题.

猜测:reset 之后大幅下降的 loss，可能是因为和惩罚率有关? 在下载 model validate 之后，发现，并不存在 overfit，是因为生成的 train 的数据和之前不同了? 换成全随机看看呢? 嗯，果然问题在于训练数据本身,reset 之后明显惩罚率很低，是 0.007，而之前是 0.015 另外，虽然 datasize 限定在 6000，但是惩罚率在后期却没有变动，大概是因为惩罚率是在 failed dataset 中提取出来的. 如此分析完的话，就是 mu1256 的 loss 并没有优势. 还是要从 mu190 下手，那么怎么改呢? 首先 r1 和 his 基本上是没问题的，然后你也看到了，改图片质量的效果不大. 有一点我们不清楚的是乘法门到底起了什么作用. 可是如果不用乘法门的话，我们应该在哪一层结合左右呢? 先试试看在第一层结合?

测试结果好像乘法门没什么用处，没有乘法门，第一层结合左右也可以达到 0.065 下一步我打算在 13 结合，并且同时使用乘法门试试看能否进一步优化. 哦，还有我们需要试下，不使用 stereo 的情况结果如何. 不过，不使用 stereo 的话，结果应该不会比 11 结合的更好了. 你可以假设 11 结合的 stereo 没有起作用的话，那么基本上就和没有 stereo 的情况类似了. 所以 13 结合 + 乘法门可以出结果的话，会说明结果不错吧? 但是有一个问题是，

对人来说呢? 有地面作为参考物的情况下，要判断距离是非常简单的吧，才不需要 stereo 当然 84 的话，精度略低，不过 256 我们也试过了. 通过地面来辨识距离，做不到吗? 果然还是网络太小了?

---

13 的训练结果也在中途崩溃了一次,reset,reset 之后,loss 进入了 0.060，也是碰撞率不足引起的.

结果是，没有 stereo，没有 reset 的情况下，得到的结果是 loss 0.070，也就是说,stereo 的确勉强的提升了一些成功率.(还是说因为 his 比较大?) 不过我们这其中还把网络结构改了，或许新的网络结构太复杂了? 看来好像是我们网络没做好，原因可能是初始参数的问题. 不过我想搞不清的一点是，学习速度和初始参数的关系，我们现在使用的学习速度是否足以改变初始参数? 实际上新网络在 14 14_2 的 w 并没有多少改变，改变全在 15，或许 14 的输出太小了，导致后面的层全都没有改变. 因此其实我们的网络实在近乎瘫痪的情况下训练的，看来每层输出不可以太小的.bn 的价值就在这里吧，输出太小会导致改变不足，输出太大则会导致波动太大总的来说，这就是问题，除非你在每层的输出都维持 1 左右的 std，太小的话就会对上层产生递减效应. 导致上层的 lr 过小. 顺便说下，其实使用 stereo 的情况下,w 的 std 也没怎么改变.

我觉得我们可以在每次网络保存的时候，记录下 weight 的变动，尤其是网络初始阶段，加大保存频率，好好记录下来参数的变动我们尝试调大参数，避免僵化，但是离散了，离散的原因似乎是因为，某些参数上产生了大幅度变动.

看来的确有调整参数的必要，调整后第 200 轮的 loss 是 0.075 比之前低了很多，不过虽然初期很快，但是到了 300 轮就停留在 0.068 了，看来是因为 lr 相对 weight 比较高吧. 不过就是要高一些，才能在初期大幅变动 weight 造成影响. 由于我们的目标 q 范围是-1,0 大致的 std 需要是 0.1 左右，超出的话，似乎就会导致发散，为此 15 就设定了 wstd0.1 再高就会出现发散了.lr 大体也是在这个范围内? 能试试看 lr=1 吗? lr 应该要和 std 处于同一个数量级，太大的话，说明 std 设定在了近似 0 的位置，也就是初始随机不够，太小的话，则无法撼动初始随机，结果网络就僵化在了初始状态因此，我们应该用这种方法来调整参数. 从最底层网上,std 设为 0，然后调整顶层的 std 和 lr，找出一个不会发散的数值，然后让 std 和 lr 处于近似的数量级，确保网络没有僵化，然后逐步向下调整 std 找出僵化和发散之间的极限数值. 这和 bn 很像吧?bn 有着对数据分布严格要求的缺陷，并且不一定足够精确. 但是 bn 能动态调整各种数值. 那么有没有什么办法可以优化 bn 呢? 似乎是有的，加大 batchsize，甚至是记录历史上的 batch 来设定 bn. 回到我们的问题: 大致上来说，每层网络的 wstd 设定在 0.01-0.1 左右，这样每层的输出 std 就会有 0.1 左右，一直到顶层，设定 w 在 0.005，比较小，但这里可以设定 lr 为 0.5，然后你会发现每层的 update 基本上都会有 max0.01 mean0.001 左右，所以波动已经开始，不用担心初始数值了. 不过有点奇怪的是 14,max 是 0.01，但是 mean 是 0.0001，说明整体上改动不大，但是存在局部的大波动. 显然局部波动很有效的传递到了下层，主要问题是，我们要提高这里的波动吗? 因为 mean 太小了. 可是如果加 10 倍，好像会出现很大的 max，可能会导致发散? 然后虽然各层的 max 比较高，但是从效果上来说，在 params 的整体 update 上，几乎没有造成什么变动. 除了 15，原本是 0.005，

