DDPG 在强化学习中的应用和改进

目录

1. 问题的定义

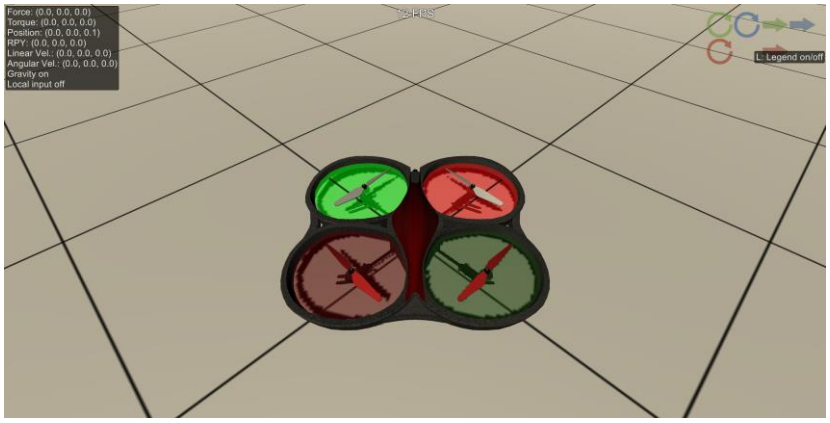
1.1. 项目概述

强化学习是机器学习的一个方向，经典的机器学习有监督学习、无监督学习，深度学习等；强化学习最大的特点是在与环境的交互中学习，完成更加适应该环境的任务。强化学习训练智能体做出在当前环境下最优的动作，通过动作获取奖励或惩罚，不断迭代优化对于动作的选择。强化学习被认为是最接近我们人类学习知识过程的机器学习方法。

强化学习，就是学习如何做出最合适当前环境的动作，以及让动作得到最大的回报。我们称学习的对象为智能体；智能体需要感知环境中的有效信息，下文称为状态；同时做出动作改变状态接近最终的目标。马尔科夫决策过程（MDPs）尝试使用这三个维度，感知、动作、目标来描述强化学习的过程，解决这个学习过程的算法我们都认为是强化学习算法。

1.2. 问题陈述

本文要解决的强化学习问题，是要训练一个四轴飞行器学习一系列复杂的动作。智能体训练的环境是一个三维空间，通过一个三维向量（x, y, z）来表示智能体的位置，以及一个四维方向 Quaternions（qx, qy, qz, qw）来表示智能体的旋转。同时，针对不同的任务，其他的特征也会加入观察空间，比如飞行器的速度(dx, dy, dz)，加速度 linear_acceration (lx, ly, lz)；智能体通过轴向的力（fx, fy, fz）使飞行器延 x, y, z 轴运动，通过旋转轴的力（tx, ty, tz）使用飞行器按 x, y, z 轴旋转。



智能体观察空间

智能体感知的是一个连续空间，飞行器的动作空间也是连续的。这给强化学习带来了一个巨大的难题，如何处理高维度连续空间特征，这个问题有点类似机器学习在视觉和语音处理中的挑战一样。

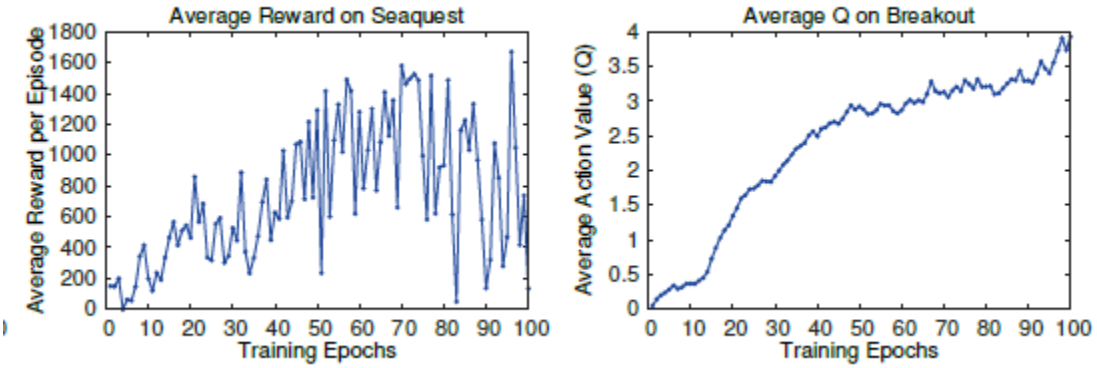
众所周知，深度学习网络已经在计算机视觉和语音处理领域取得巨大的突破。虽然使用深度学习的强化 Q 网络（Deep Q Network），我们可以很好的处理高维度的连续空间状态，但 DQN 只能处理低维度离散动作空间。

深度确定性策略梯度（Deep Deterministic Policy Gradient）算法是 Google 在 2016 年提出的，使用基于策略梯度（Policy Gradient）和深度 Q 网络的组合来解决连续动作空间的问题。DDPG 的最大优点是简单；实现简单，只需要通过扩大学习网络，即可处理更复杂，更大空间的问题。

1.3. 评价指标

深度确定性策略梯度（Deep Deterministic Policy Gradient）是非模型、离线策略、观察评论者，使用深度网络的算法，可以处理高维度、连续动作空间。DDPG 使用一个参数化的行动者函数 $\mu(s|\theta^{\mu})$ 来学习确定策略，即一个确定的状态和动作之间的关系，以及一个评论者使用贝尔曼公式学习值函数$Q(s,a)$；在使用 DQN 时，DDPG 使用一个回放缓存区存储 (s_t, a_t, r_t, s_{t+1})，缓存区的大小是有限的，当缓存区满了后，最早存放的数据会被替代，在每一个时间步行动者和评论者通过一簇缓存区的数据进行更新；行动者和评论者网络会被拷贝构成目标网络$\mu'(s|\theta^{\mu'})$，$Q'(s,a|\theta^{Q'})$ 以小步长滞后更新；这种“软”更新，有助于提高学习的稳定性；在 DDPG 算法中，我们可以把探索任务独立于学习任务，并加入一个噪声过程 N 作为我们的探索策略。

DDPG 使用行动者和评论者分别拟合策略和 Q 值函数，所以我使用以下标准评估模型的表现：“阶段总回报或平均回报曲线”和“阶段 Q 值曲线”。



图片来源 DeepMind Technologies: Playing Atari with Deep Reinforcement Learning

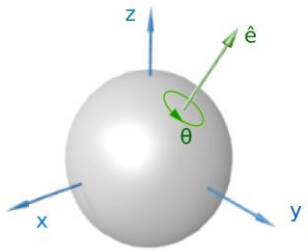
2. 分析

2.1. 数据的探索

智能体的动作空间为 x、y、z 轴向的力和扭力，值域为[-25, 25]；定义动作空间为(Box(6,))。我们在模拟任务中所处的环境是理想环境，没有风力或其他力的作用，轴向的扭力对起飞降落任务没有特别的作用，我们可以固化动作空间扭力为 0；我们的任务也并不需要在 x、y 方向移动智

