stable-baselines3

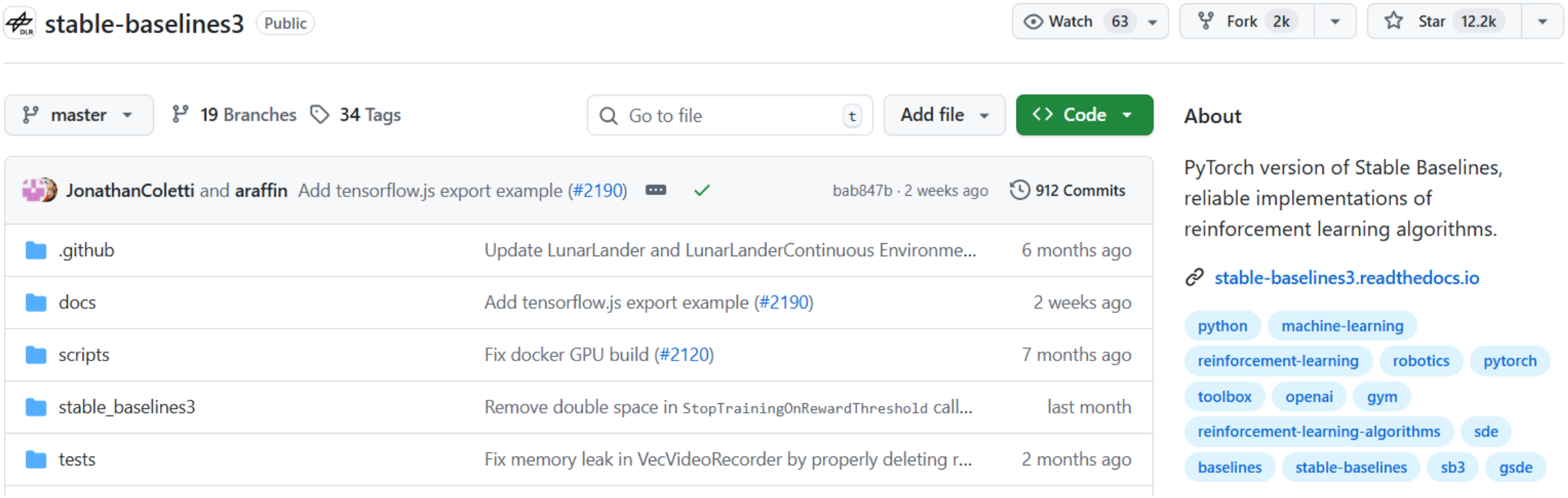
PPO
完整实现

东川路第一可爱猫猫虫

主要内容

- 完整目标函数 价值损失项 熵项
- RolloutBuffer ReplayBuffer
- 广义优势估计计算advantages
- PPO的训练流程
- 优势归一化
- 价值函数的裁剪
- KL散度早停

https://github.com/DLR-RM/stable-baselines3



stable-baselines3/stable_baselines3/ppo at master · DLR-RM/stable-baselines3
stable-baselines3/stable_baselines3/common/buffers.py at master · DLR-RM/stable-baselines3
stable-baselines3/stable_baselines3/common/evaluation.py at master · DLR-RM/stable-baselines3
stable-baselines3/stable_baselines3/common/on_policy_algorithm.py at master · DLR-RM/stable-baselines3
stable-baselines3/stable_baselines3/common/policies.py at master · DLR-RM/stable-baselines3

Clipped Surrogate Objective function

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

- 完整目标函数

  $$L_t^{CLIP+VF+S}(\theta) = \hat{\mathrm{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]$$

