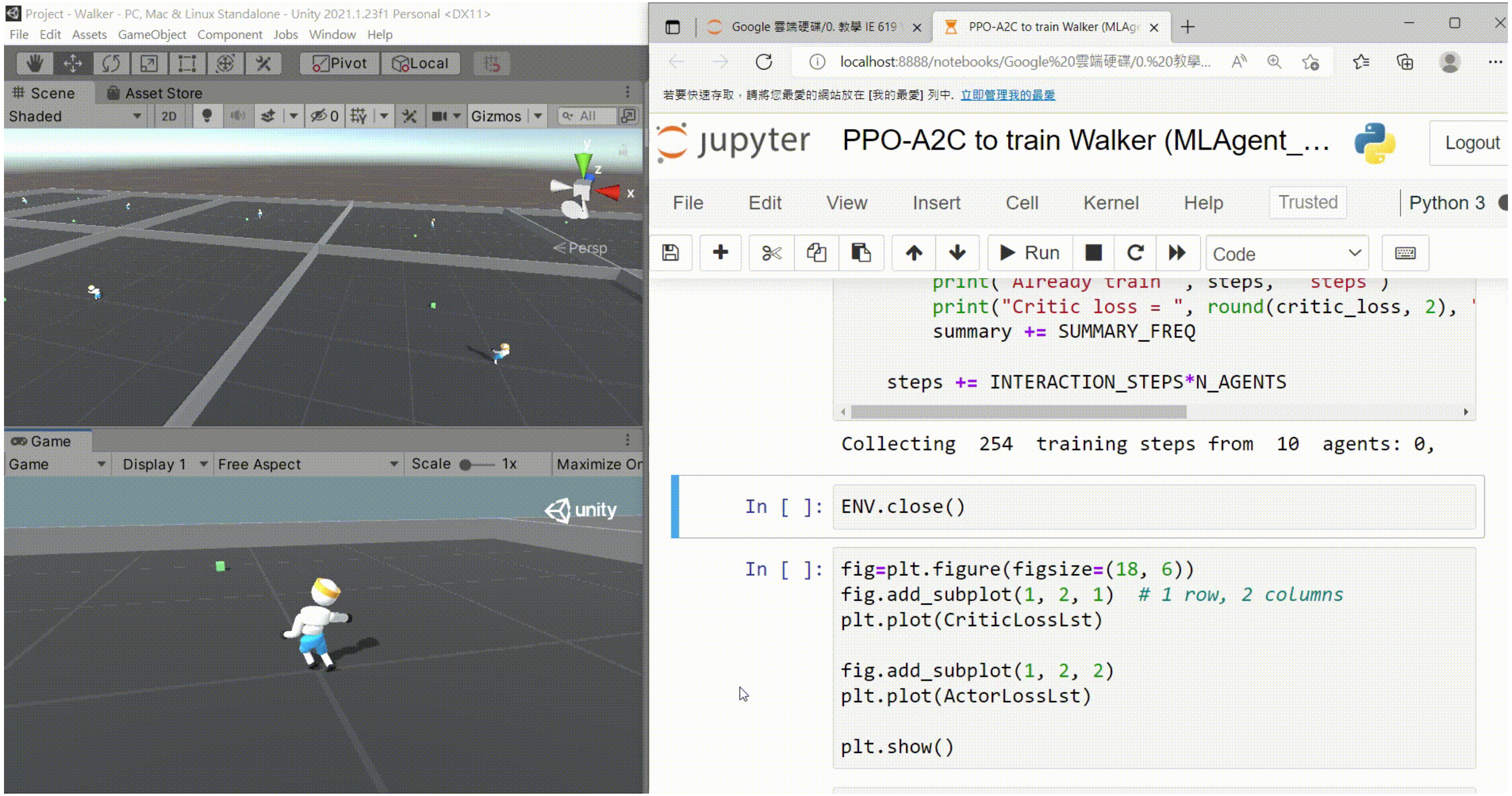
PyTorch implementation and Unity simulation environment

