强化学习第二次实验

徐宽 - SA21229033

一、实现过程

1.1 环境准备

最开始执行 `python -m atari_py.import_roms .` 的时候报错：

> (ModuleNotFoundError: No module named 'atari_py')

然后根据助教哥哥的建议改成 `python -m ale-import-roms .` 之后还是报错

> No module named ale-import-roms

根据群里小伙伴的建议重新安装： `conda install -c conda-forge atari_py` 结果又报错

> Collecting package metadata (current_repodata.json): failed

然后只好先连个 VPN，重新安装，成功！

重新运行最开始的代码，又出了点小问题，根据助教哥哥的建议把 ROMS 放到其他盘，芜湖，成功！

一个小插曲： tensorboard 打不开。报错信息

> W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudart64_110.dll'; dlerror: cudart64_110.dll not found
> I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.

然后我发现我环境中的 tensorboard.exe 修改时间是最近几天，应该就是我安装 tensorflow 的时候把原来的 tensorboard.exe 覆盖掉了，然后替换回去终于解决了这个问题！

1.2 PG算法流程

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

- 首先得到一个 observation, 并做预处理

```
# Preprocess the observation, set input to network to be difference
# image between frames
cur_x = preprocess(observation)
x = cur_x - prev_x
```

- 接着通过 policy network 估计 agent 向上的概率
- ```
  # Run the policy network and sample action from the returned probability
  prob_up = model(x)
  ```

  其中网络结构如下

  ```
  class PolicyNetwork(nn.Module):
      """ Simple two-layer MLP for policy network. """

      def __init__(self, input_size, hidden_size):
          super().__init__()

          ### TODO: e.g. a two-layer MLP with input size `input_size`
          ###       and hidden layer size of `hidden_size` that outputs
          ###       the probability of going up for a given game state.
          self.net = nn.Sequential(
              nn.Linear(input_size, hidden_size),
              nn.ReLU(),
              nn.Linear(hidden_size, 1),
              nn.Sigmoid()
          )

      def forward(self, x):

          ### TODO: Define the forward method as well
          prob_up = self.net(x)

          return prob_up
  ```

- 然后根据概率采样得到 action

  ```
  ### TODO: Sample an action and then calculate the log probability of sampling
  ###       the action that ended up being chosen. Then append to `action_chosen_log_probs`.
  action = UP if random.random() < prob_up else DOWN
  ```

- 与环境交互，更新下一步

  ```
  # Step the environment, get new measurements, and updated discounted_reward
  observation, reward, done, info = env.step(action)
  ```

- 达到退出条件后退出游戏，计算 discounted_future_rewards 并标准化
  ```

```python
# Calculate the discounted future reward at each timestep
discounted_future_rewards = calc_discounted_future_rewards(rewards, discount_factor)

discounted_future_rewards = (discounted_future_rewards - discounted_future_rewards.mean()) / discounted_future_rewards.std()
```

标准化操作可以帮助控制梯度估计的方差，它会导致大约一半的行动受到鼓励，另一半的行动受到打击，这在+1奖励信号很少的开始阶段非常有用。

- 计算损失

```python
### TODO: Calculate the loss that the optimizer will optimize
loss = -(discounted_future_rewards * action_chosen_log_probs).sum()
```

这便是 PG 算法的关键，损失前面加上负号，因为是梯度上升。

1.3 A2C算法流程

REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

A2C 算法可以视为对 PG 算法的一个改进，它使用 Q(s,a) 来代替 R，这个 Q 值是由神经网络产生的。我们不再需要等完整的序列，只需要把响应的**动作**和**状态**传到 Q 网络（critic）中，询问 Q 值，就可以大致判断出动作好坏。

PG 算法中的 Policy Network 作为 Actor 网络负责产生动作，另外再加一个 Value Network 作为 Critic 网络对这个动作进行评价，跟 GAN 有异曲同工之妙。两个网络相互配合/对抗，共同训练，不同的是这里的 Critic 网络只是作为一个辅助网络，我们真正关心的不是"裁判"而是"运动员"，所以主要还是为了训练 Actor 网络。

- Critic 网络的结构如下：

