Implementation of Reinforce Algorithm
AE598 - HW3

Maulik Bhatt *
Department of Aerospace Engineering, UIUC

The Reinforce algorithm is a well-known approach in reinforcement learning, used to learn a policy for a stochastic policy gradient problem. It is a popular algorithm for solving tasks where the agent needs to learn a policy to take actions in a dynamic environment to maximize a cumulative reward over time. The Reinforce algorithm uses a Monte Carlo approach to estimate the gradients of the policy function, which allows it to handle non-differentiable policies. Reinforce is a simple yet powerful algorithm that has been successfully applied in various domains, such as robotics, game playing, and natural language processing. In the assignment, I have applied the reinforce algorithm on a gird world environment.

I. REINFORCE

We express the control problem of dynamical systems as a reinforcement problem which is controlling a Markov Decision Process(MDP). We achieve this by a type of policy gradient algorithm called Reinforce. The key idea behind policy gradient algorithm is to estimated the value of gradient of cost function with the respect to policy parameter(θ) using data. In standard Reinforce, we do this using Monte-Carlo approach. The pseudo-code for the same is given below.

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)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Fig. 1 Pseudo Code

*NetID: mcbhatt2

Following are the details of hyperparameters and various components of the code

A. Tabular Policy

We consider a tabular policy to implement Reinforce on the grid world environment. In order to do so, we use categorical distribution over policy parameters to compute probabilties of taking action.

```
def policy(s, theta):
    logits = torch.nn.functional.softmax(theta[s,:],dim=0)
    return torch.distributions.categorical.Categorical(logits=logits)

def action(s, theta):
    return policy(s,theta).sample().item()
```

Fig. 2 Policy

B. Generating episodes

Given a policy, we use the following code to generate the episodes which essentially choose actions according to the policy given.

```
def generate_episode(env, theta):
    T = env.max_num_steps
    states = torch.zeros(T+1, dtype = int)
    actions = torch.zeros(T+1, dtype = int)
    rewards = torch.zeros(T+1)
    states[0] = env.reset()
    done = False
    t = 0
    actions[t] = action(states[t].item(),theta)
    while not done:
        t+=1
        (states[t], rewards[t], done) = env.step(actions[t-1])
        actions[t] = action(states[t].item(),theta)
    return states, actions, rewards
```

Fig. 3 Episode Generation

C. Loss function

We define the loss function as shown in the pseudo code.

```
def loss(G, theta, t, s, a, gamma):
    return -(gamma**t)*G*policy(s,theta).log_prob(torch.tensor([a]))
```

Fig. 4 Loss function

D. REINFORCE Algorithm

The REINFORCE algorithm as in pseudo code is as below. We start with a random policy, obtain the data by generating the episode and then update the policy by gradient decent using the optimizer given.

```
def reinforce(env, optim, n_episodes, gamma=1):
    n_s = env.num_states
    n_a = env.num_actions
    T = env.max_num_steps
    theta_0 = torch.rand(n_s,n_a)
    theta = theta_0.requires_grad_(requires_grad=True)
    reward_array = torch.zeros(n_episodes)

    if optim == 'SGD':
        optimizer = torch.optim.SGD([theta], lr = 1e-2)
    elif optim == 'Adam':
        optimizer = torch.optim.Adam([theta], lr = 1e-2)

    for n in range(n_episodes):
        if n%1000 == 0:
            lr = 5e-3
        states, actions, rewards = generate_episode(env, theta)
        for t in range(T):
            G = 0
            for k in range(t+1,T,1):
                G+= gamma**(k-t-1)*rewards[k]
            optimizer.zero_grad()
            loss_value = loss(G, theta, t, states[t], actions[t], gamma)
            loss_value.backward()
            optimizer.step()
        reward_array[n] = rewards.sum()
    return theta, reward_array
```

Fig. 5 REINFORCE

E. List of hyper-parameters

Hyperparameter	Value
Number of episodes	2000
gamma(γ)	1
Learning rate(initially)	1e-2
Learning rate after 1000 episodes	5e-3

I observed that decreasing learning rate after 1000 epochs was improving the performance

II. Results

A. Learning curves

As we can see, reward is increasing over number of episodes but varinace is very high as expected for a Monte-Carlo reinforce

Fig. 6 Learning Curve for easy grid world with SGD optimizer

Furthermore, the reward obtained from hard grid world environment is lower because of the randomized action part in the environment.

