实验报告

181840013 陈超

1 基本配置

1. 语言：Pytorch
2. GPU：Kaggle 平台提供的 NVIDIA Tesla P100
3. 环境：LunarLander-v2

(食用本篇实验报告前建议先阅读第六部分)

2 Lunar Lander

在具体讲 Policy Gradient 和 A2C 两种方法前，有必要具体介绍一下 Lunar Lander 的具体机制。

Lunar Lander 由 OpenAI 的 gym 包提供，可以模拟飞船在月球登陆的情景。任务的目标是让登月小艇安全地降落在两个黄色旗帜间的平地上。整个代码运行的过程其实就是 agent 和 environment 交互的过程，每个 epoch 中，agent 根据当前 environment 的 state 计算得到 action 并将其传给 environment，environment 通过 step 函数执行 action，得到新的 state 和 reward，并将其传给 agent。而我们要做的就是训练出一个可靠的 agent 来与 environment 进行交互，从而使得飞船能够成功降落。

主要要介绍的与代码有关的是 state 和 reward 的计算方式。根据 LunarLander-v2 的源码，如下所示，我们可以分析出 state 这个状态向量中到底有什么。

```
state = [
(pos.x - VIEWPORT_W/SCALE/2) / (VIEWPORT_W/SCALE/2),
(pos.y - (self.helipad_y+LEG_DOWN/SCALE)) / (VIEWPORT_H/SCALE/2),
vel.x*(VIEWPORT_W/SCALE/2)/FPS,
vel.y*(VIEWPORT_H/SCALE/2)/FPS,
```

```
self.lander.angle,
20.0*self.lander.angularVelocity/FPS,
1.0 if self.legs[0].ground_contact else 0.0,
1.0 if self.legs[1].ground_contact else 0.0]
```

不难看出，state 包含了以下 8 个参量：水平位置，垂直位置，水平速度，垂直速度，倾斜角，角速度，左着陆架是否触碰地面，右着陆架是否触碰地面。清楚了解这些参量对接下来我们修改 reward 函数十分重要。

reward 函数的计算方式可以阅读源码，也可以在 gym 官网上查到 [2]。在计算 reward 之前，作者定义了一个 shaping 参量：

```
shaping = \
- 100*np.sqrt(state[0]*state[0] + state[1]*state[1]) \
- 100*np.sqrt(state[2]*state[2] + state[3]*state[3]) \
- 100*abs(state[4]) + 10*state[6] + 10*state[7]
```

shaping 中给了位置-100 的权重，速度-100 的权重，角度-100 的权重，两个着陆架是否碰到地面各 10 权重。按照我的理解，这个 shaping 其实就应该是 reward 了。但作者偏不，他把 reward 的第一部分看成是当前 shaping 减去上一轮的 shaping，取一个差值。我想作者这样是为了表现出采取某个 action 之后 reward 是增大还是减小了，增大表示这个 action 是好的，减小则表示这个 action 不好。reward 的第二部分是点火部分，规定主引擎点火一次-0.3，左右引擎点火一次-0.03。第三部分是着陆部分，如果着陆在指定区域内 +100，否则-100。

3 Policy Gradient

3.1 原理

梯度策略的原理其实很简单，就是用一个全连接的神经网路，输入是当前状态向量 state，输出是动作向量 action，向量的每个元素大小表示概率。也就是说神经网络得到的 action 实际上是一个概率分布，agent 会根据这个概率分布来选择执行什么动作，概率值越高的动作越有可能被执行。训练时，每一轮更新参数前都会进行 episode_per_batch 次模拟，根据这几次 reward 的平均对网络进行更新。

3.2 代码分析

这里的代码主要参考了压缩包中的 Final2_reinforcement_learning.ipynb。主要分成环境搭建，Agent 构建和训练三个部分。每部分具体可以参考 ipynb 文件，这里就不再赘述。下面确定一组参数作为 Baseline，方便下面对超参数的比较。

- 训练轮数 num_batch: 1000
- 每轮训练模拟次数 episode_per_batch: 5
- 优化器 optimizer: SGD，learning_rate=0.002
- 网络结构: 如下所示

```
----------------------------------------------------------------
Layer (type)               Output Shape              Param #
================================================================
Linear-1                   [-1, 3, 16]               144
Linear-2                   [-1, 3, 16]               272
Linear-3                   [-1, 3, 4]                68
================================================================
Total params: 484
Trainable params: 484
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
----------------------------------------------------------------
```

