Open in Colab

Spring 2023 6.8200 Computational Sensorimotor Learning Assignment 2

In this assignment, we will implement model-free RL algorithms from scratch to solve `DoorKeyEnv5x5`. We will cover:

- REINFORCE
- Vanilla Policy Gradient (VPG)
- Generalized Advantage Estimation (GAE)
- Proximal Policy Optimization (PPO)

You will need to **answer the bolded questions** and **fill in the missing code snippets** (marked by **TODO**).

There are (approximately) **150** total points to be had in this PSET. `ctrl-f` for "pts" to ensure you don't miss questions.

Please fill in your name below:

Name: Olivia Siegel

Setup

The following code sets up requirements, imports, and helper functions (you can ignore this).

```
!pip3 install -i https://test.pypi.org/simple/ sensorimotor-checker==0.0.8  &>/dev/null
```

```
!pip install gym-minigrid &>/dev/null
```

```
import torch
import torch.nn as nn
from torch.distributions.categorical import Categorical
import torch.nn.functional as F
import gym_minigrid
import numpy as np
import matplotlib.pyplot as plt
import gym
from tqdm.notebook import tqdm
from gym_minigrid.envs.doorkey import DoorKeyEnv
import pandas as pd
import random
from sensorimotor_checker import hw2_tests
```

```
checker_policy_gradient = hw2_tests.TestPolicyGradients()
```

```
# Function from https://github.com/ikostrikov/pytorch-a2c-ppo-acktr/blob/master/model.py
def init_params(m):
    classname = m.__class__.__name__
    if classname.find("Linear") != -1:
        m.weight.data.normal_(0, 1)
        m.weight.data *= 1 / torch.sqrt(m.weight.data.pow(2).sum(1, keepdim=True))
        if m.bias is not None:
            m.bias.data.fill_(0)

def preprocess_obss(obss, device=None):
    if isinstance(obss, dict):
        images = np.array([obss["image"]])
    else:
        images = np.array([o["image"] for o in obss])

    return torch.tensor(images, device=device, dtype=torch.float)

class DoorKeyEnv5x5(DoorKeyEnv):
    def __init__(self):
        super().__init__(size=5)

    def _reward(self):
        """
        Compute the reward to be given upon success
```

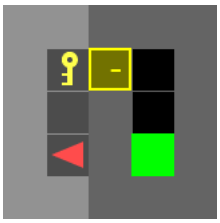
```
        """
        return 1
class Config:
    def __init__(self,
                 score_threshold=0.93,
                 discount=0.995,
                 lr=1e-3,
                 max_grad_norm=0.5,
                 log_interval=10,
                 max_episodes=2000,
                 gae_lambda=0.95,
                 use_critic=False,
                 clip_ratio=0.2,
                 target_kl=0.01,
                 train_ac_iters=5,
                 use_discounted_reward=False,
                 entropy_coef=0.01,
                 use_gae=False):

        self.score_threshold = score_threshold
        self.discount = discount
        self.lr = lr
        self.max_grad_norm = max_grad_norm
        self.log_interval = log_interval
        self.max_episodes = max_episodes
        self.use_critic = use_critic
        self.clip_ratio = clip_ratio
        self.target_kl = target_kl
        self.train_ac_iters = train_ac_iters
        self.gae_lambda=gae_lambda
        self.use_discounted_reward=use_discounted_reward
        self.entropy_coef = entropy_coef
        self.use_gae = use_gae
```

Task (Environment)

In this assignment, we will work with the `DoorKeyEnv5x5` environment from gym_miniworld. This environment is a 5×5 gridworld. The agent needs to pick up the key, open the door, and then go the the green cell. The agent gets a $+1$ reward if it reaches the green cell, and a 0 reward otherwise.