现在是 0.012，要不然直接改各层 lr 算了，比如说各层的输出乘以特定的系数比如这个样子，大概估计下首先是各层的输出波动，我们假设在 1 左右，上一层也是 1 左右，这个数值乘以 lr，大体就是 wupdate 了，然后 wstd 有必要造成一定的初始随机，但是同时也要能够被 wupdate 撼动，理想的比例或许是 10 倍，比如说 15，的 wstd 是 0.005，而 wupdate 则是 0.0007 左右

在这样的条件下,loss 进入 0.050 了．不过或许用 depthmap，调整 lr 后能够得到更好的 loss?

在某一层乘以一个系数，其效果类似于改变了其上一层 weight，其上一层的 weight，原本是由 lr 的波动幅度决定的，但是如果这个波动太小，不足以传递到下一层，波动太大，则会导致上一层崩溃，在其上一层波动固定的情况下，sgd 的算式是 update=loss/gradient*lr 换种说法就是 wstd=out_{std}/in_{std}*lr 对于 15,out_{std}=0.01,lr=0.5,wstd=0.001,in_{std} 是 0.1，并不是怎么符合这个算式,std 和实际的 update 之间有很

现在的问题是，如何升降 wstd? 提升 out 下降 in? 是的，如果 out 和 in 联动的话，那么网络的其他部分就不会被影响了．不过，理论上联动可以防止影响其他部分，但是实际上 15 在 14out 提升之后就变大了．结果就发散了因此 14 的 wupdate 似乎无法提升，那么还不如减小初始值，来防止僵化，问题是这样就连带影响了 15 的数值因此，如果降低 14std，就会导致 15 数值的改变，但是 14 就不再僵化了，不过，为了避免 15 的变动，可以给 14 乘上一定的系数．也就是说，初始值要设定在大约可以和 wstd 匹配的范围，如果这样，会导致输出不足，影响后续的网络，同时也影响到 wstd，为了解决这个问题，可以给输出乘以系数．但是在训练过程中，你会发现，按照原来不会发散的配置，按比例变动初始 w 和输出系数后发散了，说明 w 波动过大，网络承受不了了，

为了避免学术表述的问题，我们的图像输入可以设定在-128,128 的范围，这样子不需要提升输出系数，也可以保证一定的输出大小了我猜可能有这么一个矛盾．顶层网络的稳定性问题，底层不稳定，上就会发散，太过稳定，则找不出必要的解答．这也就是 bn 的手法．

---

bn 真是难用，怎么都会崩溃． 按照我们之前调的参数，可以用 lr0.5 进入 0.05 的 loss，这个训练结果测试过后，发现准确率是很高的．这也是合理的，毕竟人是不用依赖 stereo 来避障的．唯有的问题是可能是一种 overfitting，由于我们的输入数据是图片，维度太大，这和一半的一维深度数据是不同的，另一方面我们的场景中的特征又很单调． 顺便说下，在 loss0.05 的情况下,survive 可以上万，但是有很多情况下会得到 1000 以下的 survive

那么这里有没有我们进一步优化的必要呢? 我们现在的一个问题是底层网络感觉是僵化的,update 不够．总觉得那里的参数基本就没怎么改变.std 就没变过． 总觉得如果能让其不再僵化，而且又不发散的话，应该可以得到更好的结果.

其实应该是这样子的，如果你怕网络僵化，那么我们其实可以提升 input 的系数，然后对应的降低初始 w，这样子整体的 output 就不变了，也就是说，原来是 x=1，牛顿法解 $(3*x)^2=0$ 现在是 x=0.1 牛顿法解 $(30*x)^2=0$，那么也就是 $9*x^2$ 和 $900*x^2$ 导数是 18*x 和 1800*x，也就是 18 和 180,y=9 dx=y/d 也就是 0.5 和 0.05，这样子从比例上来看也不对啊，两次的 update 的比例是差不多的． 为什么会这样子呢? 折算出来的不对啊． 实际情况是,15 是因为 input 的数值大，才得到了大波动的．当然前提是 15 自身没有变．如