能体的位置，所以我们也固化动作空间中 x、y 方向的力；另外，在任务中有一个隐含的力，就是重力，在没有任何作用力作用下，飞行器也会下降；当想要使飞行器悬停或上升时，都要作用 z 轴的力来对抗重力。

智能体感知的空间是一个 300*300*300 的立方体，x、y 轴的值域为[-150,150]，z 轴的值域为[0,300]；观察空间有飞行器在立方体中的位置还有姿势，我们通过(x, y, z)的坐标表示位置，并通过一个 4 元祖(x, y, z, w)描述飞行器的姿势，与普通的轴旋转不同的是，4 元祖将基于 3 个轴的旋转简单表示为基于一个欧拉轴的简单旋转，x、y、z、w 的值域为[-1,1]；定义观察空间为(Box(7,))。另外，在力的作用下，智能体还能感知到轴向力和扭力的加速度。



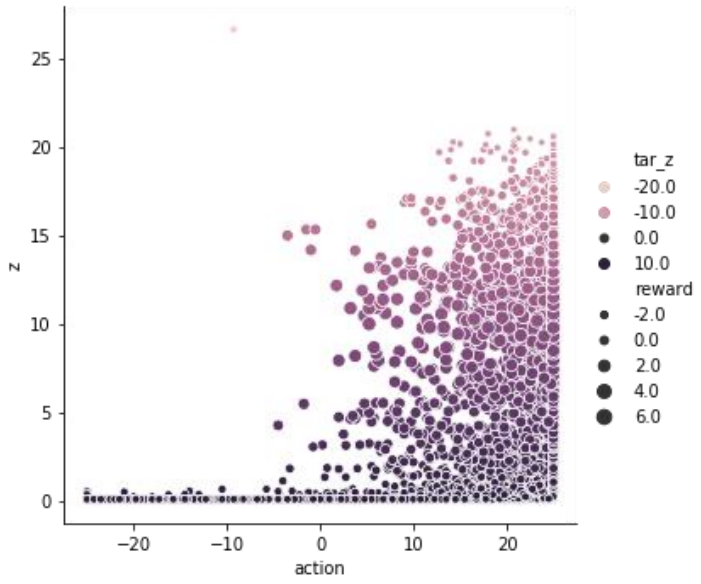
图片来源:
https://en.wikipedia.org/wiki/Euler%27s_rotation_theorem#/media/File:Euler_AxisAngle.png

2.2. 探索性可视化

我们可以将训练的数据储存下来用图表展示，数据中包含位置(x, y, z)，速度、加速度、使用的力、得到的奖励和与目标的差距等。

如下图展示，越大的点表示奖励越多，从图表中可以看到，大点集中在 z 轴 10 附近，所以，我们设置的奖励在该点附近最高，越往上或下，奖励越低；大部分的点都集中在力大于 0 的区域中，可以看到，小于 0 的力对于智能体在该任务中起到的作用较小；很多的大奖励的点出现在 z 轴 10 与力 20 附近，代表力为 20 左右可以使得智能体尽可能久的悬停在 10 的位置。

如果任务是悬停，这是个并不成功的训练，我们可以看到图中有大部分点落在了 10 以上的区域，所以可以推断出，最终智能体是在高于 10 的区域结束训练，而并非悬停在 10 的位置。



2.3. 算法和技术

2.3.1.回放缓存

设计一个列表存储之前的经验，该经验需要包含$(s_t, a_t, r_t, s_{t+1}, \mathrm{d})$，d 表示阶段是否结束；当缓存区满后，最早的缓存空间将被替换；设定经验簇的大小，当缓存区的数据量大于簇大小时，随机取出其中的数据簇作为模型的输入。

随机取出基于这样一种策略：缓存区的所有经验都同样重要，存在于回放的经验间的时序相关性被极大减弱因为经验回放机制把新旧经验混合在了一起来更新网络。

回放缓存有两个参数，缓存区的大小 buffer_size 以及簇的大小 batch_size；缓存区大小决定可以保存多久之前的经验用于更新网络，而簇大小决定使用多少数据来更新神经网络参数。

2.3.2.评论者 Q-Learning

贝尔曼公式描述的值函数优化：

$$Q^*(s,a) = \mathop{\mathrm{E}}_{s' \sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

设计一个深度网络拟合$Q_{\emptyset}(s,a)$，参数∅，输入为经验簇 D $(s_t, a_t, r_t, s_{t+1}, \mathrm{d})$；误差函数为贝尔曼均方误差(MSBE)

$$L(\phi, \mathcal{D}) = \mathop{\mathrm{E}}_{(s,a,r,s',d) \sim \mathcal{D}}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)\max_{a'} Q_\phi(s',a')\right)\right)^2\right]$$

评论者模型的输入为状态 s 和动作 a，输出为状态值函数$Q^*(s,a)$，参数有隐含层的大小 *hidden_layer* 和学习速率 *learning_rate*；

2.3.3.目标网络

在 Q 学习中使用目标网络的概念；我们使用贝尔曼均方误差，将下式作为我们 Q 函数的逼近目标；

$$r + \gamma(1-d)\max_{a'} Q_\phi(s', a')$$

从中我们可以看到问题是，目标是依赖于训练模型参数ø的，这会让贝尔曼均方误差的最小化过程非常不稳定；解决方案是，设计一套与ø相近的训练参数，这套训练参数会被延迟更新，我们称第二套网络为目标网络。

目标网络在用多个经验进行训练后，将新学习的权重复制到目标模型中；通过一个 ρ （0,1）之间的参数调整权值。

参数包含折扣系数 *gamma* 和权重更新系数 *tau(ρ)*控制。

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi,$$

2.3.4.行动者策略学习

我们需要学习确定策略 $\mu_\theta(s)$，最大化值函数$Q_\emptyset(s,a)$，我们可以使用梯度下降得到：

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathrm{E}}\left[Q_\phi(s, \mu_\theta(s))\right].$$

行动者的梯度来自于评论者的 Q 函数。行动者的输入是状态 s，输出是动作 a；参数有隐含层的大小 *hidden_layer* 和学习速率 *learning_rate*；

2.3.5.探索策略

由于 DDPG 使用确定性策略，智能体可能在训练初期就使用确定策略而造成探索不到有用的信息；为了让智能体更好的探索空间，使用时间相关的 OU （Ornstein–Uhlenbeck）噪声，并引入贪婪系数 *epsilon*，使噪声的幅值随着更多阶段完成变小。

$$\mu'(s_t) = \mu\left(s_t\middle|\theta_t^\mu\right) + \mathrm{N};$$

2.3.6.伪代码

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

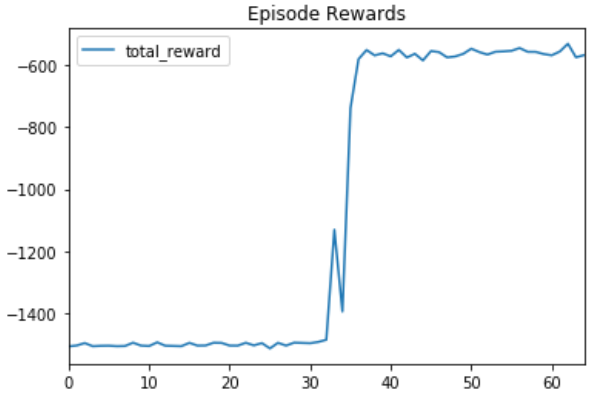
$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