The environment is visually shown below:



```
env = DoorKeyEnv5x5()
```

Question: What does `env.reset()` return? What does each item returned mean? What's the shape of the image in the observation? How about the action space? What does each action mean? (Hint: You may find the source code of gym_minigrid helpful.) (10 pts)

Answer: env.reset() returns a tuple where the first element is an integer-value image and the second element is a direction valued from 1 to 4. The shape of the image in the observtion is 7x7x3. The action space is in the superclass minigrid. Each action has a meaning where left means left move, right means right move, forward means forward move. Pickup means picking up an obkect and dropping means dropping. Toggle means activating an object and done is when the agent is done with the task.

left = 0 right = 1 forward =2 pickup = 3 drop = 4 toggle = 5 done = 6

Model

In Deep Q-Learning, we estimated the value function for each (state, action) pair. In policy gradients, we will directly learn a policy: i.e, for each state, predict an action! We call this policy network the *actor*.

Our *actor* will take in as input the `DoorKeyEnv5x5` observation (a 7x7x3 image), and output a categorical distribution over all possible actions. To choose an action, we will sample from this distribution. We suggest implementing the actor network to contain a few convolutional layers, followed by a few fully-connected layers.

In addition to the actor network, later questions in the PSET require estimating the value network, called the *critic*. The critic estimates total future reward, much like DQN in PSET 3, but is notably *on-policy*, meaning it's reward estimates are conditioned on the actor. We will use the critic to reduce variance in the policy gradient estimate. We will get to that soon.

We have provided you a reference architecture to use for your actor-critic networks. Note that we have separate networks for each that do not share weights, as this has been shown to empirically improve performance. Also note that the policy outputs a `torch.distributions.categorical.Categorical` object.

```
class ACModel(nn.Module):
    def __init__(self, num_actions, use_critic=False):
        super().__init__()
        self.use_critic = use_critic

        # Define actor's model
        self.image_conv_actor = nn.Sequential(
            nn.Conv2d(3, 16, (2, 2)),
            nn.ReLU(),
            nn.MaxPool2d((2, 2)),
            nn.Conv2d(16, 32, (2, 2)),
            nn.ReLU(),
            nn.Conv2d(32, 64, (2, 2)),
            nn.ReLU()
        )
        self.actor = nn.Sequential(
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, num_actions)
        )

        # Define critic's model
        if self.use_critic:
            self.image_conv_critic = nn.Sequential(
                nn.Conv2d(3, 16, (2, 2)),
                nn.ReLU(),
                nn.MaxPool2d((2, 2)),
                nn.Conv2d(16, 32, (2, 2)),
                nn.ReLU(),
                nn.Conv2d(32, 64, (2, 2)),
                nn.ReLU()
            )
            self.critic = nn.Sequential(
                nn.Linear(64, 64),
                nn.Tanh(),
                nn.Linear(64, 1)
            )

        # Initialize parameters correctly
        self.apply(init_params)

    def forward(self, obs):
        conv_in = obs.transpose(1, 3).transpose(2, 3) # reshape into expected order

        dist, value = None, None

        x = self.image_conv_actor(conv_in)
        embedding = x.reshape(x.shape[0], -1)

        x = self.actor(embedding)
        dist = Categorical(logits=F.log_softmax(x, dim=1))

        if self.use_critic:
            y = self.image_conv_critic(conv_in)
            embedding = y.reshape(y.shape[0], -1)

            value = self.critic(embedding).squeeze(1)
        else:
            value = torch.zeros((x.shape[0], 1), device=x.device)

        return dist, value
```