3.3 训练结果

对训练过程中的 total_reward 和 final_reward 画图，得到如下图像：

图 1: Total Reward

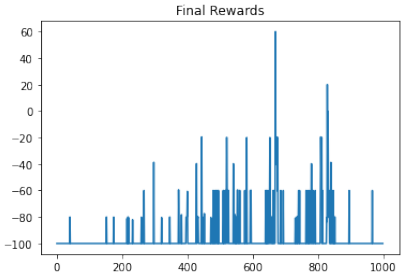

图 2: Final Reward

3.4 超参数比较

这里我选取了 episode_per_batch，optimizer 和网络结构进行改动，并对每一次改动进行比较。

3.4.1 episode_per_batch

保持其他参数不变，将 episode_per_batch 改为 1 和 10，训练 1000 轮，训练过程中的 total_reward 图像如下所示：

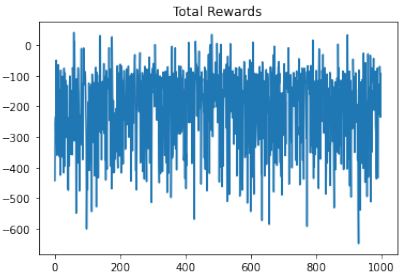

图 3: episode per batch=1

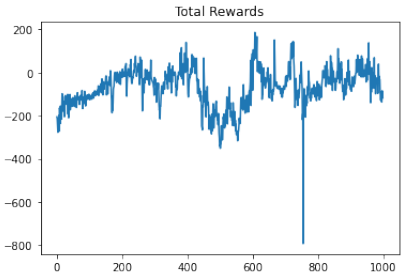

图 4: episode per batch=10

可以看到区别还是很明显的，当 episode_per_batch=1 时，由于一次模拟的随机性太强，导致梯度更新方向有很大随机性，整个训练过程自然也就很不稳定，上下震荡。当 episode_per_batch=10 时，多次模拟取平均虽然增加了计算量，但确实一定程度上减小了随机性，使得梯度更新方向更确定，整个训练过程也就更稳定。

3.4.2 optimizer

保持其他参数不变，将 optimizer 的学习率改为 0.001 和 0.005，再把 optimizer 整个改为 Adam，lr=0.002，训练 1000 轮，训练过程中的 total_reward 图像如下所示：

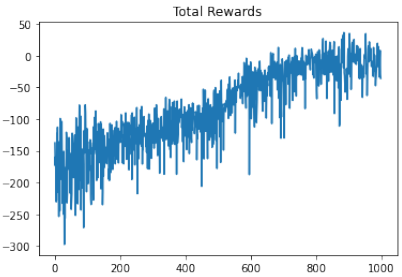

图 5: SGD lr=0.001

图 6: SGD lr=0.005

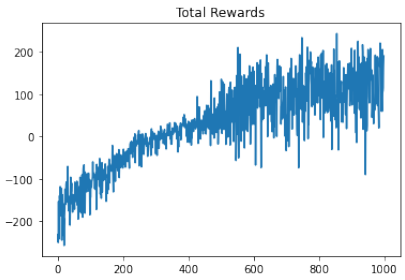

图 7: Adam lr=0.002

可以看到，SGD 情况下，lr=0.001 的训练过程明显比 lr=0.005 稳定，这也是很好理解的。而使用 Adam 则不仅可以加快训练速度，还能够取得更高的 reward。所以对于本题来说，Adam 加上较小的学习率是最好的 optimizer 组合。

3.4.3 网络结构

保持其他参数不变，将网络从原来的 3 层改为 2 层 (去掉一个全连接层) 和 4 层 (加上一个全连接层)，训练 1000 轮，训练过程中的 total_reward 图像如下所示：