果 15 的 input 很小的话，他就得不到波动．为什么呢? 为什么总的 output 在 0 附近的话，就得不到波动呢? 而且所谓波动，并不是 loss，因为波动大，我们才把 loss 快速降低了．其实我们这里说的波动是 wupdate．为什么 wupdate 只有在 output std 大的时候才会变大? 怎么说呢，不管 update 大小，初始的网络因为随机因素，其 loss 必然是差不多的．但是 std 较大，意味着存在较为明显的特征，特征足够明显的时候，应该会得到比较值得依赖的 kernal，因而才产生了大幅度的 wupdate 变动? 但是，在最初的时候，我们设定 b 是 0，这个时候 w 的系数不管怎么变，只是影响了 kernal 的大小．为什么较大的 kernal 会导致大幅 wupdate 呢? 不对，确切的说，应该是，同一个 kernal 对于 batch 中的不同图片，都给出了相近的信号．可是这还是没有解释，为什么和 w 系数有关．因为 b 全是 0，所以 w 系数怎么说都不该影响结果的．倒是可以影响 b 的导数，但是第一次训练,b 还不会反过来影响 w 啊，不过第一次训练，我们好像是不保存的，确切说，完成一次训练才保存，而第一次保存没有可对比的参照，因此，这里的确会生成 b 虽然说 b 是受 w 的影响产生的，不过 b 我们可以假设 b 的产生大概是固定在一个范围内的．这里就产生问题了，如果起始的 w 太小的话，这里很快就会被 b 干扰变成僵化的 kernal．其实神经网络的关键就是因为有 b 存在，否则就只是一个线性模型而已．那么，如果是因为 b 和 w 的匹配问题的话，增大 input，降低 w 或许倒是会有效果的? 因为单纯的降低 w 后，好像 kernal 的激活不足，但是提升 input 就能带来激活了吗? 如果可以的话，能保证 wupdate 吗? 多想也没用，试下就知道了．那么你预期会有什么结果呢? 第一，没有效果，第二，出现发散，第三，有效增大了 wupdate,3 是介于 1 和 2 之间的，有没有 3 存在的余地呢?

它们的关系绝对是非线性的，不过 20 倍而已,wupdate 就从 e-5 涨到了 e-3 左右，然后发散之后就更大了，而且本来预期输入的增大会导致 wupdate 的对应减小的，结果反而是增大，完全不和简单线性推理啊不过我们改 input，对于 b 来说是个问题啊,b 跟不上 update 速度了吧?

提升 10,20 倍,init=0.01 发散了．提升 5 倍,init=0.04,wupdate<0.001,100 轮 std=0.042，也就是 1/20 的 update，对比 15 是 0.012，是 7/5，10 倍,init=0.02，发散了．下一步,12 5 倍,init=0.02，发散了．但是优先表现出发散的是 11，不知道是不是和 11 的调整联动的，那拿到要把 11 的改动删掉试下? 2 倍,init=0.05，也发散了，11,2 倍,0.02,12,2 倍,0.05，没有发散，看来是联动了．不过 11 明明已经设定过 init=0.02 了，就算 input 加 5 倍，对于输出应该是没有影响的，可能的影响就是，如果 11 的 w 大变动的话，那么就会影响到下一层了．那么如果我们进一步减小 init 会怎么样呢? 反正减小 init 对我们想要避免网络僵化的问题有利，唯一的问题是怕 init 减小后 wupdate 也减小了，但是既然现在加倍 input 会导致发散，那么我们减小 init 会不会减小发散的可能性呢? 在 3 层 input 加倍，并且 init=0.02 的情况下，我们的到了一堆 wupdate=e-5，当然我们不算把 init 改回去，那么只有进一步增大 input 了．增大哪里的 input 可以同时调整所有的 wupdate 呢? 只能从 11 试起吧? 又是 11,5 倍，不过意外地这次没什么效果．那么进一步,12,5 倍，这种试验要做长一点，因为发散可能会在一段波动出现之后再出现，因为波动会改变统计分布，啊，果然发散了．那么把 11 的 5 倍拿到 13，结果 11,15 在 0.001 以内，其他层在 e-5，其中最小的是 14 2e-5，不过反正 14 的 init 就很小，所以没问题吧? 不过结果在 100 轮左右崩了．那么怎么办呢? 保留 11,5 倍，其他地方都去除? 结果，估计在 50 轮左右,15 的变

动依旧是 140%,11 是 10%,12,13,14 都是 1%，可以说，略微比之前好些? 之前训练完都是 0.5% 的变动但是 check 就很差了,200 轮是 0.077 之前的结果是 0.071 我猜原因是 15 的 update 变慢了? 进一步的训练后，各层 update 百分比还在提升，其实我或许本而来可以进一步降低那里的 init 的. 300 轮是 0.062，这是原来 500 轮的结果，看来是后发优势了. 顺便说来，加倍 input 本质上是因为 init 不足，如果 init 后期能自己改变，那么就不需要 input 来加强 update 了，但是 init 改变本来就是因为 update 强才能做到的. 那么就是说，如果我们在不触及 update 的情况下降低 init 的 haunted，还是会很有用的 400 轮 0.060，这就感觉没什么特别了关于 update，或许存在一些临界点,11 大概是达到了，所以可以做到 70%，但是其他层只有 5%，我们现在降低 init，但是给 14 做 input 翻倍，因为那里还没做过，不过其实那里的活跃度比 12,13 大，不给 12,13 的原因是担心那里后期统计分布改变的话会不会出现问题? 结果，这一奇怪的是反而是 14 的 update 减小了，不过有一点，原本上层网络是依赖较高的下层 init 来 update 的，但是现在下层可以通过 update 来提高 w，所以训练一段时间后，上层网路还是有机会得到必要的 kernal 来进一步 update 的. 现在的状况是,12,13 的 update 超过 10% 了，或许能突破临界点? 但是 14 很小，只有 2%，这次的 loss,200 轮 0.078,300 轮 0.0605，更明显的是后发了. 的确，在训练早期，你也看到了,update 是比较低的，后期 upate 才开始上涨在把中间层的 init 改小之后，训练到 600 轮后，网络崩溃. 看来是因为统计分布的改变导致不适应吧. 但是这个崩溃似乎是概率性的. 第二次我们训练就得到结果了. 现在唯有 14 还没能有效 update，但是 12,13 也算 update 了，所以大概算是要固定了. 大概不会崩溃了吧? 刚说完就崩了，原因是什么呢? 是 14 的僵化吗?

