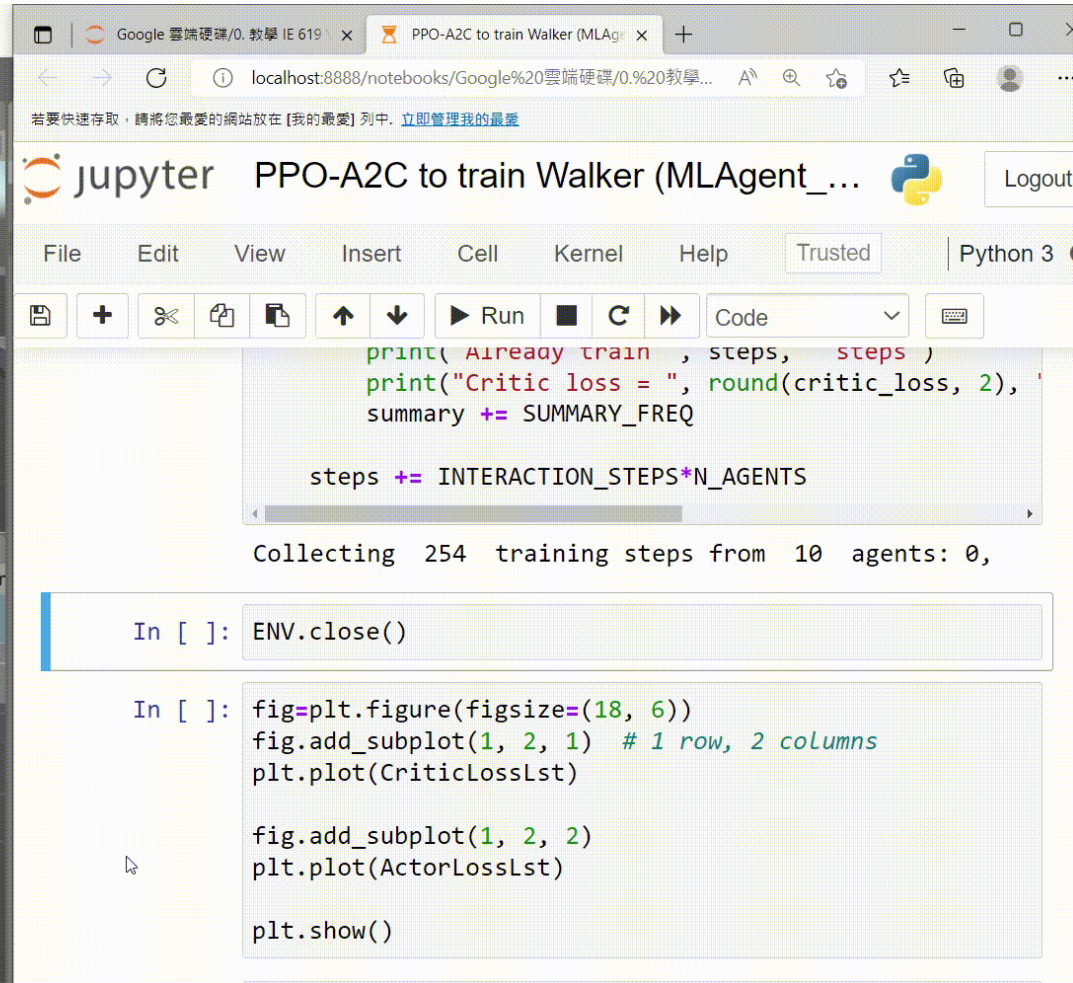


Ipython notebook version

Walker in ML agent 19 project

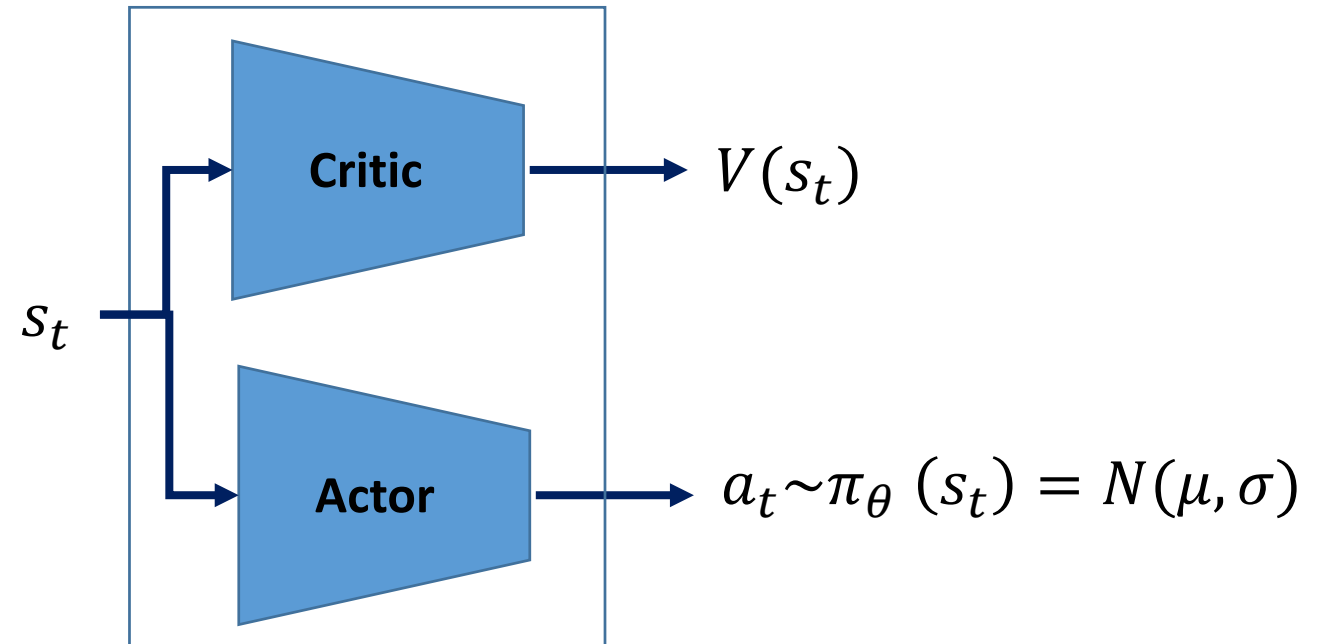
PPO-A2C to train Walker (MLAgent_19).ipynb



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

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

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

REINFORCE (Monte Carlo PG)

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

Advantage Actor-Critic

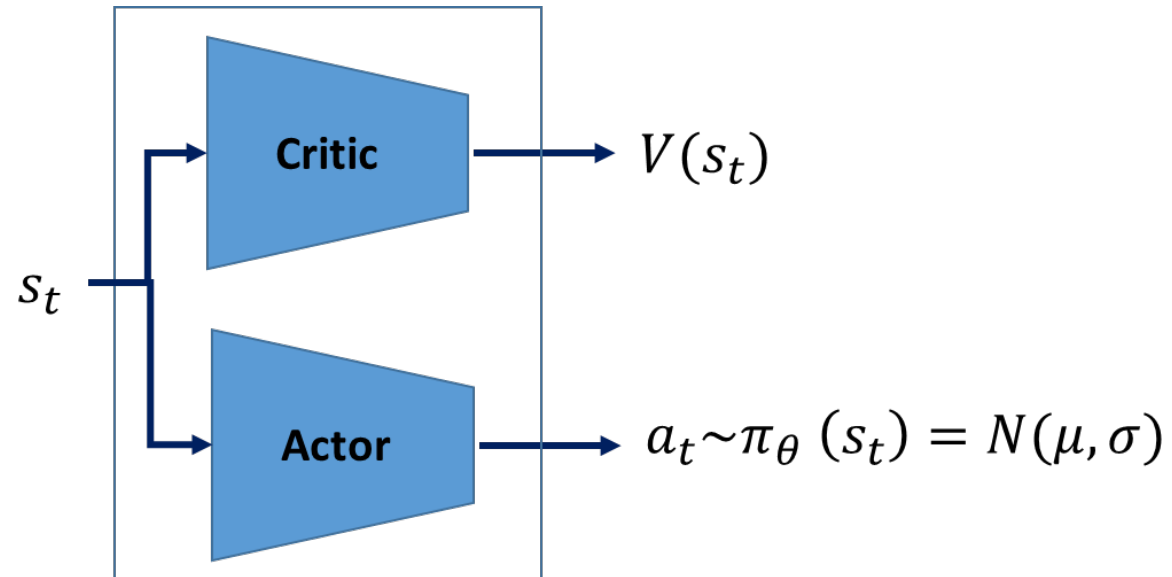
$$\begin{aligned} \nabla_{\theta} J(\pi_{\theta}) &= E_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) (Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)) \right] \\ &= E_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) (r(s_t, a_t) + \gamma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

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

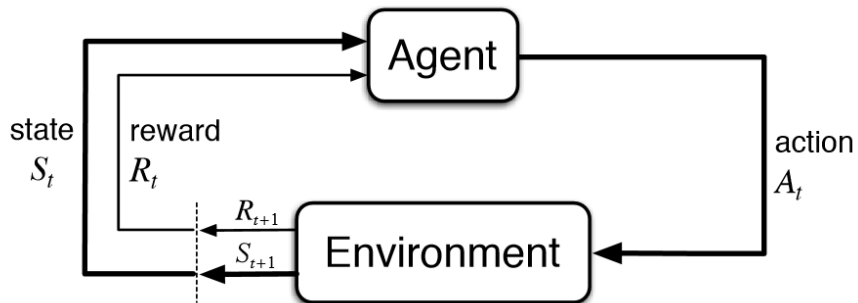
```
In [14]: a = dist.sample()
log_prob = dist.log_prob(a)
print(a, "\n", log_prob)
```

