Train ML Agent

Define hyper parameters

mlagents-learn MobilRobot.yaml --run-id=1 --force

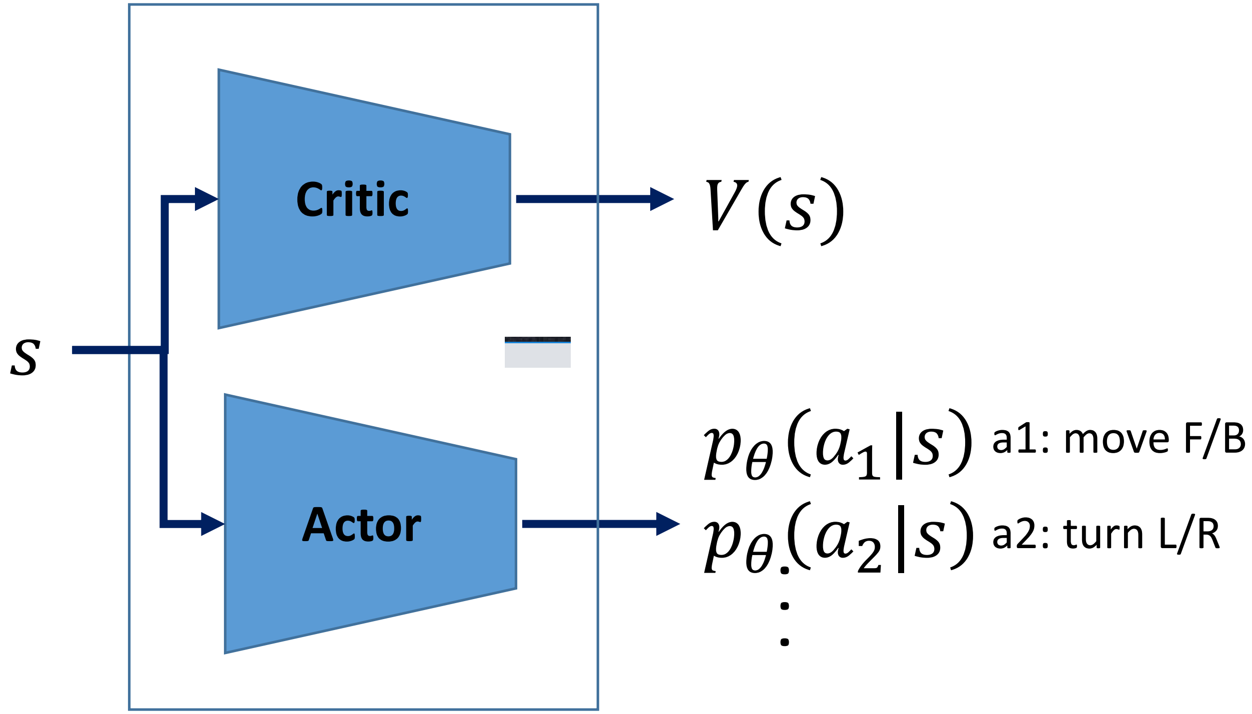
Exemplar PPO-AC code: https://github.com/TienLungSun/RL-Mobile-Robot/tree/main/LearnPPO-AC

Two neural networks

Actor – Learns the best actions (that can have maximum long-term rewards)
Critic – Learns the expected value of the long-term reward.

```
class ActorCritic(nn.Module):
    def __init__(self, num_inputs, num_outputs, hic
        super(ActorCritic, self).__init__()

        self.critic = nn.Sequential(
            nn.Linear(num_inputs, hidden_size1),
            nn.LayerNorm(hidden_size1),
            nn.Tanh(),
            nn.Linear(hidden_size1, hidden_size2),
            nn.LayerNorm(hidden_size2),
            nn.Tanh(),
            nn.Linear(hidden_size2, 1),
        )

        self.actor = nn.Sequential(
            nn.Linear(num_inputs, hidden_size1),
            nn.LayerNorm(hidden_size1),
            nn.Tanh(),
```

Exemplar PPO-AC code



Two neural networks

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

For simple problems where the correct action is a straightforward combination of the observation inputs, this should be small. For problems where the action is a very complex interaction between the observation variables, this should be larger.

For simple problems, fewer layers are likely to train faster and more efficiently. More layers may be necessary for more complex control problems.

Interact with training environment to collect data

while frame < max_frames
    while (episodes < buffer_size)
        get s_1
        for step in range(time_horizon):
            $(s_1, a_1, r_1, s_2), v_1, \log p_1$
            $(s_2, a_2, r_2, s_3), v_2, \log p_2$
                …
            $(s_N, a_N, r_N, s_{N+1}), v_N, \log p_N$
    calculate GAE
    use GAE to update NN

```
hyperparameters:
  batch_size: 2048
  buffer_size: 20480
  learning_rate: 0.0003
```

```
keep_checkpoints: 5
max_steps: 5000000
time_horizon: 1000
summary_freq: 30000
threaded: true
```

Time_horizon: This parameter trades off between a less biased, but higher variance estimate (long time horizon) and more biased, but less varied estimate (short time horizon). In cases where there are frequent rewards within an episode, or episodes are prohibitively large, a smaller number can be more ideal. This number should be large enough to capture all the important behavior within a sequence of an agent's actions.