Walker in ML agent 19 project

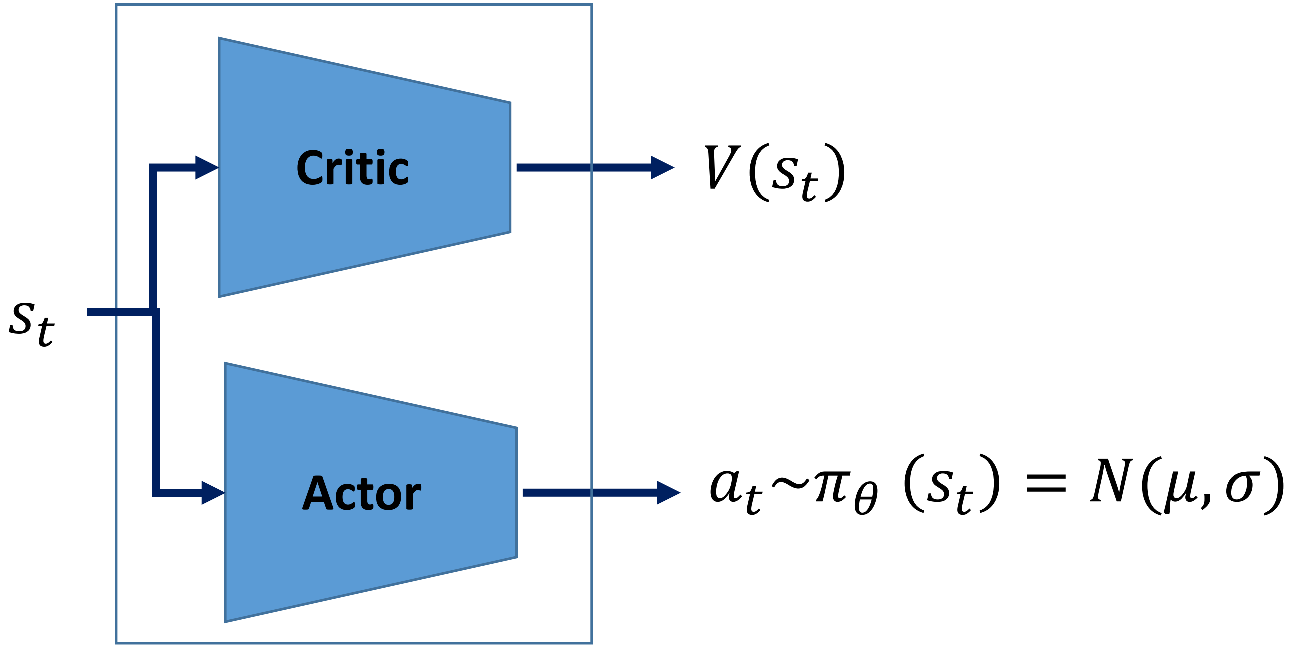
PPO-A2C to train Walker (MLAgent_19).ipynb



Actor and critic NN

```
self.critic = nn.Sequential(
    nn.Linear(N_STATES, HIDDEN_UNITS),
    nn.LayerNorm(HIDDEN_UNITS),
    nn.Linear(HIDDEN_UNITS, HIDDEN_UNITS),
    nn.LayerNorm(HIDDEN_UNITS),
    nn.Linear(HIDDEN_UNITS, 1)
)
```

```
self.actor = nn.Sequential(
    nn.Linear(N_STATES, HIDDEN_UNITS),
    nn.LayerNorm(HIDDEN_UNITS),
    nn.Linear(HIDDEN_UNITS, HIDDEN_UNITS),
    nn.LayerNorm(HIDDEN_UNITS),
    nn.Linear(HIDDEN_UNITS, N_ACTIONS)
)
```




```
def forward(self, x):
    value = self.critic(x)
    mu    = self.actor(x)
    std   = self.log_std.exp().expand_as(mu)
    dist  = Normal(mu, std)
    return dist, value
```

