Experiment 1

211220166 王诚昊

1.实验过程

依照算法原理，完成了double_dqn.py和td3.py两个训练算法，并编写了对应的验证脚本，结合TensorBoard提供的数据可视化效果对不同算法的训练结果做出评估。

- double_dqn.py

```
with torch.no_grad():
    # TODO: generate target of Double DQN
    indices = torch.argmax(self._qnet(s_), dim=1)
    q_s_a = self._target_qnet(s_).gather(1, indices.reshape(-1,
1)).reshape(-1)
    label = r + GAMMA * q_s_a * (1 - d)
    target = self._qnet(s).scatter(1, a.reshape(-1, 1), label.reshape(-1,
1).float())
```

Double_DQN在DQN基础上做出的核心改动就是，将动作的**选择和评估两步解耦**，使用qnet()来进行动作选择，使用target_qnet()来进行动作评估。所以上述代码中在DQN基础上只需将第一行的目标Q网络换成Q网络即可。

- td3.py

目标策略网络的平滑正则化：

```
def query_target_action(self, obs):
    o = torch.tensor(obs).float()
    with torch.no_grad():
        a = self._target_actor(o)
        a = a.detach().cpu().numpy()

    # TODO: apply "Target Policy Smoothing Regularization"
    noise = np.random.normal(0, STD, (self._act_dim,))
    a += noise
    return np.clip(a, -1, 1)
```

理论上需要给目标Actor网络的输出添加一个带截断的正态分布噪声：

$$\epsilon \sim clip(\mathbf{N}(0, \sigma), -c, c)$$

但我们只有对于动作输出的限制条件：

$$-1 \leq \Delta\theta \leq 1$$

所以我们先加入噪声再进行截断，以保证动作不越界。

截断的Double Q-learning:

```
with torch.no_grad():
    # TODO: generate target_q with "Clipped Double Q-Learning for Actor-
Critic"
    y1 = r_tensor + GAMMA * self._target_critic[0](next_sa_tensor)
    y2 = r_tensor + GAMMA * self._target_critic[1](next_sa_tensor)
    target_q = torch.min(y1,y2)
```

用两个目标网络分别计算TD目标后，调用torch.min()来确定最终的TD目标。

策略网络和目标网络的延迟更新：

延迟更新的延迟率是由超参数里面的$DELAY$值确定的：

```
if self._step % DELAY == 0:
```

更新1个策略网络与DDPG算法里面没有什么不同：

```
# update actor
a_loss_log = 0
new_a_tensor = self._actor(s_tensor)
new_sa_tensor = torch.cat([s_tensor, new_a_tensor], dim=1)
q = -self._critic[0](new_sa_tensor).mean()
# q = -(self._critic[0](new_sa_tensor).mean() + self._critic[1]
(new_sa_tensor).mean())
# maybe?
self._actor_opt.zero_grad()
q.backward()
self._actor_opt.step()
a_loss_log = q.detach().cpu().item()
```