Buffer size: larger value corresponds to more stable training updates

Interact with training environment to collect data

```
for step in range(num_steps):
    if(__printDetails and (step+1) % 5==0):
        print(step+1, end = ", ")
    state = torch.FloatTensor(state).to(device)
    dist, value = model(state)

    action = dist.sample()
    if(int(torch.isnan(torch.min(action))) == 1) : #we
        print("Error: distribution=", dist, "%.2f, %.2
    env.set_actions(behaviorName, np.array(action.cpu(
    env.step()

    step_result = env.get_steps(behaviorName)
    DecisionSteps = step_result[0]
    TerminalSteps = step_result[1]
    if(len(TerminalSteps) >0): # if episode is termine
        next_state = TerminalSteps.obs[0]
        reward = TerminalSteps.reward
        if(__printDetails):
            print("Reach goal, r= %.2f" % reward)
        mask=[0.0]
    elif(len(DecisionSteps) >0): #otherwise collect (s
        next_state = DecisionSteps.obs[0]
        reward = DecisionSteps.reward
        if(__printDetails and reward >= 5):
            print("Hit obstacle!, r=", reward)
        mask=[1.0]

    log_prob = dist.log_prob(action)
    entropy += dist.entropy().mean()

    log_probs.append(log_prob)
    values.append(value)
    rewards.append(torch.FloatTensor(reward).unsqueeze
    masks.append(torch.FloatTensor(mask).unsqueeze(1).

    states.append(state)
    actions.append(action)
```

```
// s = (1, 0, 0, theta, d1~dn)
sensor.AddObservation(1);
sensor.AddObservation(0);
sensor.AddObservation(0);
Vector3 targetDir = goal.transform.position - r
float facingAngle = Vector3.SignedAngle(robot.t
sensor.AddObservation(facingAngle); // theta

for (int i = 0; i < 18; i++) //add dista
{
    if (Physics.Raycast(distSensor[i].po
    {
        sensor.AddObservation(hit.distan
    }
    else
    {
        sensor.AddObservation(1);
    }
}
```

Interact with training environment to collect data

```python
for step in range(num_steps):
    if(__printDetails and (step+1) % 5==0):
        print(step+1, end = ", ")
    state = torch.FloatTensor(state).to(device)
    dist, value = model(state)

    action = dist.sample()
    if(int(torch.isnan(torch.min(action))) == 1) : #we have
        print("Error: distribution=", dist, "%.2f, %.2f" %
    env.set_actions(behaviorName, np.array(action.cpu()))
    env.step()

    step_result = env.get_steps(behaviorName)
    DecisionSteps = step_result[0]
    TerminalSteps = step_result[1]
    if(len(TerminalSteps) >0): # if episode is termined, co
        next_state = TerminalSteps.obs[0]
```

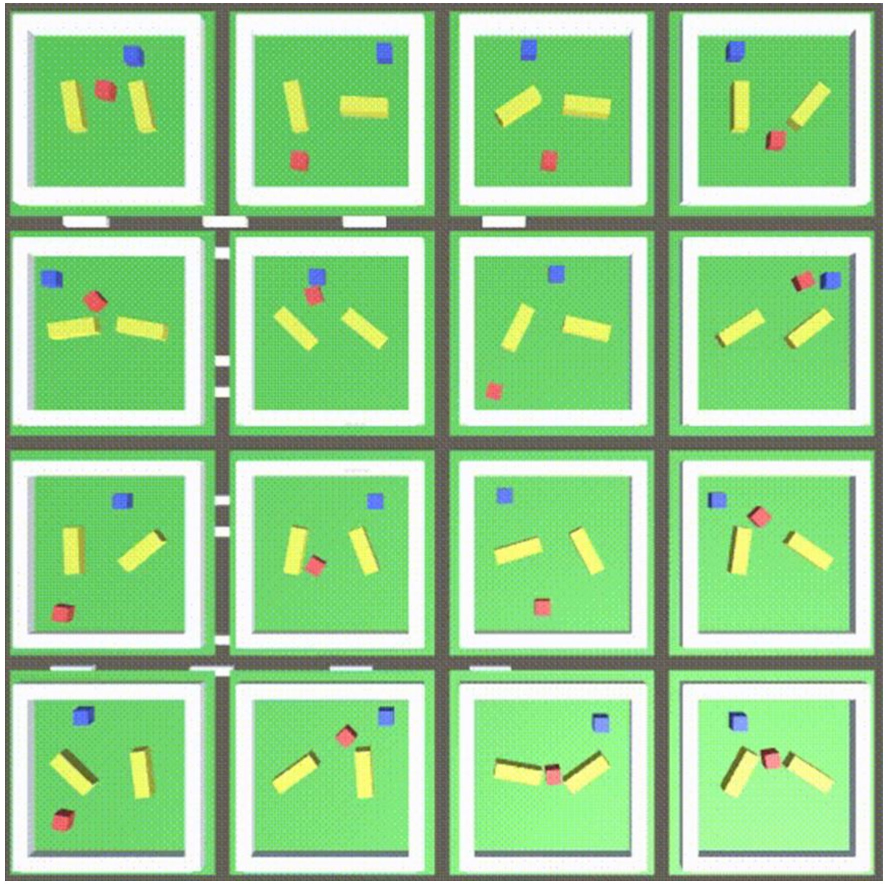
```csharp
public override void OnActionReceived(float[] vectorAction)
{
    int oldStage = DetermineStage();
    robot.transform.Translate(0, 0, vectorAction[0]*0.4f);
    robot.transform.Rotate(0, vectorAction[1]*10.0f, 0);
```