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

$$\log \pi_\theta(a_t | s_t)$$

4,	2.1507, -1.0720, 1.5543, -2.0371, -3.58
5,	-2.0914, 3.4678, -1.9880, -2.7836, 2.71
2,	2.1310, 2.9008, -3.7953, 0.6812, -1.04
0,	2.7232, 1.2273, -0.9963, -3.4695, 2.51
[2.7992, -2.0890, -2.9866, 3.2485, -1.91
8,	3.3492, 3.6204, -0.6714, 0.3952, -0.74
7,	-3.0459, -2.8508, 1.2497, -0.8889, 2.14

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

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

$T = \text{INTERACTION_STEPS}$

Agent 1: $\tau = (s_1, V_1, a_1, \log p(a_1 | s_1), r_1, \text{mask}_1, \dots, s_T, V_T, a_{1,T}, \log p(a_T | s_T), r_T, \text{mask}_T)$

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

$$\begin{bmatrix} S_{step_1, agent_1} \\ S_{step_1, agent_2} \\ \dots \\ S_{step_1, agent_{10}} \end{bmatrix} \begin{bmatrix} V_{step_1, agent_1} \\ V_{step_1, agent_2} \\ \dots \\ V_{step_1, agent_{10}} \end{bmatrix} \begin{bmatrix} a_{step_1, agent_1} \\ a_{step_1, agent_2} \\ \dots \\ a_{step_1, agent_{10}} \end{bmatrix}$$

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

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

$$\begin{bmatrix} s_{next_{step_1, agent_1}} \\ s_{next_{step_1, agent_2}} \\ \dots \\ s_{next_{step_1, agent_{10}}} \end{bmatrix}$$

Store training trajectory data

```
[25]: print(len(LOG_PROBS), LOG_PROBS[0].shape)
      print(len(VALUE), VALUE[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)
```

Training trajectory data from 10
agents each conducting 254 steps

```
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

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

```
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])
```

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

$$\begin{aligned}\Delta_{19} &= r_{19} + (\gamma * v_{20} * mask_{19} - v_{19}) \\ gae_{19 \sim 20} &= \Delta_{19} + \gamma * \tau * mask_{19} * gae_{20} \\ return_{19} &= gae_{19 \sim 20} + v_{19}\end{aligned}$$

...

$$\begin{aligned}\Delta_1 &= r_1 + (\gamma * v_2 * mask_1 - v_1) \\ gae_{1 \sim 20} &= \Delta_1 + \gamma * \tau * mask_1 * gae_{2 \sim 20} \\ return_1 &= gae_{1 \sim 20} + v_1\end{aligned}$$

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(VALUE).detach()
      MERGED_STATES     = torch.cat(STATES)
      MERGED_NEXT_STATES = torch.cat(NEXT_STATES)
      MERGED_ACTIONS    = torch.cat(ACTIONS)
      MERGED_ADVANTAGES = MERGED_RETURNS - MERGED_VALUES
```

2540 = 10 agents each conducting 254 steps

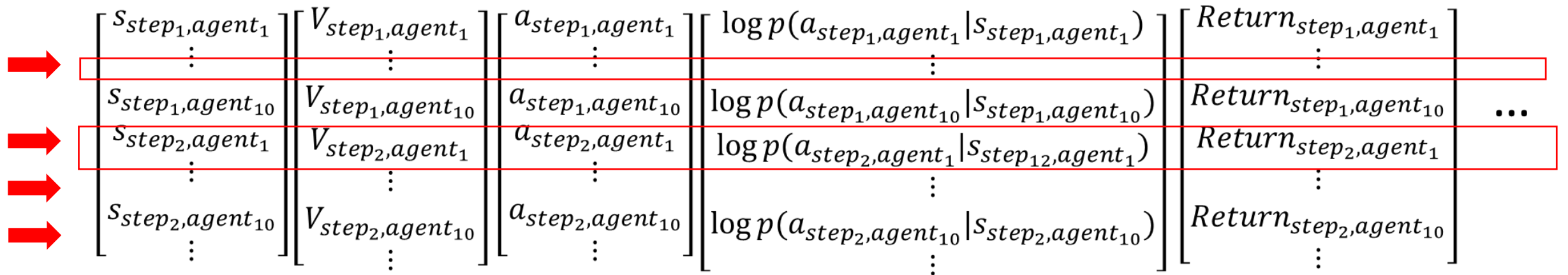
```
[30]: print(len(MERGED_RETURNS), MERGED_RETURNS[0].shape)
      print(len(MERGED_LOG_PROBS), MERGED_LOG_PROBS[0].shape)
      print(len(MERGED_VALUES), MERGED_VALUES[0].shape)
      print(len(MERGED_STATES), MERGED_STATES[0].shape)
      print(len(MERGED_NEXT_STATES), MERGED_NEXT_STATES[0].shape)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

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

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_ids, :]
```



