

Advantage Actor-Critic (A2C)

이동민

삼성전자 서울대 공동연구소
Jul 16, 2019

Outline

- Advantage Actor-Critic (A2C)
 - Learning process
 - Hyperparameter
 - Main loop
 - Train model
 - Train & TensorboardX
 - Learning curve & Test
- Comparison of algorithms for discrete action - Learning curve



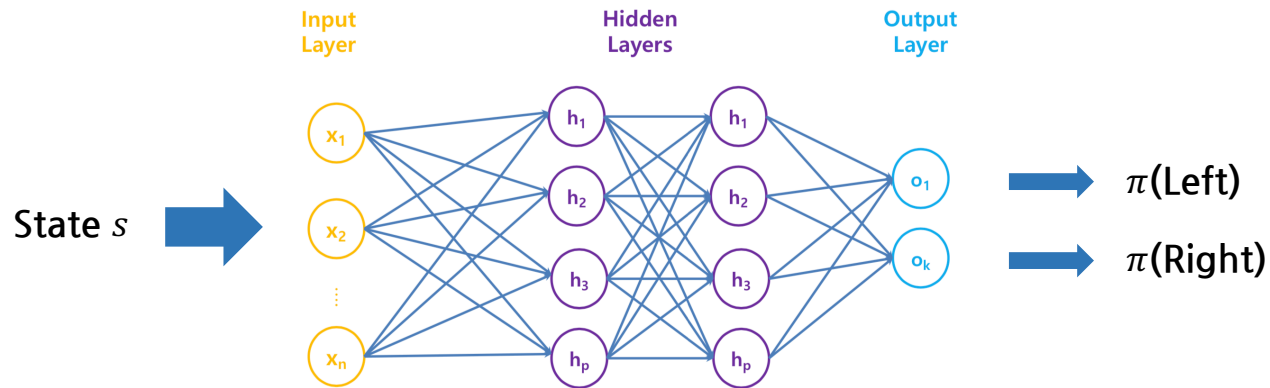
Advantage Actor-Critic (A2C)

- Learning process
 1. 상태에 따른 행동 선택
 2. 환경에서 선택한 행동으로 한 time step을 진행한 후, 다음 상태와 보상을 받음
 3. Sample (s, a, r, s') 로 Actor & Critic network 업데이트

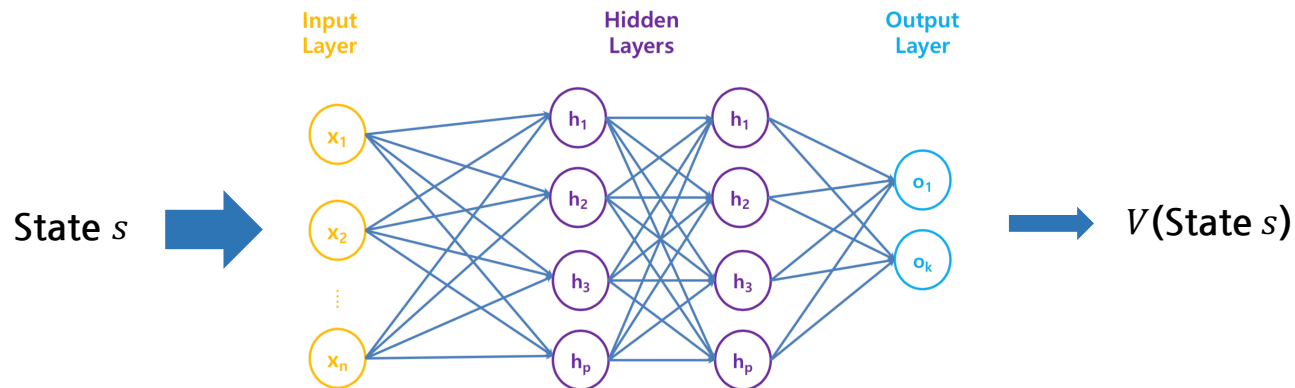


Advantage Actor-Critic (A2C)

- Actor network



- Critic network



Advantage Actor-Critic (A2C)