Interact with training environment to collect data

```python
for step in range(num_steps):
    if(__printDetails and (step+1) % 5==0):
        print(step+1, end = ", ")
    state = torch.FloatTensor(state).to(device)
    dist, value = model(state)

    action = dist.sample()
    if(int(torch.isnan(torch.min(action))) == 1) : #we
        print("Error: distribution=", dist, "%.2f, %.2
    env.set_actions(behaviorName, np.array(action.cpu(
    env.step()

    step_result = env.get_steps(behaviorName)
    DecisionSteps = step_result[0]
    TerminalSteps = step_result[1]
    if(len(TerminalSteps) >0): # if episode is termine
        next_state = TerminalSteps.obs[0]
        reward = TerminalSteps.reward
        if(__printDetails):
            print("Reach goal, r= %.2f" % reward)
        mask=[0.0]
    elif(len(DecisionSteps) >0): #otherwise collect (s
        next_state = DecisionSteps.obs[0]
        reward = DecisionSteps.reward
        if(__printDetails and reward >= 5):
            print("Hit obstacle!, r=", reward)
        mask=[1.0]

    log_prob = dist.log_prob(action)
    entropy += dist.entropy().mean()

    log_probs.append(log_prob)
    values.append(value)
    rewards.append(torch.FloatTensor(reward).unsqueeze
    masks.append(torch.FloatTensor(mask).unsqueeze(1).

    states.append(state)
    actions.append(action)
```

```csharp
AddReward(-0.005f * newStage); //punish more st
AddReward(-0.005f * (oldStage-newStage)); //pur

//Part II: rewards based on distance sensors, e
for (int i = 0; i < 18; i++)
{
    //Debug.DrawRay(distSensor[i].position, dis
    if (Physics.Raycast(distSensor[i].position,
    {
        if (hit.collider.tag == "goal" && ((i >
        {
            //print("Goal!");
            AddReward(100.0f);
            EndEpisode();
        }
        else if (hit.distance < 1.0f)  //too cl
        {
            Debug.DrawRay(distSensor[i].positic
            AddReward(-0.5f);
```

Interact with training environment to collect data



```
keep_checkpoints: 5
max_steps: 5000000
time_horizon: 1000
summary_freq: 30000
threaded: true
```

By default, model updates can happen while the environment is being stepped. This violates the on-policy assumption of PPO slightly in exchange for a training speedup. To maintain the strict on-policy of PPO, you can disable parallel updates by setting threaded to false.

Calculate GAE

After interacting with VE k steps, we collect (s_1, a_1, r_1, s_2) … (s_k, a_k, r_k, s_{k+1}).
Then we use these data to calculate GAE

```
In [4]: def compute_gae(next_value, rewards, masks, values, gamma=0.99, tau=0.95):
            values = values + [next_value]
            gae = 0
            returns = []
            for step in reversed(range(len(rewards))):
                delta = rewards[step] + gamma * values[step + 1] * masks[step] - values[step]
                gae = delta + gamma * tau * masks[step] * gae
                returns.insert(0, gae + values[step])
            return returns
```

Δ = reward of this step + expected reward of next step

gae = Δ + accumulated gae

Return = gae + v

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

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

…

$$\Delta_{1} = r_{1} + \gamma * v_{2} * mask_{1} - v_{1}$$
$$gae_{1\sim 20} = \Delta_{1} + \gamma * \tau * mask_{1} * gae_{2\sim 20}$$
$$return_{1} = gae_{1\sim 20} + v_{1}$$

Calculate GAE

```
def compute_gae(next_value, rewards, masks, values, gamma=0.99, tau=0.95):
    values = values + [next_value]
    gae = 0
    returns = []
    for step in reversed(range(len(rewards))):
        delta = rewards[step] + gamma * values[step + 1] * masks[step] - va
        gae = delta + gamma * tau * masks[step] * gae
        returns.insert(0, gae + values[step])
    return returns
```

hyperparameters:
  batch_size: 2048
  buffer_size: 20480
  learning_rate: 0.0003
  beta: 0.005
  epsilon: 0.2
  lambd: 0.95

reward_signals:
  extrinsic:
    gamma: 0.995
    strength: 1.0

$$\Delta_{20} = r_{20} + \gamma * v_{21} * mask_{20} - v_{20}$$

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

$$return_{20} = gae_{20} + v_{20}$$

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

$$gae_{19\sim20} = \Delta_{19} + \gamma * \tau * mask_{19} * gae_{20}$$

$$return_{19} = gae_{19\sim20} + v_{19}$$

...