- $L_t^{VF}(\theta)$

  平方损失项

  $$\left(V_{\text{pred}}(s_t) - V_t^{target}\right)^2$$

  使价值网络的估计更准确

- $S[\pi_\theta](s_t)$

  熵项

  鼓励探索

$L_t^{VF}(\theta)$如何使价值网络的估计更准确

- $V_{\text{pred}}(s_t)$是当前价值网络的输出
- V_t^{target}是用旧的样本数据计算出的return

  期望回报

  V_t^{target} = advantages + values
- advantages

  由广义优势估计得到
- values

  由旧的价值网络输出
- 训练的时候我们让当前价值网络的输出逼近V_t^{target}

$S[\pi_\theta](s_t)$如何鼓励探索

- 连续空间 PPO里的动作是如何采样得到的

  PPO采用的生成连续动作的技术最早是TRPO这篇论文提出的

  标量连续动作是从**一维高斯分布**中随机采样得到的

  高斯分布由均值和标准差参数化

    均值是策略网络计算的输出

    标准差是选定的超参数并保持固定

  如果想要调整PPO里智能体的探索行为

    好像只能手动调整标准差

The exploration-exploitation dilemma for adaptive agents

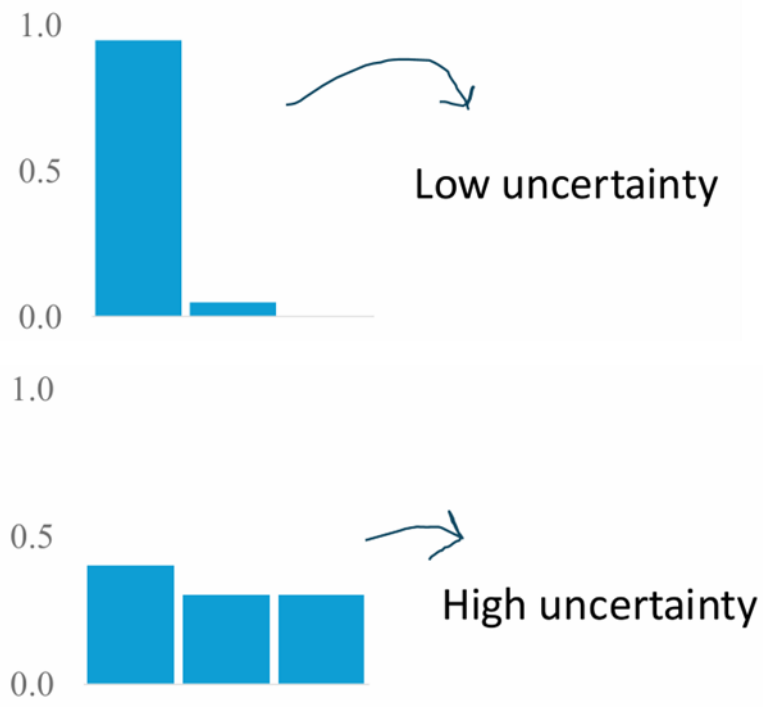
Lilia Rejeb[1], Zahia Guessoum[1,2] and Rym M'Hallah[3]

[1] CReSTIC, MODECO Team, Rue des Crayères, Reims Cedex2, France
[2] Université de Paris-VI, LIP6, OASIS Team, 4 place Jussieu, 75252 cedex 5, France
[3] Kuwait University, Dep. of Statistics and Operations Research
P.O. Box 5969, Safat 13060

- 将高斯分布的标准差
  视为**可训练参数**
- 在训练过程中
  通过随机梯度下降进行调整

```
if isinstance(self.action_space, spaces.Box):
    self.log_std = nn.Parameter(th.zeros(action_dim))
```

离散空间



- PPO依据概率分布采样
- 如何鼓励探索？

  最大探索程度的实现方式是

  为动作空间的所有元素分配相等的概率
- 熵奖励

  让智能体更倾向于对动作空间中所有可用动作生成等概率估计
- 这样智能体就能在遇到的状态里探索更多不同的动作
- SB3的PPO代码里

  离散动作空间和连续动作空间都用了熵项

PPO在训练前做的事情

- RolloutBuffer
  - 存储每个时间步的数据
- 与环境交互
  - 并行采样
  - 将得到的数据存放在RolloutBuffer里
- 这里的采样用的是当前策略（旧策略）
- 采样完成后计算GAE