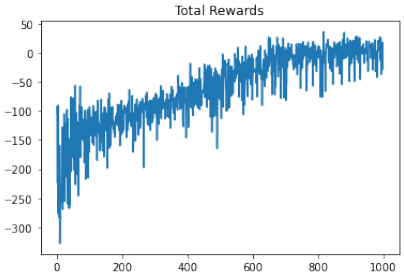

图 8: layers=2

图 9: layers=4

可以看到，当网络层数减为 2 时，Total Reward 能达到的最大值也就只有 50 左右，可见此时网络的拟合能力是不够的。当层数为 4 时，训练过程的稳定性略有下降，最后 Total Reward 和 Baseline 一样，大致为 200，并没有什么提升，可见对于这个任务而言，3 层的网络已经足够了。

3.5 修改 Reward

针对前面第二部分介绍的 state 和 reward 的计算方法，我主要做了两方面的修改，一是加上一个衰减系数，二是调整各项权重。

3.5.1 加衰减系数

鉴于整个模拟过程中一开始的 reward 的重要性显然不如后期的 reward 的重要性，我给 reward 加上了一个衰减系数 0.99，截止的数目为 100，也就是说只考虑最后 100 个 reward，并且从后往前有一个 0.99 的衰减。

3.5.2 调整权重

原先的 reward 计算方式我在第 2 节中已经详细说明了，其实我觉得这个已经很不错了。但若非要改一下，我觉得有以下几个地方可以改进：

1. 由于这个小游戏中飞船似乎不会因为落地速度太快炸掉，那么竖直方向速度也就没有那么重要，相比之下水平方向速度更为重要，那么我们就可以适当增加水平方向速度的权重，减小竖直方向速度的权重。

2. 相较于竖直位置，水平位置也更为重要，因为这决定了飞船能否落在指定区域，于是我们可以适当增加水平位置权重，减小竖直位置权重。

3. 经过观察，飞船有时会在撞地后发生侧翻，为了防止这种情况，可以适当增肌左右着陆架是否触碰地面这一项的权重。

4. 为了让飞船更注重着陆的结果，可以适当增加着陆成功的奖励和着陆失败的惩罚。

使用函数重载的方式，重写源码中 LunarLander 这个类的 step 函数，重载后的 reward 计算方式如下：

```
reward = 0
shaping = \
- 50*np.sqrt(state[0]*state[0]) - 150*np.sqrt(state[1]*state[1]) \
- 50*np.sqrt(state[2]*state[2]) - 150*np.sqrt(state[3]*state[3]) \
- 100*abs(state[4]) + 30*state[6] + 30*state[7]

if self.prev_shaping is not None:
        reward = shaping - self.prev_shaping
        self.prev_shaping = shaping

reward -= m_power*0.30
reward -= s_power*0.03

done = False
if self.game_over or abs(state[0]) >= 1.0:
        done = True
        reward = -200
if not self.lander.awake:
        done = True
        reward = +200
```

3.5.3 运行结果

其他参数与前述 Baseline 相同，运行 1000 轮。结果如下所示：

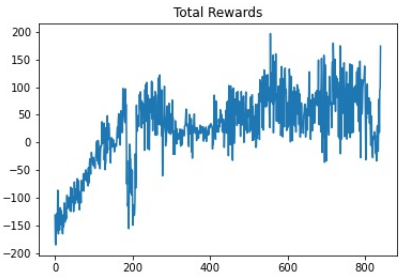

图 10: Total Reward

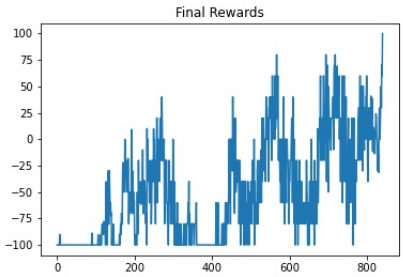

图 11: Final Reward

reward 改进之后的训练效果确实比原先要好了一些，尤其是在 final reward 这一项指标上，改进后的模型明显能够有更高的得分，也就意味着能够更平稳地降落，可见这个改进还是有效果的。不过话又说回来，也不是原来的 reward 不好，原来的 reward 在我看来已经足够好了，只要在多训练一些，一样能训练出很好的模型。

4 A2C

4.1 原理

A2C 全称为优势动作评论算法 (Advantage Actor Critic)，顾名思义，该模型中包含两个网络：Actor 和 Critic，其中 Actor 和 Policy Gradient 中的 Actor 一致，就是传入当前的 state 向量，传出一个 action 的概率分布。Critic 则是一个评价网络，传入的同样是当前的 state 向量，传出的则是一个值，表示当前状态的好坏。之后再用优势函数代替 Critic 网络传出的值，作为衡量选取动作值和所有动作平均值相比好坏的指标。优势函数 A 的计算方式为：