2.4. 基准模型

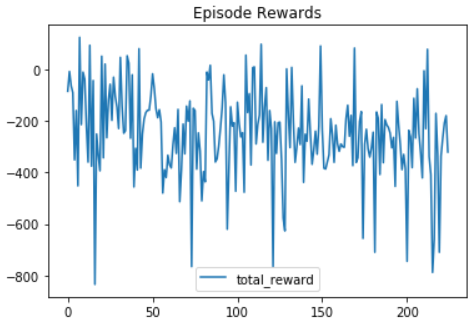
我们选择两种基于值函数的算法作为基准，Monte-Carlo 和 Temporal Difference 来对比 DDPG 在连续空间任务中的表现；随机策略只能在每一个 epoch 结束才更新策略，无法在每个时间步更新动作，所以无法做悬停和 Combined 的任务；TD 算法需要将状态空间和动作空间离散化，在这里我们选择将动作空间都化为整数，且只选择 3 个动作{15,20,25} 的动作空间，作为悬停和 Combined 任务的基准。

2.4.1.起飞任务（*随机策略*）

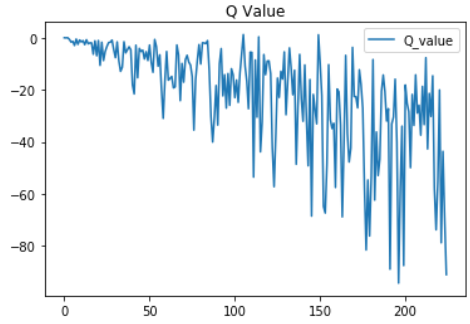


起飞任务阶段总回报

2.4.2.悬停任务（离散 TD）



悬停任务阶段总回报



悬停任务 Q 值

2.4.3.下降任务（*随机策略*）

下降任务阶段总回报

2.4.4.混合任务（离散 TD）

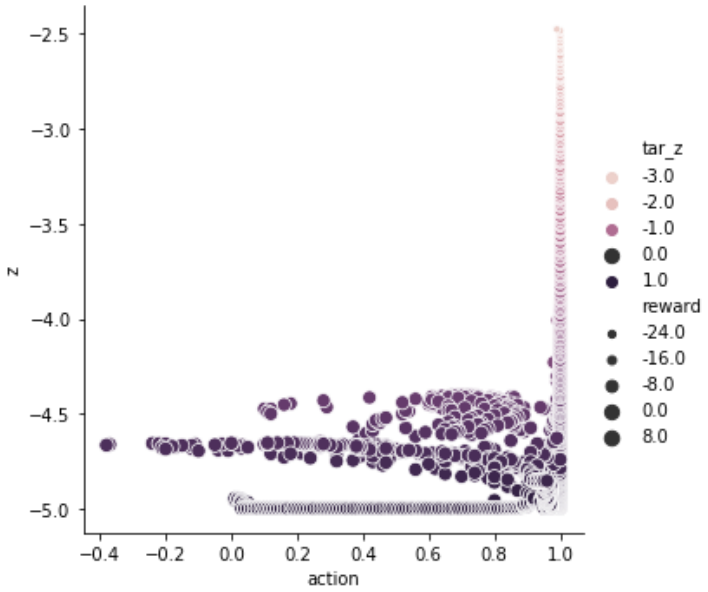
混合任务阶段总回报

混合任务 Q 值

3. 方法

3.1. 数据预处理

数据预处理包括将飞行器位置归一化到[-5,5]，目标位置归一化到[-5,5]，仅处理 z 轴方向的力归一化到[-1,1]，存储上一次的位置及时间戳用来计算速度；如下图可视化数据。

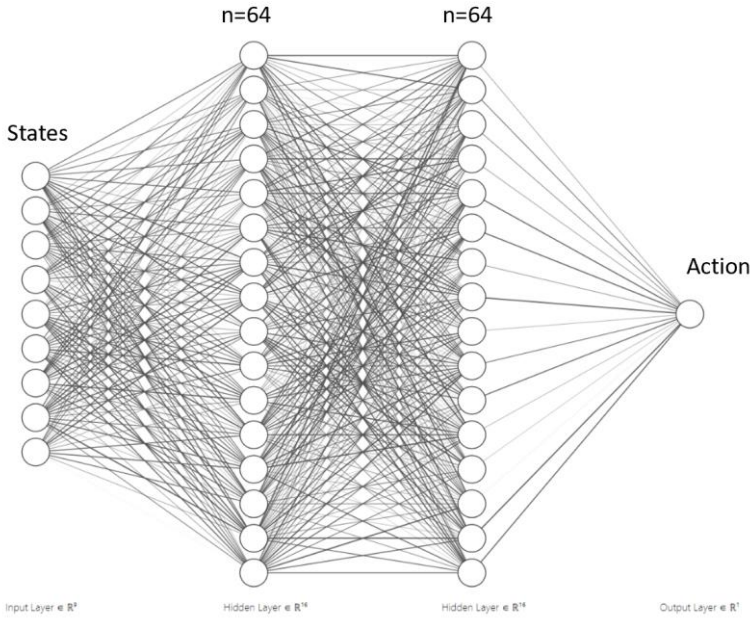


3.2. 执行过程

3.2.1.智能体

- 行动者

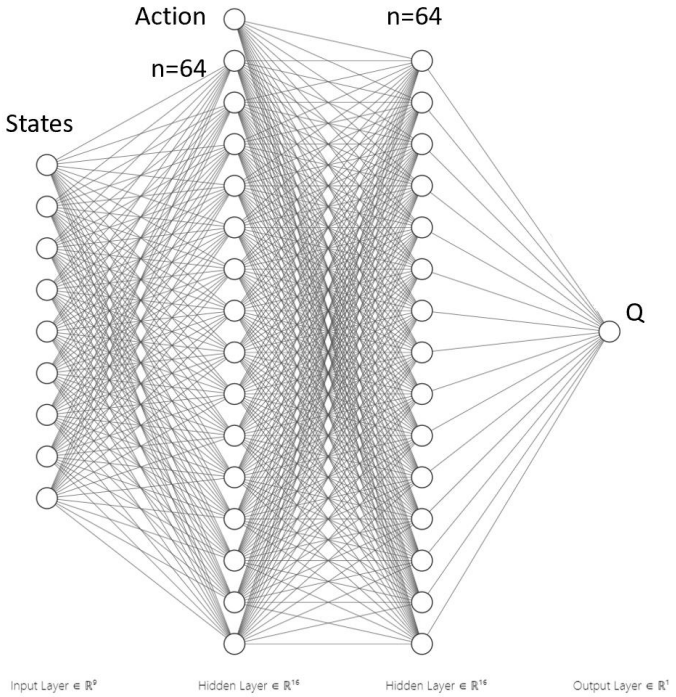
行动者模型如下图所示，包含输入层，2 个隐含层和输出层；其中隐含层使用了大小 64，relu 激活函数；输出层使用了 tanh，直接产生[-1,1]的归一化动作值；输出层使用 Lambda 直接转化为[-25,25]的力，效果不是非常理想，会造成难以学习的问题；使用 tanh 可以很好的解决这个问题，对于归一化的输出值直接乘以 25 即可转化成为智能体实际的力。