Rollout Buffer

- PPO的回放缓冲区
  - 用来存放采取当前策略下采集的完整trajectory的数据
  - 后续会用采集到的数据进行多轮训练
  - 数据单次使用，用后即弃
  - 策略更新之后清空缓冲区
- 容量
  - buffer_size = n_steps
  - 固定

另一种回放缓冲区

- ReplayBuffer
- DQN: Experience Replay
- 每次从环境中采样

  数据存放到ReplayBuffer里

  训练的时候从里面随机采样一批出来训练

```python
def add(self, obs, action, reward,
episode_start, value, log_prob):
    # ... 写入数据 ...
    self.pos += 1
    if self.pos == self.buffer_size:
        self.full = True
```

- RolloutBuffer

  容量固定，数据不会循环覆盖

```python
self.pos += 1
if self.pos == self.buffer_size:
    self.full = True
    self.pos = 0
```

- ReplayBuffer

  满了就回到开头，循环覆盖旧数据

```python
def get(self, batch_size: Optional[int] = None):
    assert self.full, ""
```

- RolloutBuffer

  强制要求full=True

  数据满了才能使用

[stable-baselines3/stable_baselines3/common/buffers.py at master · DLR-RM/stable-baselines3]

```python
def reset(self) -> None:
    self.observations = np.zeros(...)
    self.actions = np.zeros(...)
    ...
    self.generator_ready = False
    super().reset()
```

- RolloutBuffer
  - 每次使用后全部置0
  - 丢弃旧数据

```python
def reset(self) -> None:
    self.pos = 0
    self.full = False
```

- RolloutBuffer的父类BaseBuffer
  - 指针归零
  - 状态重置

```python
def sample(self, batch_size: int, env=None):
    upper_bound = self.buffer_size if self.full else self.pos
    batch_inds = np.random.randint(0, upper_bound, size=batch_size)
    return self._get_samples(batch_inds)
```

- ReplayBuffer
  - 未满也能用
  - 随机选，返回抽中的单个batch

RolloutBuffer里面存放什么

```
def reset(self) -> None:
    self.observations = np.zeros((self.buffer_size, self.n_envs, *self.obs_shape),
dtype=self.observation_space.dtype)
    self.actions = np.zeros((self.buffer_size, self.n_envs, self.action_dim),
dtype=self.action_space.dtype)
    self.rewards = np.zeros((self.buffer_size, self.n_envs), dtype=np.float32)
    self.returns = np.zeros((self.buffer_size, self.n_envs), dtype=np.float32)
    self.episode_starts = np.zeros((self.buffer_size, self.n_envs), dtype=np.float32)
    self.values = np.zeros((self.buffer_size, self.n_envs), dtype=np.float32)
    self.log_probs = np.zeros((self.buffer_size, self.n_envs), dtype=np.float32)
    self.advantages = np.zeros((self.buffer_size, self.n_envs), dtype=np.float32)
    self.generator_ready = False
    super().reset()
```

- returns和advantages通过计算得到

  advantages通过广义优势估计

  returns = advantages + values

advantages

```
def compute_returns_and_advantage(self, last_values: th.Tensor, dones: np.ndarray) -> None:
    # 将最后一步的价值估计转为numpy 用于bootstrapping
    last_values = last_values.clone().cpu().numpy().flatten()
    last_gae_lam = 0
    for step in reversed(range(self.buffer_size)):
        if step == self.buffer_size - 1:
            next_non_terminal = 1.0 - dones.astype(np.float32)
            next_values = last_values
        else:
            next_non_terminal = 1.0 - self.episode_starts[step + 1]
            next_values = self.values[step + 1]
        delta = (
            self.rewards[step] +  # r_t
            self.gamma * next_values * next_non_terminal -  # γ * V(s_{t+1}) * (1-done)
            self.values[step]  # V(s_t)
        )
        last_gae_lam = (
            delta +  # δ_t
            self.gamma * self.gae_lambda * next_non_terminal * last_gae_lam  # γλ * A_{t+1}
        )
        self.advantages[step] = last_gae_lam
    self.returns = self.advantages + self.values
```

```
for step in reversed(range(self.buffer_size)):
        if step == self.buffer_size - 1:
            next_non_terminal = 1.0 - dones.astype(np.float32)
            next_values = last_values
        else:
            next_non_terminal = 1.0 - self.episode_starts[step + 1]
            next_values = self.values[step + 1]
```

