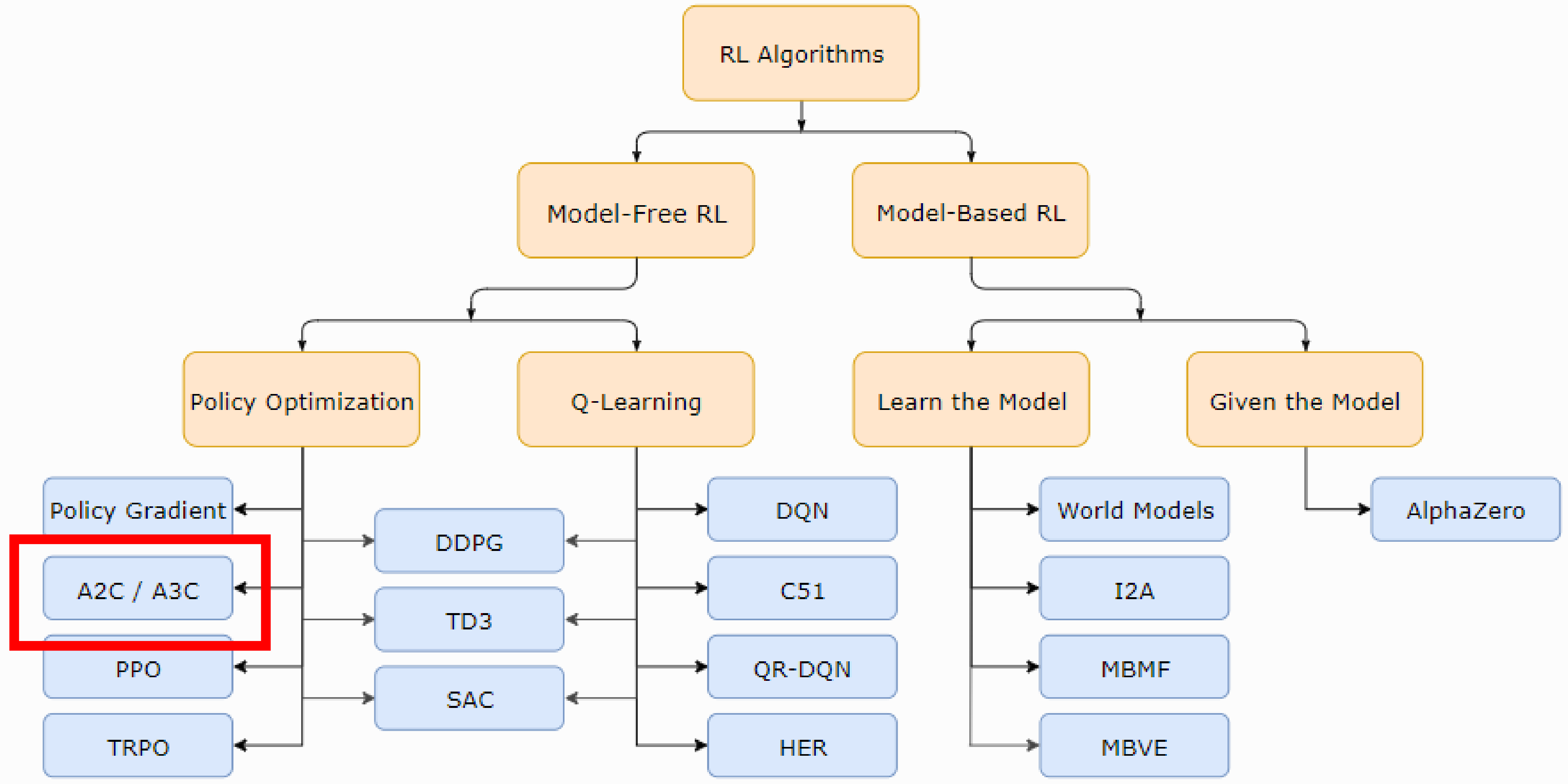
# A2C/A3C




*A non-exhaustive, but useful taxonomy of algorithms in modern RL.* *Citations below.*