我们不希望任何一层僵化，但是不僵化，概率分布就会变动，

顺便说下,11 的参数画出图来后基本是一团乱麻，看来是没训练过的.std 是 0.005,mean 是 0.195 lr0.5 似乎停留在了 0.049 了，当然 lr 还有进一步收缩的余地，和当时的 depth 网络不同，这个网络的 lr 收缩余地很大啊.

就算用 bn 锁定了分布，结果 update 的效果还是 11 最好，往后的层基本没怎么变，唯一的好处是现在的网络不会那么容易崩溃吧? 不过 dengse4 还是没有拉动. 的确很难理解. 为什么 dense4 可以不起作用呢? 可能是因为 dense4 涉及到了 7*7 的点阵，其中并不是每一个都有用处，所以没用处的那些就被抛弃了吧?

大体上来说,bn 防止了崩溃问题，不过为了将来的 validation，显然 bn 训练是必须要记录下 batch 概率分布的，这个之后再说吧. 然后没有崩溃之后，除了 14，其他层多少在起作用了，应该能追上 depthmap 了吧 1

训练一晚上 bn 的结果，进入了 loss0.032，虽然我猜其中有一定的原因是 overfit 吧? 因为我们现在用的训练方案和原本不一样，在后期的随机行为是降到 10% 的，所以碰撞率真的很低. 然后是验证环节，基本看不到明显的犯错，也就是说，即使是因为碰撞率降低，但是 loss 达到这个程度的话，算是可以避免碰撞了. 我们大概要解释一下，在被逼不断前进的情况下，即使是人也会犯错的. 基本上平均存活率在 2000 左右，不过这个数值略微有点勉强，因为 depthmap 的情况下，很多情况下是超出 5000 的，虽然也有不少情况在 1000 以下. 当然这其中可能要考虑到现在的 bn 不够准确的问题，是否需要统计下分布情况，然后重新训练，还是说我们要用原本的训练方案呢?

用 raw image 训练 nav 的 loss 是 0.083，检查了下,l1 的 kernal 图像和避障中训练出来的一样. 然后本地 validate 结果也不错. 似乎不用浪费时间去处理 bn 了，因为反正既然已经做到了，那么处理了也没有多少提升余地了. 另一个问题是，我们使用了步进奖励，而不是终点奖励，原计划是用步进奖励积累经验去训练终点奖励的，不过也挺麻烦的，而且眼下也没有提升余地，所以即使做了也无法判断好坏.

真正有价值的，是将环境迁移到真是世界会碰到的问题，包括是否我们的模型 overfit 了虚拟世界，我们虚拟世界是否过于规整，导致了即使不用 depth map 也可以简单推算出距离信息? 第二，即使没有 overfit，考虑到训练需要的数据量之大，这些数据必须要自动生成. 而这个系统在每次装车后都有重置的必要，在现实世界，这种重置或许要设计一种特殊机制，甚至是要人为的去做. 这是 agent 无法自己做到的，如果说能够让 agent 学会自动从困境恢复的话，就可以不用考虑在现实世界中重置的问题了

关于尝试放松 bn，之后的确得到了更好的 loss，但是 survive 验证却很成问题，成功率不好. 嗯，确切的说，是我感觉有些不应该失败的地方失败了，但是实际上成功率好像和 0.1bn 差不多，勉强差不多的样子

在加大训练频率，以及放松 check 频率后,loss 下降进入 0.08 了，本来以为是 overfit，但是意外的，失败率进入 1/6 了，原本是 1/4，怎么理解呢? 因为 check 频率市跟随训练频率的，所以，这次相当于加大了高 lr 的训练频率，也就是说原本维持在高 lr 的时间不足吧. 而且我们发现了样本只要 45000 就够了，如果说 1 秒 10 个样本的话，就是 4500 秒，不过一小时而已，但是考虑到一次撞击只有 50step 的话，就是撞击频率是 0.2hz，总计 1000 次撞击样本吧

尤其关键一点是，高频高 lr 训练下,q mean 明显降低了一个猜测，或许我们的场景太简单了,overfit 很容易. 不过现实或许也不过如此之后给图片加入一点光影方面的 dataaug 吧，

上面只是初始状态，后续比例依旧是 1/4

关于 bn 分布优化要怎么做? 训练的时候，样本很多，并且和 loss 有关，但是实际样本对参数的实验少，本地的时候，样本分布和训练的时候不同了，但是参数已经固定，所以可以大量实验. 由于远程的样本分布和本地不同，对于同一个样本，远程和本地给予的 bn 分布修正是不同的，除非我们用远程样本.