- dones识别的是

  buffer结束之后环境是否继续

  如果环境继续，用$r_t + \gamma V(s_{t+1}) - V(s_t)$计算$\delta_t^V$

  如果环境终止，即buffer后的下一状态不存在

  则用$r_t - V(s_t)$计算δ_t^V

- done只作用于最后一步

  反向计算的起始步

  `self.buffer_size - 1`

```
for step in reversed(range(self.buffer_size)):
        if step == self.buffer_size - 1:
            next_non_terminal = 1.0 - dones.astype(np.float32)
            next_values = last_values
        else:
            next_non_terminal = 1.0 - self.episode_starts[step + 1]
            next_values = self.values[step + 1]
```

- buffer内部的信号不由dones识别

  由episode_starts识别

- step 不等于 self.buffer_size - 1 的时候

  我们用episode_starts识别是否为完整episode

- 广义优势估计GAE

  作用于单条完整的episode

  episode内部无需mask

- buffer可能是多条episode的拼接

  需要阻止优势迭代时跨越episode

```
for step in reversed(range(self.buffer_size)):
        if step == self.buffer_size - 1:
            next_non_terminal = 1.0 - dones.astype(np.float32)
            next_values = last_values
        else:
            next_non_terminal = 1.0 - self.episode_starts[step + 1]
            next_values = self.values[step + 1]
```

- episode_starts

  识别buffer内部任意步t的下一时刻t+1是否开启新episode

  查询的是`episode_starts[step + 1]`

  episode_starts[t+1]为1

  表示t的下一个时间步是一个新episode的开始

  此时`next_non_terminal`为0

  阻断了优势和价值估计的反向传播

- buffer_size=3, n_envs=2

```
dones = np.array([
    [0., 0.],
    [1., 0.],
    [0., 0.]
], dtype=np.float32)
```

```
episode_starts = np.array([
    [1., 1.],
    [0., 0.],
    [1., 0.]
], dtype=np.float32)
```

- episode_starts[t+1] == dones[t]

  在简单的测试环境里是普遍成立的

- 符合直觉

  当一个episode结束时

  下一个时间步应该是新episode的开始

- 代码证明这一点 只需看字段是如何产生的

  stable_baselines3/common/on_policy_algorithm.py

episode_starts[t+1] == dones[t]

- 从数学的角度来看

  episode_starts和dones是一一对应的

  所以 理论上来说

  只用dones不用episode_starts也能实现GAE

【广义优势估计】GAE的原理 代码实现 Generalized Advantage Estimation

https://github.com/labmlai/annotated_deep_learning_paper_implementations/tree/master/labml_nn/rl/ppo/gae.py

$$\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$$

$$\delta_t + \gamma\lambda\hat{A}_{t+1} = \delta_t + \gamma\lambda(\delta_{t+1} + \gamma\lambda\delta_{t+2} + \cdots) = \hat{A}_t$$

```python
import numpy as np
class GAE:
    def __init__(self, n_workers: int, worker_steps: int, gamma: float, lambda_:
float):
        self.lambda_ = lambda_
        self.gamma = gamma
        self.worker_steps = worker_steps
        self.n_workers = n_workers
    def __call__(self, done: np.ndarray, rewards: np.ndarray, values: np.ndarray) ->
np.ndarray:
        advantages = np.zeros((self.n_workers, self.worker_steps), dtype=np.float32)
        last_advantage = 0
        last_value = values[:, -1]
        for t in reversed(range(self.worker_steps)):
            mask = 1.0 - done[:, t]
            last_value = last_value * mask
            last_advantage = last_advantage * mask
            delta = rewards[:, t] + self.gamma * last_value - values[:, t]
            last_advantage = delta + self.gamma * self.lambda_ * last_advantage
            advantages[:, t] = last_advantage
            last_value = values[:, t]
        return advantages
```