# Use expected value to reduce sampling variance

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

$$\Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)$$

$$\Phi_t = Q^{\pi_\theta}(s_t, a_t)$$

$$\Phi_t = A^{\pi_\theta}(s_t, a_t)$$

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

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

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

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

unstable when sampling amount is not large enough

Ref: 李弘毅 https://youtu.be/j82QLgfhFiY

# We only need to estimate $V(s)$

$$\boxed{V^{\pi_\theta}(s_t^n)}$$

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

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

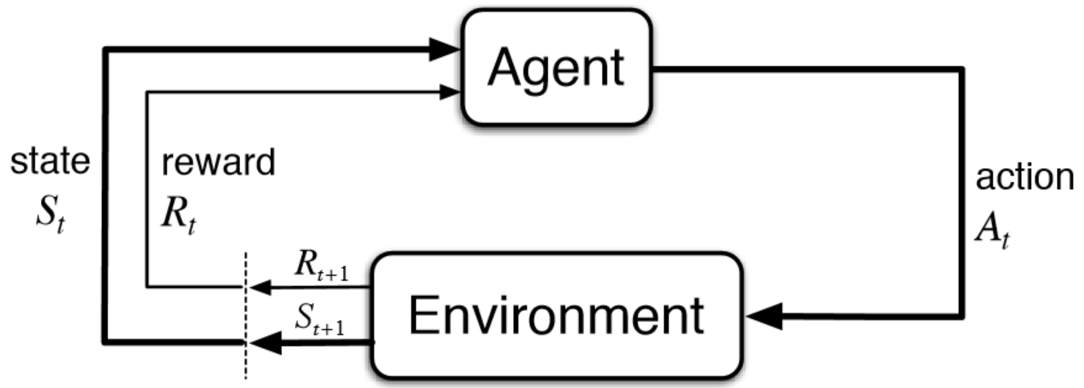
$$Q^{\pi_\theta}(s_t^n, a_t^n) = \mathrm{E}[r_t^n + V^{\pi_\theta}(s_{t+1}^n)] = r_t^n + V^{\pi_\theta}(s_{t+1}^n)$$

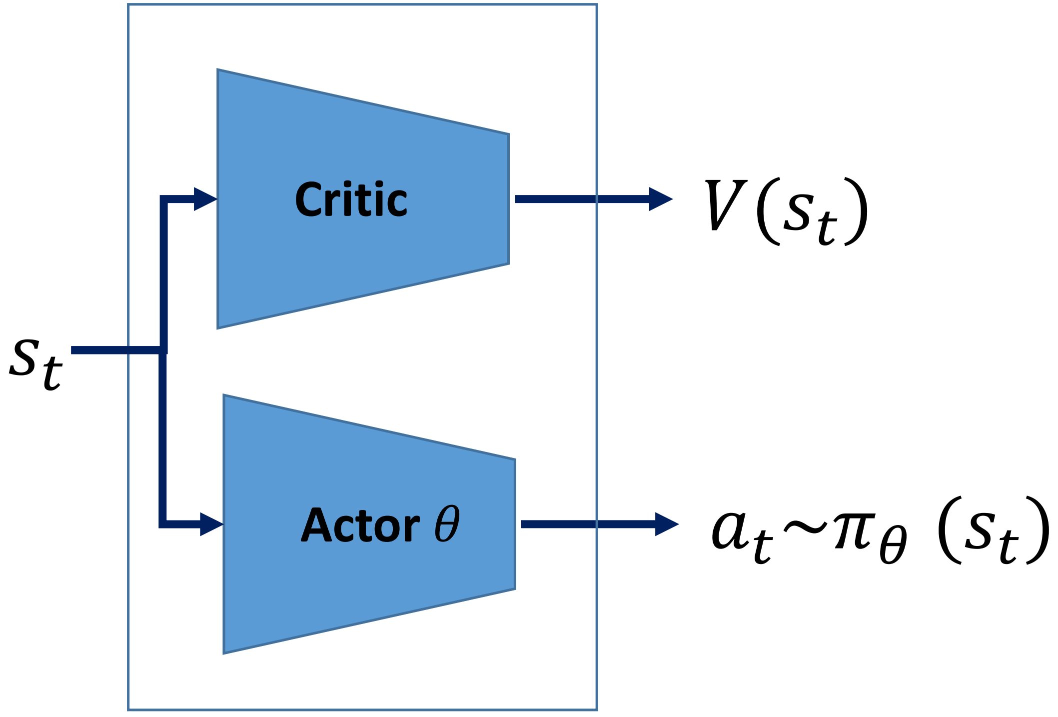
$$Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n) = r_t^n + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)$$