- Actor network

```
class Actor(nn.Module):  
    def __init__(self, state_size, action_size, args):  
        super(Actor, self).__init__()  
        self.fc1 = nn.Linear(state_size, args.hidden_size)  
        self.fc2 = nn.Linear(args.hidden_size, args.hidden_size)  
        self.fc3 = nn.Linear(args.hidden_size, action_size)  
  
    def forward(self, x):  
        x = torch.tanh(self.fc1(x))  
        x = torch.tanh(self.fc2(x))  
        policies = torch.softmax(self.fc3(x), dim=1)  
  
        return policies
```



Advantage Actor-Critic (A2C)

- Critic network

```
class Critic(nn.Module):  
    def __init__(self, state_size, args):  
        super(Critic, self).__init__()  
        self.fc1 = nn.Linear(state_size, args.hidden_size)  
        self.fc2 = nn.Linear(args.hidden_size, args.hidden_size)  
        self.fc3 = nn.Linear(args.hidden_size, 1)  
  
    def forward(self, x):  
        x = torch.tanh(self.fc1(x))  
        x = torch.tanh(self.fc2(x))  
        value = self.fc3(x)  
  
        return value
```



Learning process

1. 상태에 따른 행동 선택

```
policies = actor(torch.Tensor(state))  
action = get_action(policies)
```

```
def get_action(policies):  
    categorical = Categorical(policies)  
    action = categorical.sample()  
    action = action.data.numpy()[0]  
  
    return action
```

2. 환경에서 선택한 행동으로 한 time step을 진행한 후, 다음 상태와 보상을 받음

```
next_state, reward, done, _ = env.step(action)
```

Learning process

3. Sample (s, a, s', r) 로 Actor & Critic network 업데이트

- Critic Loss

$$J_V(\theta) = (\underbrace{r + \gamma V_\theta(s')}_{\text{Target}} - \underbrace{V_\theta(s)}_{\text{Prediction}})^2$$

```
# update critic
criterion = torch.nn.MSELoss()

value = critic(torch.Tensor(state)).squeeze(1)

next_value = critic(torch.Tensor(next_state)).squeeze(1)
target = reward + mask * args.gamma * next_value

critic_loss = criterion(value, target.detach())
critic_optimizer.zero_grad()
critic_loss.backward()
critic_optimizer.step()
```



Learning process

3. Sample (s, a, s', r) 로 Actor & Critic network 업데이트

- Actor Loss

$$J_{\pi}(\phi) = -\log \pi_{\phi}(a|s) \underbrace{(r + \gamma V_{\theta}(s') - V_{\theta}(s))}_{\text{Advantage Function}} + \underbrace{\alpha \left(- \sum_i \pi_{\phi_i} \log \pi_{\phi_i} \right)}_{\text{entropy}} \quad \text{regularization term}$$

- **Entropy** is used to **improve exploration** by limiting the premature convergence to suboptimal policy.

Learning process

3. Sample (s, a, s', r) 로 Actor & Critic network 업데이트

- Actor Loss

$$J_{\pi}(\phi) = -\log \pi_{\phi}(a|s) \left(r + \gamma V_{\theta}(s') - V_{\theta}(s) \right) + \alpha \left(-\sum_i \pi_{\phi_i} \log \pi_{\phi_i} \right)$$