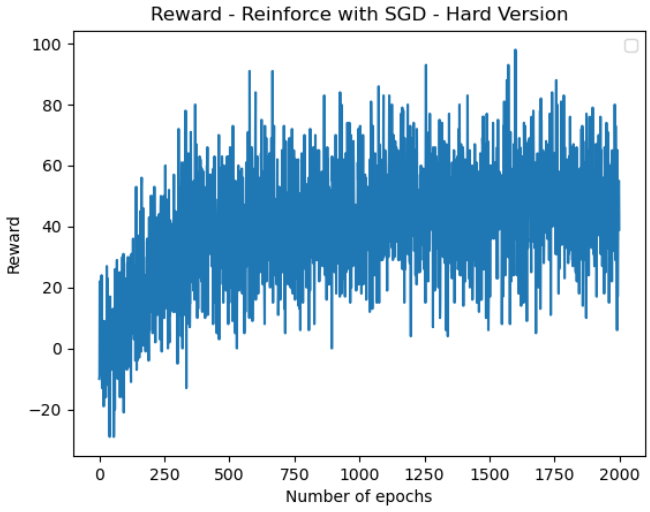

Fig. 7 Learning Curve for hard grid world with SGD optimizer

When using adam, the rewards obtained are almost similar to the SGD optimizer.

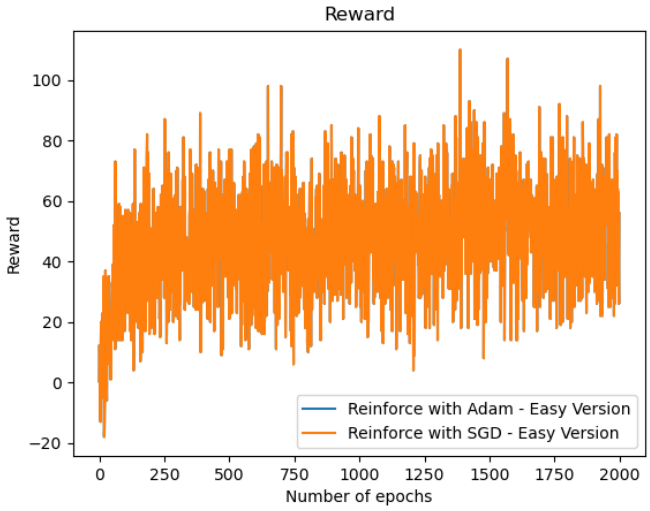

Fig. 8 Learning Curve for easy grid world with Adam optimizer

Fig. 9 Learning Curve for hard grid world with Adam optimizer

B. Policy

The policies for SGD and Adam optimizers for both the type of environments are shown below. As we can see, because of very high variance the policies generated are also random and not very close to optimal while yielding good rewards.