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

# A2C (Advantage Actor Critic)

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




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

# Use temporal difference to calculate $V(s)$

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

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

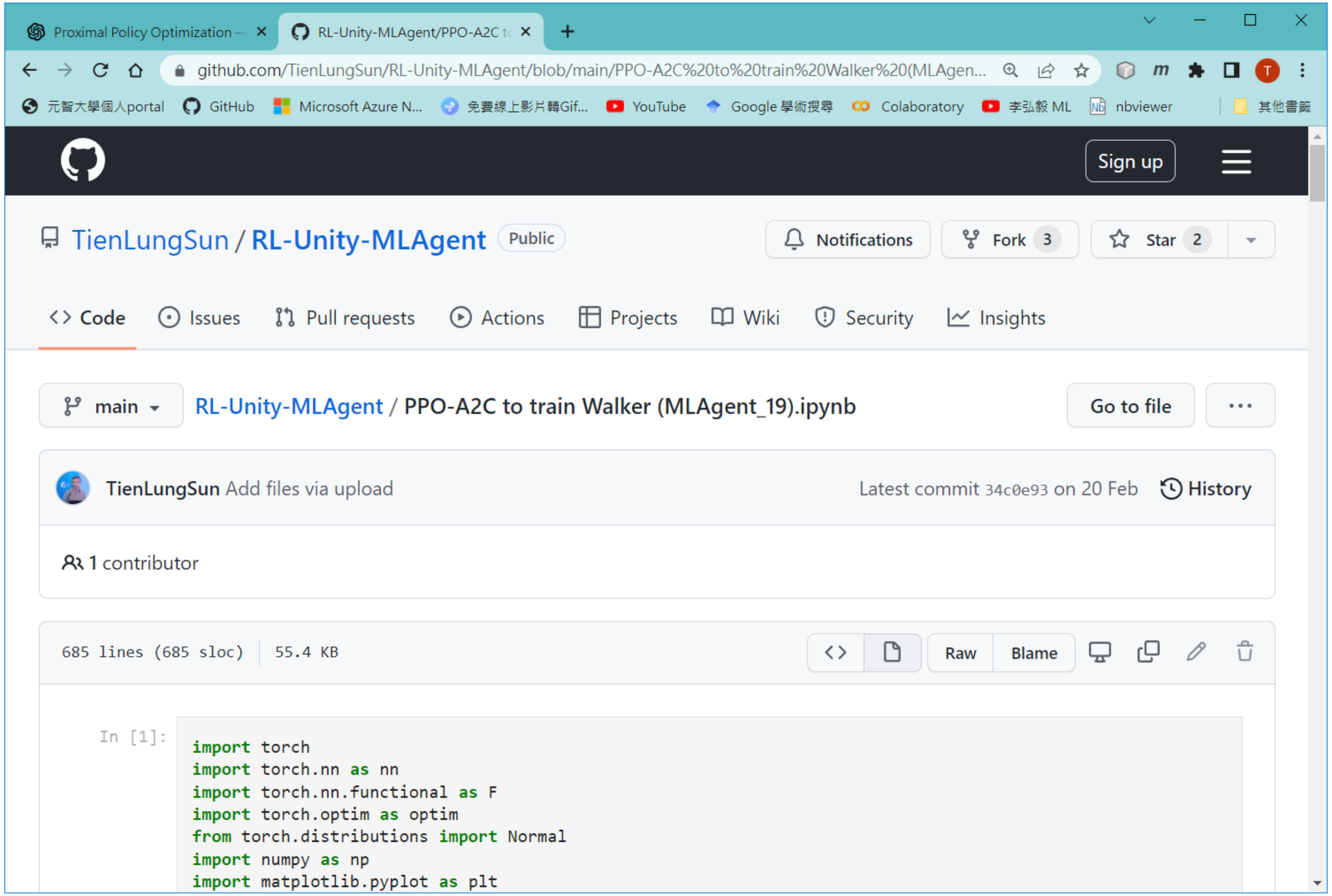
$V^{\pi_\theta}(s_a) \leftrightarrow G_a$ Monte-Carlo approach