怎么看这里都很成问题，一个解决方法是用 bn 预训练网络，之后逐步 fix 不含 bn 的网络部分，然后训练其余的部分，使用这种方法的前提假设是，预训练的层次是完美的，因为之后就没有机会 finetune 这些网络了? 因为抽离了 bn 后网络会不稳定嘛，眼下其实由于我们证明 65000 组样本已经足够训练了，所以存储样本也是一个可选项了. 我们可以把 r1 转变为 s1 问题了，

不过眼下，我们需要知道的是，是否我们能通过逐步 fix,pretrain，来得到一个远程本地 validataion loss 一致的结果，现在的不一执性，如果排除 bn 的因素的话，就会变成 overfitting 问题了，而你加大训练频率显然加重了这个问题

当前的计划，首先用 bn 训练,train 频率为高，所以有 overfit 可能，甚至

可能和 agent 随机率有关，已知的是，随着 loss 下降,q mean 也下降了．然后训练到 0.060 左右之后，会锁定一层参数，去除 bn 和 prelu 重新训练，直到锁定三层之后，如果锁定三层的时候,loss 可以进入 0.060，那么就 ok 了，可以进行本地 validation 了，但是如果做不到，就需要进行 finetune 了，

记录下现在的方案：使用固定的 bn(b=0,r=0.1) 来训练，因为这样子速度好像快很多？我们之前的确有一些快速方案，不过好像找不回来了？然后，从第三层开始，逐步解除 bn，另一个方案是从第一层开始，逐步固定 pretrain，重新训练．感觉第一种方案比较好，因为免去了 finetune 的问题．这种方案面对的问题，主要在于，解除 bn 之后是否足够稳定，以及是否能够进一步获得更好的结果．第二个问题是第二种方案无法处理的，除非有 finetune 方案．保证稳定这一点，我们可以从第三层开始解除，也可以从第一层开始解除，但是第一层解除带来的发散问题会在第三层积累下来，所以我们还不如从第三层开始避免发散积累？但另一个问题是，解除的点上，会带来不稳定性，这种不稳定性是向上扩散的可能性比较大，所以从底层开始解除，实际上，可以帮助逐步稳固下层．还是说我们应该先稳固上层？因为最主要的不稳定点是 13 和 14 之间．另一方面，由于我们使用无缝解除 bn 的方案，也会有可能实际上解除 bn 带来的波动不是那么大的．

确认了一点，就是固定的 bn 比非固定 bn 训练更快．原本使用非固定 bn 是因为担心固定 bn 潜力有限，不过，既然我们打算解除 bn，那么这就不是问题了．

关于解除 bn 的方案有很多种，一是到底从底部还是顶部出发，二是到底在什么 loss 的状态下进行？我们加大训练频率后，没有尝试过和固定 bn 组合，所以其实不知道固定 bn 的 loss 极限，是 0.060 吗？我们应该在什么样的 loss 状态下解除 bn 呢？眼下来看 loss0.080 的时候解除，对于 loss 的影响是很小的，解除 bn 之后我们的目标 loss 又是多少呢？应该不会比有 bn 更好吧，所以目标应该定在 0.060 由于当前来看解除 bn 之后带来的 loss 波动并不大，所以，其实可以在 bn 状态下接近 0.060 之后再解除 bn，时间上其实应该是差不多的，甚至,bn 还能加速训练.(甚至不知是加速，而是带来本来无法得到的结果) 我们的最初目标，并不是要超越当前的 loss，而是希望使得它脱离 bn 控制，避免被 dist 影响，所以，首先通过 bn 接近 0.060 的策略是合理的

顺便提一下，当前 1r0.5 的情况下，某些状态下，训练早期可能会出现发散，不过反正避开发散就能得到好的结果，所以这暂时不是我们需要在乎的顺便说一下，发散的原因在于训练过程中很诡异的 q mean 增长，超出 0，一直到 0.50 左右，这个过程 loss 会一同增长．这个 q mean 的范围是不合理的，但是我们不理解产生这个现象的原因，所以也无法修正，唯一知道的是，在加大训练频率后，这个值多少能下降一些，不过在有正向奖励的情况下，到底是否网络还是维持正值呢？实际实验中，我们是知道的，接近障碍物的时候,q 为负数，而接近目标的时候,q 好像为 0？这个现象产生的原因似乎是因为马克夫链的传递效率不够？能找到加速的办法吗？

