Exercise 4

Sheetal Borar - 915263
ELEC-E8125 - Reinforcement Learning

October 26, 2020

1 Task 1

I implemented policy gradient with the following parameters-

- 1a.) Basic REINFORCE without baseline

```python
    def episode_finished(self, episode_number):
        action_probs = torch.stack(self.action_probs, dim=0) \
                .to(self.train_device).squeeze(-1)
        rewards = torch.stack(self.rewards, dim=0).to(self.train_device).squeeze(-1)
        self.states, self.action_probs, self.rewards = [], [], []

        # TODO: Compute discounted rewards (use the discount_rewards function)
        d_rewards = discount_rewards(rewards, self.gamma)

        # TODO: Compute the optimization term (T1)
        loss = -torch.mean(action_probs * d_rewards)

        # TODO: Compute the gradients of loss w.r.t. network parameters (T1)
        loss.backward()

        # TODO: Update network parameters using self.optimizer and zero gradients (T1)
        self.optimizer.step()
        self.optimizer.zero_grad()
    def get_action(self, observation, episode_number, evaluation=False):
        x = torch.from_numpy(observation).float().to(self.train_device)

        # TODO: Pass state x through the policy network (T1)
        action_dist = self.policy.forward(x, episode_number)

        # TODO: Return mean if evaluation, else sample from the distribution
        if(evaluation):
            action = action_dist.mean
        else:
            action = action_dist.sample()

        # TODO: Calculate the log probability of the action (T1)
        act_log_prob = action_dist.log_prob(action)

        return action, act_log_prob
```

- 1b.) REINFORCE with a constant baseline b = 20

```python
    b = 20
    loss = -torch.mean(action_probs * (d_rewards - b))
```

- 1c.) REINFORCE with discounted rewards normalized to zero mean and unit variance

```python
    reward_mean = torch.mean(d_rewards)
    reward_std = torch.std(d_rewards)
    norm_rewards = (d_rewards - reward_mean)/reward_std
    # TODO: Compute the optimization term (T1)
    loss = -torch.mean(action_probs * norm_rewards)
```

Figure 1 shows the training plats for 1a, 1b and 1c.

1.1 Question 1.1

A good baseline is helpful in reducing the variance in the gradient update. Adding a baseline should not change the expected value of the update, but it should help to reduce the variance. A good baseline should be able to change according the state value, in some states all actions gets high reward so the baseline should be high to clearly indicate the difference in reward

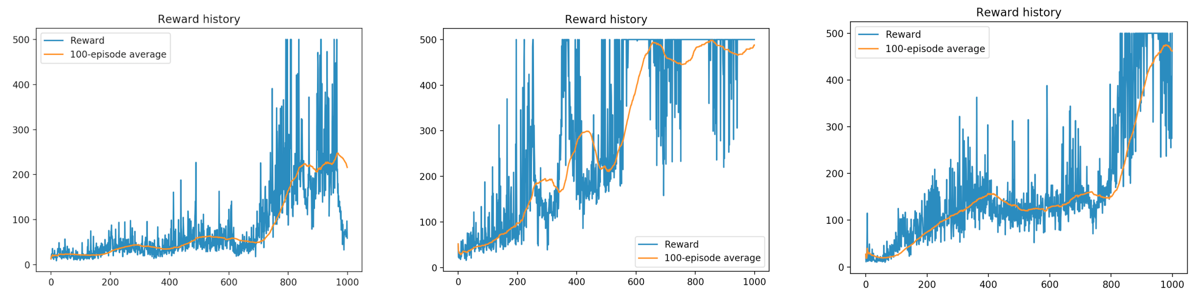

Figure 1: Learning curves for 1a ,1b ,1c from left to right

for each action. Actually a state value function estimation is a good choice for a baseline as it does not depend on action.

1.2 Question 1.2

Baseline helps to reduce the variance of the gradient update. High variance can slow down the learning process, by providing conflicting descent direction to the model. Adding a baseline helps reduce the variance and hence leads to faster convergence.

2 Task 2

- 2a.) I set initial sigma = 10 in the model init function -

  ```
          self.sigma = torch.tensor([10]).float()
  ```

  I changed sigma as follows in the policy.forward function -

  ```
          c = torch.tensor([0.0005])
          k = torch.tensor([episode_number])
          sigma = self.sigma * (torch.exp(-c * k))
  ```

- 2b.) I set initial sigma = 10 in model init function and put sigma in nn.Parameter so it gets updated with every gradient ascent step -

  ```
          self.sigma = torch.nn.Parameter(torch.tensor([10]).
              float())
  ```

  I changed sigma as follows in the policy.forward function -

  ```
          sigma = self.sigma
  ```