$S_{step_1, agent_1}$	$V_{step_1, agent_1}$	$a_{step_1, agent_1}$	$\log p(a_{step_1, agent_1} S_{step_1, agent_1})$	$Return_{step_1, agent_1}$
\vdots	\vdots	\vdots	\vdots	\vdots
$S_{step_1, agent_{10}}$	$V_{step_1, agent_{10}}$	$a_{step_1, agent_{10}}$	$\log p(a_{step_1, agent_{10}} S_{step_1, agent_{10}})$	$Return_{step_1, agent_{10}}$
$S_{step_2, agent_1}$	$V_{step_2, agent_1}$	$a_{step_2, agent_1}$	$\log p(a_{step_2, agent_1} S_{step_2, agent_1})$	$Return_{step_2, agent_1}$
\vdots	\vdots	\vdots	\vdots	\vdots
$S_{step_2, agent_{10}}$	$V_{step_2, agent_{10}}$	$a_{step_2, agent_{10}}$	$\log p(a_{step_2, agent_{10}} S_{step_2, agent_{10}})$	$Return_{step_2, agent_{10}}$
\vdots	\vdots	\vdots	\vdots	\vdots

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

```
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()
```

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

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()
```

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

py version

1. Open terminal window from Anaconda
2. cd to the directory where the file "PPO_A2C_Walker_MLAgent_19.py" is located
3. >> python PPO_A2C_Walker_MLAgent_19.py
4. Press Play in Unity to start training

 C:\Windows\system32\cmd.exe

```
(IE562) C:\Users\ADMIN>cd C:\Users\ADMIN\desktop\ppo train
```

```
(IE562) C:\Users\ADMIN\desktop\ppo train>python PPO_A2C_Walker_MLAgent_19.py
```

```
cuda GeForce GTX 1660 SUPER
```

```
Please press play in Unity editor
```

```
2022-05-21 08:51:09.126205: W tensorflow/stream_executor/platform/default/dso_loader
```

```
library 'cuda64_110.dll'; dLError: cuda64_110.dll not found
```

```
2022-05-21 08:51:09.126335: I tensorflow/stream_executor/cuda/cuda_stub.cc:29
```

```
o not have a GPU set up on your machine.
```

py version

ct - Walker - PC, Mac & Linux Standalone - Unity 2021.1.23f1 Personal <DX11>

Assets GameObject Component Jobs Window Help

The screenshot shows the Unity 2021.1.23f1 Personal interface. The main view is a 3D scene with a character (a small white figure with a yellow hat) on a grid floor. The console window is open, showing the following output:

```
C:\Windows\system32\cmd.exe - python PPO_A2C_Walker_MLAgent_19.py
(IE562) C:\Users\ADMIN>cd C:\Users\ADMIN\desktop\ppo train
(IE562) C:\Users\ADMIN\desktop\ppo train>python PPO_A2C_Walker_MLAgent_19.py
cuda GeForce GTX 1660 SUPER
Please press play in Unity editor
2022-05-21 08:51:09.126205: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cuda64_110.dll'; dlderror: cuda64_110.dll not found
2022-05-21 08:51:09.126335: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.

No of training steps = 5080
Reward = 19.01
Critic loss = 0.74 Actor loss = -16.35

No of training steps = 7620
Reward = 19.1
Critic loss = 0.37 Actor loss = -16.1

No of training steps = 10160
Reward = 19.21
Critic loss = 0.57 Actor loss = -16.48

No of training steps = 12700
Reward = 19.13
Critic loss = 0.16 Actor loss = -16.53

No of training steps = 15240
Reward = 19.25
Critic loss = 0.18 Actor loss = -16.4
```

The console window also shows the Unity logo and the text "ModularFirstPersonController" at the bottom.

Calculate reward

```
if(steps > summary):  
    print("No of training steps = ", steps)  
    mean_reward = float(torch.mean(MERGED_RETURNS))  
    print("Reward = ", round(mean_reward, 2))  
    print("Critic loss = ", round(critic_loss, 2), " Actor loss = ", round(actor_loss, 2))  
  
    writer.add_scalar("Loss/Actor loss", actor_loss, steps)  
    writer.add_scalar("Loss/Critic loss", critic_loss, steps)  
    writer.add_scalar("Reward", actor_loss, steps)  
  
    fname = "NN_" + str(steps) + ".pth"  
    torch.save(NET.state_dict(), fname)  
    summary += SUMMARY_FREQ
```

Write reward and loss to tensorboard