不过，固定 bn 的方法在第五轮 check(6000 轮 loss) 的时候,loss 就和不固定的情况差不多了，虽然初期的时候，固定的方法超过了不固定的方法．因此，我们暂且只能试验下，这个方法如果 loss 不够的话，还是不固定比较好？

结果固定 bn 的 loss 似乎停留在了 0.076，所以我就在这里解除了 bn，解除顺序是从上往下，在解除 bn1 的时候，出现了 loss0.11，但是很快回复

了 0.080 也就是有轻微的波动，实际上，由于我解除速度之快，而且每 50 次 loss 才有一次 save，所以从 save 读取的解除后的 network，实际上是相当于一次解除了 3 个 bn 限制，但是对 loss 的影响并不大，下面我们就期待，从这个 loss0.080 出发，最终可以到达 0.060 吗？ 如果不行的话，那么最初的训练只有放松 bn 来做了，很明显的 loss 差别很大啊.

突然意识到,bn 有 regularization 的作用，比如我们从原场景切换到 tiny3d 的时候，就吃了 regularization 不足的苦头，我们是这么猜的，虽然具体原因还不理解. 但是，如果我们不做 batch normalization，而是在 image 内部 normalize 内？可以吗？ 作为图片来说，其实即使这么干，仍然是可以保留图片的形状轮廓的，因此 qval 不变，所以其实应该是没问题的，这相当于 dataaug，但是上层 kernal 就不能简单这么说了，因为其中或许是包含了下层的形状信息的，也就是下层的形状信息，会在上层转变为 kernal 强度，也就是颜色，这个时候如果 normalize 的话，会导致距离评估错误的吧.

那么还有一种可能性，就是 overfit 了，当前，得到的本地 loss 是 0.11，服务器是 0.07，还是比较接近的. 但是我无法预见到 overfit 的存在，因为收敛是如此的难，应该没有 overfit 的余地

这个问题很重要，另一方面固定 bn 方案的收敛大概是停留 0.70 以上了，所以我们在远程测试这个 overfit 问题吧. 或许我们应该在远程测试这个问题，甚至是可以使用 bn 的方案，因为远程的 batchsize 更大

有一个方案是，在 train 的同时，抽取一部分的数据，作为 validate 来用，那么就可以知道大体的结果了. 这里面的问题在于，是否有必要要求 agent 在两个不同的样本群中有着同样的表现？在 classification 问题中，这代表了 overfit 问题，但是，在 rl 问题中，这个是否重要呢？因为 rl 其实不是极力避免去学习那些低 qvalue 的数据吗？因为如此才能节省下足够的学习能力，而如果完全遍历那些低 qvalue 的情况，似乎会的到指数级的大数据量，所以 rl 抛弃了它们，但是即使如此，我们可以依据同样的 epsilon 来积累 validation 和 train 的数据，这样分布问题就解决了，

因此，我们就能知道怎样的参数会带来 overfit 了，同时也知道哪些 loss 是我们有能力通过 validate 重现的现在有了 val loss 对比之后，我们需要弄清楚，调整训练频率会带来什么问题，实际上，加快频率对于训练是没有意义的，除了能加快速度，之前对我们有意义的是，给予高 lr 更多训练次数. 这是个选择，现在我们选择使用慢频率来训练，看 overfit，如果连这样也有 overfit 的话，那么高频训练的 overfit 自然就更强烈了.

实验结果来看，即使放低频率，还是出现 overfit 了. 但是其实我们之前在低 loss 的情况下，的确得到了一些很好的结果，所以应该是在 overfit 的情况下，训练也一定程度的起效果了

由于 overfit 应该主要出现在避障问题中，而且限定-1,0 的时候 loss 非常低，所以我们应该在那里测试下，到底避障问题的 overfit 有多严重. 最近我们老是用—raw —nav 在调整网络，不知道去除它们之后网络训练速度如何，毕竟原来的训练速度可是很快的

切换了-1,0 限定后，速度依旧很快，老实说，真该用这个办法 pretrain 前三层，因为-1,1 下，训练真的很慢，如果为了避免 overfit 而减低训练频率的话，那么速度就更慢了，所以最好是能在-1,0 的条件下解决 overfit，然后再训练 nav

这次 overfit 之后的一些办法,1. 调节光影，因为反正我们在房内加了灯照因素了. 不过具体怎么做不知道呢，简单的做法是很耗费计算，并且不够真实 2. 降低训练频率

顺便说一下，其实之前在其他地方读到过，在 loss 计算晚期，有时候可能会出现 overfit，但是有的时候，却是实在的降低了 validate loss，何况我们在 r1 的环境下，一直在加入新数据，所以其实我们应该更长久的保持大 lr，训练的时间足够长的话，或许 validate loss 会和 training loss 一起下降. 这个方案其实配合低训练频率应该会更好些. 甚至于在-1,0 的条件下，我们应该可以降低 batchsize，这或许也能减少 overfit，很可惜的是对计算时间没有帮助，因为计算消耗主要耗费在了生成新数据上.

