Assignment 5 AC Report

1 Introduction

Actor-Critic method reduces the variance in Monte-Carlo policy gradient by directly estimating the action-value function and is able to tackle the environment with continuous action space. However, a naive application of AC method with neural network approximation is unstable for challenging problem. Asynchronous Advantage Actor-Critic (A3C) uses parallel actor-learners which shows a stabilizing effect on training. In A3C, there are several instances of local agent and a global agent. Instead of experience replay, all the local agents asynchronously executed in parallel. The parameter of the global agent is updated by all the local experience.

In this experiment, I implement A3C algorithm and train the agent in a classical RL continuous control scenarios, Pendulum. The inverted pendulum swing up problem is a classic problem in the control literature. In gym's implementation of the problem, the pendulum starts in a random position with a random speed, and the goal is to swing it up so it stays upright. The observation we get is a three dimensional vector about pendulum's position and angular velocity, and the action is a continuous force between -2.0 and 2.0.

2 Procedure

2.1 ACNet

First, I implement a class called `ACNet` , where I build actor and critic network. Actor net has one hidden layer and two output layers. The output layer, σ^2 and μ, act as the parameter of normal distribution, which sample the final action. Critic net has one hidden layer and outputs the estimation of value function.

It's worth noting that actor and critic network don't share any parameters. Supplementary material of original paper recommend this setup in continuous control problems.

```python
def __init__(self, state_space, action_space):
    super(ACNet, self).__init__()
    self.state_dim, self.action_dim = state_space, action_space
    # Actor net
    self.p1 = nn.Linear(state_space, 200)
    self.mu = nn.Linear(200, action_space)     # parameter mu of normal distribution
    self.sigma = nn.Linear(200, action_space)  # parameter sigma of normal distribution
    self.distribution = torch.distributions.Normal
    # Critic net
    self.v1 = nn.Linear(state_space, 100)
    self.value = nn.Linear(100, 1)
    # initialization
    for layer in [self.p1, self.mu, self.sigma, self.v1, self.value]:
        nn.init.normal_(layer.weight, mean=0.0, std=0.1)
        nn.init.constant_(layer.bias, 0.0)
```

Then I also follow the setup of the original paper. In actor network, μ is modeled by a linear layer and σ 2 by a SoftPlus operation, $\log(1+\exp(x))$, as the activation computed as a function of the output of a linear layer. Value function is also modeled by linear layers.

```
def forward(self, x):
    p1 = nn.functional.relu6(self.p1(x))
    mu = 2 * torch.tanh(self.mu(p1))
    sigma = nn.functional.softplus(self.sigma(p1)) + 0.001  # avoid 0
    v1 = nn.functional.relu6(self.v1(x))
    values = self.value(v1)
    return mu, sigma, values
```

Function `choose_action()` uses distribution to sample an action and this step isn't trainable.

```
def choose_action(self, state):
    self.training = False
    mu, sigma, value = self.forward(state)  # shape of mu/sigma: (1, action_dim)
    action = self.distribution(mu.view(1, ).data, sigma.view(1, ).data).sample()
    return action.numpy()
```

Final loss consists of loss of actor net and loss of critic net. Loss of critic net is the square of n-step TD error, which also acts as advantage function $A(s_t, a_t; \theta, \theta_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$.

Then, use function `log_prob(action)` of distribution and advantage function to get the loss of actor net, that is $\log(\pi_\theta(s_t, a_t))A(s_t, a_t; \theta, \theta_v)$. Here, a differential entropy of the normal distribution is used to encourage exploration, which is defined by the output of the actor network, that is $-\frac{1}{2}(\log(2\pi\sigma^2)+1)$.

```
def calculate_loss(self, states, actions, v_t):
    self.train()
    mu, sigma, values = self.forward(states)
    td_error = v_t - values
    a_loss = td_error.pow(2)    # loss for Actor net

    m = self.distribution(mu, sigma)
    entropy = 0.5 + 0.5 * math.log(2 * math.pi) + torch.log(m.scale)  # exploration
    exp_v = m.log_prob(actions) * td_error.detach() + 0.005 * entropy
    c_loss = -exp_v             # loss for Critic net
    total_loss = (a_loss + c_loss).mean()
    return total_loss
```

2.2 Worker

In A3C, there are many workers which copy global network's parameters, intersect with environment , and use the experience to compute gradient and update global models.