Model Evaluation

The following code runs the actor critic model `acmodel` for one episode, and returns a dictionary with all the relevant information from the rollout. It relies on placeholders below for `compute_advantage_gae` and `compute_discounted_return`: you can ignore these for now, and just evaluate through to the next section. However, it might be useful to review this code just to make sure you understand what's going on.

```
def compute_advantage_gae(values, rewards, T, gae_lambda, discount):
    advantages = torch.zeros_like(values)

    #### TODO: populate GAE in advantages over T timesteps (10 pts) ############
    last = 0
    for t in reversed(range(T)):
      delta = rewards[t] + discount*values[t+1] - values[t]
      last = delta + (discount*gae_lambda) * last
      advantages[t] = last

    ############################################################################

    return advantages[:T]

#### Test GAE ####
checker_policy_gradient.test_general_advantage_estimation(compute_advantage_gae)


def compute_discounted_return(rewards, discount, device=None):
    returns = torch.zeros(*rewards.shape, device=device)

    #### TODO: populate discounted reward trajectory (10 pts) ############
    for term in range(len(rewards)):
      sum = 0
      for reward in range(len(rewards)-term):
        sum += rewards[reward+term]*(discount**reward)
        if reward + 1 == len(rewards)-term:
          returns[term] = sum
    ######################################################################
    return returns

#### Test discounted return ####
checker_policy_gradient.test_compute_discounted_return(compute_discounted_return)


def collect_experiences(env, acmodel, args, device=None):
    """Collects rollouts and computes advantages.
    Returns
    -------
    exps : dict
        Contains actions, rewards, advantages etc as attributes.
        Each attribute, e.g. `exps['reward']` has a shape
        (self.num_frames, ...).
    logs : dict
        Useful stats about the training process, including the average
        reward, policy loss, value loss, etc.
    """


    MAX_FRAMES_PER_EP = 300
    shape = (MAX_FRAMES_PER_EP, )

    actions = torch.zeros(*shape, device=device, dtype=torch.int)
    values = torch.zeros(*shape, device=device)
    rewards = torch.zeros(*shape, device=device)
    log_probs = torch.zeros(*shape, device=device)
    obss = [None]*MAX_FRAMES_PER_EP

    obs, _ = env.reset()

    total_return = 0

    T = 0

    while True:
        # Do one agent-environment interaction

        preprocessed_obs = preprocess_obss(obs, device=device)
```

```
        with torch.no_grad():
            dist, value = acmodel(preprocessed_obs)
        action = dist.sample()[0]


        obss[T] = obs
        obs, reward, done, _, _ = env.step(action.item())


        # Update experiences values
        actions[T] = action
        values[T] = value
        rewards[T] = reward
        log_probs[T] = dist.log_prob(action)


        total_return += reward
        T += 1

        if done or T>=MAX_FRAMES_PER_EP - 1:
            break

    discounted_reward = compute_discounted_return(rewards[:T], args.discount, device)
    exps = dict(
        obs = preprocess_obss([
            obss[i]
            for i in range(T)
        ], device=device),
        action = actions[:T],
        value  = values[:T],
        reward = rewards[:T],
        advantage = discounted_reward-values[:T],
        log_prob = log_probs[:T],
        discounted_reward = discounted_reward,
        advantage_gae=compute_advantage_gae(values, rewards, T, args.gae_lambda, args.discount)
    )

    logs = {
        "return_per_episode": total_return,
        "num_frames": T
    }

    return exps, logs
```

REINFORCE

Now comes the fun part! Using the `collect_experiences` function and `ACModel`, we will implement vanilla policy gradients. The following function takes in an `optimizer`, `ACModel`, batch of experience `sb`, and some arguments `args` (see `Config` in setup for fields and default values), and should perform a policy gradients parameter update using the observed experience.