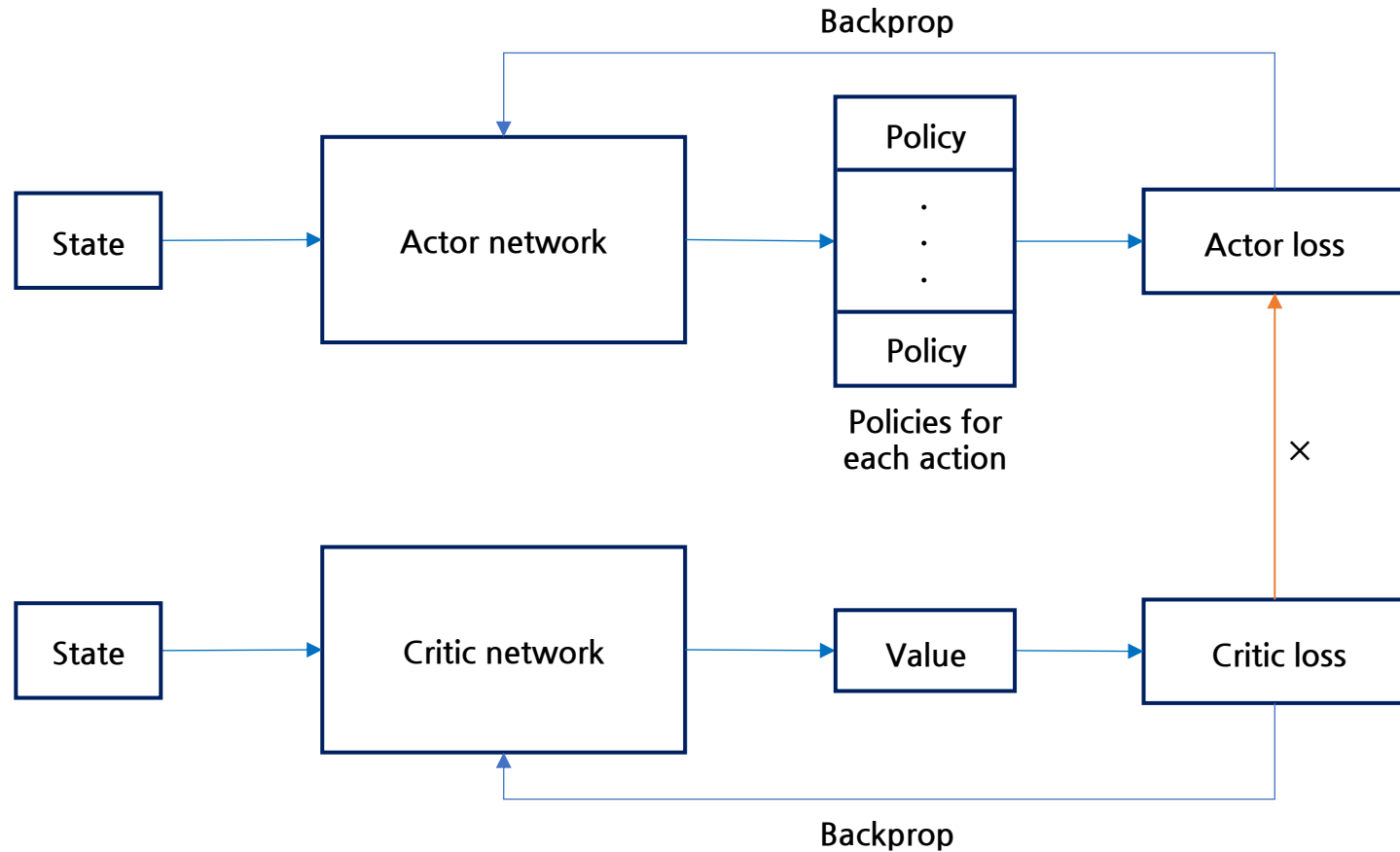
```
# update actor
categorical = Categorical(policies)
log_policy = categorical.log_prob(torch.Tensor([action]))
entropy = categorical.entropy()

advantage = target - value

actor_loss = -log_policy * advantage.item() + args.ent_coef * entropy
actor_optimizer.zero_grad()
actor_loss.backward()
actor_optimizer.step()
```



Learning process



Hyperparameter

```
parser = argparse.ArgumentParser()
parser.add_argument('--env_name', type=str, default="CartPole-v1")
parser.add_argument('--load_model', type=str, default=None)
parser.add_argument('--save_path', default='./save_model/', help='')
parser.add_argument('--render', action="store_true", default=False)
parser.add_argument('--gamma', type=float, default=0.99)
parser.add_argument('--hidden_size', type=int, default=64)
parser.add_argument('--actor_lr', type=float, default=1e-4)
parser.add_argument('--critic_lr', type=float, default=1e-3)
parser.add_argument('--ent_coef', type=float, default=0.1)
parser.add_argument('--max_iter_num', type=int, default=1000)
parser.add_argument('--log_interval', type=int, default=10)
parser.add_argument('--goal_score', type=int, default=400)
parser.add_argument('--logdir', type=str, default='./logs',
                    help='tensorboardx logs directory')
args = parser.parse_args()
```