labmlai/annotated_deep_learning_paper_implementations: 60+ Implementations/tutorials of deep learning papers with side-by-side notes; including transformers (original, xl, switch, feedback, vit, ...), optimizers (adam, adabelief, sophia, ...), gans(cyclegan, stylegan2, ...), reinforcement learning (ppo, dqn), capsnet, distillation, ...

annotated_deep_learning_paper_implementations/labml_nn/rl/ppo/gae.py at master · labmlai/annotated_deep_learning_paper_implementations

训练

stable-baselines3/stable_baselines3/ppo/ppo.py at master · DLR-RM/stable-baselines3

```
def train(self) -> None:
    """
    Update policy using the currently gathered rollout buffer.
    """
    self.policy.set_training_mode(True)
    self._update_learning_rate(self.policy.optimizer)
    clip_range = self.clip_range(self._current_progress_remaining)
    if self.clip_range_vf is not None:
        clip_range_vf =
self.clip_range_vf(self._current_progress_remaining)
```

- 学习率随着训练进度逐渐变化
  - 可以线性衰减
    - 与progress_remaining挂钩
  - 也可以是常数或者其他的衰减方式

```
self._current_progress_remaining
= 1.0 - float(num_timesteps) /
float(total_timesteps)
```

```
def train(self) → None:
    """
    Update policy using the currently gathered rollout buffer.
    """
    self.policy.set_training_mode(True)
    self._update_learning_rate(self.policy.optimizer)
    clip_range = self.clip_range(self._current_progress_remaining)
    if self.clip_range_vf is not None:
        clip_range_vf =
self.clip_range_vf(self._current_progress_remaining)
```

- clip_range

  裁剪参数

  它可以是progress remaining的函数

```
ratio = new_policy_prob / old_policy_prob
ratio = th.clamp(ratio, 1 - clip_range, 1 + clip_range)
```

- 默认是常数

  也可以衰减

```
clip_range: Union[float, Schedule] = 0.2
```

```
def train(self) -> None:
    """
    Update policy using the currently gathered rollout buffer.
    """
    self.policy.set_training_mode(True)
    self._update_learning_rate(self.policy.optimizer)
    clip_range = self.clip_range(self._current_progress_remaining)
    if self.clip_range_vf is not None:
        clip_range_vf =
self.clip_range_vf(self._current_progress_remaining)
```

- clip_range_vf
  - 对值函数的裁剪
  - specific to the OpenAI implementation
  - 裁剪的是价值函数的变化量
  - 防止价值函数更新幅度过大
- 默认是不用的
  - 用价值函数裁剪的效果不一定会变好

```python
for epoch in range(self.n_epochs):
    approx_kl_divs = []

    for rollout_data in self.rollout_buffer.get(self.batch_size):
```

- n_epochs为训练的轮次

  初始化的时候默认为10

```python
def __init__( ... , n_epochs: int = 10, ... ):
    self.n_epochs = n_epochs
```

```python
def get(self, batch_size):
    indices = np.random.permutation(self.buffer_size * self.n_envs)
    while start_idx < ...:
        yield self._get_samples(indices[start_idx : start_idx + batch_size])
```

- 采集的时候

  数据应为有序的

  才能计算advantage

- 训练的时候

  每个样本都已经计算好，可以打乱

用新策略评估样本数据

```
values, log_prob, entropy =
self.policy.evaluate_actions(rollout_data.observations, actions)
```

- 用当前策略重新评估旧样本

  生成新的log_probs和values

```
log_prob = distribution.log_prob(actions)
```

- 新的log_probs

  采取当前策略，采样到这些动作的对数概率

  后续用来求重要性采样的ratio

优势归一化

```
advantages = rollout_data.advantages
    if self.normalize_advantage and len(advantages) > 1:
        advantages = (advantages - advantages.mean()) /
(advantages.std() + 1e-8)
```