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).

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. In cases when rewards are more immediate, it can be smaller. Must be strictly smaller than 1.

Combine data collected from different agents

```
next_state = torch.FloatTensor(next_state).to(device)
_, next_value = model(next_state)
returns = compute_gae(next_value, rewards, masks, values)

returns   = torch.cat(returns).detach()
log_probs = torch.cat(log_probs).detach()
values    = torch.cat(values).detach()
states    = torch.cat(states)
actions   = torch.cat(actions)
advantage = returns - values
```

N: no. of agents
K: time horizon

$$\begin{bmatrix} \vec{s}_{1,step1} \\ \vdots \\ \vec{s}_{N,step1} \\ \vdots \\ \vec{s}_{1,stepk} \\ \vdots \\ \vec{s}_{N,stepk} \end{bmatrix} \begin{bmatrix} \vec{a}_{1,step1} \\ \vdots \\ \vec{a}_{N,step1} \\ \vdots \\ \vec{a}_{1,stepk} \\ \vdots \\ \vec{a}_{N,stepk} \end{bmatrix} \quad \begin{bmatrix} v_{1,step1} \\ \vdots \\ v_{N,step1} \\ \vdots \\ v_{1,stepk} \\ \vdots \\ v_{N,stepk} \end{bmatrix} \begin{bmatrix} return_{1,step1} \\ \vdots \\ return_{N,step1} \\ \vdots \\ return_{1,stepk} \\ \vdots \\ return_{N,stepk} \end{bmatrix} \begin{bmatrix} gae_{1,step1} \\ \vdots \\ gae_{N,step1} \\ \vdots \\ gae_{1,stepk} \\ \vdots \\ gae_{N,stepk} \end{bmatrix}$$

Sampling a batch of data to train NN

In [5]:

```
import numpy as np

def ppo_iter(mini_batch_size, states, actions, log_probs, returns, advantage):
    batch_size = states.size(0)
    for _ in range(batch_size // mini_batch_size):
        rand_ids = np.random.randint(0, batch_size, mini_batch_size)
        yield states[rand_ids, :], actions[rand_ids, :], log_probs[rand_ids, :]
```

hyperparameters:
batch_size: 2048

$$\begin{bmatrix} \vec{s}_{1,step1} \\ \vdots \\ \vec{s}_{N,step1} \\ \vdots \\ \vec{s}_{1,step20} \\ \vdots \\ \vec{s}_{N,step20} \end{bmatrix} \begin{bmatrix} \vec{a}_{1,step1} \\ \vdots \\ \vec{a}_{N,step1} \\ \vdots \\ \vec{a}_{1,step20} \\ \vdots \\ \vec{a}_{N,step20} \end{bmatrix} \qquad \begin{bmatrix} return_{1,step1} \\ \vdots \\ return_{N,step1} \\ \vdots \\ return_{1,step20} \\ \vdots \\ return_{N,step20} \end{bmatrix} \begin{bmatrix} gae_{1,step1} \\ \vdots \\ gae_{N,step1} \\ \vdots \\ gae_{1,step20} \\ \vdots \\ gae_{N,step20} \end{bmatrix} \quad N * 20$$

Update NN

beta: 0.005
epsilon: 0.2
lambd: 0.95
num_epoch: 3

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.

```
def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns, advant
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, return_, advantage in ppo_iter(mini_batch_siz
s):
            dist, value = model(state)
            entropy = dist.entropy().mean()
            new_log_probs = dist.log_prob(action)

            ratio = (new_log_probs - old_log_probs).exp()
            surr1 = ratio * advantage
            surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_param) * advantage

            actor_loss  = - torch.min(surr1, surr2).mean()
            critic_loss = (return_ - value).pow(2).mean()

            loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    return float(loss)
```

Loss function

- Define a loss function $\mathcal{L}(f)$ that describe the error between y^n and $\hat{y}^n$.
- Find the optimal parameters that minimize $\mathcal{L}(f)$

```
def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns, advant
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, return_, advantage in ppo_iter(mini_batch_siz
s):
            dist, value = model(state)
            entropy = dist.entropy().mean()
            new_log_probs = dist.log_prob(action)

            ratio = (new_log_probs - old_log_probs).exp()
            surr1 = ratio * advantage
            surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_param) * advantage

            actor_loss  = - torch.min(surr1, surr2).mean()
            critic_loss = (return_ - value).pow(2).mean()

            loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    return float(loss)
```

