Batch size

```
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, :]
```

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

Batch size

- If you are using a continuous action space, this value should be large (in the order of 1000s). If you are using a discrete action space, this value should be smaller (in order of 10s).
- Typical range: (Continuous - PPO): 512 - 5120; (Continuous - SAC): 128 - 1024; (Discrete, PPO & SAC): 32 - 512.

https://github.com/Unity-Technologies/ml-agents/blob/release_10_docs/docs/Training-Configuration-File.md

Buffer size

In [18]:
```
frame_idx  = 0
max_frames = 5000   #15000
env.reset()
early_stop = False
__printDetails = False

while frame_idx < max_frames and not early_stop: #
    if(__printDetails):
        print("Frame = ", frame_idx, end=", ")
    log_probs = []
    values    = []
    states    = []
    actions   = []
    rewards   = []
    masks     = []
    entropy = 0

    step_result = env.get_steps(behaviorName)
    DecisionSteps = step_result[0]
    state = DecisionSteps.obs[0]
    if(__printDetails):
        print("step", end = ":")
    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)
```

$$(\vec{s}_1, \vec{a}_1, v_1, r_1, log\vec{p}_1, \vec{s}_2)$$

$$(\vec{s}_2, \vec{a}_2, v_2, r_2, log\vec{p}_2, \vec{s}_3)$$

$$\vdots$$

$$(\vec{s}_{20}, \vec{a}_{20}, v_{20}, r_{20}, log\vec{p}_{20}, \vec{s}_{21})$$

Buffer size

- default = 10240 for PPO and 500k for SAC
- Typically a larger buffer_size corresponds to more stable training updates.
- Typical range: PPO: 2048 - 409600; SAC: 500k – 1M

Learning rate

- default = 0.0003 Initial learning rate for gradient descent.
- This should typically be decreased if training is unstable, and the reward does not consistently increase.
- Typical range: 0.00001 – 0.001

Learning rate

In [12]:
```
num_inputs  = behavior_spec.observation_shapes[0][0]
num_outputs = behavior_spec.action_shape
print(num_inputs, num_outputs)
```

```
19 2
```

In [13]:
```
# NN parameters
hidden_size1      =128
hidden_size2      = 64
lr               = 3e-4
```

In [14]:
```
import torch.optim as optim
model = ActorCritic(num_inputs, num_outputs, hidden_size1, hidden_size2).to(device)
optimizer = optim.Adam(model.parameters(), lr=lr)
```

beta

In [6]:
```
def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns, advanta
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, return_, advantage in ppo_iter(mini_batch_size
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()

            loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy

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

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

beta

- (default = 0.005)
- 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: 0.0001 – 0.01

epsilon

```
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()
```

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

epsilon

- (default = 0.2)
- Setting this value small will result in more stable updates, but will also slow the training process.
- Typical range: 0.1 - 0.3

lambd

```
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
```

$(\vec{s}_1, \vec{a}_1, v_1, r_1, log\vec{p}_1, \vec{s}_2)$

$(\vec{s}_2, \vec{a}_2, v_2, r_2, log\vec{p}_2, \vec{s}_3)$

$\vdots$

$(\vec{s}_{20}, \vec{a}_{20}, v_{20}, r_{20}, log\vec{p}_{20}, \vec{s}_{21})\ v_{21}$

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

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

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

...

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

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

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

$$\Delta_1 = r_1 + \gamma * v_2 * mask_1 - v_1$$

$$gae_{1\sim20} = \Delta_1 + \gamma * \tau * mask_1 * gae_{2\sim20}$$

$$return_1 = gae_{1\sim20} + v_1$$

lambd

- (default = 0.95)
- 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).
- Typical range: 0.9 - 0.95

epoch

```
In [6]: def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns, advanta
            for _ in range(ppo_epochs):
                for state, action, old_log_probs, return_, advantage in ppo_iter(mini_batch_size
        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()

                    loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy

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

epoch

- (default = 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.
- Typical range: 3 - 10

Learning rate schedule

- (default = linear for PPO and constant for SAC)
- 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.