Advantage actor-critic

$$\max J(\pi) = E_{\tau\sim\pi}\left(R(\tau)\right)$$

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)$$

REINFORCE (Monte Carlo PG)

$$\nabla_\theta J(\pi_\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)\Phi_t\right] \qquad \Phi_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

Advantage Actor-Critic

$$\nabla_\theta J(\pi_\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)\left(Q^\pi(s_t, a_t) - V^\pi(s_t)\right)\right]$$

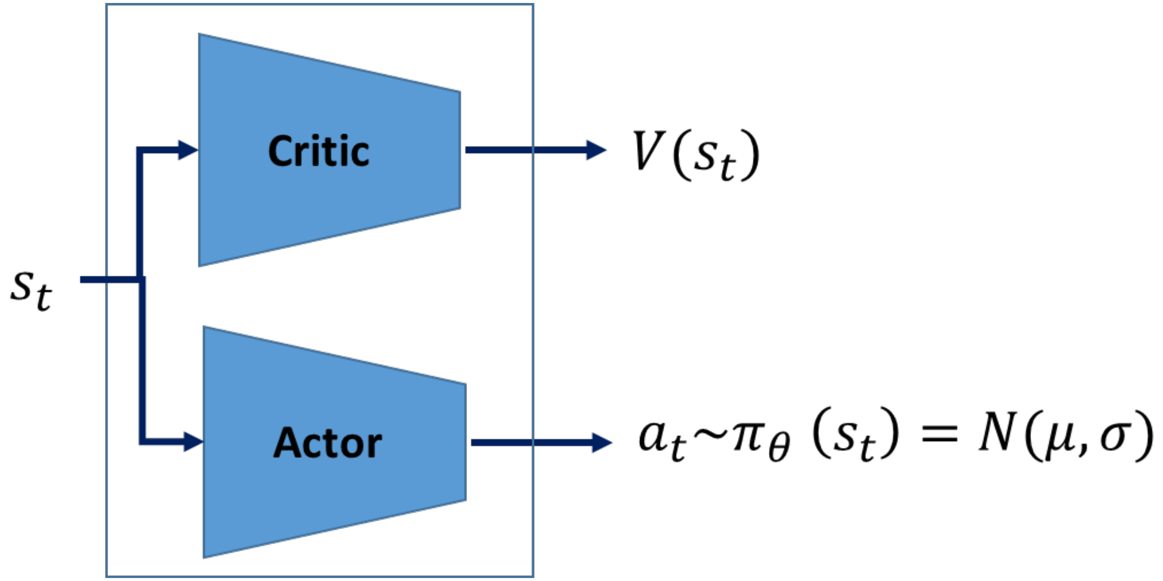
$$= E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)(r(s_t, a_t) + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)\,)\right]$$

Pass state to Actor and Critics NN

```
[13]: s = torch.FloatTensor(s)
      dist, value = NET(s.to(device))
      print(dist, "\n", value)
```

Pass s to Actor and Critic NN to get $\pi_\theta\,(s_t) = N(\mu, \sigma)$ and $V(s_t)$

```
Normal(loc: torch.Size([10, 39]), scale: torch.Size([10, 39]))
 tensor([[1.7525],
        [1.3020],
        [1.8764],
        [1.7034],
        [1.7539],
        [1.6710],
        [2.0807],
        [1.8870],
        [2.0875],
```