```python
class Critic(nn.Module):
    """ Simple two-layer MLP for value network. """
    def __init__(self, input_size: int, hidden_size: int):
        super(Critic, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 1)
```

```python
        )

    def forward(self, x):
        x = self.net(x)
        return x
```

跟 Actor 基本一样，只是减少了最后一层的 Sigmoid.

- 改变计算 loss 的方式

```python
# Calculate the loss that the optimizer will optimize
actor_loss = -((discounted_future_rewards-value) * action_chosen_log_probs).sum()
critic_loss = (discounted_future_rewards-value).pow(2).sum()
```

这里 Critic 网络的打分 value 可以视为一个 baseline.

- 两个网络同时更新

```python
# Backprop after `batch_size` episodes
actor_optimizer.zero_grad()
critic_optimizer.zero_grad()

mean_batch_loss.backward(retain_graph=True)
critic_batch_loss.backward()

actor_optimizer.step()
critic_optimizer.step()
```

这里还有一个小插曲：连续两个 backward() 会报错

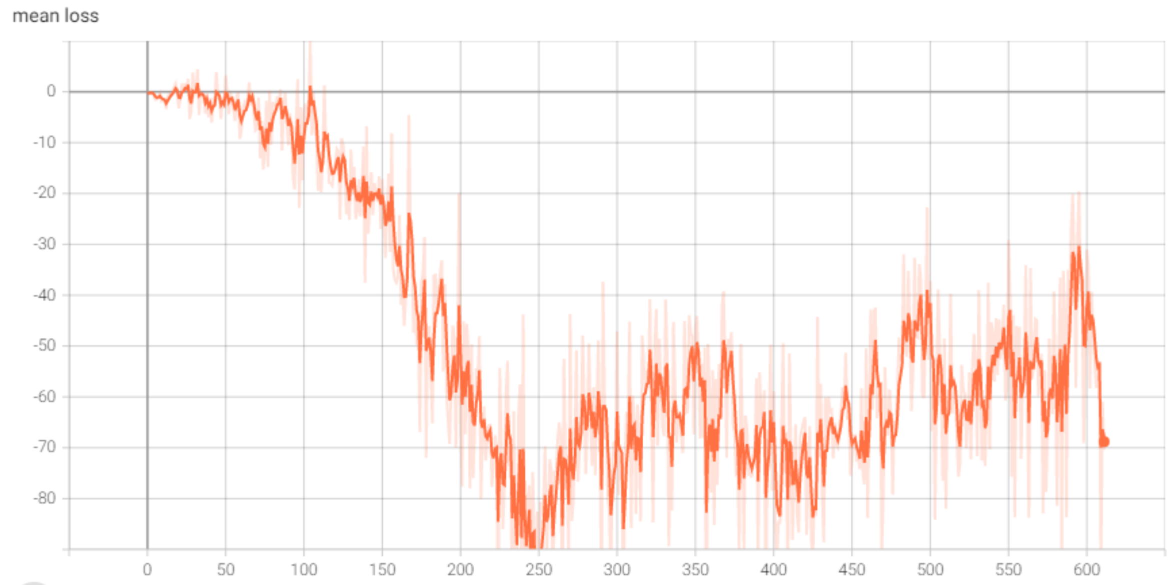
```python
mean_batch_loss.backward()
critic_batch_loss.backward()
```

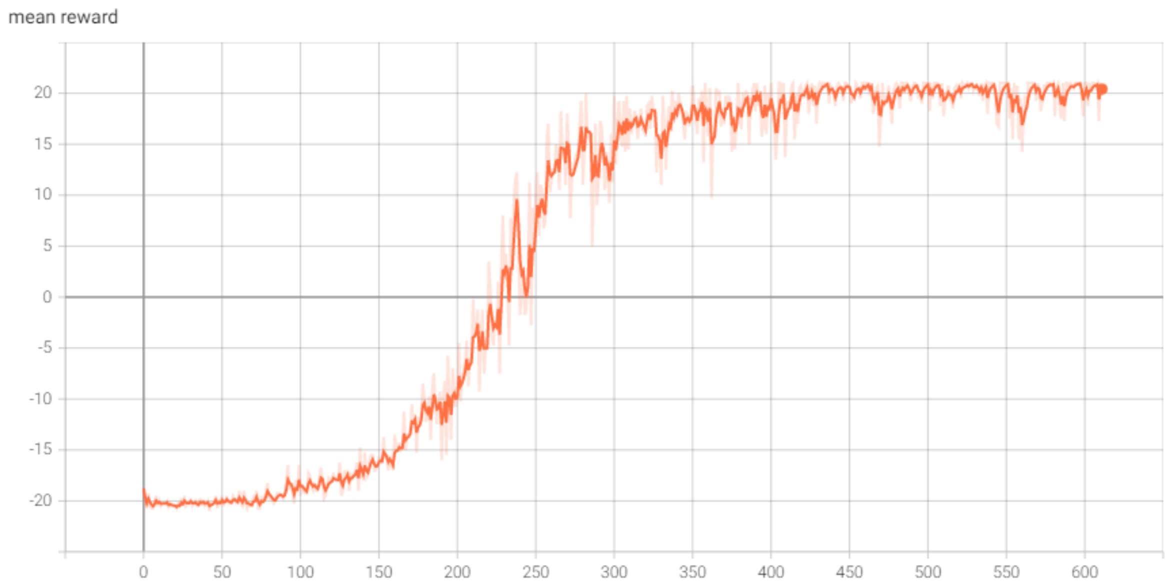