Learning rate schedule

```
In [12]: num_inputs  = behavior_spec.observation_shapes[0][0]
         num_outputs = behavior_spec.action_shape
         print(num_inputs, num_outputs)

         19 2

In [13]: # NN parameters
         hidden_size1      =128
         hidden_size2      = 64
         lr               = 3e-4

In [14]: import torch.optim as optim
         model = ActorCritic(num_inputs, num_outputs, hidden_size1, hidden_size2).to(device)
         optimizer = optim.Adam(model.parameters(), lr=lr)
```

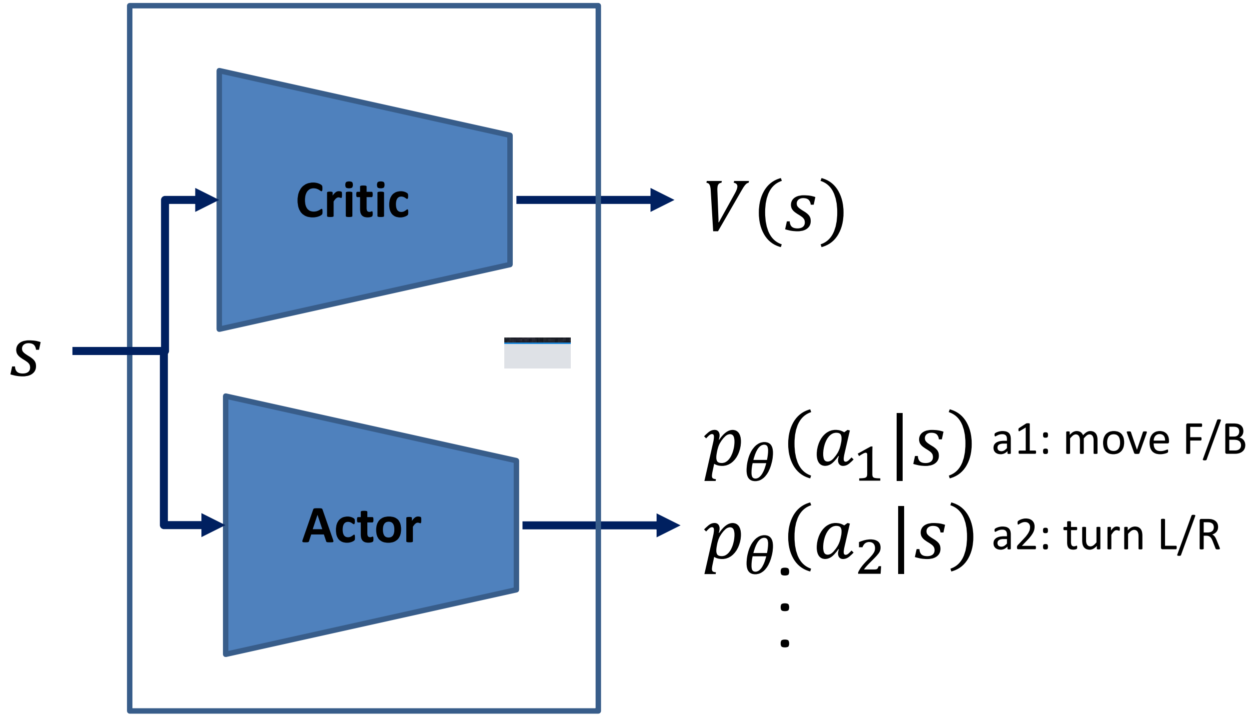
Network_settings

```
In [3]: from torch.distributions import Normal

        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(),
```



Hidden units

- (default = 128)
- 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.
- Typical range: 32 - 512

Num_layers

- (default = 2)
- For simple problems, fewer layers are likely to train faster and more efficiently. More layers may be necessary for more complex control problems.
- Typical range: 1 - 3

gamma

```
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
```

$(\vec{s}_1, \vec{a}_1, v_1, r_1, log\vec{p}_1, \vec{s}_2)$

$(\vec{s}_2, \vec{a}_2, v_2, r_2, log\vec{p}_2, \vec{s}_3)$

$\vdots$

$(\vec{s}_{20}, \vec{a}_{20}, v_{20}, r_{20}, log\vec{p}_{20}, \vec{s}_{21})\ v_{21}$

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

…

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

$$\Delta_1 = r_1 + \gamma * v_2 * mask_1 - v_1$$
$$gae_{1\sim20} = \Delta_1 + \gamma * \tau * mask_1 * gae_{2\sim20}$$
$$return_1 = gae_{1\sim20} + v_1$$