- 某些advantage的数值会过大或者过小
- 如果 batch_size=1

  标准差是 0

  无法归一化，跳过
- 每个batch内部做优势归一化

```
ratio = th.exp(log_prob - rollout_data.old_log_prob)
```

$$\text{ratio} = \exp(\log \pi_{\text{new}}(a|s) - \log \pi_{\text{old}}(a|s)) = \frac{\pi_{\text{new}}(a|s)}{\pi_{\text{old}}(a|s)}$$

- ratio

  重要性权重

  新策略与旧策略采样到相同动作的概率比值

  新策略觉得这个动作比原来更**可能**了多少

- PPO就是对ratio做裁剪

  来保证策略更新的幅度不要太大

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]$$

clipped surrogate loss

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

```
policy_loss_1 = advantages * ratio
policy_loss_2 = advantages * th.clamp(ratio, 1 - clip_range, 1 + clip_range)
policy_loss = -th.min(policy_loss_1, policy_loss_2).mean()

pg_losses.append(policy_loss.item())
clip_fraction = th.mean((th.abs(ratio - 1) > clip_range).float()).item()
clip_fractions.append(clip_fraction)
```

- `clip_fraction`

  记录被裁剪样本的比例

价值函数的裁剪

```
if self.clip_range_vf is None:
    values_pred = values


else:
    values_pred = rollout_data.old_values + th.clamp(
        values - rollout_data.old_values,
        -clip_range_vf,
        clip_range_vf
    )
```

- 限制价值函数的变化量
- 默认是不使用的

    实践发现效果不一定好

KL散度早停

```
with th.no_grad():
    log_ratio = log_prob - rollout_data.old_log_prob
    approx_kl_div = th.mean((th.exp(log_ratio) - 1) - log_ratio).cpu().numpy()
    approx_kl_divs.append(approx_kl_div)
if self.target_kl is not None and approx_kl_div > 1.5 * self.target_kl:
    continue_training = False
    if self.verbose >= 1:
        print(f"Early stopping at step {epoch} due to reaching max kl:
{approx_kl_div:.2f}")
    break
```