```
from torch.utils.tensorboard import SummaryWriter
```

```
writer = SummaryWriter()
```

```
writer.add_scalar("Loss/Actor loss", actor_loss, steps)  
writer.add_scalar("Loss/Critic loss", critic_loss, steps)  
writer.add_scalar("Reward", actor_loss, steps)
```

Write reward and loss to tensorboard

- runs
- NN_5080.pth
- NN_7620.pth
- NN_10160.pth
- NN_12700.pth
- NN_15240.pth
- NN_20320.pth
- NN_22860.pth
- NN_25400.pth
- NN_27940.pth
- PPO_A2C_Walker_MLAgent_19.py

ADMIN > desktop > ppo train > runs

名稱

May21_09-24-25_DESKTOP-AQI6BIB

> ppo train > runs > May21_09-24-25_DESKTOP-AQI6BIB

名稱

修改日期

events.out.tfevents.1653096267.DESKT... 2022/5/21 上午 09:28

Visualize training performance using tensor board

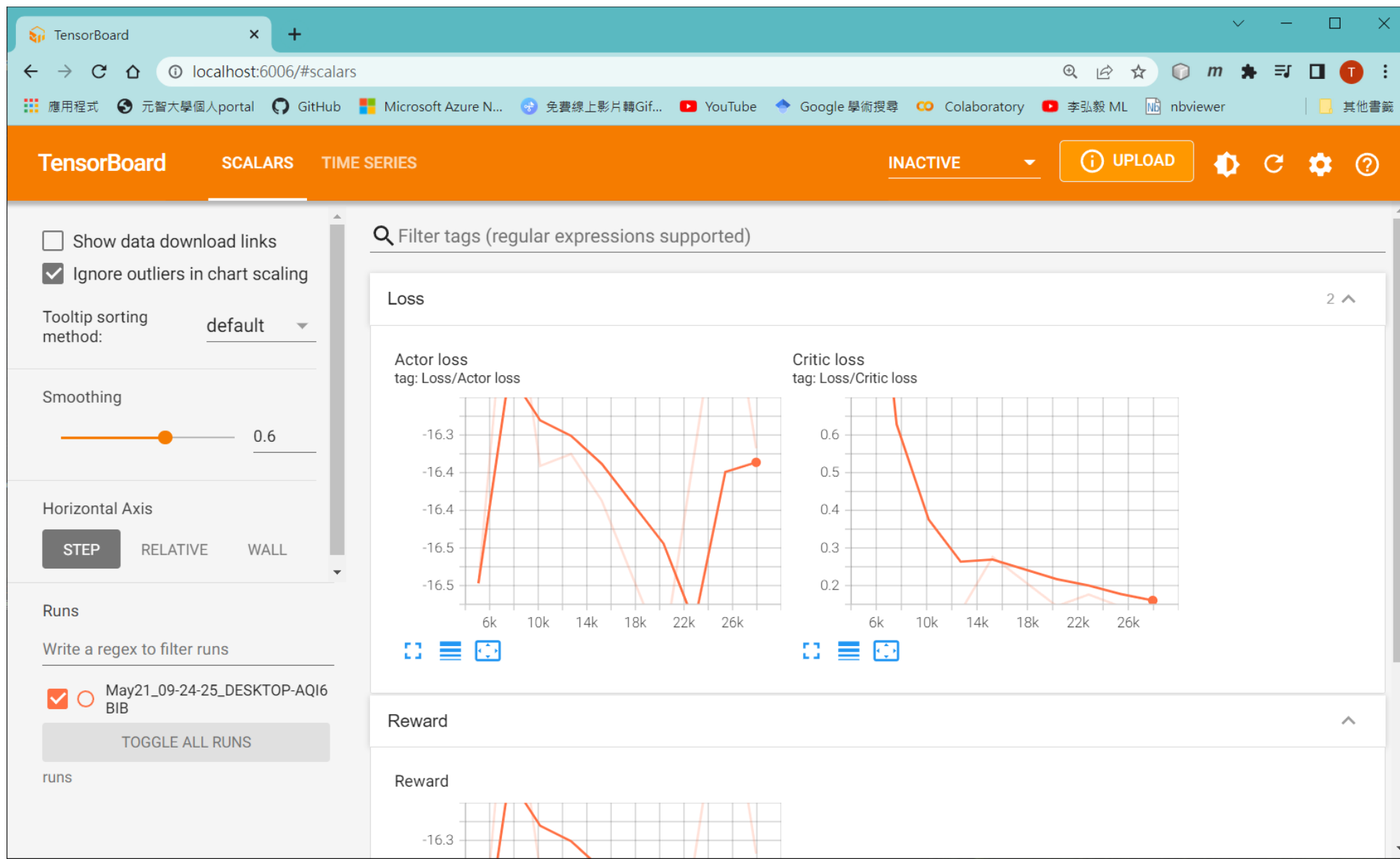
1. Open terminal window from Anaconda
2. cd to the directory where the file "PPO_A2C_Walker_MLAgent_19.py" is located
3. >> tensorboard --logdir=runs
4. Open web browser: localhost:6006