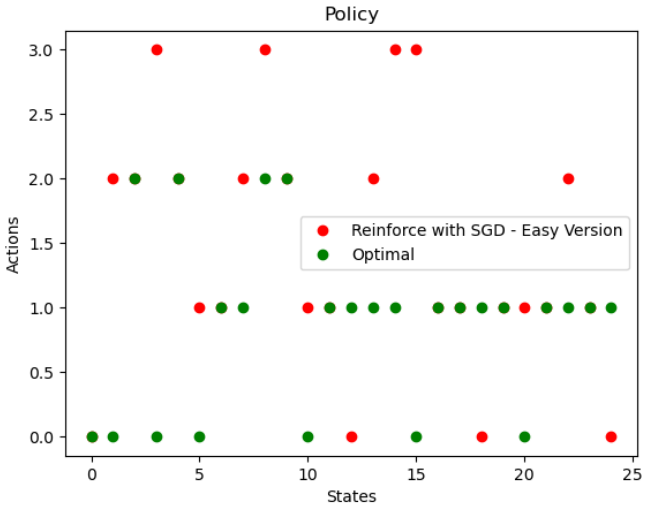

Fig. 10 Policy for easy grid world with SGD optimizer

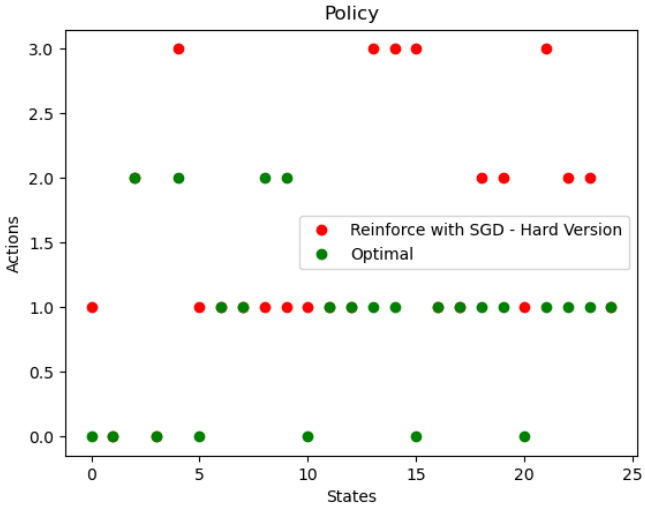

Fig. 11 Policy for hard grid world with SGD optimizer

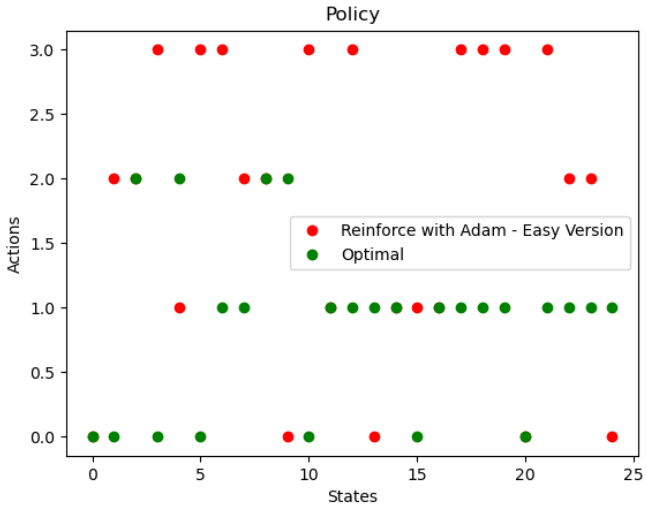

Fig. 12 Policy for easy grid world with Adam optimizer

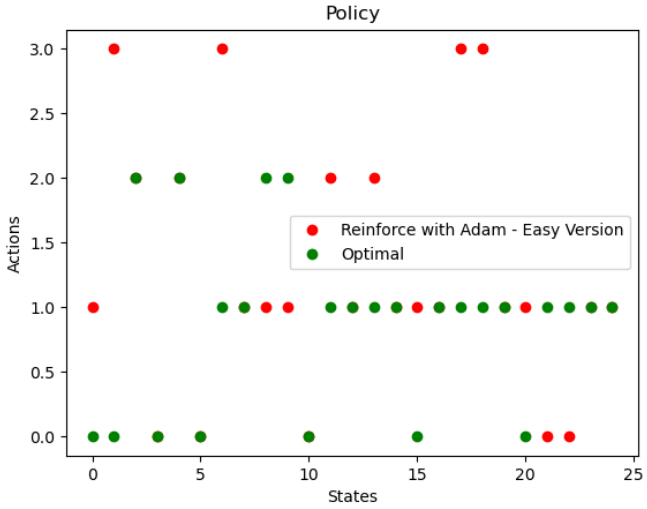

Fig. 13 Policy for hard grid world with Adam optimizer

III. Conclusion

In conclusion, we implemented REINFORCE for this homework. The key learnings were as follows,

- The REINFORCE algorithm has very high variance therefore we need to use variance reduction techniques to get better results.
- The impact of optimizer may not be apparent for a small problem like this but it may be give different results for larger problems.