- KL散度衡量新旧策略的差异
- 当KL散度大于阈值

  立刻停止当前 epoch 训练

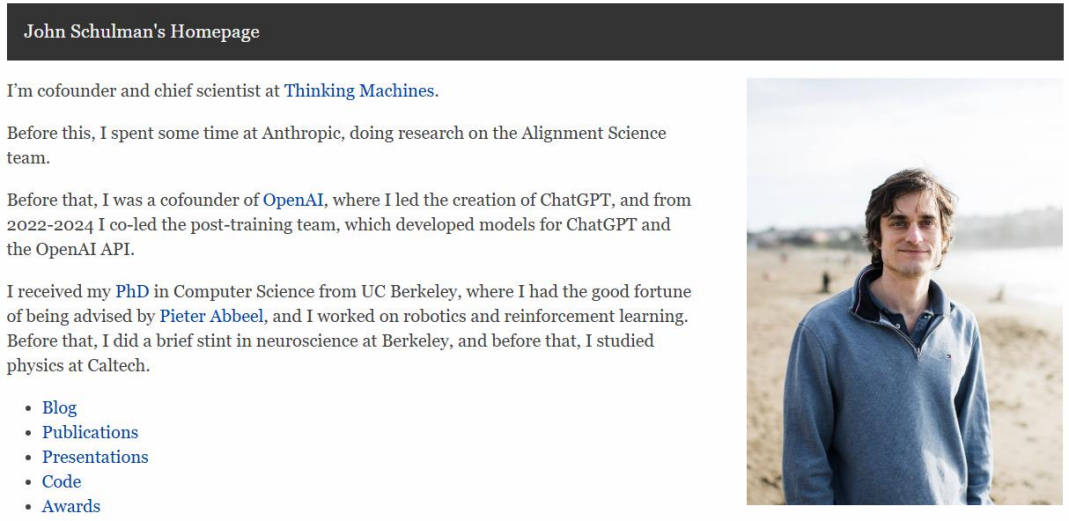
  防止策略更新幅度过大

KL散度的蒙特卡洛估计

- http://joschu.net/blog/kl-approx.html
- 用$r-1-\log r$来估计KL散度

  无偏

  方差小
- DeepSeek GRPO也用了类似的估计方法


John Schulman's Homepage

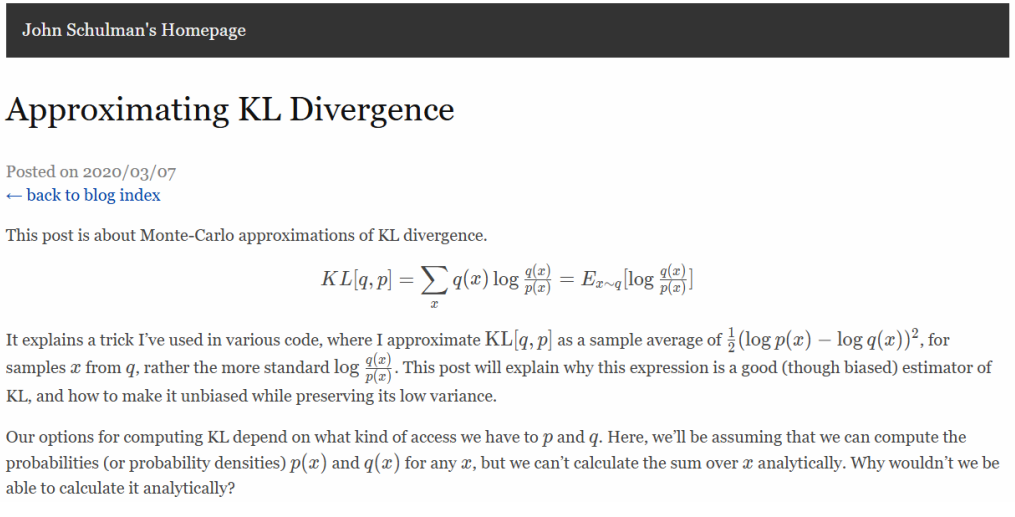
I'm cofounder and chief scientist at Thinking Machines.

Before this, I spent some time at Anthropic, doing research on the Alignment Science team.

Before that, I was a cofounder of OpenAI, where I led the creation of ChatGPT, and from 2022-2024 I co-led the post-training team, which developed models for ChatGPT and the OpenAI API.

I received my PhD in Computer Science from UC Berkeley, where I had the good fortune of being advised by Pieter Abbeel, and I worked on robotics and reinforcement learning. Before that, I did a brief stint in neuroscience at Berkeley, and before that, I studied physics at Caltech.

- Blog
- Publications
- Presentations
- Code
- Awards


John Schulman's Homepage

Approximating KL Divergence

Posted on 2020/03/07
← back to blog index

This post is about Monte-Carlo approximations of KL divergence.

$$KL[q,p] = \sum_x q(x) \log \frac{q(x)}{p(x)} = E_{x\sim q}[\log \frac{q(x)}{p(x)}]$$

It explains a trick I've used in various code, where I approximate $\mathrm{KL}[q,p]$ as a sample average of $\frac{1}{2}(\log p(x) - \log q(x))^2$, for samples x from q, rather the more standard $\log \frac{q(x)}{p(x)}$. This post will explain why this expression is a good (though biased) estimator of KL, and how to make it unbiased while preserving its low variance.

Our options for computing KL depend on what kind of access we have to p and q. Here, we'll be assuming that we can compute the probabilities (or probability densities) $p(x)$ and $q(x)$ for any x, but we can't calculate the sum over x analytically. Why wouldn't we be able to calculate it analytically?