```
( IE562) C:\Users\ADMIN>cd C:\Users\ADMIN\desktop\ppo train  
( IE562) C:\Users\ADMIN\desktop\ppo train>tensorboard --logdir=runs  
2022-05-21 09:05:45.345759: W tensorflow/stream_executor/platform/default/dyn  
ry 'cudart64_110.dll': dLError: cudart64_110.dll not found
```

```
please make sure the missing libraries mentioned above are installed properly if you would like  
side at https://www.tensorflow.org/install/gpu for how to download and setup the required lib  
Skipping registering GPU devices...  
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all  
TensorBoard 2.6.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

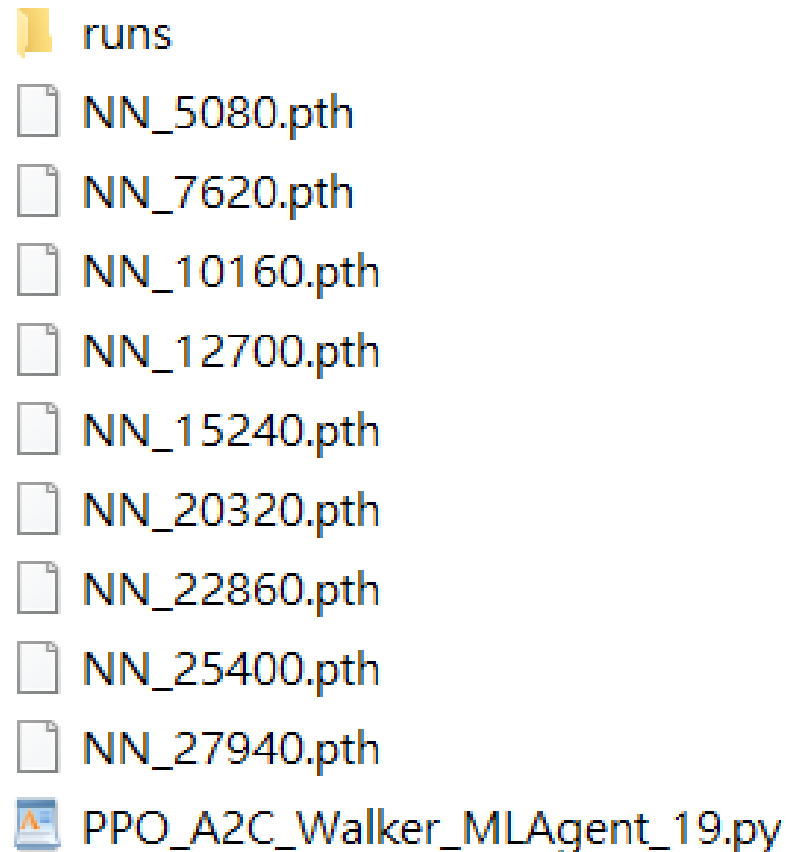

Visualize training performance using tensor board

23




Save NN


```
fname = "NN_" + str(steps) + ".pth"  
torch.save(NET.state_dict(), fname)
```




Training parameter setting

- Unity ML Agent Github → docs → Training configuration file

 main ▾ [ml-agents](#) / [docs](#) / Training-Configuration-File.md

 miguelalonsojr Removed mention of release 19 (#5637) ... ✕

👤 12 contributors 

☰ 229 lines (180 sloc) | 58.3 KB

Training Configuration File

Table of Contents

- [Common Trainer Configurations](#)
- [Trainer-specific Configurations](#)
 - [PPO-specific Configurations](#)
 - [SAC-specific Configurations](#)
- [Reward Signals](#)