Fill in todos below to implement vanilla policy gradients (20 pts).

```
def compute_policy_loss_reinforce(logps, returns):
    policy_loss = torch.tensor(0)

    #### TODO: complete policy loss (10 pts) ###
    policy_loss = -torch.sum(torch.dot(logps,returns))/returns.shape[0]
    ############################################

    return policy_loss

#### Test policy loss for REINFORCE algorithm ####
checker_policy_gradient.test_compute_policy_loss_reinforce(compute_policy_loss_reinforce)


def update_parameters_reinforce(optimizer, acmodel, sb, args):

    logps, reward = None, None

    ### TODO: compute logps and reward from acmodel, sb['obs'], sb['action'], and sb['reward'] ###
    ### If args.use_discounted_reward is True, use sb['discounted_reward'] instead. ##############
    ### (10 pts) ########################################
    distribution, val = acmodel.forward(sb['obs'])
    logps = distribution.log_prob(sb['action'])
```

```
    reward = sb['reward']
    if args.use_discounted_reward :
      reward = sb['discounted_reward']


    ############################################################################################

    policy_loss = compute_policy_loss_reinforce(logps, reward)
    update_policy_loss = policy_loss.item()

    # Update actor-critic
    optimizer.zero_grad()
    policy_loss.backward()

    # Perform gradient clipping for stability
    for p in acmodel.parameters():
        if p.grad is None:
            print("Make sure you're not instantiating any critic variables when the critic is not used")
    update_grad_norm = sum(p.grad.data.norm(2) ** 2 for p in acmodel.parameters()) ** 0.5
    torch.nn.utils.clip_grad_norm_(acmodel.parameters(), args.max_grad_norm)
    optimizer.step()

    # Log some values
    logs = {
        "policy_loss": update_policy_loss,
        "grad_norm": update_grad_norm
    }

    return logs
```

Now, let's try to run our implementation. The following experiment harness is written for you, and will run sequential episodes of policy gradients until `args.max_episodes` timesteps are exceeded or the rolling average reward (over the last 100 episodes is greater than `args.score_threshold`. It is expected to get highly variable results, and we'll visualize some of this variability at the end.

The method accepts as arguments a `Config` object `args`, and a `parameter_update` method (such as `update_parameters_reinforce`).

```
def run_experiment(args, parameter_update, seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    env = DoorKeyEnv5x5()

    acmodel = ACModel(env.action_space.n, use_critic=args.use_critic)
    acmodel.to(device)

    is_solved = False

    SMOOTH_REWARD_WINDOW = 50

    pd_logs, rewards = [], [0]*SMOOTH_REWARD_WINDOW

    optimizer = torch.optim.Adam(acmodel.parameters(), lr=args.lr)
    num_frames = 0

    pbar = tqdm(range(args.max_episodes))
    for update in pbar:
        exps, logs1 = collect_experiences(env, acmodel, args, device)
        logs2 = parameter_update(optimizer, acmodel, exps, args)

        logs = {**logs1, **logs2}

        num_frames += logs["num_frames"]

        rewards.append(logs["return_per_episode"])

        smooth_reward = np.mean(rewards[-SMOOTH_REWARD_WINDOW:])

        data = {'episode':update, 'num_frames':num_frames, 'smooth_reward':smooth_reward,
                'reward':logs["return_per_episode"], 'policy_loss':logs["policy_loss"]}

        if args.use_critic:
            data['value_loss'] = logs["value_loss"]
```

```
            pd_logs.append(data)

            pbar.set_postfix(data)

            # Early terminate
            if smooth_reward >= args.score_threshold:
                is_solved = True
                break

        if is_solved:
            print('Solved!')

        return pd.DataFrame(pd_logs).set_index('episode')
```

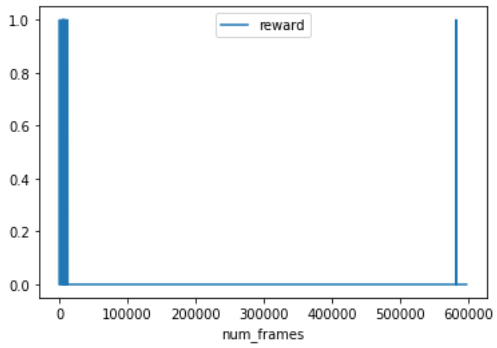
Run Reinforce

Great! Now let's run our implementation, and see how we do.

```
args = Config()
df = run_experiment(args, update_parameters_reinforce)

df.plot(x='num_frames', y='reward')
```

```
100%                 2000/2000 [41:09<00:00, 1.15s/it, episode=1999, num_frames=597044, smooth_reward=0, reward=0, policy_loss=-]
<AxesSubplot:xlabel='num_frames'>
```