Main loop

- Initialization
 - Seed - random number 고정
 - Actor & Critic network
 - Actor & Critic optimizer
 - TensorboardX

```
def main():  
    env = gym.make(args.env_name)  
    env.seed(500)  
    torch.manual_seed(500)  
  
    state_size = env.observation_space.shape[0]  
    action_size = env.action_space.n  
    print('state size:', state_size)  
    print('action size:', action_size)  
  
    actor = Actor(state_size, action_size, args)  
    critic = Critic(state_size, args)  
  
    actor_optimizer = optim.Adam(actor.parameters(), lr=args.actor_lr)  
    critic_optimizer = optim.Adam(critic.parameters(), lr=args.critic_lr)  
  
    writer = SummaryWriter(args.logdir)  
  
    running_score = 0
```



Main loop

- Episode 진행
 - Reshape state vector (4) \rightarrow (1,4)
 - 상태에 따른 행동 선택
 - 다음 상태와 보상을 받음
 - Reshape next state vector
 - Reward, mask 설정
 - Transition list에 저장
 - Train model
 - Running score 설정

```
for episode in range(args.max_iter_num):
    done = False
    score = 0

    state = env.reset()
    state = np.reshape(state, [1, state_size])

    while not done:
        if args.render:
            env.render()

        policies = actor(torch.Tensor(state))
        action = get_action(policies)

        next_state, reward, done, _ = env.step(action)

        next_state = np.reshape(next_state, [1, state_size])
        reward = reward if not done or score == 499 else -1
        mask = 0 if done else 1

        transition = [state, action, reward, next_state, mask]

        actor.train(), critic.train()
        train_model(actor, critic, actor_optimizer, critic_optimizer,
                    transition, policies)

        state = next_state
        score += reward

    score = score if score == 500.0 else score + 1
    running_score = 0.99 * running_score + 0.01 * score
```



Main loop

- Print & Visualize log
- Running score > 400
 - Save model
 - 학습 종료

```
if episode % args.log_interval == 0:
    print('{} episode | running_score: {:.2f}'.format(episode, running_score))
    writer.add_scalar('log/score', float(score), episode)

if running_score > args.goal_score:
    if not os.path.isdir(args.save_path):
        os.makedirs(args.save_path)

    ckpt_path = args.save_path + 'model.pth'
    torch.save(actor.state_dict(), ckpt_path)
    print('Running score exceeds 400. So end')
    break
```



Train model

- Transition List → state, action, next_state, reward, mask 각각 나누기

```
def train_model(actor, critic, actor_optimizer, critic_optimizer, transition, policies):  
    state, action, reward, next_state, mask = transition
```

- state - (1, 4)
- action
- reward
- next_state - (1, 4)
- mask

```
state [[-0.09066657 -1.56210361 -0.02062117 1.61977871]]  
action 0  
reward 1.0  
next_state [[-0.12190864 -1.75697662 0.01177441 1.90596389]]  
mask 1
```

```
state [[-0.12190864 -1.75697662 0.01177441 1.90596389]]  
action 0  
reward 1.0  
next_state [[-0.15704817 -1.95222371 0.04989369 2.20227582]]  
mask 1
```

```
state [[-0.15704817 -1.95222371 0.04989369 2.20227582]]  
action 1  
reward 1.0  
next_state [[-0.19609265 -1.75761556 0.0939392 1.92538951]]  
mask 1
```



Train model

- Update critic - MSE Loss

- $$J_V(\theta) = (\underbrace{r + \gamma V_\theta(s')}_{\text{Target}} - \underbrace{V_\theta(s)}_{\text{Prediction}})^2$$

```
# update critic
criterion = torch.nn.MSELoss()

value = critic(torch.Tensor(state)).squeeze(1)

next_value = critic(torch.Tensor(next_state)).squeeze(1)
target = reward + mask * args.gamma * next_value

critic_loss = criterion(value, target.detach())
critic_optimizer.zero_grad()
critic_loss.backward()
critic_optimizer.step()
```



Train model

- Update actor

- $$J_{\pi}(\phi) = -\log \pi_{\phi}(a|s) (r + \gamma V_{\theta}(s') - V_{\theta}(s)) + \alpha \left(-\sum_i \pi_{\phi_i} \log \pi_{\phi_i} \right)$$