> RuntimeError: Trying to backward through the graph a second time (or directly access saved tensors after they have already been freed). Saved intermediate values of the graph are freed when you call .backward() or autograd.grad(). Specify retain_graph=True if you need to backward through the graph a second time or if you need to access saved tensors after calling backward.

要再前面加上参数 `retain_graph=True` 才可以

二、实现结果

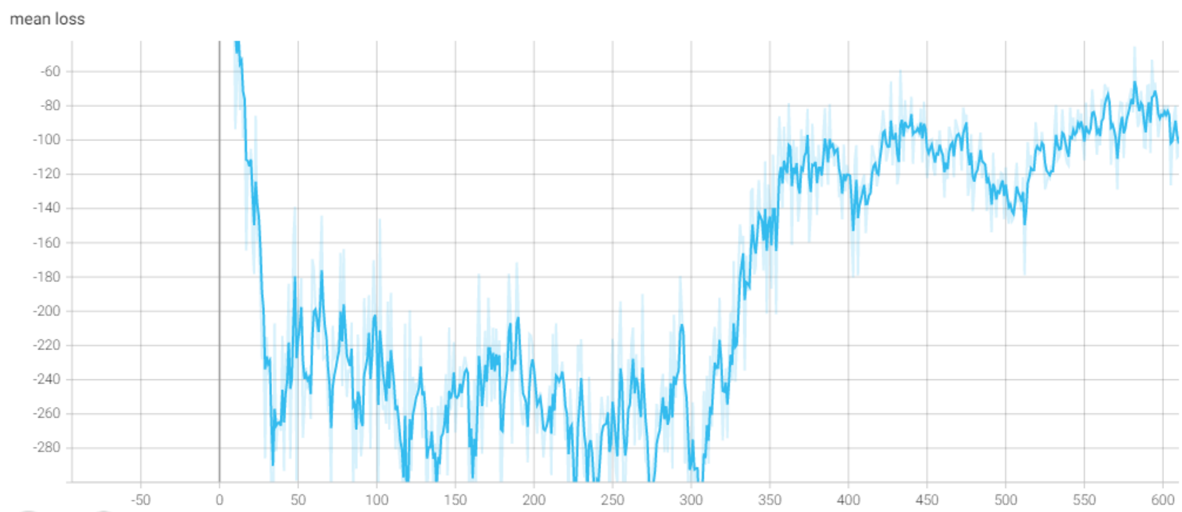
2.1 PG算法结果

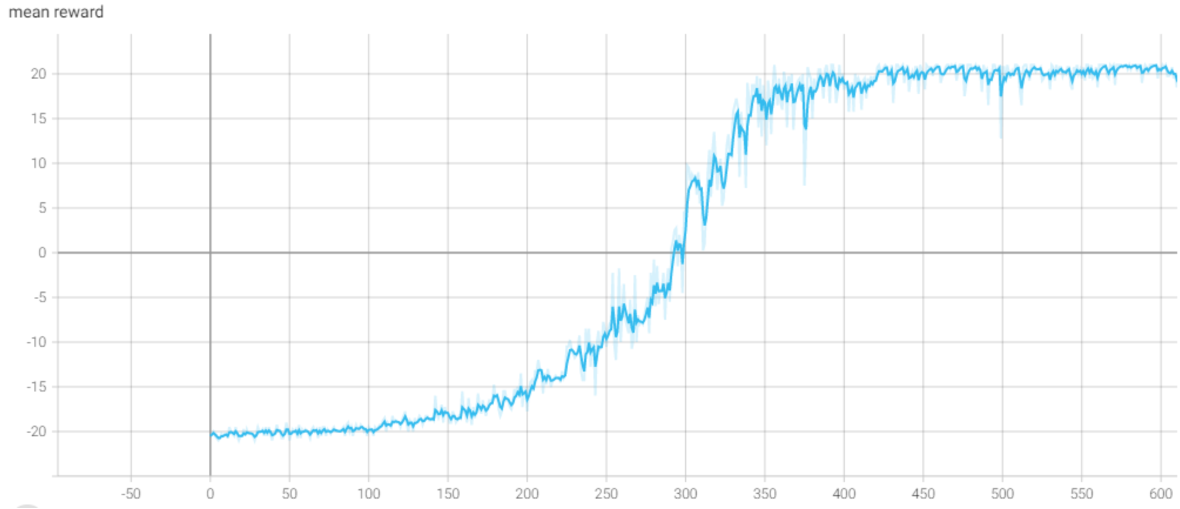




大概训练到400个 batch 的时候就收敛了。

2.2 A2C算法结果





