Natural Actor-Critic

May 16, 2021

1 Natural Actor-Critic for Long-Run Average Reward

Import packages.

[17]:
```
import gym
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.optim as optim
import torch.nn as nn
from torch.distributions import Categorical

import dual_sourcing
from utility import convergence_test, evaluate
from TBS_module import TBS
```

Set up configurations and make an instance.

[7]:
```
CONFIG = {'Lr': 2, 'Le': 1, 'cr': 100, 'ce': 105, 'lambda': 2,
          'h': 1, 'b': 19, 'starting_state': [0]*4, 'max_order': 4,␣
 ↪'max_inventory': 10
env = gym.make('DualSourcing-v0', config=CONFIG)
env.seed(0)
```

[7]:
```
0
```

Define policy network class and value network class.

[10]:
```
# Not share the parameters

state_size = env.observation_space.shape[0]
action_space = env.nA
m = env.max_order + 1

class Policy(nn.Module):
    # policy network
    def __init__(self, hidden_size1, hidden_size2):
        super(Policy, self).__init__()
```

```
        self.fc1 = nn.Linear(state_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, action_space)

        self.relu = nn.ReLU()
        self.action_output = nn.Softmax(dim = 0)

        self.save_actions = []

    def forward(self, s):
        out = self.fc1(s)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)

        action_probs = self.action_output(out)

        return action_probs


class Value(nn.Module):
    # value network
    def __init__(self, hidden_size1, hidden_size2):
        super(Value, self).__init__()
        self.fc1 = nn.Linear(state_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.relu = nn.ReLU()

        self.value_output = nn.Linear(hidden_size2, 1)

    def forward(self, s):
        out = self.fc1(s)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)

        value = self.value_output(out)

        return value
```

Construct our networks

[11]:
```
policy = Policy(4, 8)
value = Value(4, 4)
```

Define auxiliary functions to initialize network, sample action, and return NN policy.

[13]:
```
def init_weights(layer):
    # initialize network parameters
    if type(layer) == nn.Linear:
        nn.init.xavier_uniform_(layer.weight)
        layer.bias.data.fill_(0.01)

def sample_action(state):
    # sample action at current state according to policy network
    state = torch.from_numpy(state).float()
    action_probs = policy(state)

    dist = Categorical(action_probs)
    action = dist.sample()

    action = np.asarray([action.item() // m, action.item() % m])
    return action

def nn_policy(env):
    return sample_action(env.state)
```

Use supervised learning to approximate a TBS policy as our initialization.

[20]:
```
np.random.seed(0)
torch.manual_seed(100)
policy.apply(init_weights)

a = 0.01
maxit_init = 10000
for i in range(maxit_init):
    s = np.random.rand(4)
    s = np.floor(s * np.asarray([5]*3 + [21]))
    s[-1] -= 10
    env.state = s

    qr, qe = TBS(env, 2, 2)
    action = 5 * qr + qe
    action = int(action)
    probs = policy(torch.from_numpy(s).float())

    policy.zero_grad()
    loss = (probs ** 2).sum() - 2 * probs[action]
    loss.backward()

    for name, layer in policy.named_modules():
        if type(layer) == nn.Linear:
            layer.weight.data -= a * layer.weight.grad
            layer.bias.data -= a * layer.bias.grad
```

Assess initial policy performance.