图片使用 http://alexlenail.me/NN-SVG/ 绘制

- 评论者

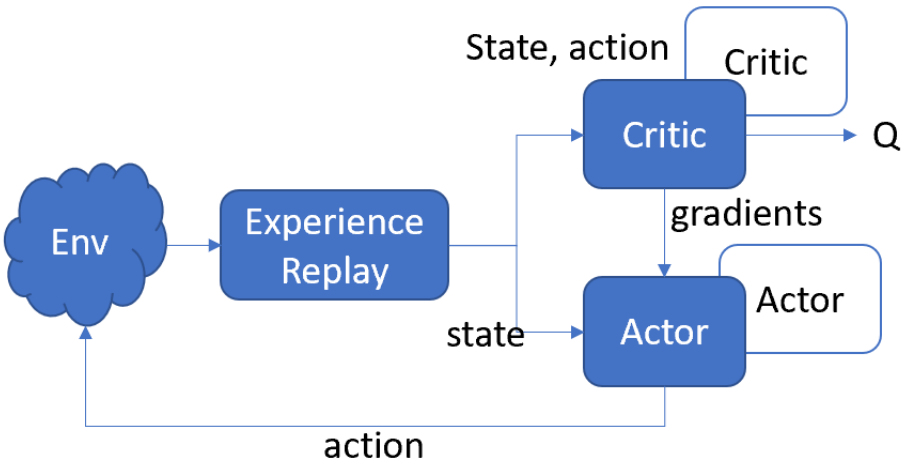
评论者模型如下图所示，包含输入层 States 和 Action 的拼接，2 个隐含层和输出层；其中隐含层使用了大小 64，relu 激活函数；输出层直接拟合 Q 函数。



图片使用 http://alexlenail.me/NN-SVG/ 绘制

- DDPG

DDPG 控制流程如下：环境的数据会存储在经验回放单元内，满足一定数量后会作为行动者和评论者网络的训练输入；行动者模型通过状态输入，直接拟合最优策略，输出动作，在这里就是作用在智能体上的力；同时评论者模型通过状态、动作输入获取最优 Q 值，并提供行动者动作的梯度，作为行动者的标签。



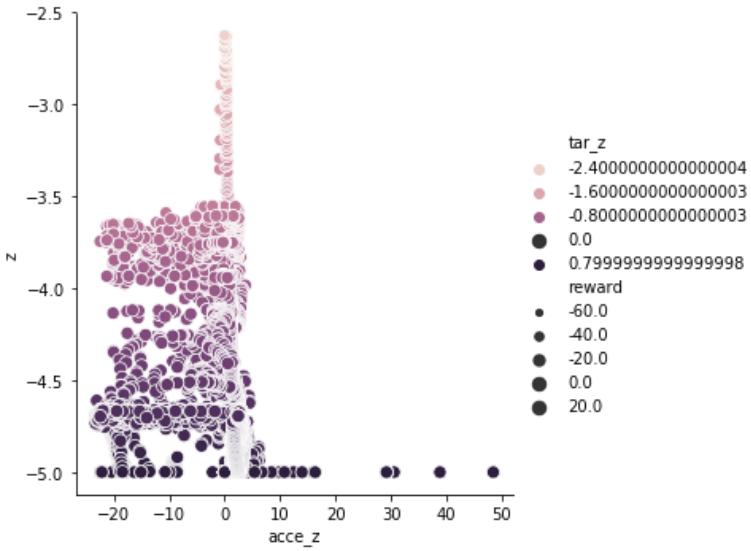
3.2.2.任务

- 悬停任务

悬停任务定义状态为 飞行器的位置(x, y, z)，速度(Vx, Vy, Vz)，距离目标的位置距离(tar_x, tar_y, tar_z)。

奖励距离目标位置最近的点，以及惩罚在最近点速度或加速度大的点；以下图为例，图中点的大小代表奖励的大小，悬停任务的最大奖励是在目标位置（z 归一化后为-4.7 附近），加速度 0 点周围。

从图中不难发现，在目标位置附近，没有加速度大于 0 的点；这就是惩罚的效果。

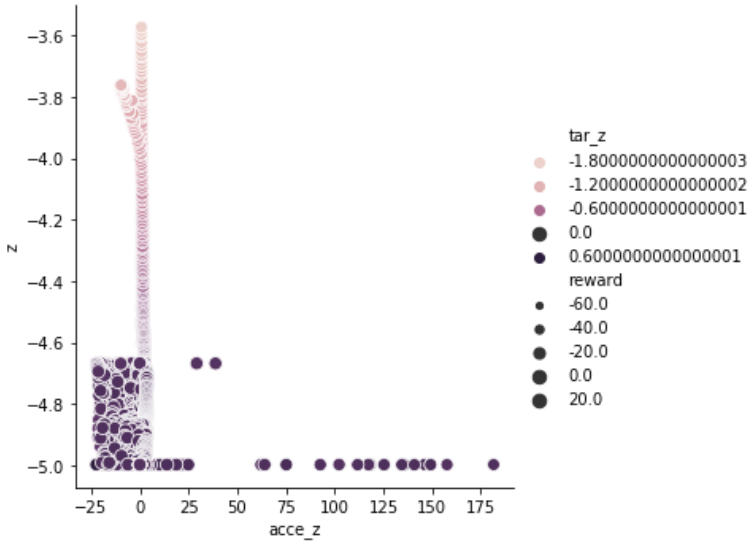


在执行的过程中，一个难点是确认距离奖励和加速度惩罚的系数；如果奖励系数过高，会无法悬停，而在目标高度上下震荡；如果惩罚系数过高，会造成智能体无法达到一定高度，或无法在限定时间内达到指定高度；在这里的策略是以距离奖励为主，加速度惩罚大概差一个数量级左右。

- 降落任务

降落任务定义状态为 飞行器的位置(x, y, z)，速度(Vx, Vy, Vz)，距离目标的位置距离(tar_x, tar_y, tar_z)，使用与悬停任务相同的智能体，初始位置为高度 10（归一化值-4.7）。