$$L = c_v L_v + L_\pi - \beta L_{reg}$$

Critic loss

```
def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns, advant
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, return_, advantage in ppo_iter(mini_batch_siz
s):
            dist, value = model(state)
            entropy = dist.entropy().mean()
            new_log_probs = dist.log_prob(action)

            ratio = (new_log_probs - old_log_probs).exp()
            surr1 = ratio * advantage
            surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_param) * advantage

            actor_loss  = - torch.min(surr1, surr2).mean()
            critic_loss = (return_ - value).pow(2).mean()

            loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    return float(loss)
```

$$L_v = MSE\ of\ (return - v)$$

$$return_i = gae_{i\sim K} + v_i$$

$$gae_{i\sim K} = \Delta_i + \gamma * \tau * mask_i * gae_{i+1\sim K}$$

Actor loss

```
def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns, advant
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, return_, advantage in ppo_iter(mini_batch_siz
s):
            dist, value = model(state)
            entropy = dist.entropy().mean()
            new_log_probs = dist.log_prob(action)

            ratio = (new_log_probs - old_log_probs).exp()
            surr1 = ratio * advantage
            surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_param) * advantage

            actor_loss  = - torch.min(surr1, surr2).mean()
            critic_loss = (return_ - value).pow(2).mean()

            loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy
```

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

Actor loss

Setting epsilon small will result in more stable updates, but will also slow the training process.

buffer_size: 20480
learning_rate: 0.0003
beta: 0.005
epsilon: 0.2

```
def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns, advantages, clip_param=0.2):
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, return_, advantage in ppo_iter(mini_batch_size, states, actions, lo
vantages):
            dist, value = model(state)
            entropy = dist.entropy().mean()
            new_log_probs = dist.log_prob(action)

            ratio = (new_log_probs - old_log_probs).exp()
            surr1 = ratio * advantage
            surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_param) * advantage

            actor_loss  = - torch.min(surr1, surr2).mean()
            critic_loss = (return_ - value).pow(2).mean()
```

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

Entropy regularization

```
def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_prob
m=0.2):
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, return_, advantage in ppo_i
ons, log_probs, returns, advantages):
            dist, value = model(state)
            entropy = dist.entropy().mean()
            new_log_probs = dist.log_prob(action)

            ratio = (new_log_probs - old_log_probs).exp()
            surr1 = ratio * advantage
            surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_p

            actor_loss  = - torch.min(surr1, surr2).mean()
            critic_loss = (return_ - value).pow(2).mean()

            loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy

            optimizer.zero_grad()
            loss.backward()
```

```
behaviors:
  MobileRobot:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.005
```

$$L = c_v L_v + L_\pi - \beta L_{reg}$$

Increasing beta will ensure more random actions are taken. Beta should be adjusted such that the entropy slowly decreases alongside increases in reward. If entropy drops too quickly, increase beta. If entropy drops too slowly, decrease beta.

Learning rate

hyperparameters:
  batch_size: 2048
  buffer_size: 20480
  learning_rate: 0.0003

epsilon: 0.2
lambd: 0.95
num_epoch: 3
learning_rate_schedule: linear

```
def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns, advant
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, return_, advantage in ppo_iter(mini_batch_siz
s):
            dist, value = model(state)
            entropy = dist.entropy().mean()
            new_log_probs = dist.log_prob(action)

            ratio = (new_log_probs - old_log_probs).exp()
            surr1 = ratio * advantage
            surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 +

            actor_loss  = - torch.min(surr1, surr2).mean()
            critic_loss = (return_ - value).pow(2).mean()

            loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    return float(loss)
```

For PPO, we recommend decaying learning rate until max_steps so learning converges more stably. Linear decays the learning_rate linearly, reaching 0 at max_steps, while constant keeps the learning rate constant for the entire training run.

Maximum the expected reward

$$\tau = \left(s_1, a_1, r_1, s_2, a_2, r_{2,} \cdots s_T, a_T\right)$$

$$p_\theta(\tau) = p(s_1)p_\theta(a_1|s_1)p(s_2|s_1,a_1)p_\theta(a_2|s_2)p(s_3|s_2,a_2)\cdots$$

$$\mathrm{R}(\tau) = \sum_{\mathrm{t}=1}^{\mathrm{T}} \mathrm{r_t}$$

$$\bar{R}_\theta = \sum R(\tau)\, p_\theta(\tau) = E_{\tau\sim p_\theta(\tau)}[R(\tau)]$$

