机器学习工程师纳米学位

杜 利强

Version v0.1, 01.04.2019

Table of Contents

1. 项目背景

近年来，**深度学习（Deep Leaning，DL）**
得到了快速的发展。在2015年底，微软亚洲研究院的研究员提出了一个新的基于CNN的深度模型，称之为残差网络，该残差网络深度高达152层，取得了当时图像识别比赛上面最好的成绩。
在2016年初，DeeopMind基于深度强化学习技术开发的AlphaGo打败了围棋世界冠军李世石。深度学习的成功主要归功于三大因素：**大数据、大模型和大计算**。
现阶段可以利用的数据特别是标注数据非常多，使得我们能够从数据中学到以前没法学习的东西。另外，技术上的发展使得训练大模型成为可能，如上千层的深度神经网络。最后，可获取的计算资源越来越丰富，从CPU到GPU到FPGA，这属于大计算。

以上提及的三大要素成就了深度学习，但同时也为深度学习的进一步发展和普带来了一些制约因素。
首先，标注数据代价昂贵，从无标注的数据里学习成为深度学习的一个前沿，已经有这方面的研究工作，包括生成式对抗网络和对偶学习。
二是如何得到一些比较小的模型使得深度学习技术能够在移动设备和各种场所里面得到更广法的应用。
三是如何设计更快更高效的深度学习算法。四是如何把数据和知识结合起来。五是如何把深度学习的成功从一些静态的任务扩展到复杂的动态决策任务上去。

前述提到的第五点，深度强化学习（Deep Reinforcement Learning, DRL）就是针对其做的一个很好的尝试。
2015年，DeepMind的Volodymyr Mnih等研究员在《自然》杂志上发表论文 *Human-level control through deep reinforcement learning*， 该论文提出了一个结合深度学习（DL）技术和强化学习（RL）思想的模型Deep Q-Network(DQN)，在Atari游戏平台上展示出超越人类水平的表现。
自此以后，结合DL与RL的深度强化学习（Deep Reinforcement Learning, DRL）迅速成为人工智能界的焦点。

强化学习（Reinforcement Learning， RL）是与决策执行（decision making）和电机控制（motor control）有关的机器学习的子领域。它研究agent如何在复杂、不确定的环境中学习如何实现目标。

强化学习能做什么以及表现：

- **RL非常通用，涵盖了涉及做出一系列决策的所有问题。**
  例如，控制机器人的电机，使其能够跑和跳，做出商业决策，如定价和库存管理，或玩视频游戏和棋牌游戏。RL甚至可以应用于具有顺序或结构化输出的监督学习问题。
- **RL算法已经开始在许多困难环境中取得良好效果。**
  正如前面提到的，深度学习和强化学习的结合使DRL成为了人工智能的焦点，有很多亮眼表现：在视频游戏（Dota）、棋类游戏（围棋）上打败人类顶尖高手；控制复杂的机械进行操作；调配网络资源；为数据中心大幅节能；甚至对机器学习算法自动调参。

然而，RL的研究也存在制约因素：

- **需要建立更好的基准。**
  在监督学习中，ImageNet这样的大型标记数据集驱动了其发展。在RL中，最接近的等价物是大量且多样化的环境集合。
  但是，RL环境的现有开源集合没有足够的多样性，并且它们通常难以设置和使用。在这方面，OpenAI做出了突出的贡献。
- **使用的环境缺乏标准化。**
  问题定义中的细微差别，例如奖励函数或动作集，可以极大地改变任务的难度。这个问题导致DRL的可复现性危机。

调查发现，RL已经在各个领域被广泛使用：

1. 控制领域。是RL思想的发源地之一，也是RL技术应用最成熟的领域。

2. 自动驾驶领域。从80年代的ALVINN、TORCS到如今的CARLA。
3. NLP领域。对话系统、机器写作等。
4. 推荐系统与检索系统领域。
5. 金融领域。
6. 进行数据选择。
7. 通讯、生产调度、规划和资源访问控制等运筹领域。

四轴飞行器在个人和专业应用领域都变得越来越热门。它易于操控，并广泛应用于各个领域，从最后一公里投递到电影摄影，从杂技表演到搜救，无所不包。

四轴飞行器控制是强化学习在控制领域的一个典型应用，在该项目中需要设计一个深度强化学习智能体，来控制几个四轴飞行器的飞行任务，包括起飞、悬停和着陆。相比而言，DL中学习器（learner）都是学会怎么做，而在RL中，智能体需要在与环境不断互动和探索的过程中根据奖励或惩罚选择动作不断学习，这十分有趣，并且具有挑战性。