收敛速度跟 PG 算法差不多，也是在400个 batch 的位置差不多达到收敛。

三、两个算法的对比

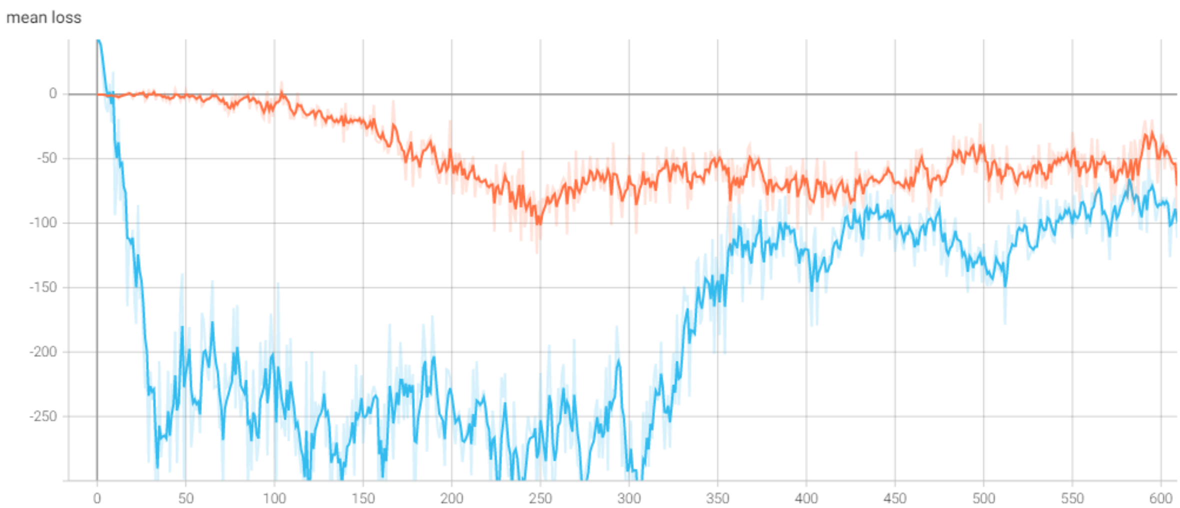
3.1 优缺点对比

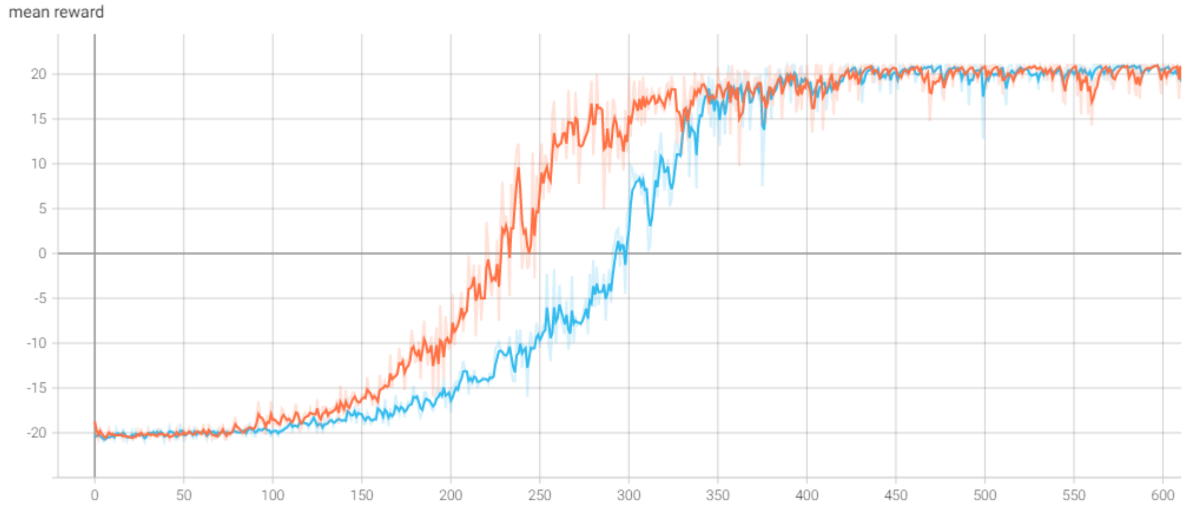
PG算法的缺点：

- 需要一个完整的序列，才能够计算出来 R_t, 是基于Monte Carlo算法的。
- 在一个总奖励的期望较高的序列中，可能存在个别的**很差的动作a**。由于我们优化目标是总奖励尽可能高，优化时会忽视这个问题。 造成的结果就是，为了得到**最优策略**，我们很多次采样来消除个别差的动作干扰。

A2C 算法就可以解决这两个问题。A2C 算法使用 Q(s,a) 来代替 R，这个 Q 值是由神经网络产生的，它本质上就是梯度权值，也可以说是评价梯度的重要性。我们不再需要等完整的序列，只需要把响应的**动作**和**状态**传到 Q 网络（critic）中，询问 Q 值，就可以大致判断出动作好坏。

3.2 结果对比





收敛的速度基本相同，PG 算法的曲线（橙色）在 A2C 算法的曲线（蓝色）上方，它在前期平均 reward 上升较快，但 A2C 算法在 250 个 batch 之后就快速赶上了。