REINFORCE with Discounted Reward

Uh oh! Even after 300,000 steps, our policy does not converge. One reason for failure is the way rewards are generated in the real-world. In an ideal world, the agent would be rewarded at every timestep in a manner that perfectly corresponded to the quality of the action taken in a particular state. However, this is rarely the case; for example, in Doorkey we only get reward at the very end of the episode (i.e., the sparse reward scenario).

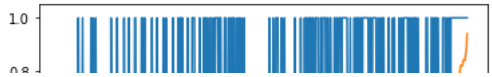
In DQN, we tackle this with a discount factor `gamma` on future rewards. In policy gradients, we'll simply rewrite all of our step rewards to be discounted from the past episode reward.

Fill in `compute_discounted_return` code block above, then run code cell below to see the effect of discounted reward trajectories. This should converge, so if it doesn't, you've made an error (although try re-running the cell with a few different seeds to make sure it's not an error). (10 pts)

```
args = Config(use_discounted_reward=True)
df = run_experiment(args, update_parameters_reinforce)

df.plot(x='num_frames', y=['reward', 'smooth_reward'])
```

```
24%  485/2000 [07:40<06:56, 3.64it/s, episode=485, num_frames=113831, smooth_reward=0.94, reward=1, policy_loss=1.27]
Solved!
<AxesSubplot:xlabel='num_frames'>
```



Vanilla Policy Gradients

You may have noticed that the REINFORCE training curve is extremely unstable. It's time to bring in our *critic*! We can prove from the Expected Grad-Log-Prob (EGLP) lemma that we can subtract any function $b(x)$ from our reward without changing our policy in expectation:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)\right)\right].$$

Baseline Proof

Question: Prove that adding a baseline doesn't change the policy in expectation using the EGLP lemma. This can be a loose proof as long as you convey the intutition. (10 pts)

Proof:

Empirically, using the on-policy value function as the baseline (b) reduces variance in the policy gradient sample estimate, leading to faster and more stable learning. We can estimate the b using an L2 loss to the true rewards (or in our case, the discounted rewards), and constitutes an additional loss term in the overall objective. The baseline substracted return term, $R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)$ is already computed for you, and is referred to as the *advantage*.

```
def compute_policy_loss_with_baseline(logps, advantages):
    policy_loss = 0

    ### TODO: implement the policy loss (5 pts) ######
    policy_loss = -torch.sum(torch.dot(logps,advantages))/advantages.shape[0]
    ################################################

    return policy_loss

#### Test discounted return ####
checker_policy_gradient.test_compute_policy_loss_with_baseline(compute_policy_loss_with_baseline)


def update_parameters_with_baseline(optimizer, acmodel, sb, args):

    def _compute_value_loss(values, returns):
        value_loss = 0

        ### TODO: implement the value loss (5 pts) #######
        loss = nn.MSELoss()
        value_loss = loss(values,returns)
        ################################################

        return value_loss

    logps, advantage, values, reward = None, None, None, None

    ### TODO: populate the policy and value loss computation fields using acmodel, sb['obs'], sb['action], and sb['discounted_rewa
    ### For the advantage term, use sb['advantage_gae'] if args.use_gae is True, and sb['advantage'] otherwise.
    ### 10 pts
    distribution, val = acmodel.forward(sb['obs'])
    logps = distribution.log_prob(sb['action'])
    reward = sb['reward']
    if args.use_discounted_reward :
      reward = sb['discounted_reward']
    if args.use_gae:
      advantage = sb['advantage_gae']
    else:
      advantage = sb['advantage']
    values = val

    ###############################################################################################
```

```
    policy_loss = compute_policy_loss_with_baseline(logps, advantage)
    value_loss = _compute_value_loss(values, reward)
    loss = policy_loss + value_loss

    update_policy_loss = policy_loss.item()
    update_value_loss = value_loss.item()

    # Update actor-critic
    optimizer.zero_grad()
    loss.backward()
    update_grad_norm = sum(p.grad.data.norm(2) ** 2 for p in acmodel.parameters()) ** 0.5
    torch.nn.utils.clip_grad_norm_(acmodel.parameters(), args.max_grad_norm)
    optimizer.step()

    # Log some values

    logs = {
        "policy_loss": update_policy_loss,
        "value_loss": update_value_loss,
        "grad_norm": update_grad_norm
    }

    return logs
```