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

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

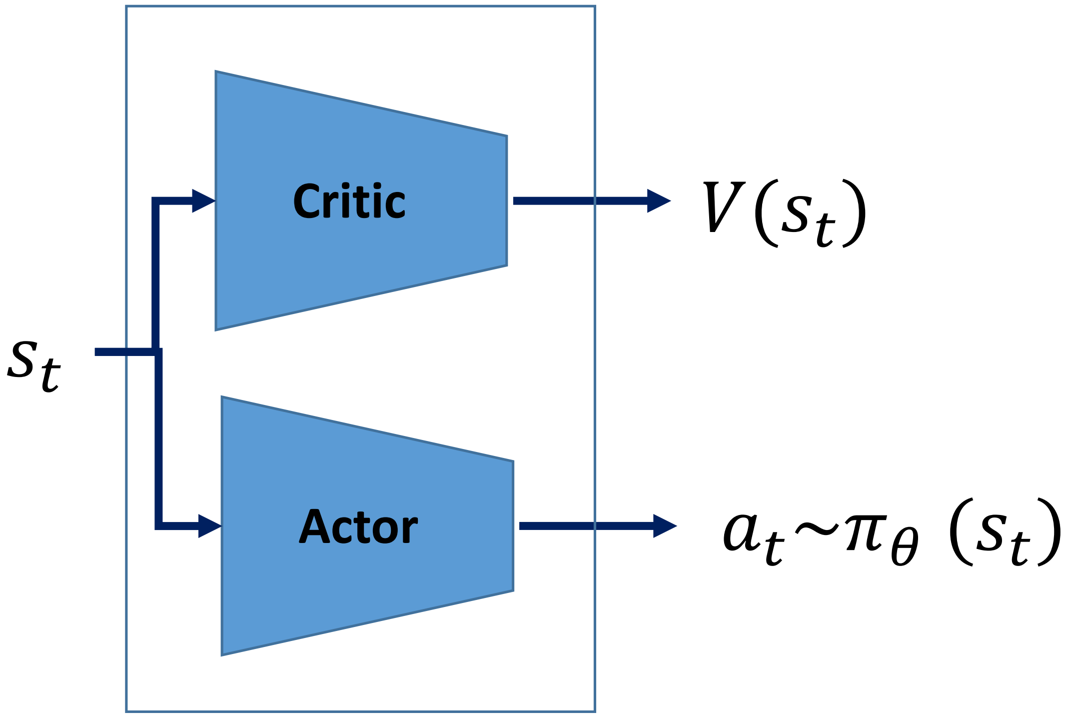
# PyTorch Implementation

## My GitHub → RL-Unity-MLAgent → PPO-A2C.ipynb

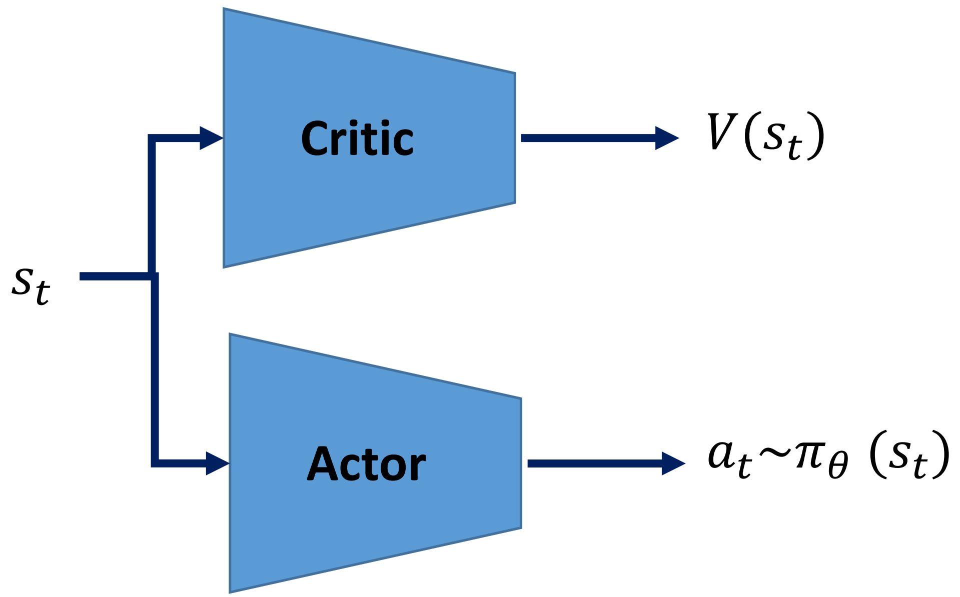


# PyTorch implementation of A2C

```
class Net(nn.Module):
    def __init__(self, ):
        super(Net, self).__init__()

        self.critic = nn.Sequential(
            nn.Linear(N_STATES, 128),
            nn.LayerNorm(128),
            nn.Linear(128, 128),
            nn.LayerNorm(128),
            nn.Linear(128, 1)
        )

        self.actor = nn.Sequential(
            nn.Linear(N_STATES, 128),
            nn.LayerNorm(128),
            nn.Linear(128, 128),
            nn.LayerNorm(128),
            nn.Linear(128, N_ACTIONS)
        )
        self.log_std = nn.Parameter(torch.ones(1,
        self.apply(init_weights)

    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
```