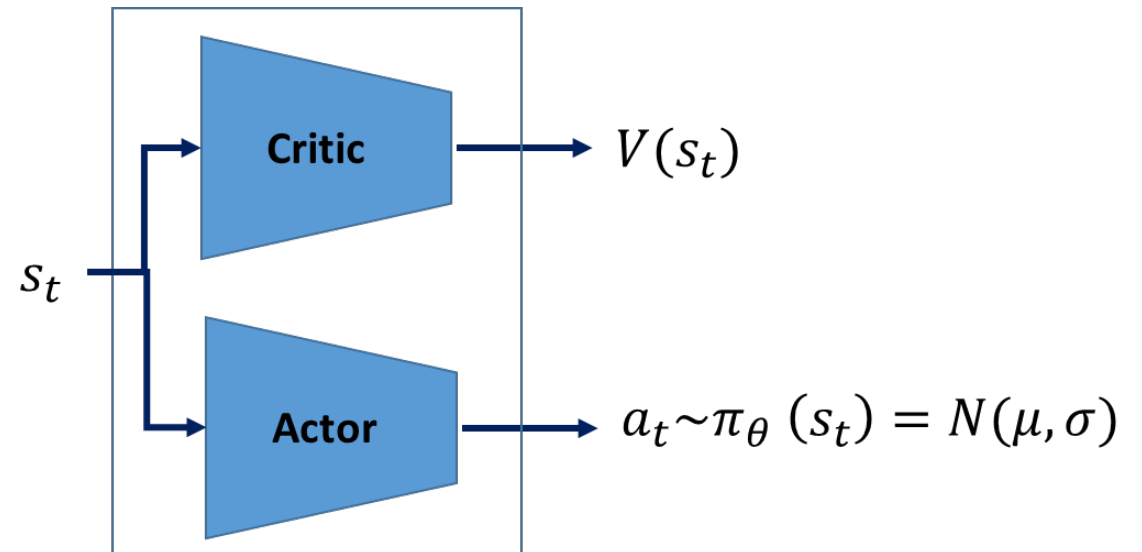
Class practice – NN structure

Unity ML Agent implementation

```
network_settings:  
  normalize: true  
  hidden_units: 512  
  num_layers: 3  
  vis_encode_type: simple
```

My py implementation

```
N_STATES = 243  
N_ACTIONS = 39  
HIDDEN_UNITS = 512
```



NN structure

<code>network_settings -></code> <code>hidden_units</code>	<p>(default = 128) Number of units in the hidden layers of the neural network. Correspond to how many units are in each fully connected layer of the neural network. For simple problems where the correct action is a straightforward combination of the observation inputs, this should be small. <u>For problems where the action is a very complex interaction between the observation variables, this should be larger.</u></p> <p>Typical range: 32 - 512</p>
<code>network_settings -></code> <code>num_layers</code>	<p>(default = 2) The number of hidden layers in the neural network. Corresponds to how many hidden layers are present after the observation input, or after the CNN encoding of the visual observation. For simple problems, fewer layers are likely to train faster and more efficiently. <u>More layers may be necessary for more complex control problems.</u></p> <p>Typical range: 1 - 3</p>
<code>network_settings -></code> <code>normalize</code>	<p>(default = false) Whether normalization is applied to the vector observation inputs. This normalization is based on the running average and variance of the vector observation. Normalization can be helpful in cases with complex continuous control problems, but may be harmful with simpler discrete control problems.</p>

Class practice – Number of training steps and summarize frequency

Unity ML Agent implementation

```
keep_checkpoints: 5  
max_steps: 30000000  
time_horizon: 1000  
summary_freq: 30000
```

My py implementation

```
MAX_STEPS = 30000  
SUMMARY_FREQ = 3000  
TIME_HORIZON = 1000
```

Class practice – Buffer size and batch size

Unity ML Agent implementation

batch_size: 2048
buffer_size: 20480
learning_rate: 0.0003

My py implementation

```
INTERACTION_STEPS = 254  
BATCH_SIZE = 254  
  
LEARNING_RATE = 0.0003  
N_AGENTS = 10 #The num
```

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

Buffer size = No_Agents * Interaction_Steps

Buffer size and batch size

hyperparameters ->
batch_size

Number of experiences in each iteration of gradient descent. **This should always be multiple times smaller than `buffer_size`**. If you are using continuous actions, this value should be large (on the order of 1000s). If you are using only discrete actions, this value should be smaller (on the order of 10s).

Typical range: (Continuous - PPO): 512 - 5120 ; (Continuous - SAC): 128 - 1024 ; (Discrete, PPO & SAC): 32 - 512 .

hyperparameters ->
buffer_size

(default = 10240 for PPO and 50000 for SAC)

PPO: Number of experiences to collect before updating the policy model. Corresponds to how many experiences should be collected before we do any learning or updating of the model. **This should be multiple times larger than `batch_size`**. Typically a larger `buffer_size` corresponds to more stable training updates.

SAC: The max size of the experience buffer - on the order of thousands of times longer than your episodes, so that SAC can learn from old as well as new experiences.