2. 问题描述

项目的目的为训练一个四轴飞行器学会如何飞行。具体为设计一个深度强化学习智能体，来控制几个四轴飞行器的飞行任务，包括起飞、悬停和着陆。在正式开始之前，需要安装ROS以及下载对应的模拟器，然后运行ROS和模拟器并确保两者能够正常连接，详细的步骤参见 README [https://github.com/udacity/RL-Quadcopter]。
这一切就绪后，最后所要做的就是为每个任务正确定义 **Task** 和 **Agent** ，共涉及四个任务:

- 任务1： 起飞

  训练一个智能体成功地从地面起飞且到达某个阈值高度（如地上10个单位）。

- 任务2： 悬停

训练智能体使其起飞且悬停在设定的高度一段时间。

- 任务3： 着陆

  训练智能体学会从地上某个位置温和地着陆。

- 任务4： 全套动作

  实现一个端到端的任务，即起飞，原地悬停某个时长，然后着陆。

3. 数据集和输入

在这里，整个系统包括ROS，环境和智能体。环境和动作都是连续的，所以都被定义为 Box [https://github.com/openai/gym/blob/master/gym/spaces/box.py] 空间。环境为一个 `cube, cube_size x cube_size x cube_size`， 每个状态表示为一个7维向量 `<position_x, ..._y, ..._z, orientation_x, ..._y, ..._z, ..._w>`，
前三个分量表示位置（pose），后四个分量组成一个四元数表示方向（orientation)。每个动作表示为一个6维向量 `<force_x, ..._y, ..._z, torque_x, ..._y, ..._z>`，
前三个分量组成力（force）的表示，后三个分量组成扭矩（torque）的表示。

针对不同的任务，状态空间和动作空间可以进行一定的优化。比如，在起飞任务中，我们可以不关注飞行器的方向，只需知道飞行器的位置（**状态向量的前3个分量**）。同样，只需要应用线性推力，不需要应用扭矩（**这样还会防止出现不必要的旋转和歪斜**）。在这个任务中，我们需要飞行器离地且达到某个高度，奖励可以根据飞行器当前高度与目标高度的距离来度量，当其达到或超过目标高度时进行额外奖励，如果在设定时间都没有完成，则进行额外惩罚。

此外，每种任务下飞行器的起始状态不同。比如在起飞（Takeoff）任务中，飞行器一开始处于某个随机高度（如可以采样自均值为0.5，标准差为0.1的正态分布）。而在着陆（Landing）任务中，飞行器一开始处于离地某个高度（如离地10个单位），并且具有一定的线性推力。

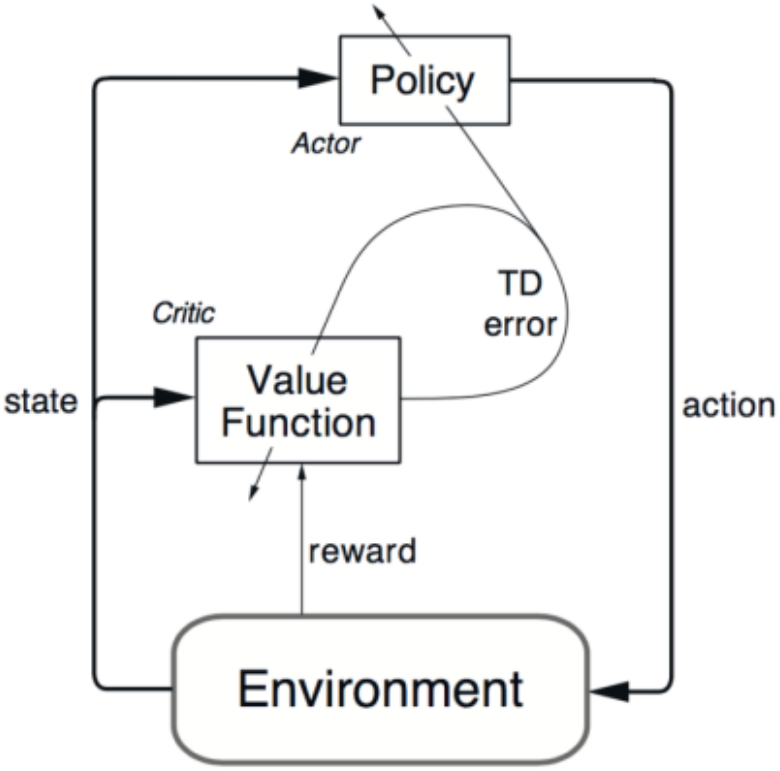
4. 解决方案描述

项目涉及的环境和状态都是连续空间，这意味着只能选择适合连续状态和动作空间的算法或者将连续空间离散化。DQN的适用范围是低维、离散动作空间，在连续空间不能适用：

1. 如果采用把连续动作空间离散化，动作空间则会过大，极难收敛。
2. 即使有些DQN的变种如VAE能提供连续动作的方案，但DQN只能给出一个确定性的动作，无法给出概率值。