# Define loss function to optimize the Actor-Critic networks



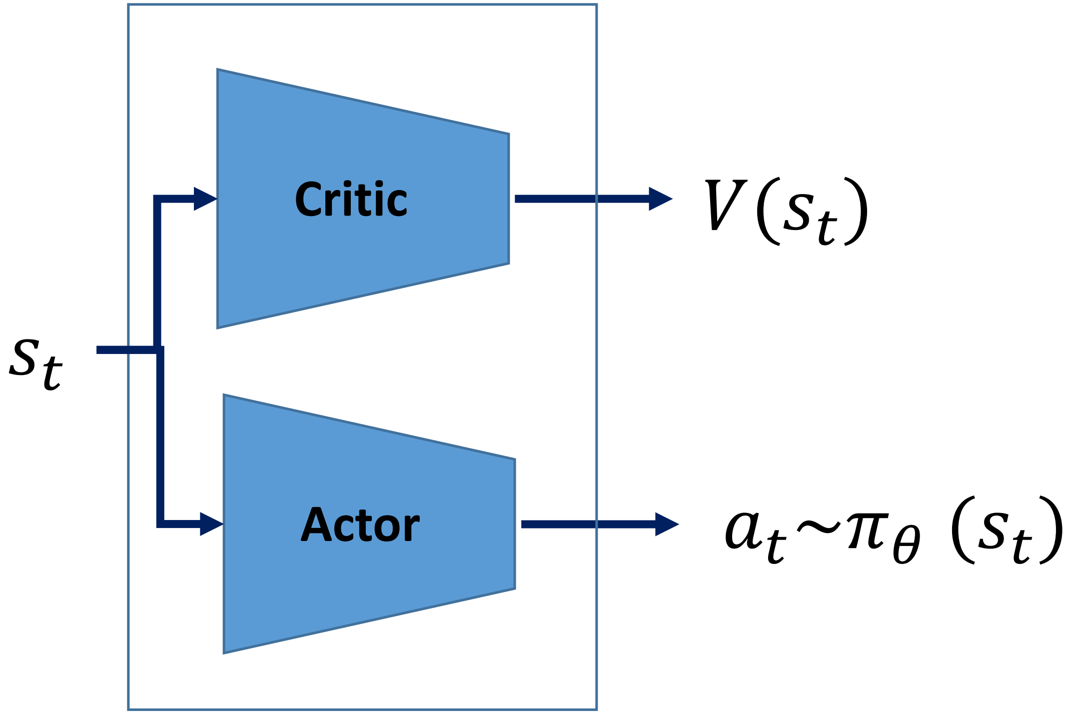

$$L = L_\pi + c_v L_v$$

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

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

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

# Add entropy-based regularization




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

# PyTorch implementation of A2C

```
dist, value = net(batch_state.to(device))
critic_loss = (batch_return.to(device) - value).pow(2).mean()
entropy = dist.entropy().mean()
batch_action = dist.sample()
batch_new_log_probs = dist.log_prob(batch_action)
ratio = (batch_new_log_probs - batch_old_log_probs.to(device)).exp()
surr1 = ratio * batch_advantage.to(device)
surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_param) * batc
actor_loss  = - torch.min(surr1, surr2).mean()
loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy
```