↑
Entropy coefficient : 0.1

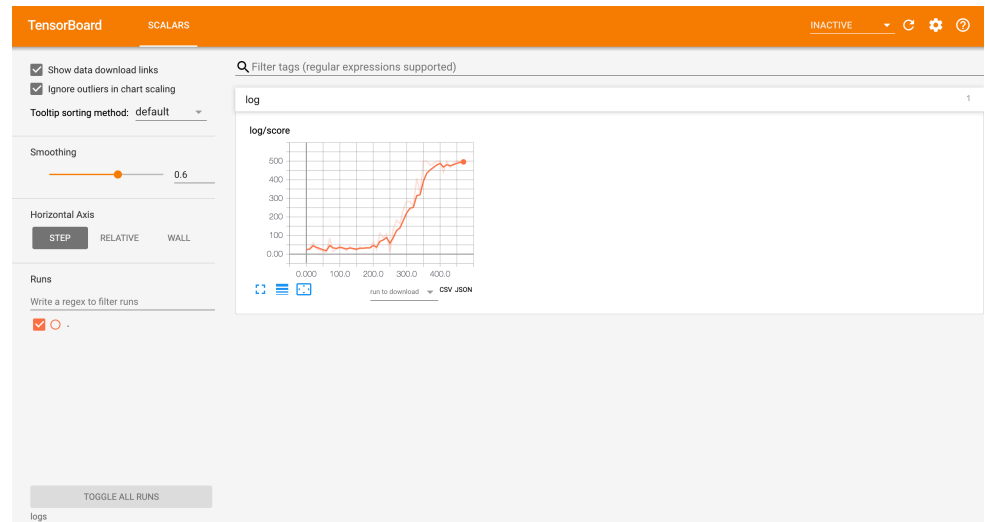
```
# update actor
categorical = Categorical(policies)
log_policy = categorical.log_prob(torch.Tensor([action]))
entropy = categorical.entropy()

advantage = target - value

actor_loss = -log_policy * advantage.item() + args.ent_coef * entropy
actor_optimizer.zero_grad()
actor_loss.backward()
actor_optimizer.step()
```

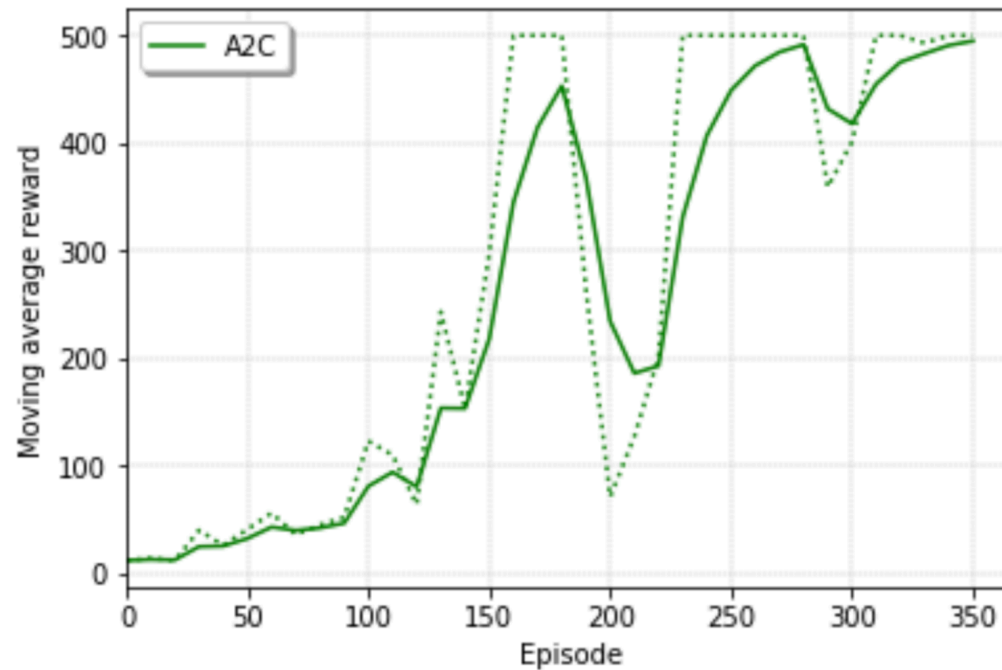
Train & TensorboardX

- Terminal A - train 실행
 - `conda activate env_name`
 - `python train.py`
- Terminal B - tensorboardX 실행
 - `conda activate env_name`
 - `tensorboard --logdir logs`
 - (웹에서) `localhost:6006`