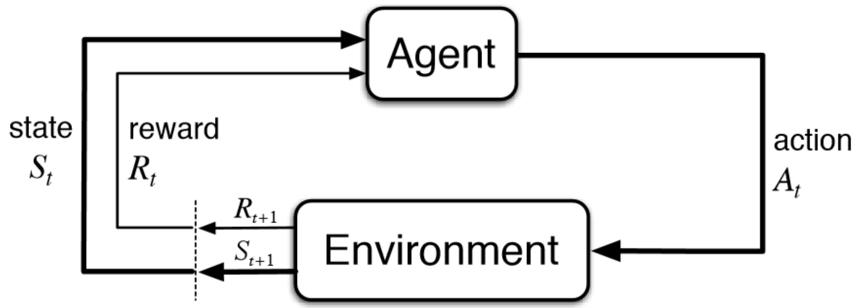
Sampling action values

```
In [14]: a = dist.sample()
         log_prob = dist.log_prob(a)
         print(a, "\n", log_prob)
```

$$a_t \sim \pi_\theta\,(s_t) = N(\mu, \sigma)$$
$$log\pi_\theta(a_t|s_t)$$

```
         2.1507, -1.0720,  1.5543, -2.0371, -3.5
5,
        -2.0914,  3.4678, -1.9880, -2.7836,  2.7
2,
         2.1310,  2.9008, -3.7953,  0.6812, -1.0
0,
         2.7232,  1.2273, -0.9963, -3.4695,  2.5
       [ 2.7992, -2.0890, -2.9866,  3.2485, -1.9
8,
         3.3492,  3.6204, -0.6714,  0.3952, -0.7
7,
        -3.0459, -2.8508,  1.2497, -0.8889,  2.1
```

One interaction step between Unity and PyTorch




```
def Interact_with_Unity_one_step (DecisionSteps):
    # ENV and NET are global variables
    s = DecisionSteps.obs[0]
    s = torch.FloatTensor(s)
    dist, value = NET(s.to(device))
    a = dist.sample()
    log_prob = dist.log_prob(a)

    a = a.cpu().detach().numpy()
    a = ActionTuple(np.array(a, dtype=np.float32))
    ENV.set_actions(BEHAVIOR_NAME, a)
    ENV.step()
    a = a._continuous #convert from ActionTuple to np
    a = torch.FloatTensor(a) # convert from np.array
    return s, value, a, log_prob
```

Collect training trajectories

```
def collect_training_data (print_message):
```

while step < INTERACTION_STEPS

If we have no decision agents → continue next loop without increase step

else

Interacts with Unity one step

If this or next decision step misses some agents → Continue next loop without increase step and do not collect data

else this and next decision steps includes all agents

(This ensures that we can collect s and s_next from all agents, otherwise program will have error!)

Collect (s, V, a, r, s_next) from all agents

Collect reward and mask from next terminal steps

Collect reward and mask from next decision steps

step = step +1

Collect training trajectories

$$\nabla_\theta J(\pi_\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)\left(Q^\pi(s_t,a_t) - V^\pi(s_t)\right)\right]$$

$$= E_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)(r(s_t,a_t) + \gamma V^\pi(s_{t+1}) - V^\pi(s_t))\right]$$

T = INTERACTION_STEPS

Agent 1: $\tau = \left(s_1, V_1, a_1, \log p(a_1|s_1), r_1, mask_1 \ldots, s_T, V_T, a_{1,T}, \log p(a_T|s_T), r_T, mask_T\right)$

Agent 2: τ

…

Agent 10: τ

Store training trajectory data

```
def collect_training_data (print_message):
AgentID = DecisionSteps.agent_id[idx]
STATES[step][AgentID]=s[idx]
VALUES[step][AgentID]=value[idx]
ACTIONS[step][AgentID]=a[idx]
LOG_PROBS[step][AgentID]=log_prob[idx]
```

```
def Collect REWARDS and MASKS
AgentID = AgentSteps.agent_id[idx]
REWARDS[step][AgentID]=r[idx]
MASKS[step][AgentID]= flag
NEXT_STATES[step][AgentID]=s[idx]
```