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{1}$$

其中 Q 为当前状态 s 下的 a 动作对应的值函数，V 为当前状态下所有可能动作所对应的动作值函数乘以采取该动作的概率的和。两个一减就是动作值函数相比于当前状态值函数的优势。如果大于 0，则说明该动作比平均动作好，如果小于 0，则说明该动作不如平均动作。

于是，我们就可以把-A 作为更新 Actor 网络的 loss，Critic 网络的输出值和实际 reward 的差值的平方作为更新 Critic 网络的 loss，每轮训练中依次更新 Actor 和 Critic，经过若干轮更新就可以得到一个能够给出河里 Action 的 Actor 和一个能给出和实际 reward 相近的值的 Critic。

4.2 代码分析

这边的代码我参考了 github 上的相关文档 [3]，相较于 Policy Gradient 的代码主要要两处改动，一是网络结构，二是训练策略，下面逐个进行介绍。

4.2.1 网络结构

如前所述，A2C 模型又两个网络，分别是 Actor 和 Critic，根据 Policy Gradient 中改变网络深度的实验结果，三层网络对这个任务已经足够，但是保险起见，我还是稍微加大了网络大小，把中间层的神经元数量由 16 个改为 64 个, 并且加入 dropout。Actor 输出层为四个神经元，并且有 softmax，Critic 输出层为 1 个神经元，没有 softmax，其余相同。网络结构具体如下：
Actor:

```
----------------------------------------------------------------
Layer (type)               Output Shape              Param #
================================================================
Linear-1                   [-1, 1, 64]               576
Dropout-2                  [-1, 1, 64]               0
Linear-3                   [-1, 1, 64]               4,160
Dropout-4                  [-1, 1, 64]               0
Linear-5                   [-1, 1, 4]                260
================================================================
Total params: 4,996
Trainable params: 4,996
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
```

Params size (MB): 0.02
Estimated Total Size (MB): 0.02

Critic:

Layer (type)	Output Shape	Param #
Linear-1	[-1, 1, 64]	576
Dropout-2	[-1, 1, 64]	0
Linear-3	[-1, 1, 64]	4,160
Dropout-4	[-1, 1, 64]	0
Linear-5	[-1, 1, 1]	65

Total params: 4,801
Trainable params: 4,801
Non-trainable params: 0

Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.02
Estimated Total Size (MB): 0.02

4.2.2 训练策略

如前所述，更新 Actor 是根据优势函数 A，更新 Critic 则是根据其输出与实际 reward 的差值，具体到代码如下所示：

```
loss_actor = ((R - values.detach()) * -log_probs).mean()
loss_critic = ((R - values) ** 2).mean()
```

其中 R 是加上衰减系数后的 reward，values 就是 Critic 网络的输出，log_probs 就是 Actor 网络的输出。算出 loss 之后反向传播更新 Actor 和 Critic 即可。

另外，A2C 方法虽然理论上来说确实可以不用模拟完一整个回合再更新，但保险起见，我还是和 Policy Gradient 一样每模拟十次更新一次，这样可以使训练过程更加稳定。

4.2.3 相关参数

- 训练轮数 num_batch: 1000
- 每轮训练模拟次数 episode_per_batch: 10
- 优化器 optimizer: Adam，learning_rate=0.002
- 最大计算步数 num_steps: 100
- 衰减系数 gamma: 0.99

4.3 训练结果

本来我是想跑 1000 轮的，奈何 Kaggle 上内存不够，跑到 400 轮左右就会爆内存，没办法，我只好跑个 300 轮看看，结果如下：

图 12: Total Reward

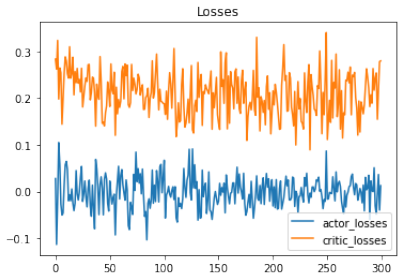

图 13: Actor & Critic loss

可以看到，训练过程还是非常稳定的，不过相较于 Policy Gradient，收敛速度有些慢。这也是可以理解的，毕竟这里要同时训练两个网络 (网络也稍微大了些)，要是能多训练几百轮，应该能得到一个不错的模型，不过我这里由于内存限制跑不了了，希望助教能够帮助我完成这个愿望 (x。