Learning curve & Test

- Learning curve

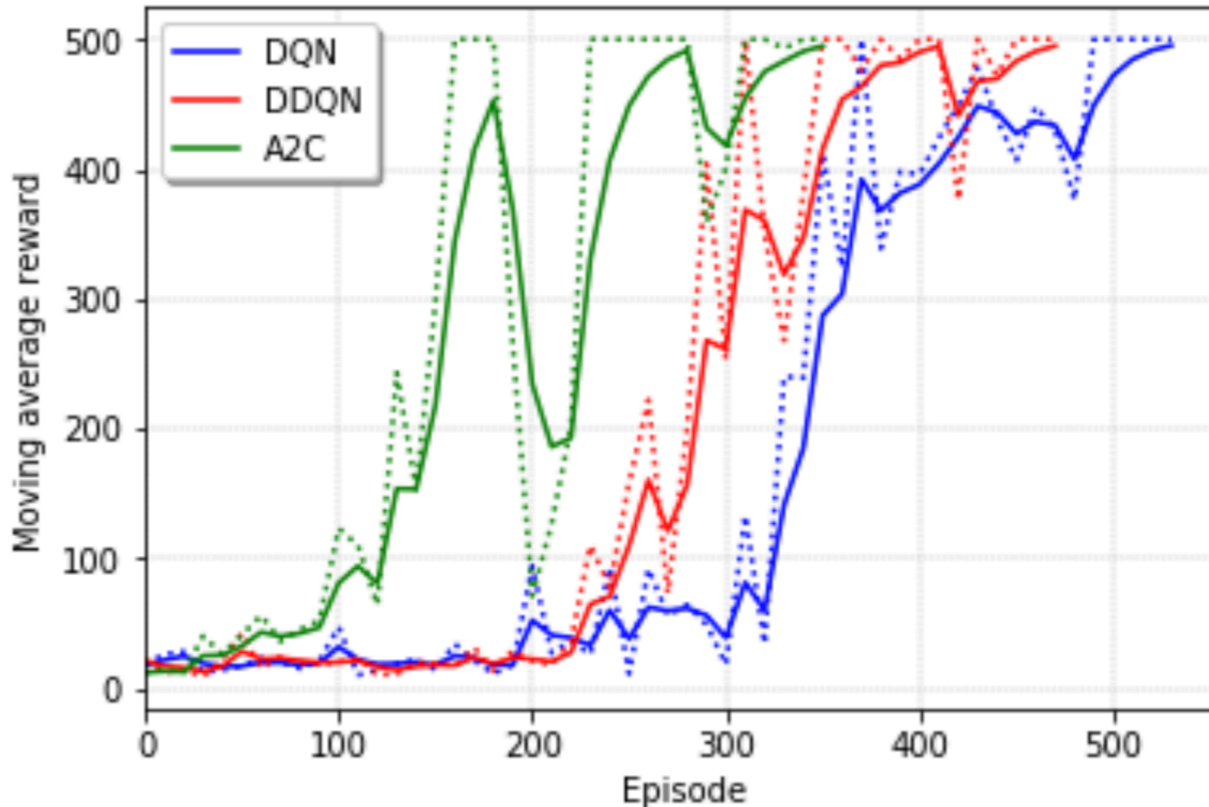


- Test
 - `python test.py`



Comparison of algorithms for discrete action

- Learning curve



Comparison of algorithms for discrete action

Discrete action일 때는 어떤 알고리즘이 제일 좋을까요? → A2C
그리고 그 알고리즘이 왜 좋을까요?



Comparison of algorithms for discrete action

Discrete action일 때는 어떤 알고리즘이 제일 좋을까요? → A2C
그리고 그 알고리즘이 왜 좋을까요?

- DQN, DDQN (Off-policy algorithms)
 - Replay buffer의 크기 → 컴퓨터의 메모리를 많이 차지하며 느린 학습 속도의 원인
 - Replay buffer를 통해 학습을 진행
→ 지금 policy가 아닌 이전 policies를 통해 모은 sample로 학습
 - ϵ -greedy policy를 통한 action 선택
- A2C (On-policy algorithm)
 - Replay buffer가 필요하지 않음
 - 현재 policy를 통해 학습
 - Actor에서 나오는 policy 자체가 action에 대한 확률이므로 따로 action에 대한 exploration을 정해주지 않아도 됨
 - Actor와 Critic 나눠서 업데이트

Thank you