Max $E[\bar{R}_\theta]$

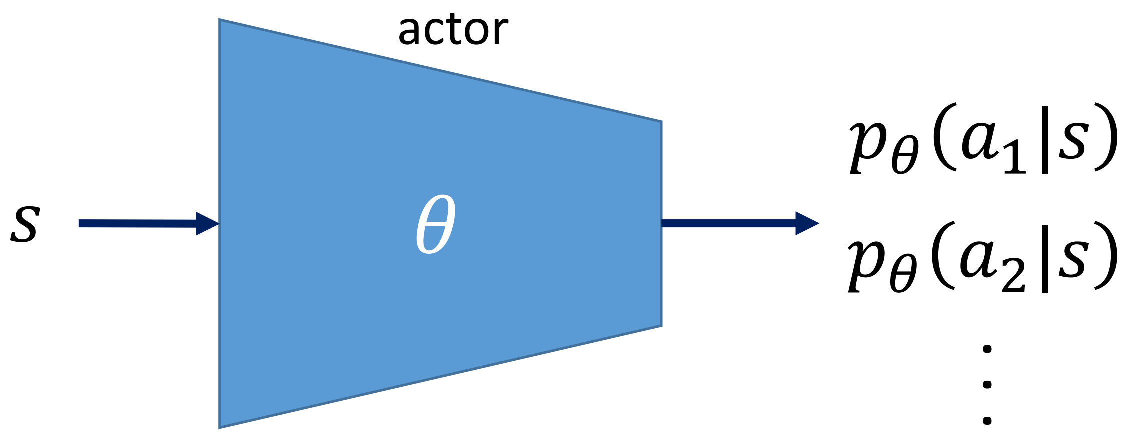
$$\max_\theta E[\bar{R}_\theta]$$

Gradient of the expected value

$$\nabla\bar{R}_\theta = \sum R(\tau)\,\nabla p_\theta(\tau) = E_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla \log p_\theta(\tau)] \approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)\nabla \log p_\theta(\tau^n)$$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla \log p_\theta(a_t^n|s_t^n)$$

Use $\nabla \bar{R}_\theta$ to update policy network



$$\theta^{\pi'} \leftarrow \theta^{\pi} + \eta \nabla \bar{R}_\theta$$

$$\nabla \bar{R}_\theta = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n)$$

Tips to calculate $\nabla\bar{R}_\theta$

$$\nabla\bar{R}_\theta = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla\log p_\theta(a_t^n|s_t^n)$$

Add a baseline to calculate the reward

$$\nabla\bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} (R(\tau^n) - b)\nabla\log p_\theta(a_t^n|s_t^n), \qquad b \approx E[R(\tau)]$$

$$\nabla\bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} \left(\sum\nolimits_{t'}^{T_n} r_{t'}^n - b\right)\nabla\log p_\theta(a_t^n|s_t^n)$$

Assign suitable time delayed credit

$$\nabla\bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} \left(\sum\nolimits_{t'}^{T_n} \gamma^{t'-t} r_{t'}^n - b\right)\nabla\log p_\theta(a_t^n|s_t^n), \gamma < 1$$

$$A^\theta(s_t, a_t) = \left(\sum\nolimits_{t'}^{T_n} \gamma^{t'-t} r_{t'}^n - b\right)$$

Off-policy $\nabla \bar{R}_\theta$

On-policy

$$\nabla \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} A^\theta(s_t, a_t)\nabla \log p_\theta(a_t^n|s_t^n), \gamma < 1 \qquad A^\theta(s_t, a_t) = \left(\sum_{t'}^{T_n} \gamma^{t'-t} r_{t'}^n - b\right)$$

Importance sampling

$$E_{x\sim \mathrm{p}}[f(x)] = E_{x\sim \mathrm{q}}\left[f(x)\frac{p(x)}{q(x)}\right]$$

$$Var_{x\sim \mathrm{q}}\left[f(x)\frac{p(x)}{q(x)}\right] = E_{x\sim \mathrm{q}}\left[\left(f(x)\frac{p(x)}{q(x)}\right)^2\right] - \left(E_{x\sim \mathrm{q}}\left[f(x)\frac{p(x)}{q(x)}\right]\right)^2$$

$$= E_{x\sim \mathrm{p}}\left[f(x)^2\frac{p(x)}{q(x)}\right] - \left(E_{x\sim \mathrm{p}}[f(x)]\right)^2$$

Off-policy

$$\nabla \bar{R}_\theta = E_{(s_t, a_t)\sim \pi_{\theta'}}\left[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t, a_t)\nabla \log p_\theta(a_t^n|s_t^n)\right]$$

Loss function derived from $\nabla \bar{R}_\theta$

Off-policy

$$\nabla \bar{R}_\theta = E_{(s_t,a_t)\sim\pi_{\theta'}}\left[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t,a_t) \nabla \log p_\theta(a_t^n|s_t^n)\right]$$

Sampling efficiency

Loss function

$$J^{\theta'}(\theta) = E_{(s_t,a_t)\sim\pi_{\theta'}}\left[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t,a_t)\right]$$

Proximal policy optimization (PPO)

$$J_{PPO}^{\theta'}(\theta) = J^{\theta'}(\theta) - \beta KL(\theta,\theta')$$

$$J_{PPO2}^{\theta'}(\theta) = \sum_{(s_t,a_t)} min\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t,a_t), clip\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon\right) A^{\theta'}(s_t,a_t)\right)$$

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Actor-critic strategy to calculate $\nabla \bar{R}_\theta$

$$\nabla \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}\left(\sum_{t'}^{T_n}\gamma^{t'-t}r_{t'}^{n} - b\right)\nabla \log p_\theta(a_t^n|s_t^n)$$

$$G_t^n = \sum_{t'}^{T_n}\gamma^{t'-t}r_{t'}^{n}$$

unstable when sampling amount is not large enough

Use expected value to reduce sampling variance

$\boxed{V^{\pi_\theta}(s_t^n)}$ Expected value of b

$$\nabla \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}\left(\boxed{\sum_{t'}^{T_n}\gamma^{t'-t}r_{t'}^{n}} - \boxed{b}\right)\nabla \log p_\theta(a_t^n|s_t^n)$$