Step *i*

$$\begin{bmatrix} s_{step_1,agent_1} \\ s_{step_1,agent_2} \\ \cdots \\ s_{step_1,agent_{10}} \end{bmatrix} \begin{bmatrix} V_{step_1,agent_1} \\ V_{step_1,agent_2} \\ \cdots \\ V_{step_1,agent_{10}} \end{bmatrix} \begin{bmatrix} a_{step_1,agent_1} \\ a_{step_1,agent_2} \\ \cdots \\ a_{step_1,agent_{10}} \end{bmatrix}$$

$$\begin{bmatrix} \log p(a_{step_1,agent_1} | s_{step_1,agent_1}) \\ \log p(a_{step_1,agent_2} | s_{step_1,agent_2}) \\ \cdots \\ \log p(a_{step_1,agent_{10}} | s_{step_1,agent_{10}}) \end{bmatrix}$$

$$\begin{bmatrix} r_{step_1,agent_1} \\ r_{step_1,agent_2} \\ \cdots \\ r_{step_1,agent_{10}} \end{bmatrix} \begin{bmatrix} mask_{step_1,agent_1} \\ mask_{step_1,agent_2} \\ \cdots \\ maks_{step_1,agent_{10}} \end{bmatrix}$$

$$\begin{bmatrix} s\_next_{step_1,agent_1} \\ s\_next_{step_1,agent_2} \\ \cdots \\ s\_next_{step_1,agent_{10}} \end{bmatrix}$$

Store training trajectory data

Training trajectory data from 10 agents each conducting 254 steps

```
[25]: print(len(LOG_PROBS), LOG_PROBS[0].shape)
      print(len(VALUES), VALUES[0].shape)
      print(len(REWARDS), REWARDS[0].shape)
      print(len(MASKS), MASKS[0].shape)
      print(len(STATES), STATES[0].shape)
      print(len(ACTIONS), ACTIONS[0].shape)
      print(len(NEXT_STATES), NEXT_STATES[0].shape)
```

```
254 torch.Size([10, 39])
254 torch.Size([10, 1])
254 torch.Size([10, 1])
254 torch.Size([10, 1])
254 torch.Size([10, 243])
254 torch.Size([10, 39])
254 torch.Size([10, 243])
```

Calculate Advantage from training trajectories

$$\nabla_\theta J(\pi_\theta)=E_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T}\nabla_\theta log\pi_\theta(a_t|s_t)\left(Q^\pi(s_t,a_t)-V^\pi(s_t)\right)\right]$$
$$= E_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T}\nabla_\theta log\pi_\theta(a_t|s_t)(r(s_t,a_t)+\gamma V^\pi(s_{t+1})-V^\pi(s_t))\right]$$

```
def compute_gae(next_value):
    return returns
```

```
[28]: RETURNS = compute_gae(
```

```
254 torch.Size([10, 39])
254 torch.Size([10, 1])
254 torch.Size([10, 1])
254 torch.Size([10, 1])
254 torch.Size([10, 243])
254 torch.Size([10, 39])
254 torch.Size([10, 243])
```

$$\Delta_{20}= r_{20} + (\gamma * v_{21} * mask_{20} - v_{20})$$
$$gae_{20} = \Delta_{20} + \gamma * \lambda * mask_{20} * gae_{initial}$$
$$return_{20} = gae_{20} + v_{20}$$

$$\Delta_{19}= r_{19} + (\gamma * v_{20} * mask_{19} - v_{19})$$
$$gae_{19\sim20} = \Delta_{19} + \gamma * \tau * mask_{19} * gae_{20}$$
$$return_{19} = gae_{19\sim20} + v_{19}$$

...

$$\Delta_{1}= r_{1} + (\gamma * v_{2} * mask_{1} - v_{1})$$
$$gae_{1\sim20} = \Delta_{1} + \gamma * \tau * mask_{1} * gae_{2\sim20}$$
$$return_{1} = gae_{1\sim20} + v_{1}$$