从另一个角度看，DQN是值基于的方法，最终还是求解策略。何不一开始就求解策略，这就是策略梯度（Policy Gradient）方法。在策略梯度方法中，设定参数化策略，然后计算得到动作上的策略梯度，沿着梯度方向，逐步调整动作，逐渐得到最优策略。策略梯度方法包括随机策略梯度（SPG）和确定性策略梯度（DPG）。

DDPG [https://arxiv.org/pdf/1509.02971.pdf] 是结合了DQN和DPG，把DRL推向了 **连续动作空间控制**。
DDPG实际为一种行动者-评价者（actor-critic）方法，其架构图如下所示：



Actor网络的输入是state，输出是action，使用DNN进行函数拟合，NN输出层使用tanh或sigmoid激励。Critic网络的输入是state和action，输出为Q值。此外，DDPG中借鉴了DQN中的经验回放和target网络，并且Actor和Critic的target网络均以小步长滞后更新，目的是让模型训练的更稳定。最后，通过在action的基础上增加随机噪声让智能体有机会探索环境。

在该项目中，选择实现DDPG方法以完成飞行器的控制任务。

5. 基准模型

这里选择线性策略作为基准模型，设置如下：

1. 随机初始化策略参数

```
# Policy parameters
self.w = np.random.normal(
    size=(self.state_size, self.action_size),  # weights for simple
linear policy: state_space x action_space
    scale=(self.action_range / (2 * self.state_size)).reshape(1, -1))  #
start producing actions in a decent range
```

2. 使用线性策略选择动作

```
def act(self, state):
   # Choose action based on given state and policy
   action = np.dot(state, self.w)  # simple linear policy
   return action
```

3. 更新策略参数

```
self.w = self.w + self.noise_scale * np.random.normal(size=self.w.shape)
# equal noise in all directions
```

基准模型的存在是为了衡量要实现的DDPG方法的控制效果，这可以通过下个部分指定的评估指标进行对比。

6. 评估指标

在对强化学习解决方案进行迭代时，需要了解智能体的表现，这就需要建立评估指标。在该项目中，一个简单有效的评价指标是智能体在每个阶段获取的总奖励或最近若干个（比如10个）阶段的平均奖励变化，这可以通过保存阶段统计信息到文件，绘制episode rewards进行对比。

7. 项目设计

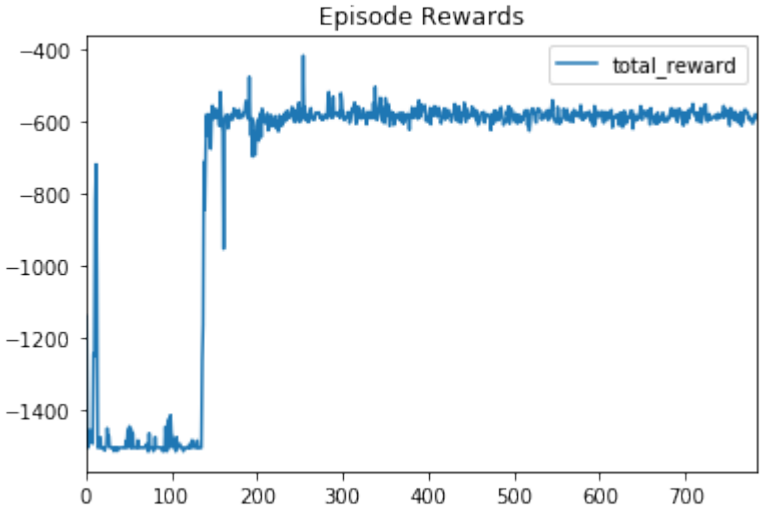
在解决方案描述部分已经确定了要使用的算法为DDPG。整个流程如下：

1. 定义任务（从 `BaseTask` 继承定义具体任务类）
    - 在 `__init__()` 构造函数中完成观察空间（observation space)和动作空间（action space）的定义，设置一些任务相关的参数，如目标高度，最大用时等。
    - 实现 `reset()` 方法和 `update()` 方法，前者在一开始或者阶段结束时调用，用于重置飞行器到起始状态。后者根据当前的状态对智能体进行奖惩，判断阶段是否结束，最后返回一个动作。
2. 定义智能体（从 `BaseAgent` 继承定义具体Agent类）
    - 在 `__init__()` 构造函数中：
    - 定义智能体的任务，状态和空间大小；
    - 定义Actor和Critic网络（包括本地和target）并初始化；
    - 设置经验回放和噪声过程；
    - 设置其他参数，如gamma（折扣率），tau（软更新因子）以及回报统计保存文件名等。
    - 实现 `step()` 方法。在该方法中，我们根据当前的状态s，回报以及完成标签选择下一个动作，使用经验回放技术训练模型以及保存回报统计信息。
3. 运行任务
    - 运行ROS和模拟器
    - 录屏最终的训练效果
4. 绘制阶段回报并进行性能分析