$\boxed{E[G_t^n] = Q^{\pi_\theta}(s_t^n, a_t^n)}$ Expected value of G_t^n

Use one neural network that estimates V

$$Q^{\pi_\theta}(s_t^n, a_t^n) = \mathrm{E}[r_t^n + V^{\pi_\theta}(s_{t+1}^n)] = r_t^n + V^{\pi_\theta}(s_{t+1}^n)$$

$$Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n) = r_t^n + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)$$

$$\boxed{A^\theta(s_t, a_t) = \left(r_t^n + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)\right)}$$

Temporal difference to calculate V

$$A^{\theta}(s_t, a_t) = \left(r_t^n + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)\right)$$

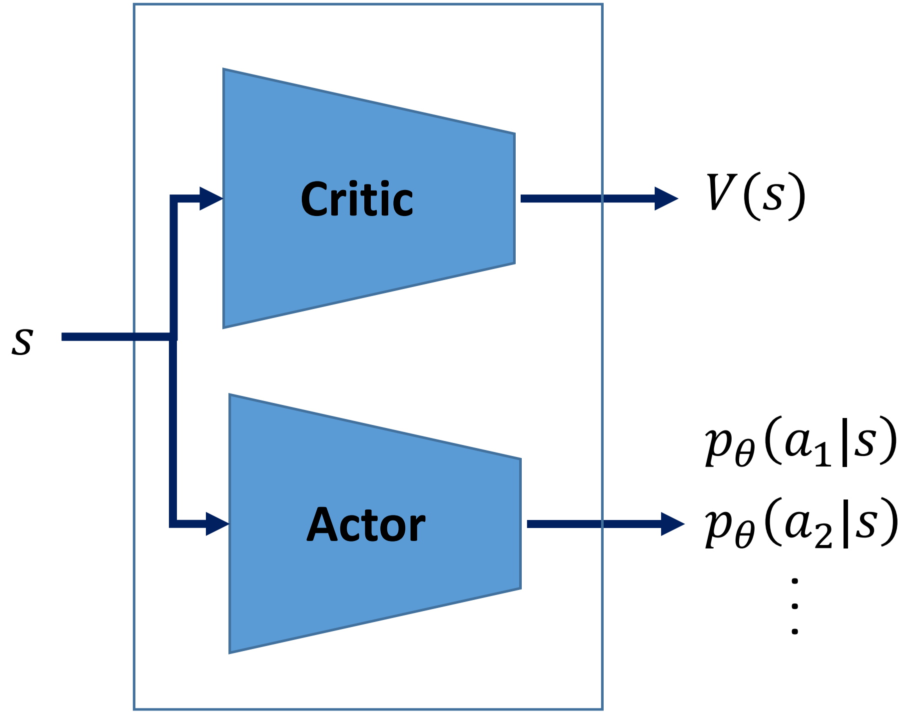
Monte-Carlo approach $V^{\pi_\theta}(\ s_a) \leftrightarrow G_a$ Until the end of the episode, the cumulated reward is G_a