5 PG 与 A2C 的比较

以我的理解，A2C 其实就是 PG 的加强版，A2C 的 Actor 网络与 PG 的 Actor 的输入输出是完全一致的，唯一不同的就是 loss 的计算方式，A2C 是以优势函数 A 乘以-log_prob 为 loss，也就是相较于平均动作，当前的动作是好还是坏，而 PG 则是直接以 reward 乘以-log_prob 作为 loss。除了 Actor，A2C 还增加了一个 Critic 网络用于评估当前状态，这就相当于引入了一个随机初始化的 Q 函数，在训练中不断逼近真实的 Q。换句话说，我们在梯度更新中引入了噪声。

就具体应用而言，在 lunar lander 这个任务中，说实话 A2C 并没有表现出什么优势，因为这个任务中原来的 reward 计算方式本身就很不错，而且网络也并不大，只有三层。两种方法理论上最后都能收敛到一个不错的模型。不过在训练时间上 A2C 比 PG 明显长了不少 (这里我的 PG 和 A2C 都是进行几次完整的模拟后再更新参数的)。我们不妨再来对比一下 PG 和 A2C 训练过程中 total reward 的图像：

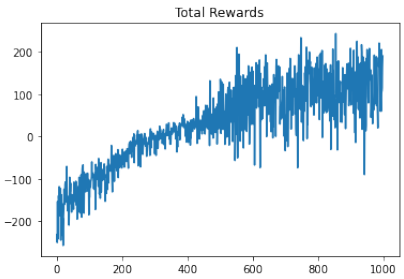

图 14: PG

图 15: A2C

虽然 A2C 只训练了 300 轮，但是已经足够说明问题了。300 轮时，PG 方法达到了 0 左右的 reward，而 A2C 只有-50，可见在同样的参数下，二者的训练收敛速度还是有差异的。不过如前所述，A2C 的一个优势是，由于有 critic 作为近似的评价函数，可以不进行完整的一轮模拟就更新参数，我也尝试过，但是发现 Critic 很难收敛，导致 Actor 也不能收敛，最后还是放弃了不完整模拟就更新这个方法，老老实实完整模拟完再更新。

总而言之，加上了 Critic 的 A2C 方法，如果考虑到其可以不进行完整模拟就更新参数，在训练速度上可能确实会比 PG 要快不少，但这是建立在 Critic 可以收敛的基础上的。否则如果都是完整模拟完再更新参数，A2C 的

训练速度和收敛速度相比 PG 都会差一些。就本任务而言，PG 应该是优于 A2C 的。

6 舍不得删掉的废话

1. 提交的 project 中,Figures 文件夹包含了所有原始图像,Network.param 为训练好的模型。train(pg).py 为使用 Policy Gradient 的训练代码，直接运行即可训练，并对训练结果进行测试。train(a2c).py 为使用 A2C 的训练代码,直接运行即可训练,并对训练结果进行测试。test.py 为使用 Actor.param 进行测试的代码，直接运行即可进行测试，其中 show 参数控制是否显示图像。

2. 由于内存问题，我实在没法完成整个 A2C 网络的训练，只好训练个 300 轮看看效果，不过助教的电脑应该是可以跑的。

3. 提交的 Network.param 是用 PG 训练的，效果应该是不错的，成功率能达到 80% 以上。

4. 感谢：感谢徐满杰同学告诉我 A2C 的正确写法，一开始我的 A2C 就是个智障。感谢王子静同学跟我讨论她对于 reward 函数计算方式的想法，根据她的实验结果，我也少走了一些弯路。感谢孙舒禺同学提醒我我的 Critic 网络最后多了一个 Softmax 层。

5. 如果有任何问题，请邮箱联系我 181840013@smail.nju.edu.cn

7 参考资料

[1] https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar_lander.py

[2] https://gym.openai.com/envs/LunarLander-v2/

[3] https://github.com/tejaskhot/pytorch-LunarLander/blob/master/a2c.py