Class `Worker` is derived from `torch.multiprocessing.Process`. Every worker has their own environment and local network, but share optimizer, global number of episode, global episode reward(Given multi-processing, episode rewards are unstable, so we average several consecutive episode rewards.), result queue(for average episode rewards). `max_ep` is the max number of total episode. `max_ep_step` is the max number of step of an episode. `g_update_iter` is the frequency of updating global network.

```python
class Worker(mp.Process):
    def __init__(self, env_n, global_net, opt, global_ep, global_ep_r, res_queue,
                 idx, max_ep=3000, max_ep_step=300, g_update_iter=5, gamma=0.9):
        super(Worker, self).__init__()
        self.env = gym.make(env_n).unwrapped
        self.l_net = ACNet(self.env.observation_space.shape[0], self.env.action_space.shape[0])
        self.g_net, self.opt = global_net, opt
        self.g_ep, self.g_ep_r, self.res_queue = global_ep, global_ep_r, res_queue
        self.name, self.gamma = 'worker%i' % idx, gamma
        self.max_ep, self.max_ep_step, self.g_update_iter = max_ep, max_ep_step, g_update_iter
        self.buffer_s, self.buffer_a, self.buffer_r = [], [], []
```

When worker runs, it will start an episode if the global number of episode is less than the maximum. During the episode, it uses local network to interact with its own envronment and then stores results in buffer. When it reaches the frequency of global updating, it updates global network parameters and copies them back and clears the old buffer.

```python
def run(self):
    while self.g_ep.value < self.max_ep:
        s = self.env.reset()
        ep_r, done, t = 0.0, False, 1
        while not done:
            a = self.l_net.choose_action(torch.tensor(s[None, :]).float())
            s_, r, done, _ = self.env.step(a.clip(-2, 2))  # action range:[-2, 2]
            if t == self.max_ep_step:   done = True
            ep_r += r
            self.buffer_a.append(a)
            self.buffer_s.append(s)
            self.buffer_r.append((r + 8.1) / 8.1)    # normalize
            if t % self.g_update_iter == 0 or done:  # update global and assign to local net
                self.push_and_pull(done, s_)         # sync with global net
                self.buffer_s, self.buffer_a, self.buffer_r = [], [], []
            s = s_
            t += 1
        self.record(ep_r)
    self.res_queue.put(None)
```

Function `push_and_pull()` calculates n-step TD target and total loss, and then calculates gradient, pushes the local gradient to global network, and then global network is updated. The optimizer's parameters are global network's parameters, so we can only pass local net's gradients rather than parameters and then load parameters from global network.

```python
def push_and_pull(self, done, s_):
    v_s_ = 0 if done else self.l_net.forward(torch.tensor(s_[None, :]).float())[-1].data.numpy()[0, 0]
    buffer_v_target = []
    for r in self.buffer_r[::-1]:  # reverse buffer r
        v_s_ = r + self.gamma * v_s_
        buffer_v_target.append(v_s_)
    buffer_v_target.reverse()
    loss = self.l_net.calculate_loss(torch.tensor(np.vstack(self.buffer_s)).float(),
        torch.tensor(np.vstack(self.buffer_a)).float(),
        torch.tensor(np.array(buffer_v_target)[:, None]).float())
    # calculate local gradients and push local parameters to global
    self.opt.zero_grad()
    loss.backward()
    for lp, gp in zip(self.l_net.parameters(), self.g_net.parameters()):
        gp._grad = lp.grad
    self.opt.step()
    self.l_net.load_state_dict(self.g_net.state_dict())    # copy global parameters
```

Function `record()` used at the end of each episode in `run()` aims to update those global variables. Because of multi-processing, we need to use `get_lock()` to avoid conflicts. Given that multi-processing brings unstable episode rewards, I use `g_ep_r` , a moving average reward which can be roughly seen as the mean of ten consecutive episode rewards.

```python
def record(self, ep_r):
    with self.g_ep.get_lock():
        self.g_ep.value += 1
    with self.g_ep_r.get_lock():
        if self.g_ep_r.value == 0.:
            self.g_ep_r.value = ep_r
        else:
            self.g_ep_r.value = self.g_ep_r.value * 0.8 + ep_r * 0.2
    self.res_queue.put(ep_r)
```

2.3 Adam Optimizer

The original paper found that sharing parameter of optimizer brought better performance, so we need to modify the optimizer and share some parameters. I also do experiments about not sharing parameters which can be found in part 5.

```python
class SharedAdam(torch.optim.Adam):
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.99), eps=1e-8,
                 weight_decay=0):
        super(SharedAdam, self).__init__(params, lr=lr, betas=betas,
                                         eps=eps, weight_decay=weight_decay)
        # State initialization
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['step'] = 0
                state['exp_avg'] = torch.zeros_like(p.data)
                state['exp_avg_sq'] = torch.zeros_like(p.data)
                # share in memory
                state['exp_avg'].share_memory_()
                state['exp_avg_sq'].share_memory_()
```

It's worth noting that I use Adam while the paper uses RMSProp. Adam can be viewed as the combination of RMSProp and Momentum and it usually performs better than RMSProp. I also compare their performance in later experiments.