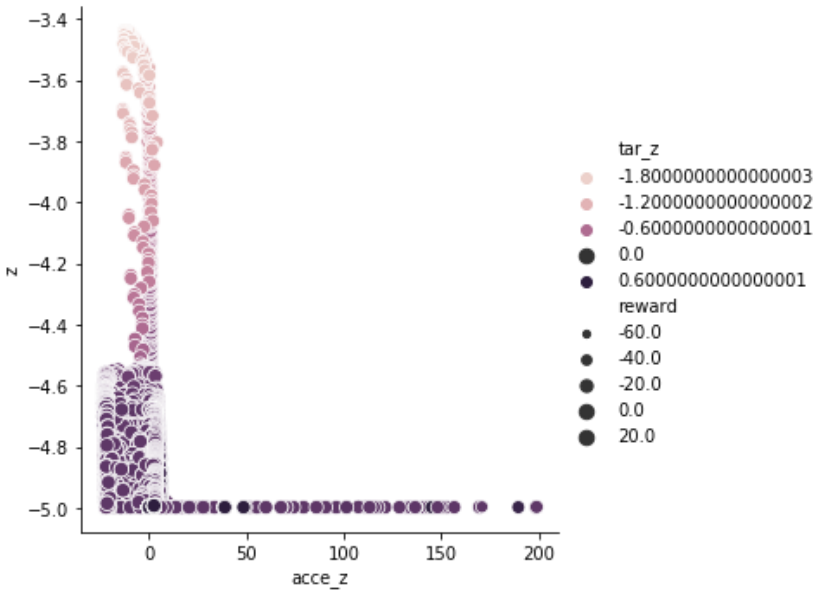
降落任务的目标是让智能体缓慢的下降落地，我们在悬停任务的基础上，让 tar_z 随时间不断减小来实现降落的任务。下图很明显的表现了降落应有的数据，在小于目标高度的范围里，都是向下的速度（下降），且奖励值高于大于目标高度的范围。在加速度为正的区域，我们可以观察到，这些点都在地面高度，代表智能体从高处落下并重新弹起来获取奖励。



- 混合任务

混合任务定义状态为 飞行器的位置(x, y, z)，速度(Vx, Vy, Vz)，距离目标的位置距离(tar_x, tar_y, tar_z)，使用与悬停和降落任务相同的智能体。

混合任务的目标是让智能体先在指定位置悬停然后缓慢的下降落地，我们需要设置时间轴在悬停任务的基础上，随后让 tar_z 随时间不断减小来实现混合任务。



3.3. 完善

3.3.1.对稳定性的改进

在飞行器空间中一个难点是重力，除降落任务外，其他的任务都是从平地起飞；这时需要以一个 z 方向 0.8 的力才能使飞行器开始上升；在执行中遇到的第一个问题就是不稳定，飞行器经常训练以-1 收敛，原因是当飞行器尝试增加力时，没有得到奖励，就会尝试减小力。

```
# Learn, if enough samples are available in memory
if len(self.memory) > self.batch_size:
    experiences = self.memory.sample(self.batch_size)
    #print('experience:', experiences)
    self.learn(experiences)
else:
    action = np.array([0.8]).reshape(1,-1)
```

在本文 DDPG 的实现中，在收集到足够的数据支撑训练之前，会加载一个 0.8 的力让智能体开始探索空间，并收集克服重力的经验用于训练；这个实现能明显改善智能体无法收敛的结果，增强稳定性。

3.3.2.加入贪婪系数

贪婪系数的加入有助于帮助模型更快的收敛，在这里设置每 30 个 epoch 降低一次贪婪系数;

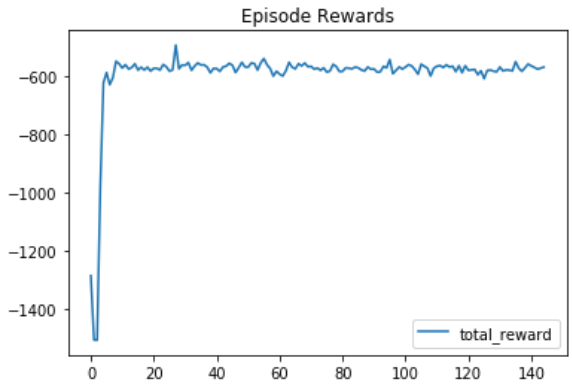
```
  noise_epsilon = self.epsilon / ( int(self.episode_num / 30 ) + 1)
```

4. 结果

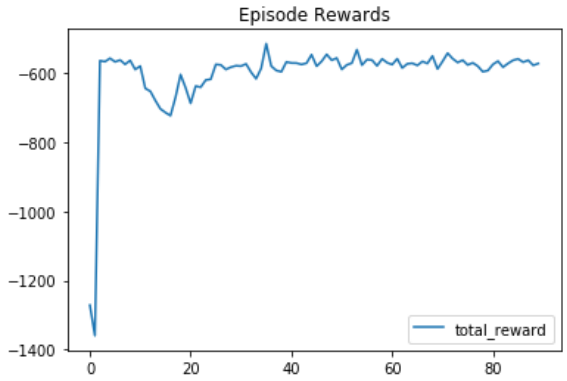
4.1. 模型的评价与验证

- 模型参数

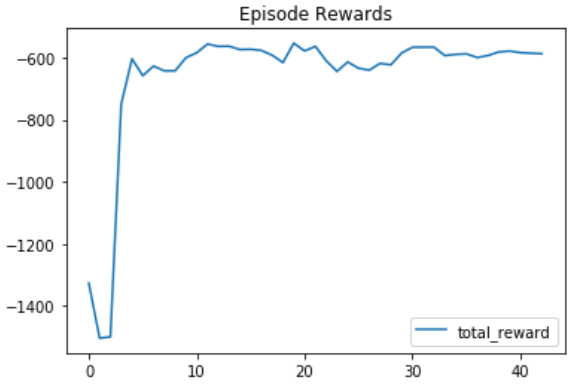
Actor_lr = 0.0001, batch = 128, hidden layer=64, gamma=0.99，模型没有太大震荡，稳定可复现。