tloss 从 0.062 到 0.058 的时候,vloss 从 0.069 提到了 0.063，平方根是: 0.25 到 0.24 0.262 到 0.251 -1,1 收敛的确很快,6000 轮 train lr0.5 可以进入 0.042，这个时候 vloss 是 0.061，因为其实看起来挺显眼的，我们应该把这个 0.061 在本地测试下，但即使如此，本地结果还是很糟糕，仔细检查之后，其实不是很糟，实际上那是初始阶段碰撞率太高造成的，实际上现在的状况是，本地的 batch 很小，所以有些 batch 大概是没有碰撞的，结果就是，没有碰撞的 batch 可以得到很低的 loss,0.03 左右，而另一些则比较高了. 说明 overfit 主要在于不同的碰撞形式上. 当前状态下,validation loss0.062 应该是极限了

眼下看似乎结果挺不错的,validation 能够紧跟 training，暂时还没有到 check 轮，就是数据生成的速度慢了 3 倍，看来大数据量是必要的.

现在的状况来看，的确是严重的 overfit，并且看来是可以通过低频训练解决的，剩下的问题是，即使我们消除 overfit，是否能够带来更好的结果呢?

当前的实验方向，通过低频训练把 overfit 解决掉，通过 dynamic cmd，在此基础上消除 bn，进一步确保本地可以重现服务器上的性能. 因为独立训练的低效率的原因，在 dynamic cmd 中，固定前面的 13，训练 (-1,1)nav，由于原本的问题反正在于训练太低效，所以可以再次解除 13 限制，做 finetune. 最后获得一个没有 bn 限制,valloss 确定的 model. 消除 bn 的主要原因是,qval 和 train,validate 的时候的分布是不一样的，为了以一样的分布得到一样的结果，才需要消除 bn，因此消除 bn 的效果大概无法从 validate 中看到，只有通过 survive 验证了，

在 loss0.050 的时候，下载了 model，因为本地 batch 很小，所以不知道整体情况，但是按照少量样本评估，基本上本地 val 重现远程是没有问题了，我们最好如此训练一组但是其实之前没有查看 val 的情况下，已经训练过低频 bn raw lr0.5 了，唯有 lr check 的变动速度比较快，且 bn 是 fix 的之前的 nav，在低频率训练下,loss 是 0.081，当然其中也有 check 或许给的太小的因素，因为 nav 比起 raw 到达状态需要更多时间啊.

不过，我很怀疑，使用当前的 train rate 到底能否带 vloss 进入 0.033? 还是说需要我们通过 dynamic cmd 调整 train rate? 我们需要找出来一个合适的 training rate，可以给予近似的 vloss 和 tloss，

四分之一 batch 左右大小的 trainingrate 依旧带来很强的 overfit，我想或许办法只有减小 dataset size 了，我倾向于认为,overfit 是因为 dataset size 太大，导致了图片的重复使用引起的. 那么终极的办法就是每次生成

的图片，都只用两次（左右对称），不可以重复用，重复自然会有 overfit 在-1,0 限制的条件下，或许我们可以降低 batchsize 也不会出现发散，当前来说,batchsize 非常重要了，直接和数据生成速度挂钩了．由于 dqn 所提到的耦合度问题，我们或许需要 dataset，但是或许这个问题并不是实际存在的? 这样的话，即使我们去除 dataset 或许关系也不大? 出问题了，原来 dataset 的 pop,append 机制不对，以前没有注意过 overfit，也很少有超出 datasize 的限制的情况出现，所以从来没注意到这个 bug 吧．现在为了 overfit 需要更新 dataset，自然就注意到了．不过我们最好依旧把全新数据的训练法加进去，如果原先的 training rate 依旧无法解决问题的话．

不过即使是-1,0,batch 变小的时候还使用 pop 方案的话，连续好久都不会有碰撞，而随机取出的时候，其中是夹杂碰撞的．这会导致接近碰撞的 batch q 偏小，而其余的则是 q 偏大，这一定程度上是 ocsillation，降低了学习速度．不过，我们有 failed dataset，一个 batch 它占 1/16，调整 failed dataset 是一个调节失败率的手段．但是如果说这个 dataset 占比重过大的话，在后期失败率低的情况下有可能导致极慢的学习速度，毕竟有的情况下数千个 step 中，都有可能没出现失败．实际上，为了避免 overfit 到这些失败数据，你一开始就不应该使用 failed dataset，关于失败率太少的问题，或许我们只能把 randomaction 提升到 50% 了这构成了一个很大的矛盾，一方面，我们需要高失败率来提供各种失败场景，确保 agent 能学到，另一方面，失败率太高得话，就没有成功率了，而我们另一方面未来预想的是依靠成功率来学习奖励．

对于当前来说，很明显的是，高失败率有助于避开障碍物，高成功率却没有什么帮助，但是训练中高成功率会带来更低的 loss 但是我们的目标是提升 survivemode 的成功率．没有办法,batch 太小的时候 loss 波动实在太大，这样实在无法收敛，可惜的是，这个问题不是 dataaug 可以解决的，如果说小的 batchsize 因为没有失败率而无法稳定的话，那么我们只有用大的 batchsize 了，也就是依据失败率决定 batchsize 或者说调高随机 action 概率，来增大失败率．