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

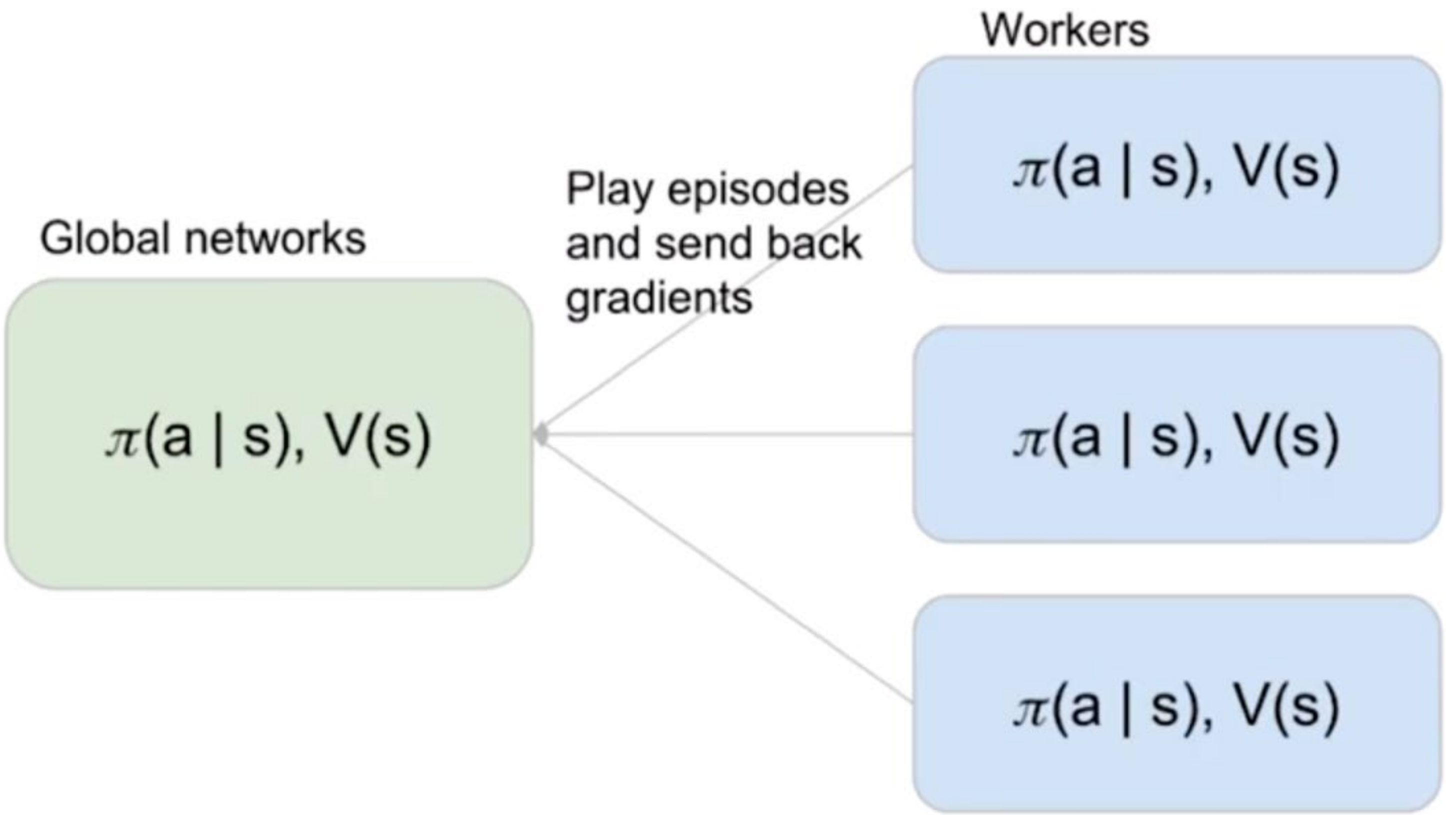
$$Loss_v = \left(r_t^n + \gamma V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)\right)^2$$

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

# PyTorch implementation of A2C

```
while (frame_idx < MAX_STEPS):
    print("\nframe idx = ", frame_idx)
    print("Interacts with Unity to collect training data")
    states, actions, log_probs, values, rewards, masks, next_state = collect_training_data (N_AGEN
    _, next_value = net(next_state.to(device))

    print("Compute GAE of these training data set")
    returns = compute_gae(TIME_HORIZON, next_value, rewards, masks, values, GAMMA, LAMBD)

    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)
    advantages = returns - values

    print("Optimize NN with PPO")
    critic_loss, actor_loss = ppo_update(N_EPOCH, BATCH_SIZE, states, actions, log_probs, returns,
    CriticLossLst.append(critic_loss)
    ActorLossLst.append(actor_loss)

    frame_idx += TIME_HORIZON
```