2.4 Training

First, initialize global network and share its memory. Then, initialize the optimizer of global network, some global variables shared among workers, and workers. The number of workers is the number of CPU kernel. Then, start the workers and get results from global queue.

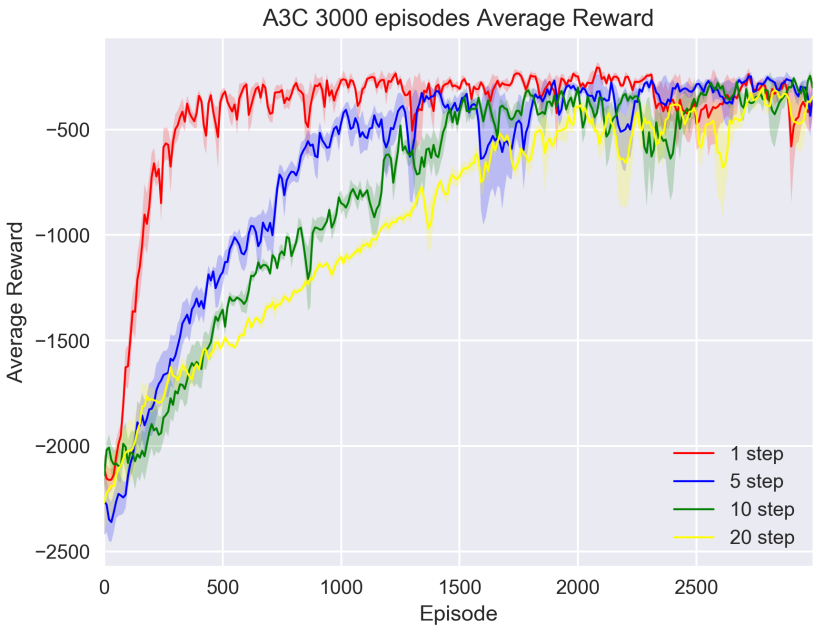
To increase the credibility of experiments, I repeat the training step for several times and use their mean as final results.

```
for i in range(rep_time):
    global_net = ACNet(state_space, action_space)
    global_net.share_memory()
    opt = SharedAdam(global_net.parameters(), lr=1e-4, betas=(0.95, 0.999))
    global_ep, global_ep_r, res_queue= mp.Value('i', 0), mp.Value('d', 0.), mp.Queue()
    workers = [Worker(env_name, global_net, opt, global_ep, global_ep_r, res_queue, idx=i,
                      g_update_iter=5) for i in range(mp.cpu_count())]
    [w.start() for w in workers]
    res = []  # record episode reward to plot
    while True:
        r = res_queue.get()
        if r is not None:
            res.append(r)
        else:
            break
    [w.join() for w in workers]
    total_res.append(res)
```

3 Results

Here are the main parameters I use in training and they remain the same when I try different global updating frequency . Because there is no terminal state in Pendulum, so it's a good idea to limit the max step of an episode and I set it to 300 step. Then I limit the total number of episodes to 3000 and set gamma to 0.9.

I have tried different global updating frequency and it turns out that the smaller the global updating step is, the faster the agent converges. But when global updating step is small, it takes a longer time for local worker to push and pull the parameters of global network, that is agent spends longer time to finish 3000 episodes.