唯一不确定的一点就是应该使用critic[0]还是critic[1]或是同时考虑二者的方式来计算Q值。

更新3个目标网络：

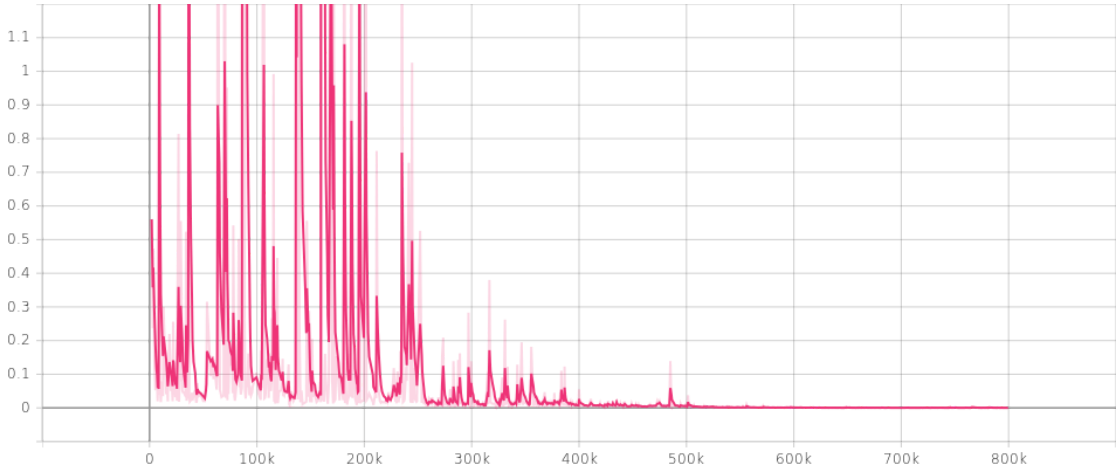
```
# update target network
self.soft_upd()
```

2.训练结果可视化展示

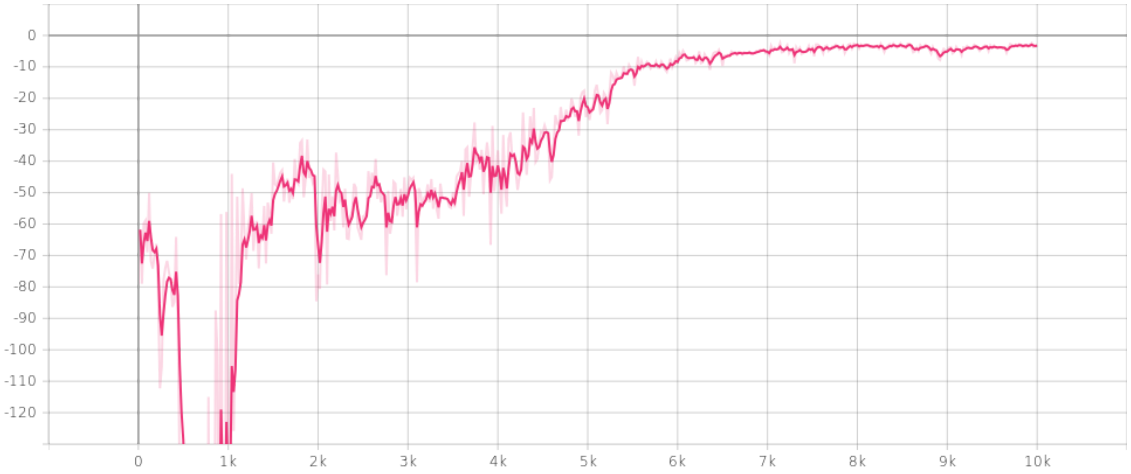
事实上对每个算法都进行了多次训练，下面对每个算法均只展示效果最理想的一次：

1. DQN

loss:

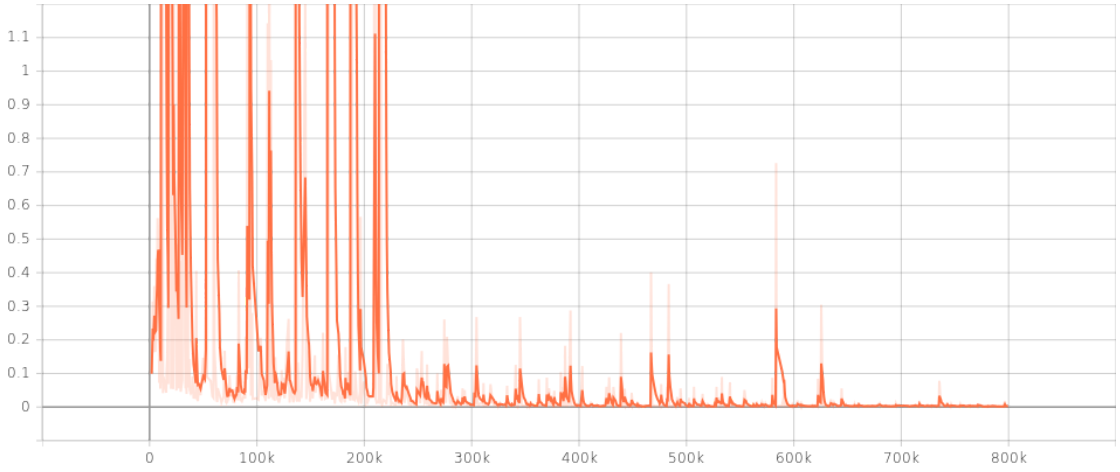


train_rew:

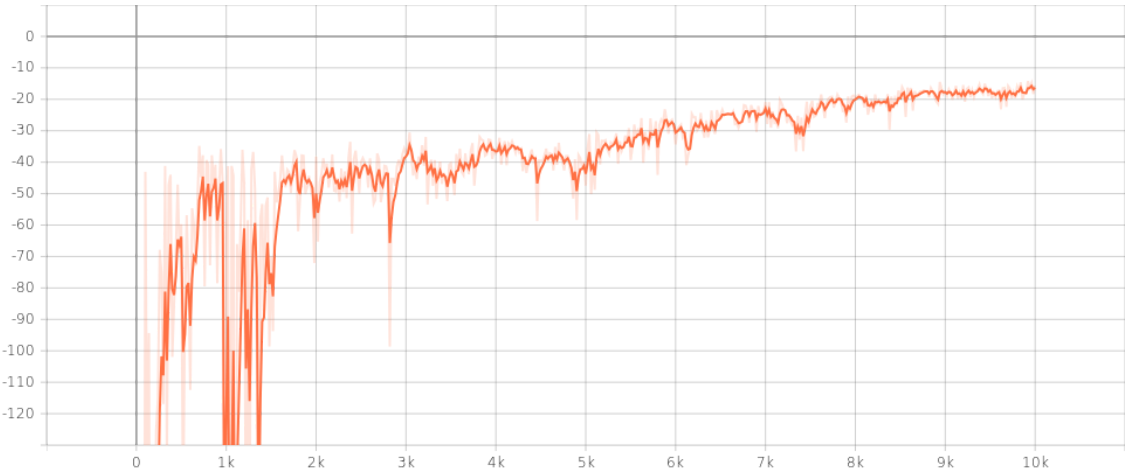


2. DDQN

loss:

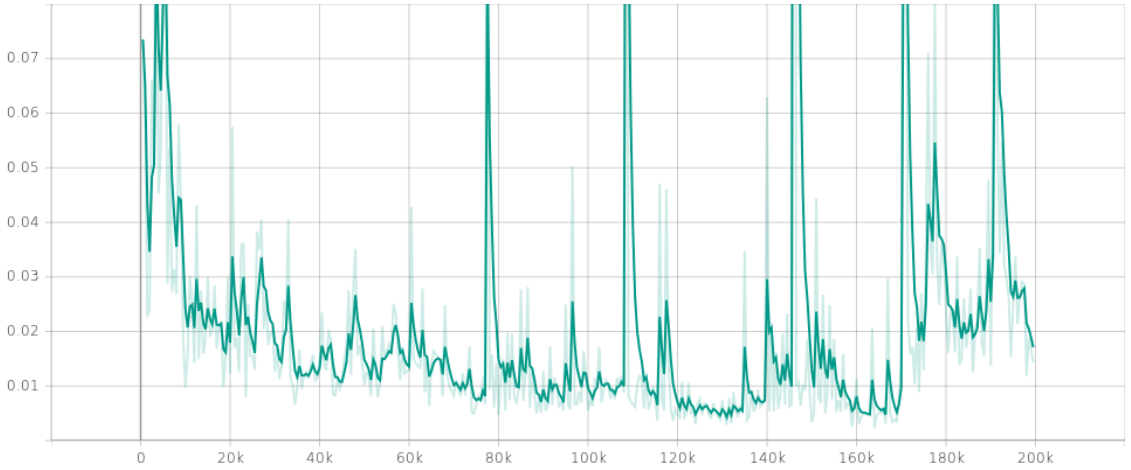


train_rew:

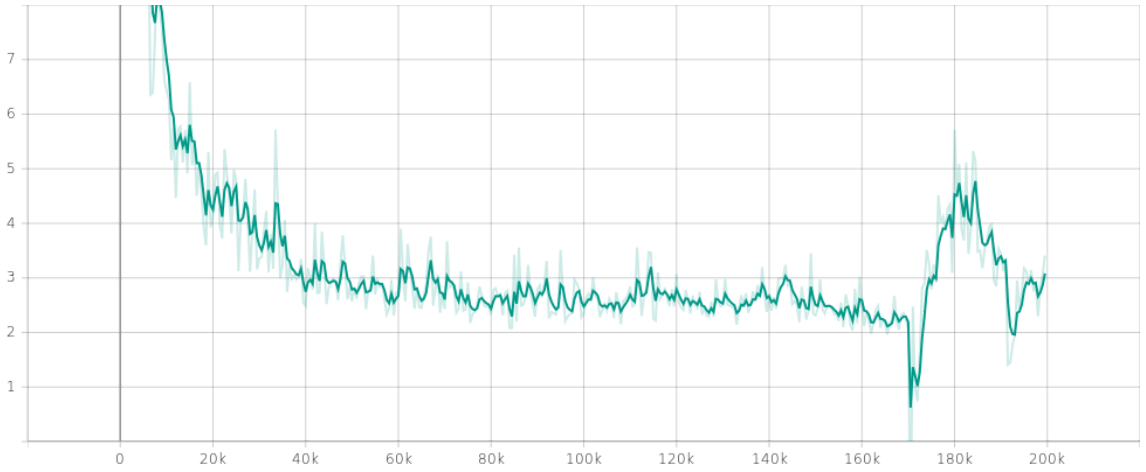


3. DDPG

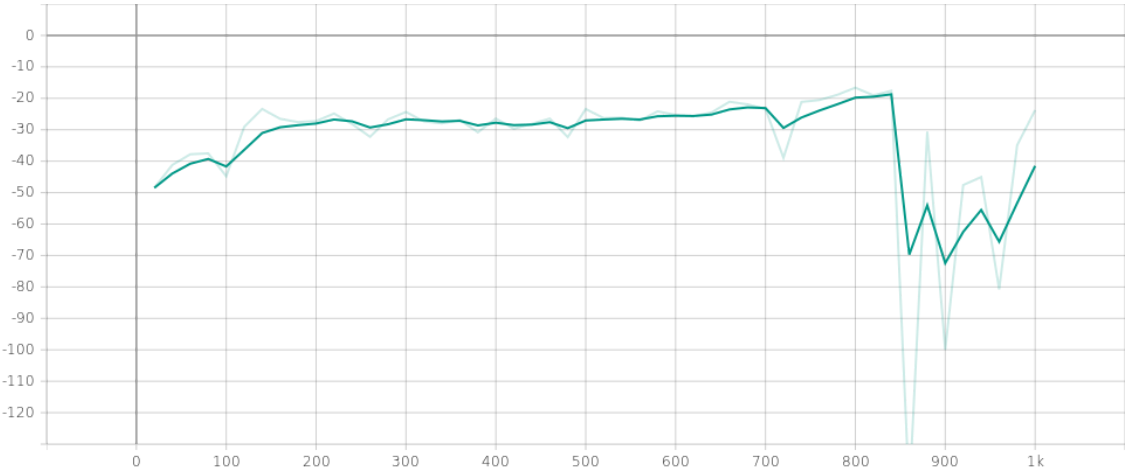
critic_loss:



actor_loss:

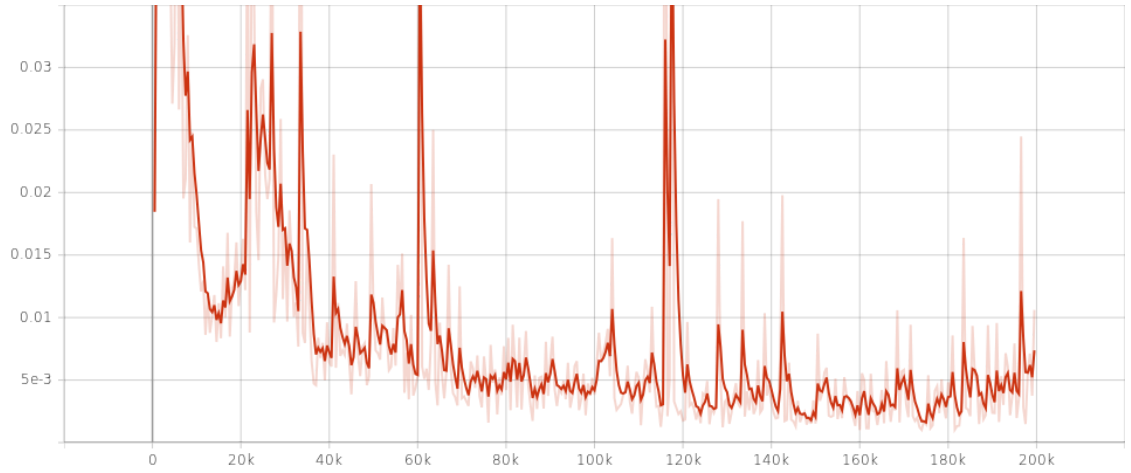


train_rew:

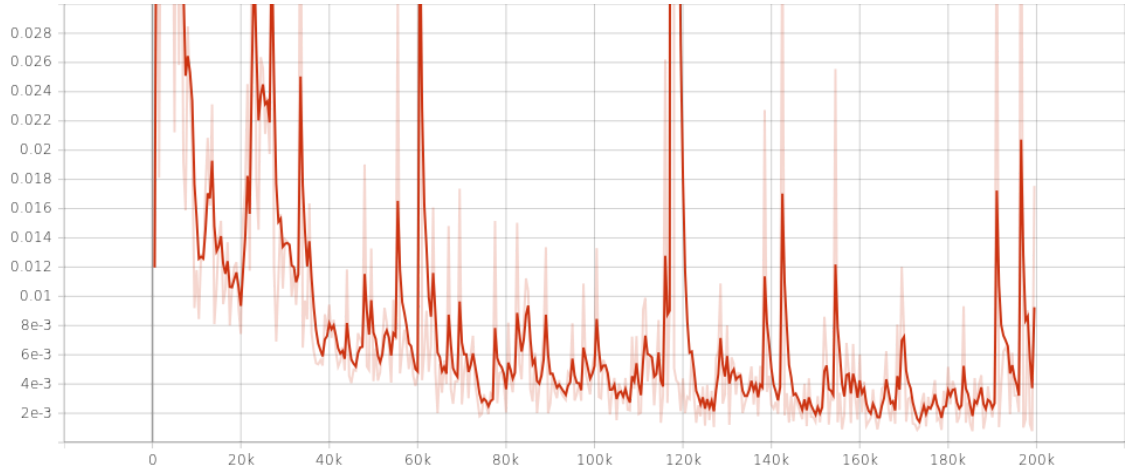


4. TD3

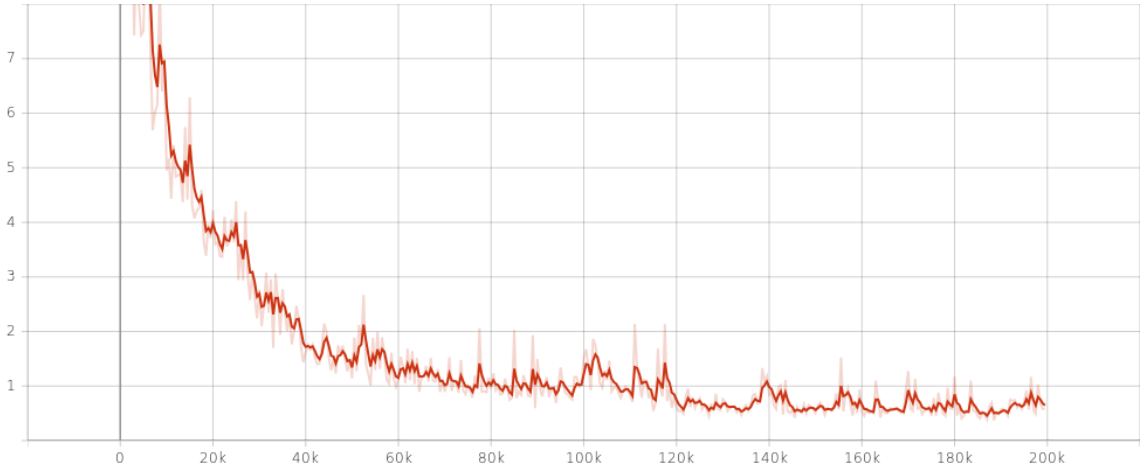
critic0_loss:



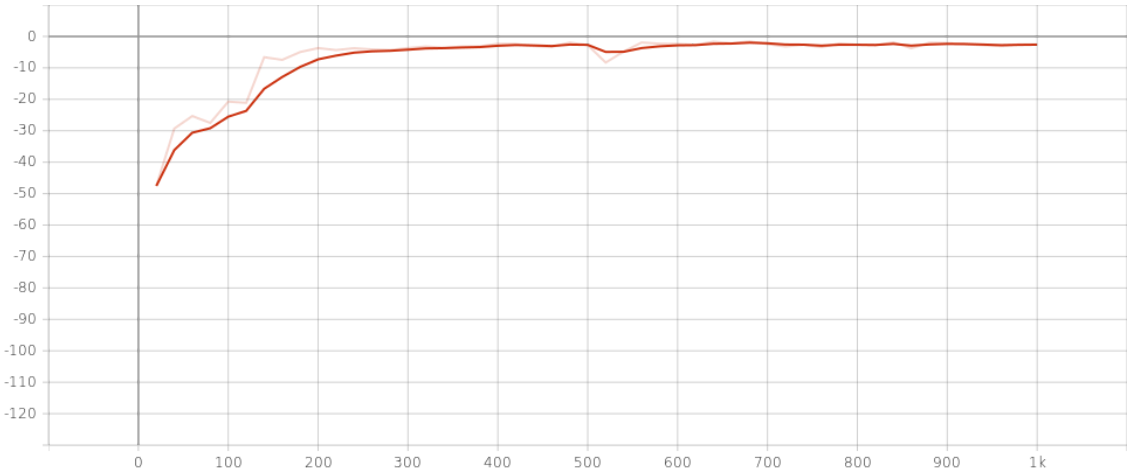
critic1_loss:



actor_loss:



train_rew:



3.总结

总体来说，DQN和TD3算法训练出的模型效果很好，尤其是TD3算法，仅仅训练了一次就得到了非常好且稳定的结果。然而另外的两个算法的结果并不尽人意，尽管训练了多次但各次结果之间偏差较大且均不是很好，说明其性能和稳定性均一般。

另外loss和reward并不能完全反映训练出的模型的性能，应该以实际渲染出的效果图为重要参考。因为我们的任务是控制无人机飞出一个圆形，但实验中有模型出现了**飞出一个圆弧后原路掉头返回的情形**，这时它的reward仍可能处于一个较高的水平。这个现象可能是因为我们的奖赏函数只考虑了几何位置关系，而忽略了运动的方向。