- Learning rate Actor_lr=0.0001->0.0005 学习速率过大，会造成结果的震荡；降低学习速率，能够使奖励曲线更平滑。

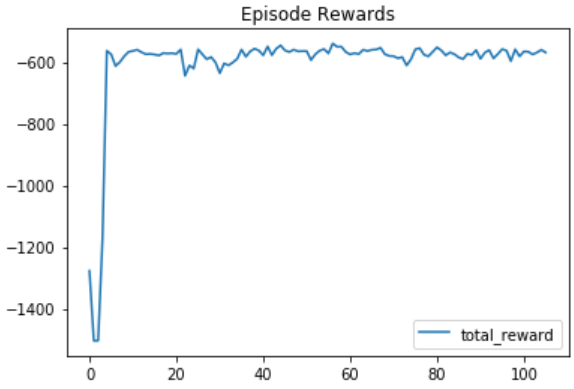


- Batch size = 128->32，对任务没有太大影响。



- 神经网络大小 hidden layer 64->32，过小的网络会导致学习问题。

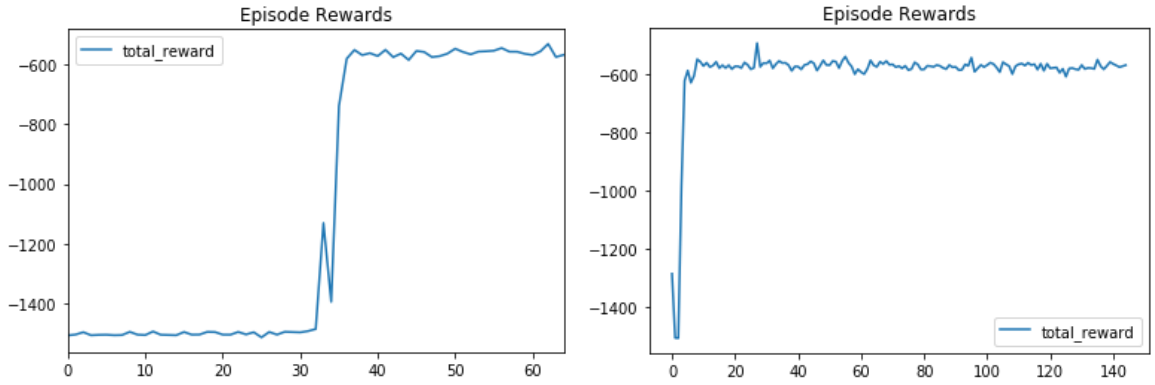
- 折扣因子 gamma 0.99->0.5，会造成一些震荡。



4.2. 合理性分析

- 起飞任务

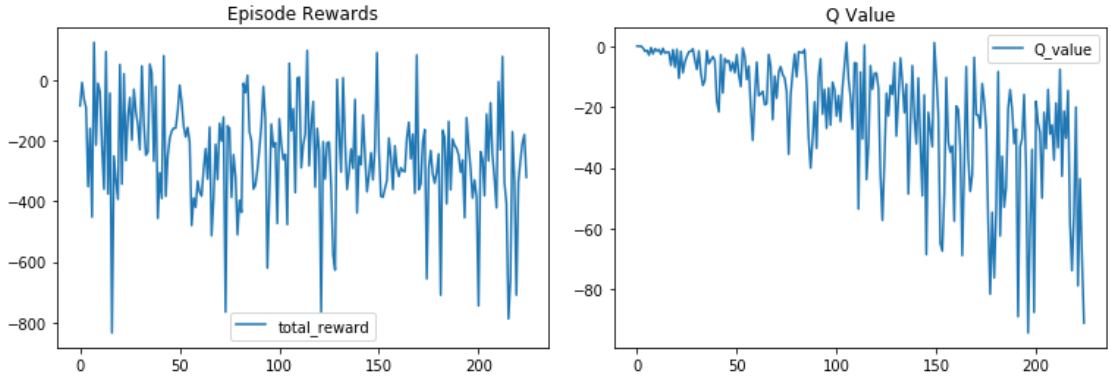
起飞任务是任务中相对简单的；随机策略（左）能够很好的完成起飞的任务，只是需要的时间会比较长，且随机性很大；DDPG（右）基本上在很早的时间步就已经可以起飞了，而且奖励平稳。



- 悬停任务

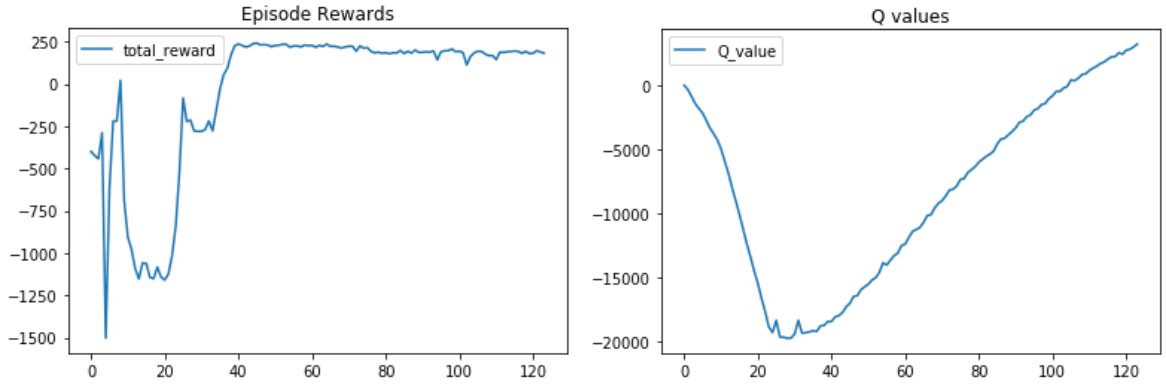
TD 算法在悬停任务上，效果不好，原因是即便空间离散化了，需要存储的 Q 值键值对数量可观，而且随着输入状态维度的提高，成指数上升，需要训练的时间会明显增加，从结果上看是不收敛的。

TD:

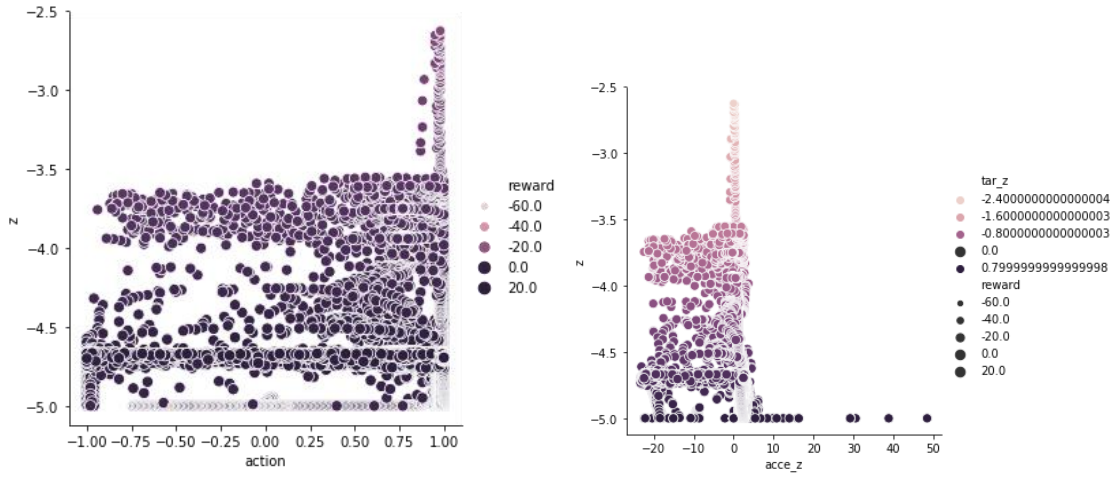


DDPG 可以利用大型神经网络，很完美的解决这个问题；我们可以获得稳定的最佳奖励，线性度较好。而且，该模型也早于 TD 算法收敛到最佳策略。

DDPG:



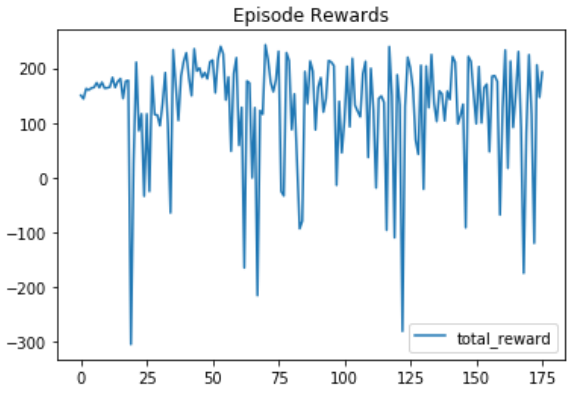
我们可以明显看到在目标高度-4.7，有一个由点组成的平台形成，聚集了很多高奖励值的点。而且动作值均匀分布，这是悬停的特征，因为需要互相中和向上和下的力。而且除了起飞时，其他时间几乎都没有大的向上的加速度产生，整个过程是平稳上升并悬停的。



- 降落任务

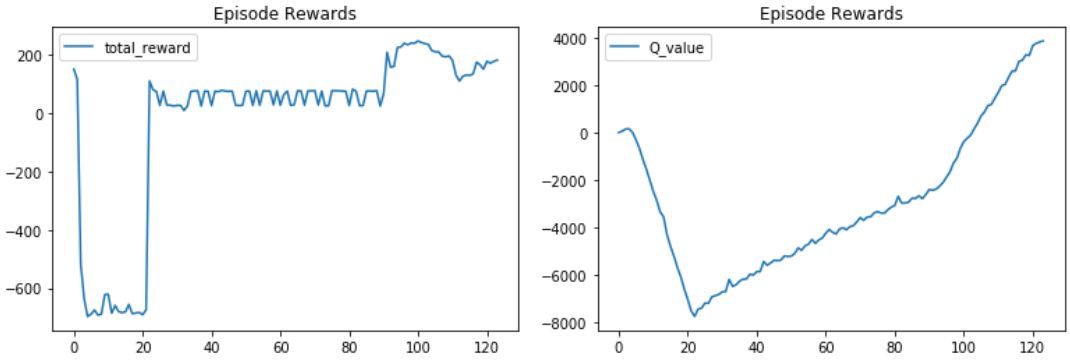
随机策略基本上能够完成任务，让人非常吃惊，原以为随机策略因为每次迭代只能选择一个力，所以无法完成降落任务；个人理解是由于重力的原因，与作用力产生了多个力，协作完成了降落任务。但即便可以有一定效果，从整体上看，仍旧是没有收敛。

随机策略:

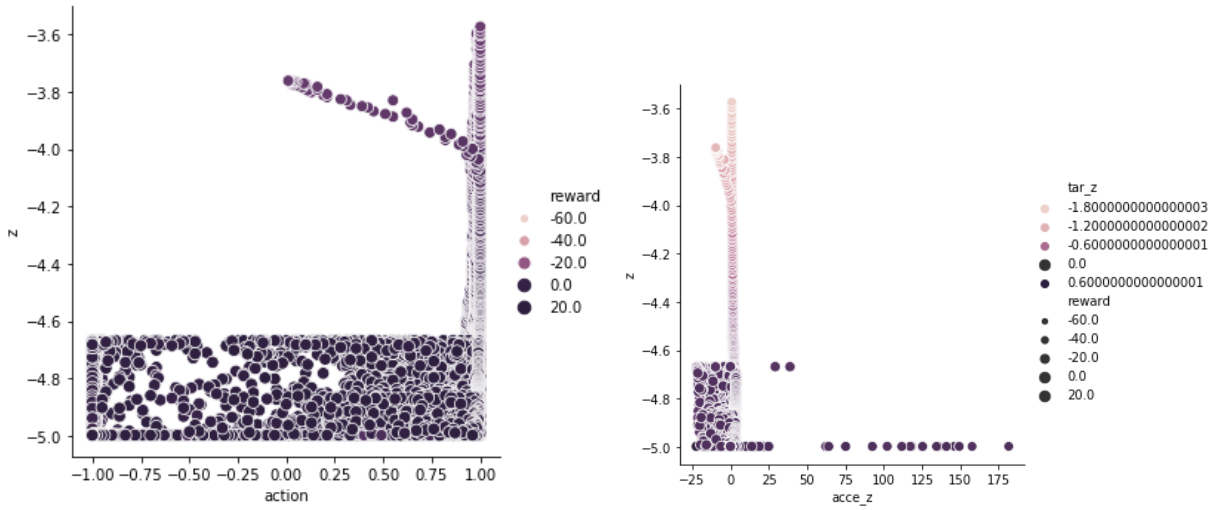


DDPG 能够比较完美的收敛，但由于重力的作用，有一些震荡，如果想要改善可以考虑降低目标高度变化的速度，让智能体有一个反应调整的时间。

DDPG:



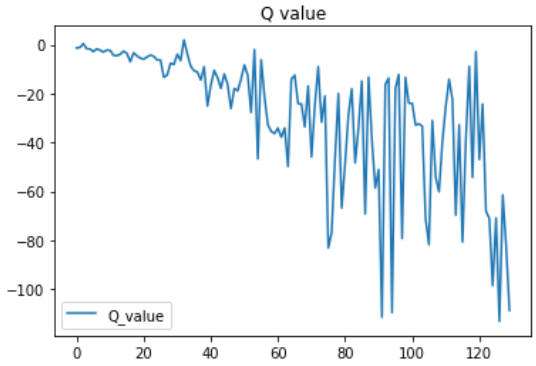
由于目标高度下降，所以在目标高度以下的高度都有类似悬停的效果；图中目标高度以上的这部分数据是智能体尝试降低高度的表现，代表模型已经学习到了目标高度的信息，但智能体仍出现大量从地面弹起的现象，可以考虑惩罚向上加速度来优化。



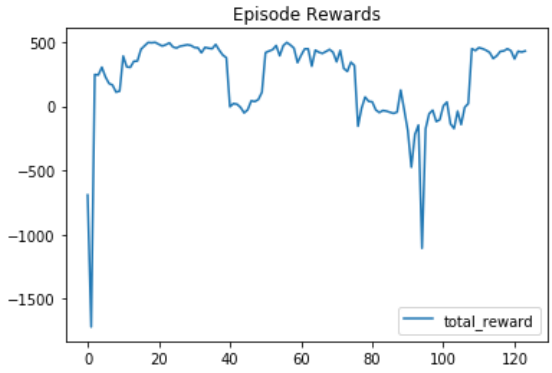
- 综合任务

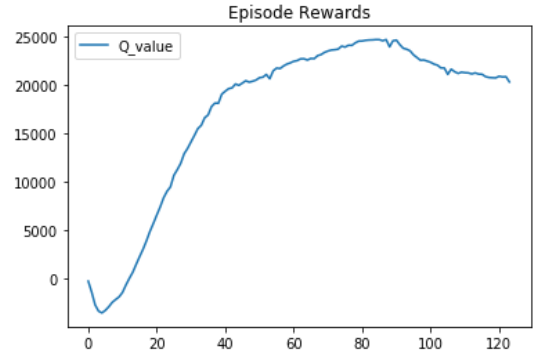
与悬停任务类似的，由于空间、动作离散化带来的复杂程度，让模型很难收敛；DDPG 能够较早收敛，但存在一些震荡，与降落一样，可以通过减慢目标高度变化的速度来优化。

TD:

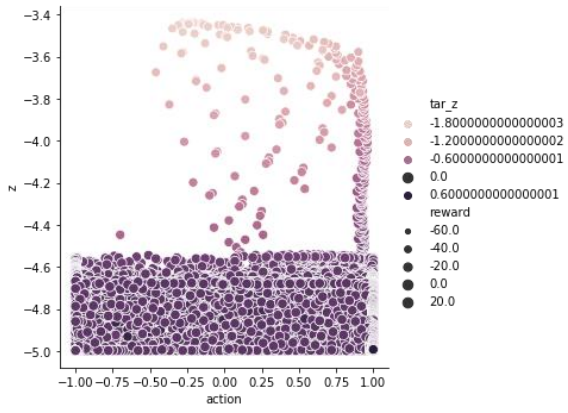


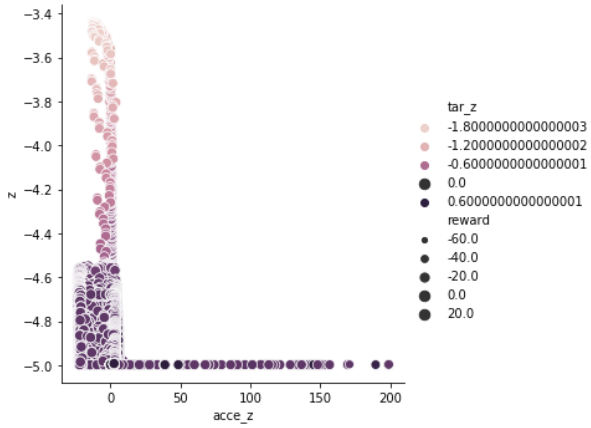
DDPG:





也可以从可视化图表中看出综合任务结合了前两个任务的特点。整个目标高度下的动作分布均匀，代表整个过程比较平稳；从加速度图表中也可以看出。



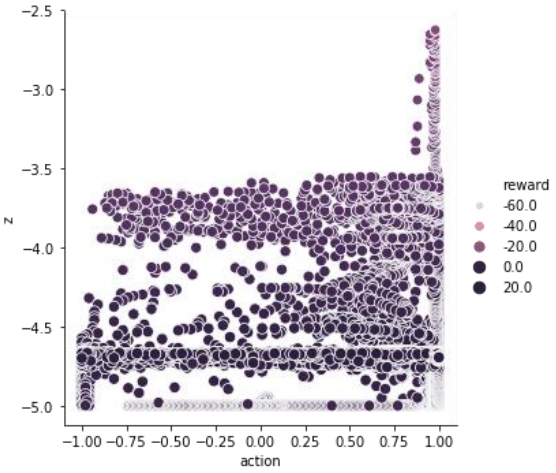


5. 项目结论

DDPG 算法相比随机策略或传统的 TD 算法，充分利用了深度神经网络处理连续空间数据，并使用评论-观察者模型解决了连续动作空间的问题；算法简单，易训练，能够比传统随机策略和 TD 算法更早收敛。

5.1. 结果可视化

在 DDPG 的交互中，我们能够比较直观的可视化智能体的学习过程，这也是体现 DDPG 算法简单易用的很重要的一点。从我们的图中可以看出智能体在 120 个时间步内，充分探索了状态空间和有效的动作空间；通过 DDPG，智能体能够更好的理解环境的特性；在目标高度-4.7 能够收敛，而且动作值分布均匀；在动作 1 上，探索了更高的高度的可能性；在>-4 的高度尝试下降等。这些数据，能够直观的反应 DDPG 的学习过程。



5.2. 对项目的思考

总体来说，这次的尝试是降低了难度实现的；因为在项目过程中，限制了其他维度的力，只保留了 z 方向的力；在项目的开始阶段，尝试了实现更多维度的力，难度很大。在代码实现过程中，收获很多，DDPG 的模型里，即用到了深度神经网络，又用到了值函数的逼近；对强化学习和深度学习了做了一次很好的复习，对知识有了更清晰的理解；代码实现的时候参考了课程中给的代码，这段代码对理解模型和原理有很大的帮助。也参考这个模型，尝试自己写了离散化状态空间和动作空间的 TD 算法，对 MC、TD 和 DDPG 的差异有了非常清楚的认识。

在项目中遇到的两个大的问题，阻碍了很长的时间；第一个问题是模型一直无法收敛，不是收敛于动作值 1 就是-1，很头疼，试了很多方法，改目标值、改参数，发现都没有关系；然后因为我使用了虚机里的 miniconda，发现里面的 tf 和 keras 版本很低，觉得可能是这个的原因，后来调试后发现也不是；最后发现是必须在每次跑模型之前都要运行 catkin_make 和 source bash，也是在重新安装虚机和库的时候，在向导中的一段看到了这句话。第二个问题是模型不稳定，很容易就收敛到-1，原因就是因为在起飞的时候正向的力没有得到奖励，所以就会往负向收敛；期初我的改进是在 batch 没有满足训练条件前给一个 1 的力，这样探索空间非常大，发现这样对有些

任务有比较大的影响，比如悬停或综合任务；后来修改成为 0.8，刚刚满足智能体能够起飞，又不会在太大的空间积累错误的经验。

最终的模型和结果，和我的预期还是符合的；在附带的视频里可以看到智能体的表现比较稳定，虽然在有一些图表中有一些震荡，但实际训练的效果还是很好的；当然，如果在真实的环境中会有风力的影响会让环境有比较大的改变，当前的这个模型还不足以范化到通用场景；

5.3. 需要作出的改进

接下来的改进，我希望可以尝试更多的状态空间；比如可以移动到场景中的石头上，需要拓展 x、y 方向的力来训练；或者模拟一些不规则的风力，训练智能体可以自动调整平滑回到水平；

在算法上，DDPG 还有一些改进算法，比如 Twin Delayed DDPG(TD3)，其中提到有时 DDPG 评论者模型会过度估计 Q 值而使策略无法收敛，有点类似这个项目中遇到收敛到-1 的情况；TD3 可以更慢的延迟更新策略，让策略更平滑稳定。现在使用的人为的加一些经验有点类似人工智能，加了一些人类的经验，并不是最佳的解决方案。

参考文献

1. Richard S. Sutton and Andrew G. Barto (2014, 2015, 2016, 2017, 2018) Reinforcement Learning: An Introduction

2. https://spinningup.openai.com/en/latest/algorithms/ddpg.html#background Deep : Deterministic Policy Gradient

3. https://zhuanlan.zhihu.com/p/25239682 清淞 深度强化学习（Deep Reinforcement Learning）入门: RL base & DQN-DDPG-A3C introduction

4. Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Martin Riedmiller @DeepMind Technologies: Playing Atari with Deep Reinforcement Learning

5. Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver & Daan Wierstra @Google Deepmind: CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING

6. https://zhuanlan.zhihu.com/p/60162556 猴哥wing 论文解读之一种基于优先级经验回放的DDPG算法

7. https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation : Quaternions and spatial rotation

8. https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process : Ornstein–Uhlenbeck process