和其他 r1 比较起来，我们的 r1 中惩罚事件的概率非常小，因此再无法有效学习的情况下，网络的从优解是放弃这些小概率事件，而主要优化其余的部分，由于为了避免这个问题，我们才希望加大惩罚概率，希望网络可以偏袒惩罚事件．其实对于人来说，惩罚事件即使小概率，却是绝对的，怎样的学习都不会影响到这种绝对性．我们这里情况却是，通过网络对大量普通事件的学习，网络会很快的遗忘惩罚事件

这是一个很大的问题，甚至会影响到向现实迁移，也就是学习对小概率碰撞事件的依赖性，和 overfit 问题之间的矛盾．我们有一个 failedset，专门用来强调失败例子的存在，如果说为了 overfit 问题而不让网络重复看到数据，那么这个 set 就不该存在．可是这个 dataset 的存在，明显的加大了小概率事件的频率

实际上发现 dataset 设定到 30000,tloss,vloss 就基本同步了，差别在 0.001，我们也可以把 dataset 设定到 10000，我想问题应该不大．

用以上方案训练的结果，直接转向 nav，受到波动不是很大,q 维持在 0.1 左右，本地验证的确 loss 进入了 0.05．但是实际结果并没有导航能力，避障能力也算一般．明明 loss 下降了，但是能力却变差了．我怀疑原因是之前的 targetpositon 的 kernal 没有起到什么作用，所以 weight 基本都被消 0

了，所以没法训练吧? 另一方面，也说明存在另一个局部最优解，就是完全不去考虑 nav 赏罚,nav 赏罚在 loss 上的作用太过轻微，在稳定的 w 面前没什么影响

不过有一点，就是现在的 dataset 很小，所以 vaildate 的时候，很快就降低了碰撞率.

训练 nav 的一个问题是，其要求的 batch size 很大，否则，会发散，虽然说原因未知，但是应该多少是和惩罚率不足导致大波动有关吧?

一个办法是，修改 15 bias 到 0.5，这样就不会发散了. 但是收敛速度还是很慢另一个方案是同时在学习过程中调整 batchsize 和 train rate，初期需要大的 batch 和高的 train rate 加快收敛，后期则是缩小两者，减小 overfit

---

由于当前来看，加入 vloss 后知道，其实还有优化余地，所以之后我们可以尝试下在避撞问题中使用 stereo 网络测试下. 眼下，则是用 raw+nav+ 大 history+ 低 train rate，来找出办法，至少训练出原有的水平吧? 原来至少失败率是 28%，现在都快有 50% 以上了.

或许训练 stereo 后再固定 13 再训练 nav 比较好吗?

一个问题，发现对于大 history 一个 batch 生成需要 16 秒，是因为硬盘速度，还是因为 batch 太大?

这里遇到了一些 bug?

4.13 Reward Rules

如果 agent 在某处停留很久，那么我们就假设它在那里停留到永远，

我们尝试用两种不同的奖励函数，一个是高斯，另一个是距离，

前者的结果不好，

前者在大部分区域的差别都太小，所以基本没有效果.

4.14 Network Architecture

××××××××××××××××××××××

Chapter 5

Reinforcement Learning

5.1 Deep Q Learning

Chapter 6

Other Works in Obstacle Avoidance

RL 是一个 AI 中活跃的研究领域，研究如何让 agent 在与复杂环境交互中获得最大奖励，RL 是研究的问题是，在一个环境中，一个个体只是被告知他的行为可能带来的奖励，但是并不告诉他正确的行为。目标是希望个体能够自行学会最大化奖励的行为方法。

多写些学术的重述，翻译成中文记录下来，再翻译回去.

关于神经网络，关于 RL，关于 BP，关于深度学习，找下你以前写过的一些出来.

神经网络是什么，

神经网络被用作了什么.

什么是 BP，

什么是 CNN

什么是 RL，如何选择参数.

为什么使用 RELU，有哪些激活函数

使用 dropout?

RL 的方程 q value

Chapter 7

Function Merging

7.1 Network Architecture

7.2 Pretrained Models

我们打算使用

Chapter 8

Experiments

Chapter 9

Conclusions

9.1 {Future}

未来的工作? 我们

我们的工作主要集中在模拟环境，模拟环境相对于真实环境的一个优势是，模拟环境中可以在 agent 碰撞后让电脑自动重置系统，但是真是环境中重置可能需要人为的操作，由于需要大量的失败数据来训练，由人为操作来重置 agent 是不可行的. 因此 agent 需要有自己从异常状态中恢复过来的能力.

为了做到这一点，我们可以在模拟环境中通过物理引擎模拟碰撞造成的异常状态，让 agent 从中学习.?

1.3D 环境的训练 2. 从失败中恢复的能力，为向真是环境迁移 3. 切换使用深度算法