Merge training trajectory data from multiple agents

[29]:
```
MERGED_RETURNS    = torch.cat(RETURNS).detach()
MERGED_LOG_PROBS = torch.cat(LOG_PROBS).detach()
MERGED_VALUES     = torch.cat(VALUES).detach()
MERGED_STATES     = torch.cat(STATES)
MERGED_NEXT_STATES    = torch.cat(NEXT_STATES)
MERGED_ACTIONS    = torch.cat(ACTIONS)
MERGED_ADVANTAGES = MERGED_RETURNS - MERGED_VALUES
```

[30]:
```
print(len(MERGED_RETURNS), MERGED_RETURNS[0].shape)
print(len(MERGED_LOG_PROBS), MERGED_LOG_PROBS[0].shape)
print(len(MERGED_VALUES), MERGED_VALUES[0].shape)
print(len(MERGED_STATES), MERGED_STATES[0].shape)
print(len(MERGED_NEXT_STATES), MERGED_NEXT_STATES[0].shape)
print(len(MERGED_ACTIONS), MERGED_ACTIONS[0].shape)
print(len(MERGED_ADVANTAGES), MERGED_ADVANTAGES[0].shape)
```

```
2540 torch.Size([1])
2540 torch.Size([39])
2540 torch.Size([1])
2540 torch.Size([243])
2540 torch.Size([243])
2540 torch.Size([39])
2540 torch.Size([1])
```

2540 = 10 agents each conducting 254 steps

$$
\begin{bmatrix} s_{step_1,agent_1} \\ \vdots \\ s_{step_1,agent_{10}} \\ s_{step_2,agent_1} \\ \vdots \\ s_{step_2,agent_{10}} \\ \vdots \end{bmatrix}
\begin{bmatrix} V_{step_1,agent_1} \\ \vdots \\ V_{step_1,agent_{10}} \\ V_{step_2,agent_1} \\ \vdots \\ V_{step_2,agent_{10}} \\ \vdots \end{bmatrix}
\begin{bmatrix} a_{step_1,agent_1} \\ \vdots \\ a_{step_1,agent_{10}} \\ a_{step_2,agent_1} \\ \vdots \\ a_{step_2,agent_{10}} \\ \vdots \end{bmatrix}
\begin{bmatrix} \log p(a_{step_1,agent_1} | s_{step_1,agent_1}) \\ \vdots \\ \log p(a_{step_1,agent_{10}} | s_{step_1,agent_{10}}) \\ \log p(a_{step_2,agent_1} | s_{step_{12},agent_1}) \\ \vdots \\ \log p(a_{step_2,agent_{10}} | s_{step_2,agent_{10}}) \\ \vdots \end{bmatrix}
\begin{bmatrix} Return_{step_1,agent_1} \\ \vdots \\ Return_{step_1,agent_{10}} \\ Return_{step_2,agent_1} \\ \vdots \\ Return_{step_2,agent_{10}} \\ \vdots \end{bmatrix}
\cdots
$$

Sampling a batch of training data from buffer

```
def ppo_iter():
    buffer_size = MERGED_STATES.size(0)
    for _ in range(buffer_size// BATCH_SIZE ):
        rand_ids = np.random.randint(0, buffer_size, BATCH_SIZE )
        yield MERGED_STATES[rand_ids, :], MERGED_ACTIONS[rand_ids,
              MERGED_LOG_PROBS[rand_ids, :], MERGED_RETURNS[rand_i
```