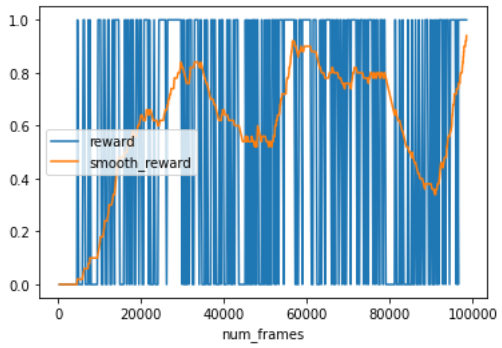
Run REINFORCE with baseline

If you did everything right, you should be able to run the below cell to run the vanilla policy gradients implementation with baseline. This should be somewhat more stable than without the baseline, and likely converge faster.

```
args = Config(use_critic=True)
df_baseline = run_experiment(args, update_parameters_with_baseline)

df_baseline.plot(x='num_frames', y=['reward', 'smooth_reward'])
```

```
27%                                547/2000 [06:57<05:02, 4.81it/s, episode=547, num_frames=98477, smooth_reward=0.94, reward=1, policy_loss=1.33, valu
Solved!
<AxesSubplot:xlabel='num_frames'>
```



Reinforce with GAE

The advantage we computed above seemed to work, and hopefully improved our results! Fortunately, we can do even better. The paper Generalized Advantage Estimation describes a nifty method for building a strong advantage estimate (see formula 16 in the paper) that empirically outperforms a naive subtraction (and includes reward shaping).

Fill in the `compute_advantage_gae` method above according to the formula in the paper (10 pts), and then run the below cell. GAE should further improve convergence time and stability.

```
args = Config(use_critic=True, use_gae=True)
df_gae = run_experiment(args, update_parameters_with_baseline)

df_gae.plot(x='num_frames', y=['reward', 'smooth_reward'])
```

Proximal Policy Optimization

Our work is not yet done! There are some surprisingly powerful additional tweaks we can make to our GAE implementation to further improve performance.

The current standard in policy gradients today is Proximal Policy Optimization, which improves on GAE by taking multiply policy update steps per minibatch, enabled by policy update clipping (this is a specific variant called *PPO-Clip*). This leads to greater sample efficiency, as larger steps can be taken from the same data samples.

We've implemented most of PPO for you: all that's left for you are the policy and value loss computations (note that you'll have to evaluate the `acmodel` each time you compute them). Note that for the policy loss, we also ask that you return the approximate KL divergence between the new and old action distributions notated as `approx_kl`; this is used to facilitate an early stopping condition in policy updates. This blog post shares a simple formula for approximating KL divergence that you can use.