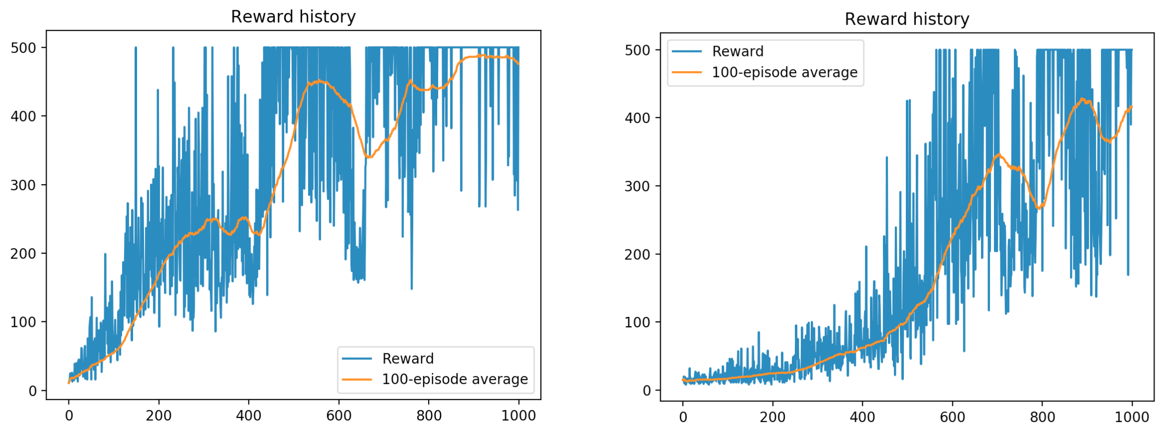

Figure 2: Learning curves for 2a and 2b from left to right

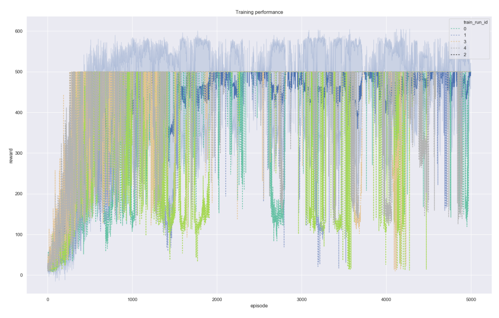

Figure 3: Learning curve for 2b multiple cartpole runs

Figure 2 shows the learning plots for 2a, 2b. Figure 3 shows the learning plots 2b for multiple cartpole runs.

2.1 Question 3.1

Variance in the Gaussian policy helps to add some noise to the choice of action to ensure that in case of partially observable scenarios, the policy does not get stuck between some states. It adds some stochasticity to the policy.

- Constant variance - During the later stages of training when the model has already learnt a near optimal policy, having the constant variance can lead to unnecessary exploration. Hence, this approach leads to unstable results.
- Decaying variance - As the policy learns and improves, adding noise to the actions become less useful and hence decaying variance helps the policy choose the optimal action more often towards the later learning stages, leading to more stable results. But if the model gets stuck in some local maxima towards the later stages with with low variance, it become hard to get out of the local maxima.

- Learned Variance - This is a smarter way of updating the variance in choosing action, as this is basing the change in variance on the gradient update rather than decaying the variance continuously. This would allow the variance to increase in the middle of training, if the policy gets stuck in some local maxima and help it get out of the maxima. But this approach is sensitive to initial variance selection as mentioned in 3.2.

2.2 Question 3.2

Learned variance approach is sensitive to initial variance selection. If the initial variance is too low, the policy does not explore enough and it could get stuck in some local maxima. Whereas if the initial variance is too high, it takes unnecessary actions due to which the learning rate is too slow. If we give it enough episodes, it will be able to bring the variance to a level at which it can learn better though.

2.3 Question 4.1

The method implemented in this exercise was REINFORCE, which is a Monte Carlo based on- policy method. On-policy method cannot be used directly with experience replay, because our current parameters are different to those used to generate the samples in the replay. We are taking actions as per another policy, and checking the rewards from the current policy.
We can however use experience replay with off-policy policy gradient methods, where we can use importance sampling to get the probability distribution of actions as per our current policy using the other policy.

2.4 Question 4.2

As mentioned in 4.1, if we use experience replay with REINFORCE, which is a Monte Carlo based on-policy method, there is a mismatch as we are taking actions as per another policy and trying to optimize the returns from running trajectories from our current policy. This makes convergence difficult when use on-policy methods with experience replay.

2.5 Question 5.1

Unbounded action space means that an action could be to increase speed to +inf or -inf as an example. Due to stochastic policy, it could sometimes choose extreme actions like go to position 10000, or go at speed 100000, which could break the system in a physical environment.

2.6 Question 5.2

Even when the action space is unbounded, it is possible to control the variance, which can make sure that the agent does not deviate too much from the mean action (optimal action) in a state and as the reward is given for staying alive, the pole will try to stay within the bounds if we keep the variance of choosing action low. We could also change the reward function to give negative rewards for some state action pairs, but this is similar to putting hard limits on the actions through the reward function.

2.7 Question 6.1

Policy gradient methods can be applied to discrete action spaces, if we use softmax policy which can return a probability for choosing each action.

The problem could arise if we have a large number of discrete actions, in those cases the neural network approach would struggle to choose the right action due the high dimensionality of the action space.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[1] [?]