在这部分，我们绘制阶段回报信息：

```
import pandas as pd

df_stats = pd.read_csv(<csv_filename>)
df_stats[['total_reward']].plot(title="Episode Rewards")
```



7.1. Takeoff 任务

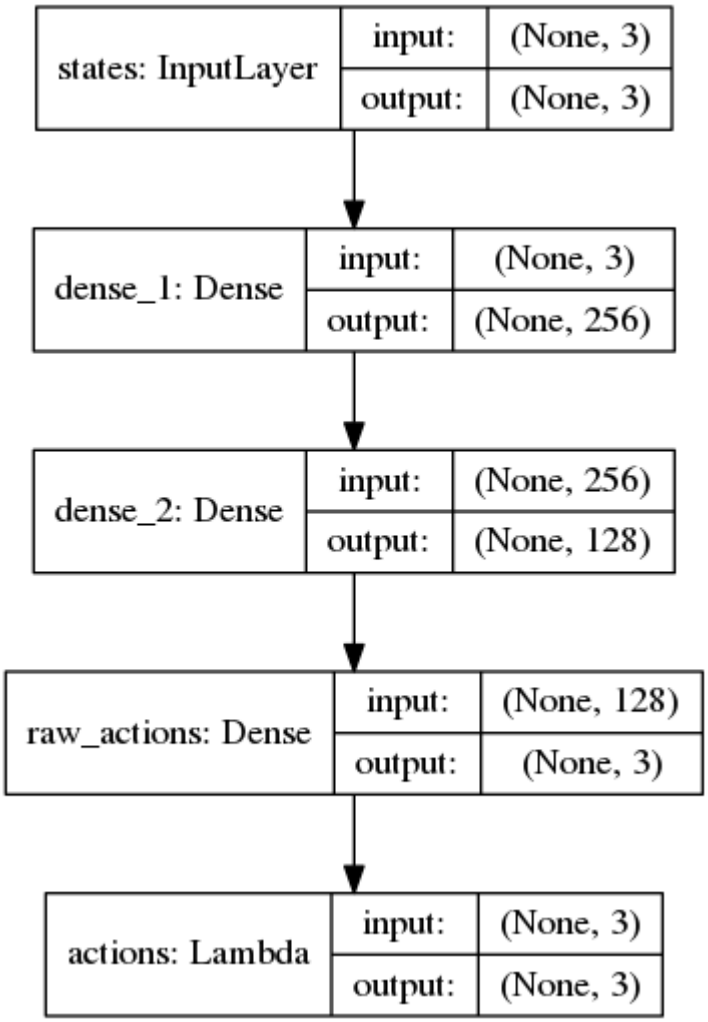
目标：训练智能体成功地从地面起飞，然后达到某个高度。

Episode Rewards Plot:

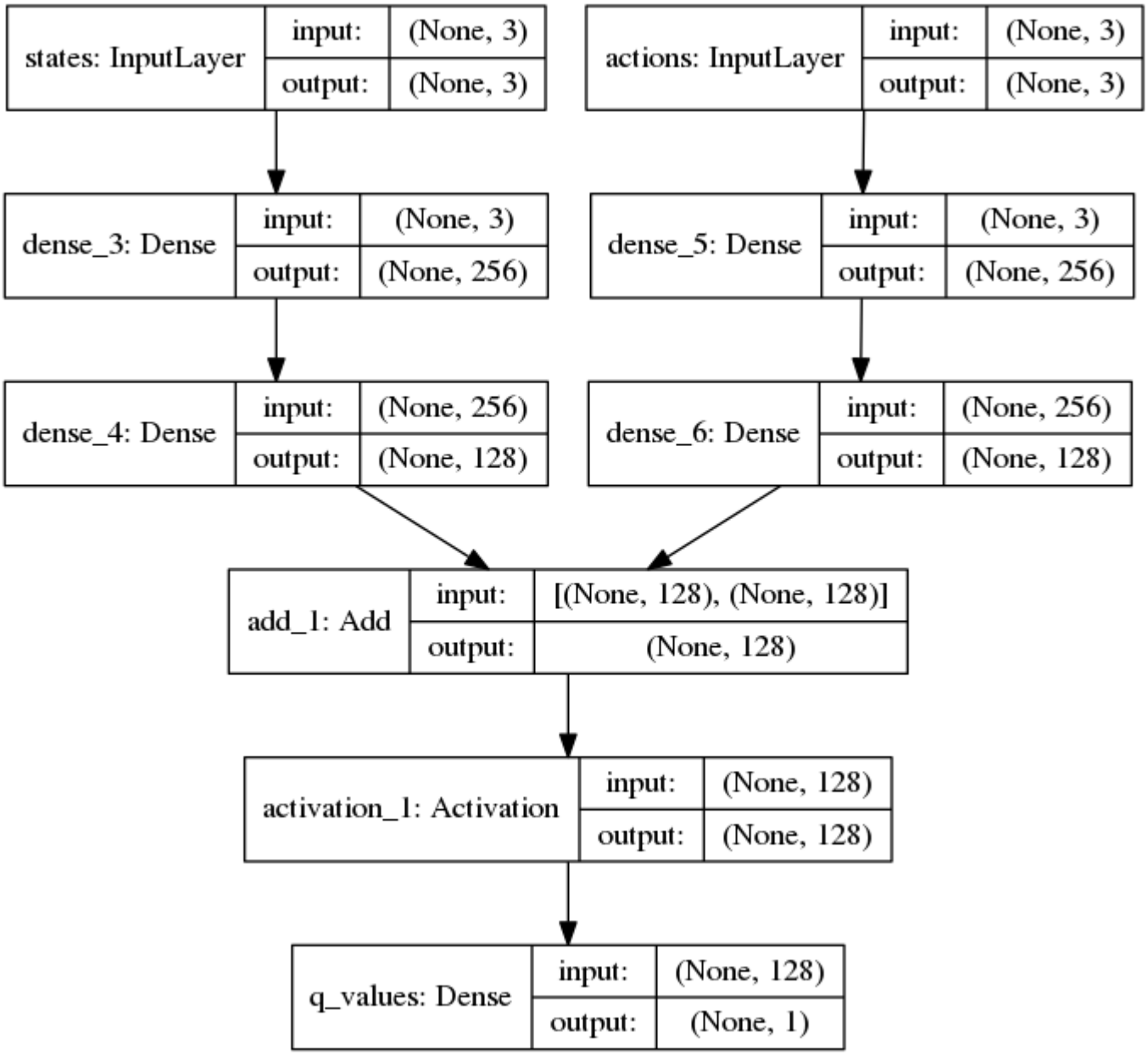
Q: What algorithm did you use? Briefly discuss why you chose it for this task.

A: I used an algorithm called DDPG, which is short for Deep Deterministic Policy Gradients and comes from the paper Continuous control with deep reinforcement learning. DDPG is a model-free policy based learning algorithm in which the agent learns directly from the unprocessed observation space without knowing the domain dynamic information, making it suited to solve control problems in continuous action space. It also employs actor-critic method in which actor model maps states into actions while critic model maps state-action pair into Q value. The visualization of actor and critic models is as following:

Actor:


states: InputLayer
input: (None, 3)
output: (None, 3)
dense_1: Dense
input: (None, 3)
output: (None, 256)
dense_2: Dense
input: (None, 256)
output: (None, 128)
raw_actions: Dense
input: (None, 128)
output: (None, 3)
actions: Lambda
input: (None, 3)
output: (None, 3)

Critic:



Q: Using the episode rewards plot, discuss how the agent learned over time.

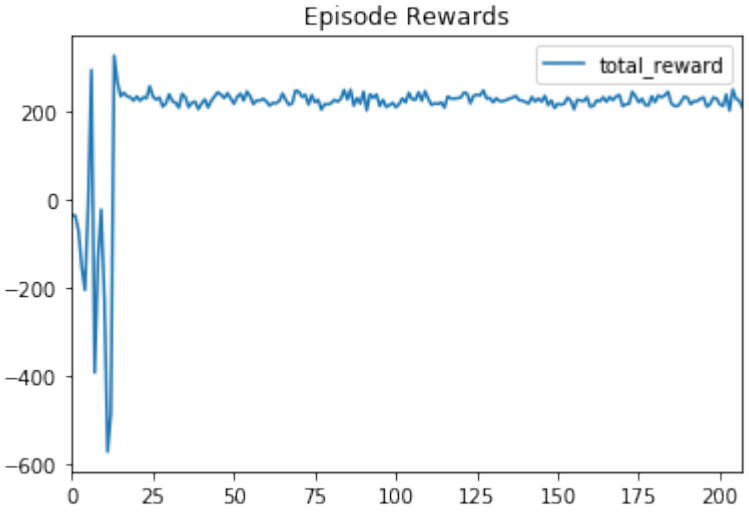
- Was it an easy task to learn or hard?
- Was there a gradual learning curve, or an aha moment?
- How good was the final performance of the agent? (e.g. mean rewards over the last 10 episodes)

A: It was an easy task to learn, from episode rewards plot we can see after around only 25 episodes the agent manages to learn how to take off. In this training case, there was not a gradual learning curve but an aha moment, which is around the 25th episode, before that moment the agent moves around but its height doesn’ t increase, leading to unchanged reward, after that moment the agent manages to learn how to lift from ground. The mean rewards over the last 10 episodes is around -600.

