HW1 Notes

Sunday, March 31, 2024 8:24 PM

Installation: https://stackoverflow.com/questions/60268769/gyms-box-2d-openai-doesnt-install-successfully-pip-error

- This worked for me:
  - conda create -c conda-forge -n gymenv swig pip
  - conda activate gymenv
  - pip install Box2D gym
  - pip install pygame
  - pip install "gymnasium[box2d]"
  - pip install "gymnasium[classic-control]"
  - Conda install mujoco==2.3.7
- I think I need pygame as a prerequisite to install the other packages

Following the openai gym tutorial on inverted pendulum

Policy, neural network (NN)

- Linear layers: torch.nn, nn.Linear(in_features, out_features, bias=true, device=None, dtype=None)
- So create a network with nn.Sequential(, then Linear, Tanh, Linear, Tanh
- "Tanh is used as a non-linearity between the hidden layers" --> sets value to between -1 and 1, see graph of tanh
- This NN is to estimate a mean and std dev from a normal distribution that the action is sampled from
- First is the shared net with 2 hidden layers, then do a mean net and a std dev net that take outputs from the shared NN
- For "forward" function, takes in x which is torch.Tensor, observation of environment, as well as self, the program
- Returns the mean and std dev of a normal distribution, from which an action is sampled from
- It runs the shared net to output "shared_features", then runs mean net to output "action_means" and std dev net to output "action_stddevs"
  - Note: stddev is torch.log(1+ torch.exp(self.policy_stddev_net(shared_features)))
  - Not sure why it's 1+e^stddev…

REINFORCE agent (see algorithm on the website)

- Init takes in the dimension of observation and action space, as well as self
- Sets hyperparameters of self: learning rate, discount factor, small number for stability
- Create vector for probability values of sampled action, and corresponding rewards
- Self.net is set to the policy network class created above
- Then set an optimizer torch.optim.AdamW(self.net.parameters(), lr=self.learning_rate)
  - Input parameters of the net, input learning rate
- AdamW: https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html
  - Optimization algorithm to optimize the parameters of the policy net
- Sample action function takes in self and observation of environment "state", type np.ndarray, returns float action to be performed
- Set state to be torch.tensor(np.array[state]), convert to right format
- Use the self.net to calculate action_means and action_stddevs
  - Only argument is state? So it calls self.net which is defined above to be Policy_Network(obs_space_dims,action_space_dims) which are both inputs to init.

    - And those get put through the Policy_Network class and used in shared_net, and other nets
- After, create a normal dist from mean and std dev using Normal(), adding self.eps small number for stability
- Action is random sample (distrib.sample()) and also return log of the probability of the probability density function at that action
- Then convert using action.numpy(), which turns tensor object into numpy.ndarray
- Append probability to the self.probs, then return the action
- Then update the policy network's weights:
- Iterate over items R in self.rewards[::-1] which is just the rewards vector backwards
- Running_g = R + discount*running_g ==> add discounted returns to reward vector each item
- Gs.insert(0,running_g) ==> insert value into gs vector
- Set deltas to be torch.tensor(gs) and loss = 0
- Define loss function as sum of -1 * prob * reward, for all probs and rewards
- Update policy network with self.optimizer.zero_grad() [resets gradients in torch.Tensor], loss.backward [calculates gradient during backward pass of NN], self.optimizer.step() [updates parameters]
- Then empty out the variables probs and rewards

Some Notes

- Sample_action(obs) calls self.net(state), which calls Policy_Network(obs_space_dims,action_space_dims) class, which runs __init__ and assigns "shared_net" NN to self, assigns "policy_mean_net" and "policy_stddev_net" to self as well
- THIS ONLY HAPPENS ONCE
- Then, after running __init__, "forward" is called
- And each time afterwards, whenever sample_action(obs) is called, "forward" is called returning sample mean and std dev

Actually training REINFORCE

- Procedure overview:
    - For seed in random seeds
        - Reinitialize agent
        - For episode in range of max number of episodes
            - Sample action based on observation
            - Take action and receive reward and next observation
            - Store action take, probability, and observed rewards
        - Update policy
- Note: test various seeds, RL is brittle concerning random seed in common uses cases
- First create the environment
- And create wrapped environment, which records episode statistics: gym.wrappers.RecordEpisodeStatistics(env, 50)
- 5000 episodes via int(5e3)
- Set observation space dimensions and action space dimensions of the current environment, which is set to InvertedPendulum, and called via env.observations_space.shape[0] and likewise with actions
    - So environment defines a space of actions to take and observations
- Start rewards_over_seeds empty array
- Set seed with three things seed setters?
- Reinitialize agent = REINFORCE(obs_space_dims, action_space_dims) and reset rewards_over_seeds
- For episode in range of max episodes:
- Set seed and reset environment
- Sample action using action = agent.sample_action(obs)
- Take a step in the environment: returns obs, reward, terminated, truncated, info

    - See documentation on each variable type
- Then append reward to agent.reward
- End when terminated or truncated
    - Truncated: episode reaches max number of timesteps
    - Terminated: state space values no longer finite
- Append to reward_over_episodes: wrapped_env.return_queue[-1]
    - Which seems to be the reward queue output from the environment wrapper
    - The environment wrapper also returns episode length via wrapped_env.length_queue
- Update agent with agent.update() [no arguments]
- Every 1000 episodes, print episode and average reward
- Then append reward_over_episodes to rewards_over_seeds for each episode

I don't get:

- Super().__init__()
- In update(self), how the discounted return works, running_g = R + self.gamma*running_g, then put that value in gs via gs.insert(0, running_g)
- And then why the deltas is torch.tensor of gs
- What is wrapped environment?
    - https://gymlibrary.dev/api/wrappers/
    - Answered above:
- Seed setting?
- Where is the reward defined? Where is the goal defined? Is it just for the inverted pendulum?
    - Based on each environment
    - Can alter reward w/ wrapper or manually before sending back.
- **Can I change this goal to be, for instance, swinging at a certain speed?**
    - Yes! I kind of did it

At least the code works!