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

where $$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0. \end{cases}$$

A commonly used technique is to add entropy regularization for policy gradient methods as shown in equation 2 of this paper, and there is some discussion of entropy regularization in this reference tutorial. You should compute the policy loss as defined above and also add in an entropy term for the updated policy (note that we've provided to you an entropy coefficient in `args.entropy_coef`).

```
def update_parameters_ppo(optimizer, acmodel, sb, args):
    def _compute_policy_loss_ppo(obs, old_logp, actions, advantages):
        policy_loss, approx_kl = 0, 0

        ### TODO: implement PPO policy loss computation (30 pts).  #######
        dist, _ = acmodel(sb['obs'])
        logp = dist.log_prob(sb['action'])
        ratio = torch.exp(logp-old_logp)
        first = advantages*ratio
        second = torch.clamp(ratio, 1-args.clip_ratio, 1+args.clip_ratio)*advantages
        loss = -torch.min(first, second).mean() - args.entropy_coef*dist.entropy().mean()
        approx_kl = torch.mean(old_logp - logp).item()
        ##################################################################

        return policy_loss, approx_kl

    def _compute_value_loss(obs, returns):
        ### TODO: implement PPO value loss computation (10 pts) ##########

        ##################################################################
        _, values = acmodel(sb['obs'])
        value_loss = F.mse_loss(values.reshape((1,-1)), returns.reshape((1,-1)))
        return value_loss


    dist, _ = acmodel(sb['obs'])
    old_logp = dist.log_prob(sb['action']).detach()

    advantage = sb['advantage_gae'] if args.use_gae else sb['advantage']
```

```
    policy_loss, _ = _compute_policy_loss_ppo(sb['obs'], old_logp, sb['action'], advantage)
    value_loss = _compute_value_loss(sb['obs'], sb['discounted_reward'])

    for i in range(args.train_ac_iters):
        optimizer.zero_grad()
        loss_pi, approx_kl = _compute_policy_loss_ppo(sb['obs'], old_logp, sb['action'], advantage)
        loss_v = _compute_value_loss(sb['obs'], sb['discounted_reward'])

        loss = loss_v + loss_pi
        if approx_kl > 1.5 * args.target_kl:
            break

        loss.backward(retain_graph=True)
        optimizer.step()

    update_policy_loss = policy_loss.item()
    update_value_loss = value_loss.item()

    logs = {
        "policy_loss": update_policy_loss,
        "value_loss": update_value_loss,
    }

    return logs


args = Config(use_critic=True, use_gae=True)
df_ppo = run_experiment(args, update_parameters_ppo)

df_ppo.plot(x='num_frames', y=['reward', 'smooth_reward'])
```

```
  0%                                          0/2000 [00:02<?, ?it/s]
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-48-b3d54863f874> in <module>
      1 args = Config(use_critic=True, use_gae=True)
----> 2 df_ppo = run_experiment(args, update_parameters_ppo)
      3
      4 df_ppo.plot(x='num_frames', y=['reward', 'smooth_reward'])

                                  1 frames
<ipython-input-47-e28b0e69e73e> in update_parameters_ppo(optimizer, acmodel, sb,
args)
     44         optimizer.step()
     45
---> 46     update_policy_loss = policy_loss.item()
     47     update_value_loss = value_loss.item()
     48

AttributeError: 'int' object has no attribute 'item'

SEARCH STACK OVERFLOW
```

Fancy Plots

If you've gotten to this point, congrats: you've succesfully implemented REINFORCE, VPG, GAE, and PPO! While we've been able to anecdotally compare their performance, we don't have any sense of *scientific rigor*. Notably, given the variance you've likely seen between runs of these models, a single run may not reflect how strong a model really is.

For this problem, train each of these 4 methods using multiple seeds (at least 5, but more if you feel you need them). Then, generate a high quality reward curve plot comparing each algorithm. The plot should be clean and legible, and clearly demonstrate the performance and variance of each of the approaches. As an example, see Figure 3 of the PPO paper (although we're only evaluating on a single environment).

Be creative and make something pretty: it matters for good science!

Note: you should leave a few hours for this to run.

+ Code + Text

Survey (bonus points, 10 pts)

Please fill out this anonymous survey and enter the code below to receive credit. Thanks!

Code:

```
variance_reduction
```

2s completed at 8:37 PM