Temporal-difference approach

$$V^{\pi_\theta}(\ s_t) + r_t = V^{\pi_\theta}(s_{t+1})$$

$$V^{\pi_\theta}(\ s_t) - V^{\pi_\theta}(s_{t+1}) \ \leftrightarrow r_t$$

Train the network



TD Error

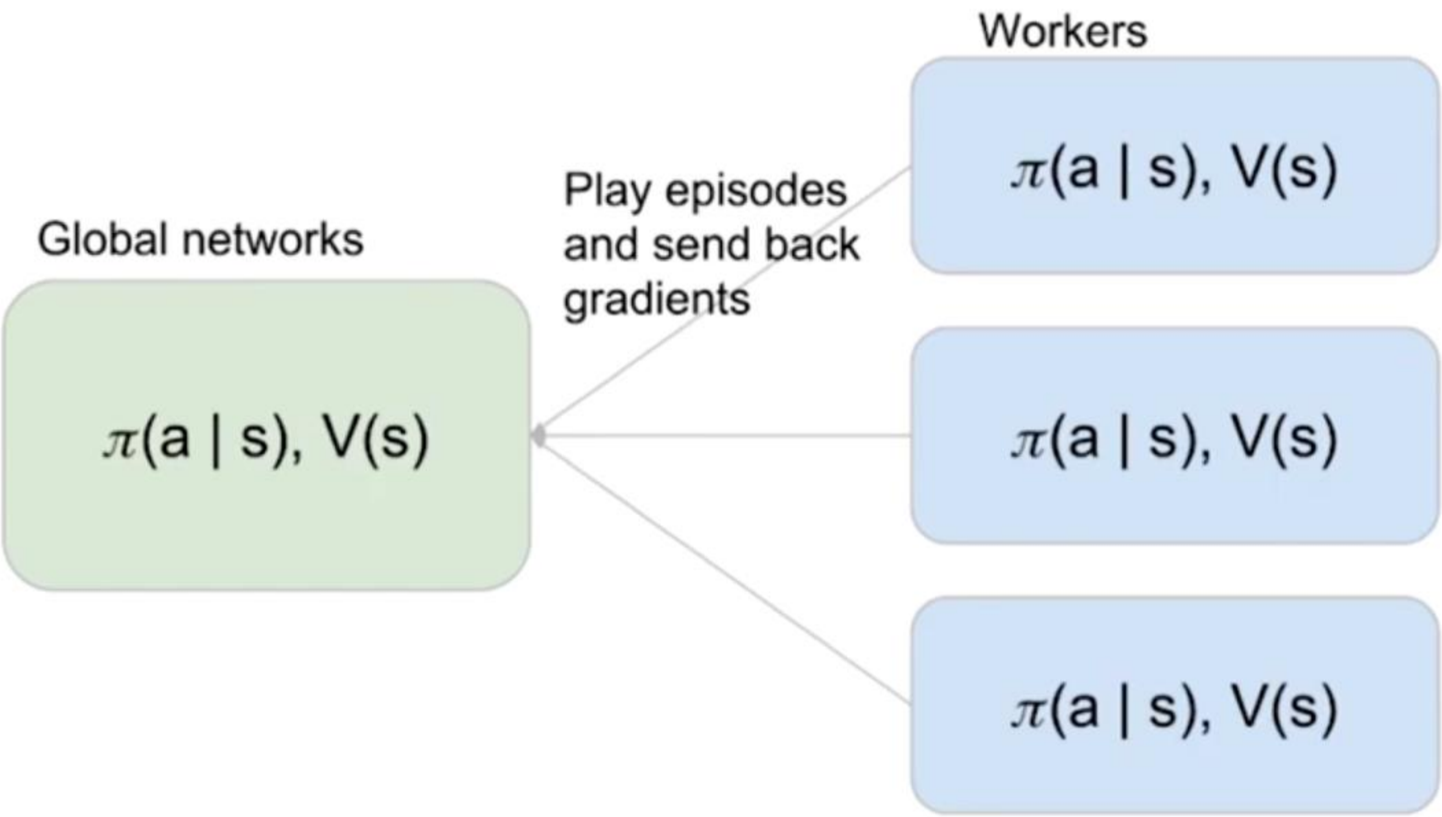
$$L = L_\pi + c_v L_v + c_{reg} L_{reg}$$

$$A^\theta(s_t, a_t) = G_t^n - V^{\pi_\theta}(s_t^n) = Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n) = r_t^n + \gamma V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)$$

$$L_v = (G_t^n - V^{\pi_\theta}(s_t^n))^2 = \left(r_t^n + \gamma V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)\right)^2$$

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

A3C



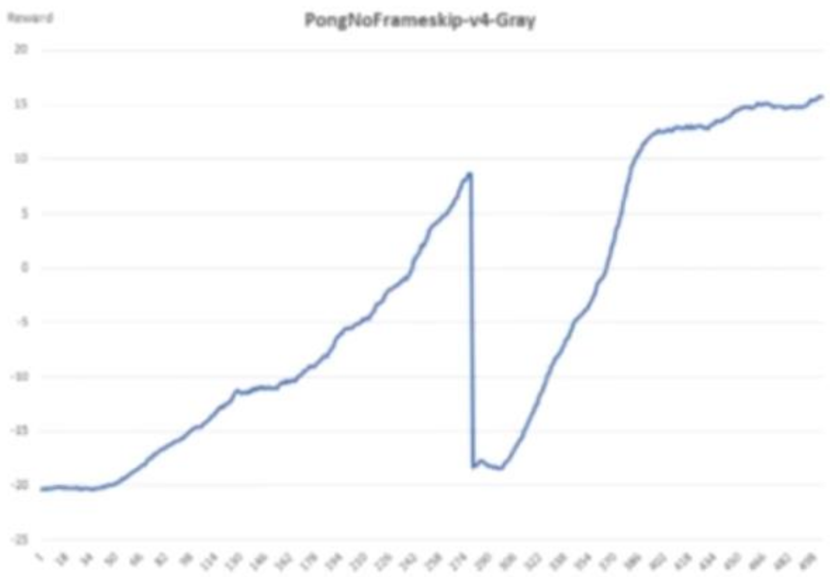
Reference: https://youtu.be/iCV3vOl8IMk

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... & Kavukcuoglu, K. (2016, June). Asynchronous methods for deep reinforcement learning. In International conference on machine learning (pp. 1928-1937).

A3C

Stability

- Each episode will progress randomly
- Each action is sampled probabilistically
- Occasionally, performance of agent can drop off due to bad update
  - Well, this can still happen with A3C so don't think you are immune



Reference: https://youtu.be/iCV3vOl8IMk