# A3C (Asynchronous Advantage Actor Critic)

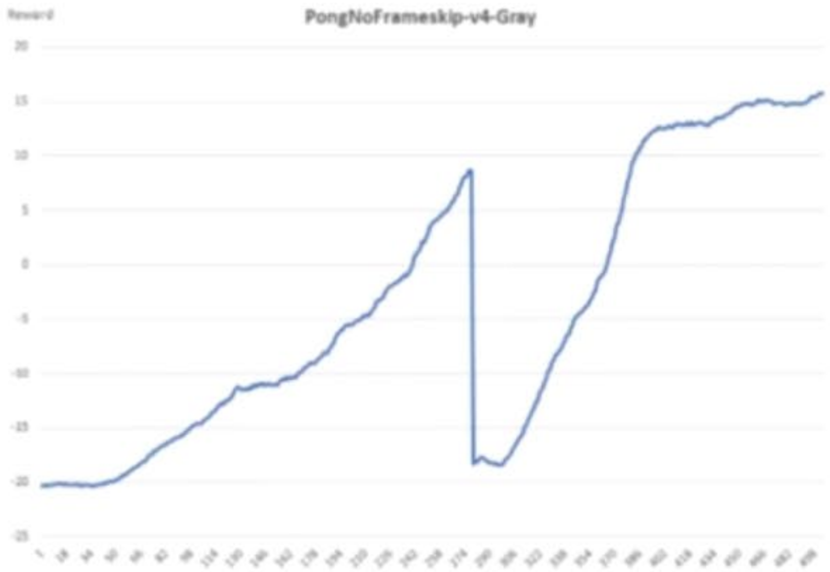



Reference: [https://youtu.be/iCV3vOl8IMk](https://youtu.be/iCV3vOl8IMk)

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

- 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](https://youtu.be/iCV3vOl8IMk)

# A3C

- DQN is also interested in stabilizing learning
- Techniques:
  - Freezing target network
  - Experience replay buffer
- Use experience replay to look at multiple examples per training step
- A3C simply achieves stability using a different method (parallel agents)
- Both solve the problem: how to make neural networks work as function approximators in classic RL algorithms?

Reference: [https://youtu.be/iCV3vOl8IMk](https://youtu.be/iCV3vOl8IMk)

# A3C

- Remember: the theory part is not new, just need to create multiple parallel agents and asynchronously update/copy parameters
- 3 files:
  - main.py (master file; global policy and value networks)
    - Create and coordinate workers
  - worker.py (contains local policy and value networks)
    - Copy weights from global nets
    - Play episodes
    - Send gradients back to master
  - nets.py
    - Definition of policy and value networks

Reference: [https://youtu.be/iCV3vOl8IMk](https://youtu.be/iCV3vOl8IMk)

# A3C



## main.py

Instantiate global policy and value networks

Check # CPUs available, create threads and workers

Initialize global thread-safe counter, so every worker knows when to quit (when # of total steps reaches a max.)

Reference: https://youtu.be/iCV3vOl8IMk

# A3C

## worker.py

```
def run():
  in a loop:
    copy params from global nets to local nets
    run N steps of game (and store the data - s, a, r, s')
    using gradients wrt local net, update the global net
```

Conceptually, it's like:

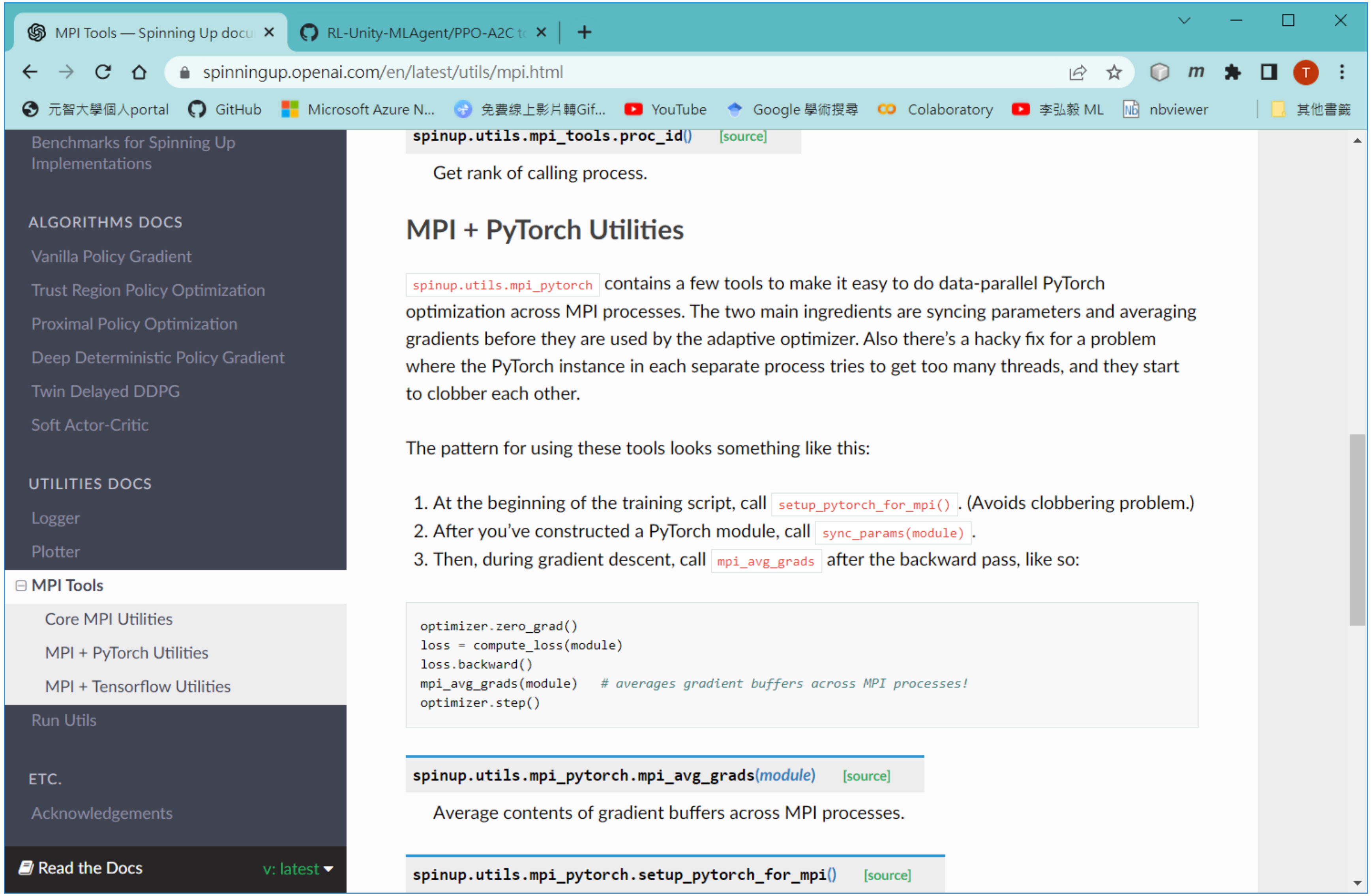
1) $$g_{local} = \frac{\partial L(\theta_{local})}{\partial \theta_{local}}$$

2) $$\theta_{global} = \theta_{global} - \eta g_{local}$$

But in reality, we'll use RMSprop

Reference: [https://youtu.be/iCV3vOl8IMk](https://youtu.be/iCV3vOl8IMk)

# Speed up PPO with MPI




`spinup.utils.mpi_tools.proc_id()` [source]

Get rank of calling process.

## MPI + PyTorch Utilities

`spinup.utils.mpi_pytorch` contains a few tools to make it easy to do data-parallel PyTorch optimization across MPI processes. The two main ingredients are syncing parameters and averaging gradients before they are used by the adaptive optimizer. Also there's a hacky fix for a problem where the PyTorch instance in each separate process tries to get too many threads, and they start to clobber each other.

The pattern for using these tools looks something like this:

1. At the beginning of the training script, call `setup_pytorch_for_mpi()`. (Avoids clobbering problem.)
2. After you've constructed a PyTorch module, call `sync_params(module)`.
3. Then, during gradient descent, call `mpi_avg_grads` after the backward pass, like so:

```
optimizer.zero_grad()
loss = compute_loss(module)
loss.backward()
mpi_avg_grads(module)   # averages gradient buffers across MPI processes!
optimizer.step()
```

`spinup.utils.mpi_pytorch.mpi_avg_grads(module)` [source]

Average contents of gradient buffers across MPI processes.

`spinup.utils.mpi_pytorch.setup_pytorch_for_mpi()` [source]

# Combine data collected from different agents

## Use PPO to update NN weights and biases

```
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
```

```
print(len(returns), returns[0].shape)
print(len(log_probs), log_probs[0].shape)
print(len(values), values[0].shape)
print(len(states), states[0].shape)
print(len(actions), actions[0].shape)
print(len(advantage), advantage[0].shape)
```

```
60 torch.Size([1])
60 torch.Size([2])
60 torch.Size([1])
60 torch.Size([8])
60 torch.Size([2])
60 torch.Size([1])
```

N: no. of agents
K: time horizon

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