Typical range: PPO: 2048 - 409600 ; SAC: 50000 - 1000000

Class practice – Training epochs

Unity ML Agent implementation

lambd: 0.95
num epoch: 3
 learning_rate_schedule

My py implementation

```

GAMMA = 0.995
LAMBD = 0.95
BETA = 0.005
EPSILON = 0.2
N_EPOCH = 3
  
```

```

def ppo_update():
    for epoch in range(N_EPOCH):
        for b_s, b_a, b_s_, b_old_LOG_PROBS, b_
            dist, value = NET(b_s.to(device))
            critic_loss = (b_return.to(device)
            entropy = dist.entropy().mean()
            b_s_new = dist.sample()
  
```

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

Training epochs

hyperparameters ->
num_epoch

(default = 3) Number of passes to make through the experience buffer when performing gradient descent optimization. The larger the batch_size, the larger it is acceptable to make this. Decreasing this will ensure more stable updates, at the cost of slower learning.

Typical range: 3 - 10

Class practice – Reward signals

- Test with ML agent and my py

Unity ML Agent implementation

```
reward_signals:
  extrinsic:
    gamma: 0.995
    strength: 1.0
```

My py implementation

```
GAMMA = 0.995
LAMBD = 0.95
BETA = 0.005
EPSILON = 0.2
N_EPOCH = 3
```

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

$$gae_{19 \sim 20} = \Delta_{19} + \gamma * \tau * mask_{19} * gae_{20}$$

$$return_{19} = gae_{19 \sim 20} + v_{19}$$

Reward signals

Setting	Description
extrinsic -> strength	<p>(default = 1.0) Factor by which to multiply the reward given by the environment. Typical ranges will vary depending on the reward signal.</p> <p>Typical range: 1.00</p>
extrinsic -> gamma	<p>(default = 0.99) Discount factor for future rewards coming from the environment. This can be thought of as how far into the future the agent should care about possible rewards. In situations when the agent should be acting in the present in order to prepare for rewards in the distant future, this value should be large. <u>In cases when rewards are more immediate, it can be smaller.</u> Must be strictly smaller than 1.</p> <p>Typical range: 0.8 - 0.995</p>

Class practice – Accumulated rewards

Unity ML Agent implementation

```

hyperparameters:
  batch_size: 2048
  buffer_size: 20480
  learning_rate: 0.0003
  beta: 0.005
  epsilon: 0.2
  lambda: 0.95
  num_epoch: 3
  learning_rate_schedule: linear

```

My py implementation

```

GAMMA = 0.995
LAMBD = 0.95
BETA = 0.005
EPSILON = 0.2
N_EPOCH = 3

```

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

$$gae_{19 \sim 20} = \Delta_{19} + \gamma * \tau * mask_{19} * gae_{20}$$

$$return_{19} = gae_{19 \sim 20} + v_{19}$$

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

Accumulated rewards

hyperparameters ->

lambd

(default = 0.95) Regularization parameter (lambda) used when calculating the Generalized Advantage Estimate (GAE). This can be thought of as how much the agent relies on its current value estimate when calculating an updated value estimate. Low values correspond to relying more on the current value estimate (which can be high bias), and high values correspond to relying more on the actual rewards received in the environment (which can be high variance). The parameter provides a trade-off between the two, and the right value can lead to a more stable training process.

Typical range: 0.9 - 0.95

Class practice – Entropy regularization

Unity ML Agent implementation

```
learning_rate: 0.  
beta: 0.005  
epsilon: 0.2  
lambda: 0.95
```

My py implementation

```
GAMMA = 0.995  
LAMBDA = 0.95  
BETA = 0.005  
EPSILON = 0.2  
N_EPOCH = 3
```

$$L = 0.5 \cdot Loss_V + Loss_\pi - 0.005 \cdot entropy$$

Entropy regularization

hyperparameters ->
beta

(default = $5.0e-3$) Strength of the entropy regularization, which makes the policy "more random." This ensures that agents properly explore the action space during training. Increasing this will ensure more random actions are taken. This should be adjusted such that the entropy (measurable from TensorBoard) slowly decreases alongside increases in reward. If entropy drops too quickly, increase beta. If entropy drops too slowly, decrease beta.

Typical range: $1e-4$ - $1e-2$

HW4 – PPO-A2C training

- Group of max. 3
- Select a rewarding scheme you like the most from HW3
- Try different PPO-AC training parameters (smaller NN, larger buffer size and epochs, larger entropy) and train with both ML agent implementation and my py implementation
- Use tensorboard visualization (reward and loss plots) to compare and discuss
- Due: Next class meeting
- Upload ppt to Teams

HW4 – PPO-A2C training

- Experiment 1: Smaller NN (243-512-512-512-39 \rightarrow 243-N-N-N-39) (try N=254 or 128)
- Experiment 2: Larger buffer and batch size (20480, 2048) \rightarrow ... (expect more stable training)
- Experiment 3: Larger training epochs (3 \rightarrow 10) (expect more unstable training)
- Experiment 4: Larger entropy regularization (0.005 \rightarrow 0.05) (expect more unstable training but more interesting behaviors)

Reference – OpenAI version

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