4 Discussion

4.1 N step Forward View(Global Updating Frequency)

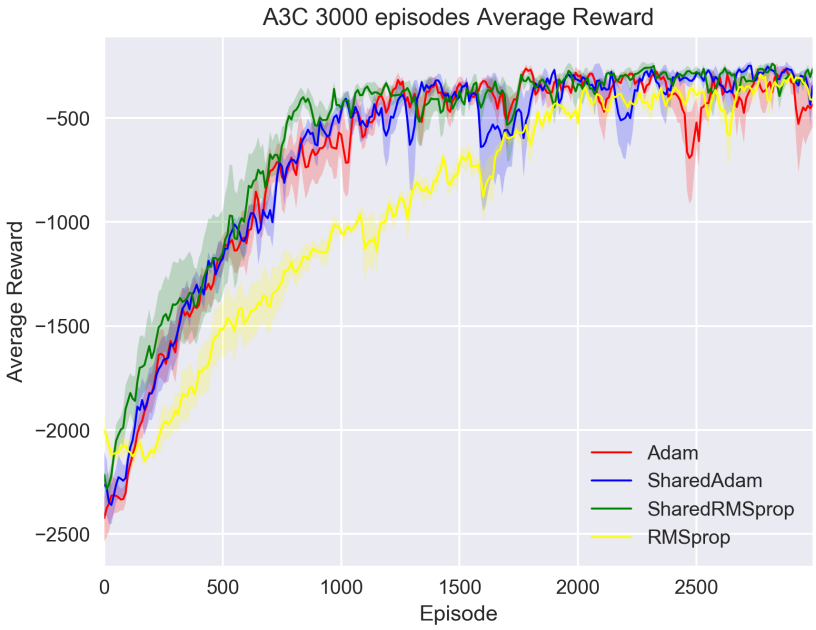
As global update step become smaller, it takes longer time to finish an episode but the performance is better. I guess the larger the global update step is , the more possibly the gradients of different workers will conflict, which may lead to the poor performance.

But it takes longer time to finish an episode when global update step is small, so there is a trade off between training time and performance. Finally, I set it to 5 for most experiments.

4.2 Optimizer

The original paper used RMSprop as optimizer, but usually, Adam performs better than RMSprop. So I do an experiment to compare them.

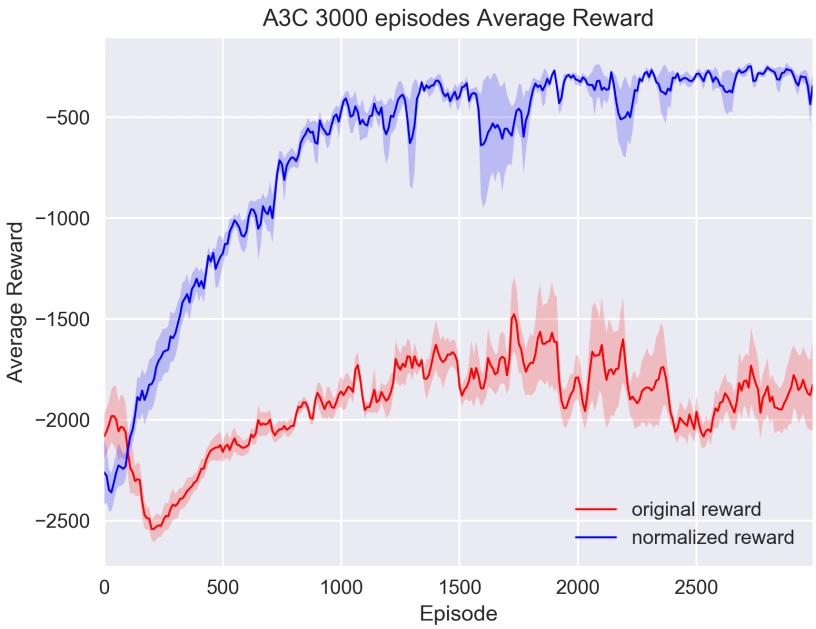
The original paper said that SharedRMSprop performed better that RMSprop, and I also do some experiments on Adam and SharedAdam.



As the digram shows, SharedRMSprop does performed better than RMSprop, but the difference among Adam, SharedAdam, and SharedRMSprop is very small. Maybe it's because this game is not very complex and it's hard to tell their difference.

4.3 Reward Modification

Original step reward is $-(\theta^2 + 0.1 * \theta_{dt}^2 + 0.001 * action^2)$ and in range of -16.2 to 0, where θ is the angle and θ_{dt} is angular velocity. In my experiments, I normalize step reward to $(rewrd - 8.1)/8.1$, which makes it in range of -1 to 1. Under the current setting, modified reward performs better while it's hard for original reward to converge.



As the digram shows, reward modification does improve agent's performance. But it's also possible that the current parameters are suitable for modified reward, but not for original reward. Maybe, it performs better if I find suitable parameters.

4.4 Number of Workers

Given that I set the a fixed total number of episode (the sum of workers' episode is fixed), the performance of 4 workers and 8 workers are about the same. 4 workers performs slightly better and the reason behind that may be 4 worker bring less gradient conflict than 8 worker, but 4 workers spend much longer time to finish 3000 episodes than 8 workers. So the leading effects of A3C may lay in reducing training time in this environment.