[18]:
```
evaluate(env, 100, 5000, nn_policy, env)
```

[18]:
```
(-217.985780210962, 2.382745749028857)
```

[ ]:
```
# reset environment
env.reset()

state = env.state
maxit = 500
total_reward = 0
J = 0
z = 0
Lambda = 0

m = env.max_order + 1

# initialize value network
value.apply(init_weights)

w = {}
for name, layer in policy.named_modules():
    if type(layer) == nn.Linear:
        w[name+'_weight'] = torch.zeros(layer.weight.size())
        w[name+'_bias'] = torch.zeros(layer.bias.size())

z = {}
for name, layer in value.named_modules():
    if type(layer) == nn.Linear:
        z[name+'_weight'] = torch.zeros(layer.weight.size())
        z[name+'_bias'] = torch.zeros(layer.bias.size())

# step size
alpha0 = 1e-1
beta0 = 1e-2
alphac = 1000
betac = 100000
c = 0.8

# use multiple epochs of TD learning
TD_epochs = 10

for i in range(maxit):

    # step size calculations
    alpha = alpha0 * alphac / (alphac + i**(2/3))
```

```
    beta = beta0 * betac / (betac + i)
    xi = c * alpha

    # draw action
    state = torch.from_numpy(state).float()
    action_probs = policy(state)
    dist = Categorical(action_probs)
    action = dist.sample()

    # get next state & reward
    next_state, reward, _, _ = env.step(np.asarray([action.item() // m, action.
↪item() % m]))
    next_state = torch.from_numpy(next_state).float()

    # value nn back prop
    value.zero_grad()
    value_loss = value(state)
    value_loss.backward()

    # TD learning with multiple epochs
    state_prime = np.asarray(state, dtype = int).copy()
    state_prime = torch.from_numpy(state_prime).float()
    next_state_prime = np.asarray(next_state, dtype = int).copy()
    next_state_prime = torch.from_numpy(next_state_prime).float()
    delta = 0
    for j in range(TD_epochs):

        # update average reward
        J = J + xi * (reward - J)
        # update temporal difference
        delta = delta + reward - J + value(next_state_prime).item() -␣
↪value(state_prime).item()

        # enter next state
        state_prime = np.asarray(next_state_prime, dtype = int).copy()
        state_prime = torch.from_numpy(state_prime).float()

        # draw action
        action_probs = policy(state_prime)
        dist = Categorical(action_probs)
        action = dist.sample() # 0, 1, ..., 24

        # draw next state and collect reward
        next_state_prime, reward, _, _ = env.step(np.asarray([action.item() //␣
↪m, action.item() % m]))
        next_state_prime = torch.from_numpy(next_state_prime).float()
```

```
    # !! reset environment state back to where we started
    env.state = next_state

    # average TD
    delta = delta / TD_epochs

    # critic update
    stepsize = alpha * delta
    for name, layer in value.named_modules():
        if type(layer) == nn.Linear:
            z[name+'_weight'] = Lambda * z[name+'_weight'] + layer.weight.grad
            z[name+'_bias'] = Lambda * z[name+'_bias'] + layer.bias.grad
            layer.weight.data += stepsize * z[name+'_weight']
            layer.bias.data += stepsize * z[name+'_bias']

    # actor update
    psi_wt = 0
    for name, layer in policy.named_modules():
        if type(layer) == nn.Linear:
            psi_wt += (layer.weight.grad * w[name+'_weight']).sum().item()
            psi_wt += (layer.bias.grad * w[name+'_bias']).sum().item()

    stepsize = alpha * (delta - psi_wt)
    for name, layer in policy.named_modules():
        if type(layer) == nn.Linear:
            w[name+'_weight'] += stepsize * layer.weight.grad
            w[name+'_bias'] += stepsize * layer.bias.grad
            layer.weight.data += beta * w[name+'_weight']
            layer.bias.data += beta * w[name+'_bias']

            layer.weight.data = torch.clamp(layer.weight.data, min=-1, max=1)
            layer.bias.data = torch.clamp(layer.bias.data, min=-1, max=1)

    # transition to next state
    state = np.asarray(next_state, dtype = int).copy()

    print('\n')
```

Assess output policy.

[23]:
```
evaluate(env, 100, 5000, nn_policy, env)
```

[23]:
```
(-218.25038327404772, 1.9000560543719498)
```

1.1 Conclusion

The algorithm does not provide significant improvement upon the initial policy. We suspect that the value function is relative in the sense that its accuracy relies on the accuracy of the estimate of the long-run average reward, making it highly unstable and extremely difficult to learn.

In the future, it might be worthwhile to investigate how to adaptively tune the step sizes and employ other tricks to stablize the algorithm and boost learning.