gamma

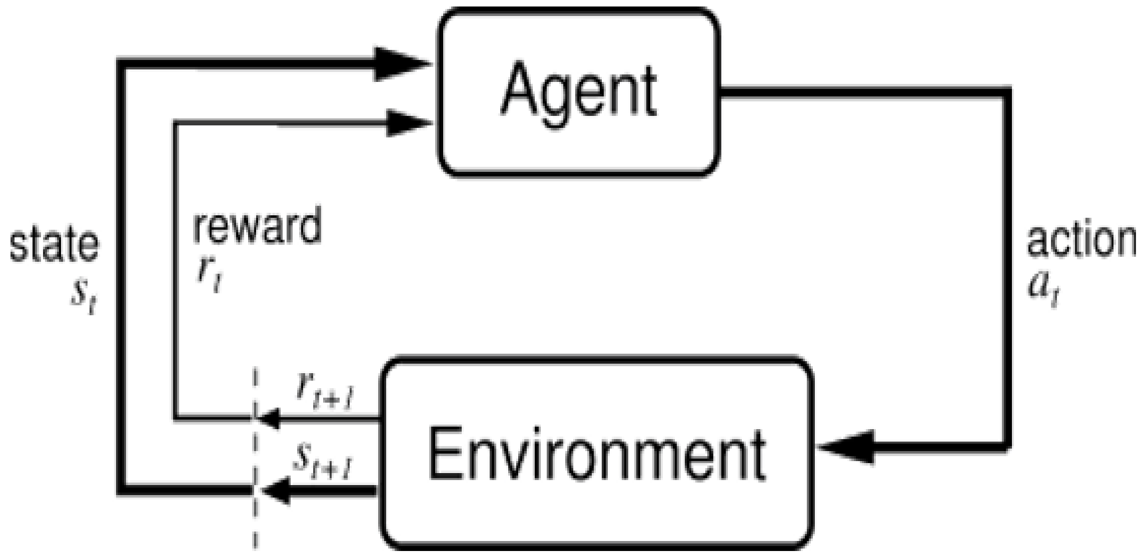
- (default = 0.99)
- In situations when the agent should be acting in the present in order to prepare for rewards in the distant future, this value should be large. In cases when rewards are more immediate, it can be smaller. Must be strictly smaller than 1.
- Typical range: 0.8 - 0.995

Max steps

```
In [18]: frame_idx  = 0
         max_frames = 5000   #15000
         env.reset()
         early_stop = False
         __printDetails = False

         while frame_idx < max_frames and not early_stop: #
             if(__printDetails):
                 print("Frame = ", frame_idx, end=", ")
             log_probs = []
             values    = []
             states    = []
             actions   = []
             rewards   = []
             masks     = []
             entropy = 0

             step_result = env.get_steps(behaviorName)
             DecisionSteps = step_result[0]
             state = DecisionSteps.obs[0]
             if(__printDetails):
                 print("step", end = ":")
             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)
```



(Sutton and Barto, 1998)

Max steps

- (default = 500k)
- Typical range: 500k – 10M

Time horizon

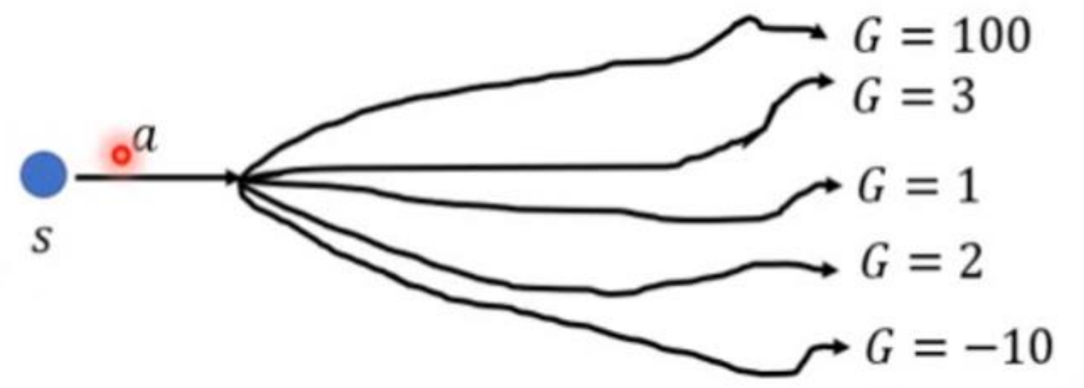
- (default = 64) Typical range: 32 - 2048
- This parameter trades off between a less biased, but higher variance estimate (long time horizon) and more biased, but less varied estimate (short time horizon).
- In cases where there are frequent rewards within an episode, or episodes are prohibitively large, 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.

Time horizon

```
In [18]: frame_idx  = 0
max_frames = 5000   #15000
env.reset()
early_stop = False
__printDetails = False

while frame_idx < max_frames and not early_stop: #
    if(__printDetails):
        print("Frame = ", frame_idx, end=", ")
    log_probs = []
    values    = []
    states    = []
    actions   = []
    rewards   = []
    masks     = []
    entropy = 0

    step_result = env.get_steps(behaviorName)
    DecisionSteps = step_result[0]
    state = DecisionSteps.obs[0]
    if(__printDetails):
        print("step", end = ":")
    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)
```



(Reference: 李弘毅 RL 影片)

threaded

- (default = 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-policyness of PPO, you can disable parallel updates by setting threaded to false.