$$
\begin{bmatrix} s_{step_1,agent_1} \\ \vdots \\ \\ s_{step_1,agent_{10}} \\ s_{step_2,agent_1} \\ \vdots \\ s_{step_2,agent_{10}} \\ \vdots \end{bmatrix}
\begin{bmatrix} V_{step_1,agent_1} \\ \vdots \\ \\ V_{step_1,agent_{10}} \\ V_{step_2,agent_1} \\ \vdots \\ V_{step_2,agent_{10}} \\ \vdots \end{bmatrix}
\begin{bmatrix} a_{step_1,agent_1} \\ \vdots \\ \\ a_{step_1,agent_{10}} \\ a_{step_2,agent_1} \\ \vdots \\ a_{step_2,agent_{10}} \\ \vdots \end{bmatrix}
\begin{bmatrix} \log p(a_{step_1,agent_1} | s_{step_1,agent_1}) \\ \vdots \\ \\ \log p(a_{step_1,agent_{10}} | s_{step_1,agent_{10}}) \\ \log p(a_{step_2,agent_1} | s_{step_{12},agent_1}) \\ \vdots \\ \log p(a_{step_2,agent_{10}} | s_{step_2,agent_{10}}) \\ \vdots \end{bmatrix}
\begin{bmatrix} Return_{step_1,agent_1} \\ \vdots \\ \\ Return_{step_1,agent_{10}} \\ Return_{step_2,agent_1} \\ \vdots \\ Return_{step_2,agent_{10}} \\ \vdots \end{bmatrix}
\cdots
$$

Loss function for critic NN

```
def ppo_update():
```

```
for b_s, b_a, b_s_, b_old_LOG_PROBS, b_return, b_advantage in ppo_iter():
```

```
    dist, value = NET(b_s.to(device))
    critic_loss = (b_return.to(device) - value).pow(2).mean()
```

BS = batch size

$$Loss_V = \frac{1}{BS}\sum_{i=1}^{BS}(Return_i - V^{\pi_\theta}(s_i))^2$$

PPO update

```python
def ppo_update():

    for b_s, b_a, b_s_, b_old_LOG_PROBS, b_return, b_advantage in ppo_iter():

        entropy = dist.entropy().mean()
        b_a_new = dist.sample()
        b_new_LOG_PROBS = dist.log_prob(b_a_new)
        ratio = (b_new_LOG_PROBS - b_old_LOG_PROBS.to(device)).exp()
        surr1 = ratio * b_advantage.to(device)
        surr2 = torch.clamp(ratio, 1.0-EPSILON, 1.0+EPSILON) * b_advantage.to(device)
        actor_loss  = - torch.min(surr1, surr2).mean()
```

$$Loss_{\pi} = \sum_{(s_t,a_t)} min\left(\frac{p_{\theta}(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t), clip\left(\frac{p_{\theta}(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon\right)A^{\theta'}(s_t,a_t)\right)$$

Entropy regularization

```
def ppo_update():
```

```
for b_s, b_a, b_s_, b_old_LOG_PROBS, b_return, b_advantage in ppo_iter():
```

```
loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy
OPTIMIZER.zero_grad()
loss.backward()
OPTIMIZER.step()
```

$$L = 0.5 \cdot Loss_V + Loss_\pi - 0.01 \cdot entropy$$

Walker training parameters

```
behaviors:
  Walker:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate:
0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
```

```
network_settings:
      normalize: true
      hidden_units: 512
      num_layers: 3
      vis_encode_type:
simple
    reward_signals:
      extrinsic:
        gamma: 0.995
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 30000000
    time_horizon: 1000
    summary_freq: 30000
```

- beta is the weight of entropy regularization
- lambd is the weight to calculate GAE

Reference – training parameters from OpenAI

Documentation: PyTorch Version

```
spinup.ppo_pytorch(env_fn, actor_critic=<MagicMock spec='str' id='140554322637768'>, ac_kwargs={}, seed=0, steps_per_epoch=4000, epochs=50, gamma=0.99, clip_ratio=0.2, pi_lr=0.0003, vf_lr=0.001, train_pi_iters=80, train_v_iters=80, lam=0.97, max_ep_len=1000, target_kl=0.01, logger_kwargs={}, save_freq=10)
```

Proximal Policy Optimization (by clipping),

with early stopping based on approximate KL

Class practice

Open terminal window from Anaconda
cd to the directory where the file “PPO_A2C_Walker_MLAgent_19.py” is located
>> python PPO_A2C_Walker_MLAgent_19.py
Press Play in Unity to start training