7.2. Hover 任务

目标： 训练智能体起飞并且悬停在某个点（如地上10个单位高度）。

Episode Rewards Plot:



Q: Did you change the state representation or reward function? If so, please explain below what worked best for you, and why you chose that scheme. Include short code snippet(s) if needed.

A: Yes, I changed the state representation, reward function was also changed. The new state representation is as following:

```
if self.last_pose is None:
            dist_last_pose = np.array([0., 0., 0.])
else:
    dist_last_pose = np.array([
        abs(pose.position.x - self.last_pose.position.x),
        abs(pose.position.y - self.last_pose.position.y),
        abs(pose.position.z - self.last_pose.position.z)
    ])
dist_target_z = abs(pose.position.z - self.target_z)

state = np.concatenate([
    np.array([pose.position.x, pose.position.y, pose.position.z]),
    dist_last_pose,
    np.array([dist_target_z])
])
```

As you can see, I included the difference between current pose and last pose, and the difference between current pose and target in z direction. These two components serves to make the agent sense whether leave target or current pose too far, thus making a well hovering.

As for reward function, the new one is as following:

```
reward = (10.0 - dist_target_z) * 0.8
if pose.position.z >= self.target_z:
    reward += 2.0  # give a small bonus
    if dist_target_z <= 2.0:
        reward += 5.0  # bonus reward, agent starts to hover
        if linear_acc:
            reward -= 0.1 * linear_acc
        if self.start_hover is None:
            self.start_hover = timestamp
        elif timestamp - self.start_hover >= 0.5:  # give reward if hover
for some time
            reward += 2.0
    elif dist_target_z > 5.0:  # agent leaves target too far
        if self.count % 20 == 0:
            print("last duration: {}".format(timestamp))
        if self.start_hover is not None:
            print("last duration: {:.4f}".format(timestamp -
self.start_hover))
        done = True
    else:
        self.start_hover = None
if timestamp > self.max_duration:  # agent has run out of time
    reward -= 5.0
    done = True
```

The reward function seems a litte complex. The base reward function is just like the one in takeoff task, when the agent leaves the target too far, it gets a smaller reward. Then if it cross over the target, it gets a small reward and if it stays in a some range between target, it gets a slight big reward. At the same time, the agent gets a penalty if it has a big linear acceration in z direction when hovering.

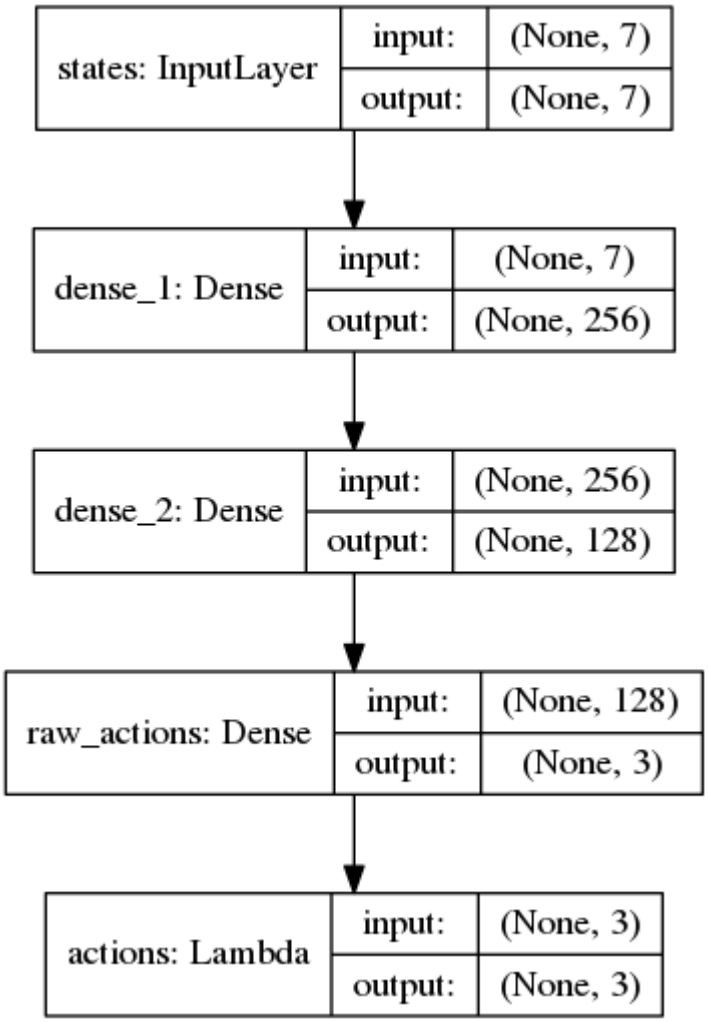
实现notes

Q: Discuss your implementation below briefly, using the following questions as a guide:

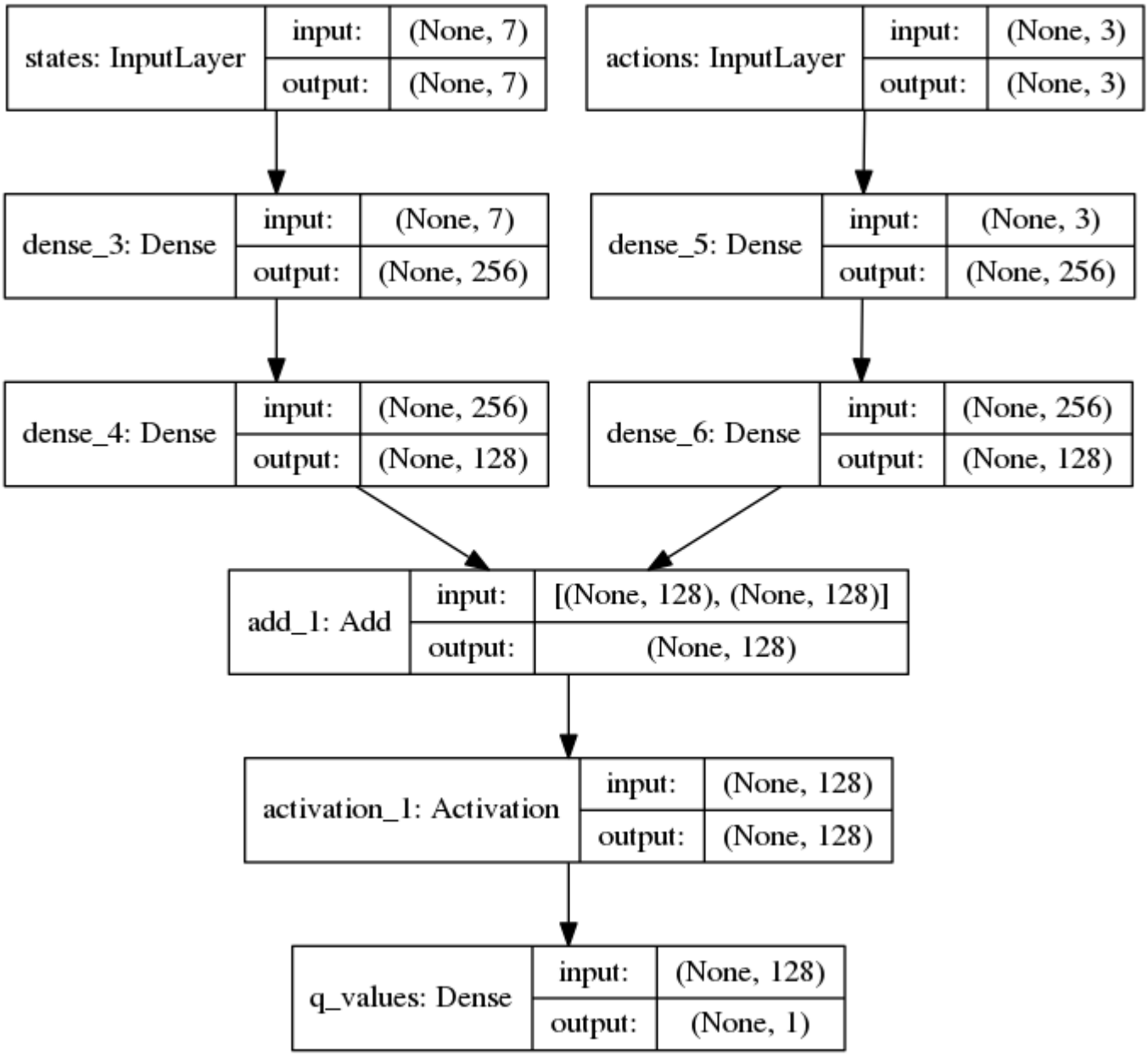
- What algorithm(s) did you try? What worked best for you?
- What was your final choice of hyperparameters (such as ☒ , ☒ , ☒ , etc.)?
- What neural network architecture did you use (if any)? Specify layers, sizes, activation functions, etc.

A: Again I used DDPG algorithm, it’ s same with that used in takeoff task. Because the state representation was changed, so the states input layer of actor and critic models has different input shape as following:

Actor:


states: InputLayer
input: (None, 7)
output: (None, 7)
dense_1: Dense
input: (None, 7)
output: (None, 256)
dense_2: Dense
input: (None, 256)
output: (None, 128)
raw_actions: Dense
input: (None, 128)
output: (None, 3)
actions: Lambda
input: (None, 3)
output: (None, 3)

Critic:

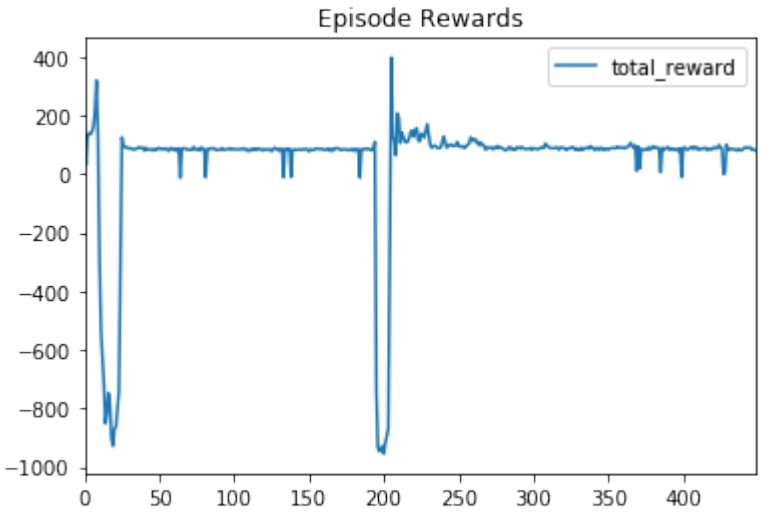


- The hyperparameters ☒ , ☒ used in this task is 0.001 and 0.99. A noise produced by Ornstein-Uhlenbeck process is added to action, where ☒ =0.0, ☒ =0.15 and ☒ =0.3.
- For neural network architecture please see beforemetioned actor and critic models diagram, the activation function is relu.

7.3. Landing 任务

目标：让智能体温和地着地。

Episode Rewards Plot:



初始条件，状态和回报

Q: How did you change the initial condition (starting state), state representation and/or reward function? Please explain below what worked best for you, and why you chose that scheme. Were you able to build in a reward mechanism for landing gently?

A: Because this time we need to make the quadcopter land, so the initial condition is the quadcopter is in the air as following:

```
def reset(self):
    ...
    return Pose(
                    position=Point(0.0, 0.0, 10.0),  # Land from height of
10 units above ground
                    orientation=Quaternion(0.0, 0.0, 0.0, 0.0),
                ), Twist(
                    linear=Vector3(0.0, 0.0, 0.0),
                    angular=Vector3(0.0, 0.0, 0.0)
                )
```

The state representation here is same with which used in hover task. The reward function (core part) is as following:

```
reward = (10.0 - dist_target_z) * 0.8
if pose.position.z <= 0.5:
    reward += 5.0  # give a small bonus
    if linear_acc:
        reward -= 0.1 * linear_acc
    done = True
elif timestamp > self.max_duration:  # agent has run out of time
    reward -= 5.0
    done = True
```

As before, the base part of reward function states that when the agent has a smaller difference with target, it gets a higher reward. When the agent falls in some small range between target, it gets a bonus, at the same time it gets penalty because of linear acceleration in z direction.

实现notes

Q: Discuss your implementation below briefly, using the same questions as before to guide you.

A: The agent used here is totally same with the one used in hover task.

7.4. Reflections

Q: Briefly summarize your experience working on this project. You can use the following prompts for ideas.

What was the hardest part of the project? (e.g. getting started, running ROS, plotting, specific task, etc.) How did you approach each task and choose an appropriate algorithm/implementation for it? Did you find anything interesting in how the quadcopter or your agent behaved?

A:

- The hardest part of the project for me was composing a well working state representation and reward function.
- All tasks are of continuous controlling, so a working agent implementation is needed. DDPG is a model-free policy based learning algorithm in which the agent learns directly from the unprocessed observation space without knowing the domain dynamic information, making it suited to solve control problems in continuous action space. So I choosed DDPG. For each task, to compose a well working reward function and hyperparameters, constant tries are needed and observe the performance of the agent and make a proper change.
- Interesting things in how the quadcopter behaved are:
  - At some time, the agent does the right thing at the first start;
  - At other time, the agent learns nothing;
  - Even for the agent learns how to well behaved, fluctuation may occurr.

8. 参考

- https://www.zhihu.com/question/47602063/answer/150845355
- https://openai.com/blog/openai-gym-beta
- https://zhuanlan.zhihu.com/p/39999667
- https://zhuanlan.zhihu.com/p/25239682
- https://arxiv.org/pdf/1509.02971.pdf