Assignment 5

Name：Shiqu Wu

Student ID：518021910665

1 . Actor-Critic (AC)

(1.1)

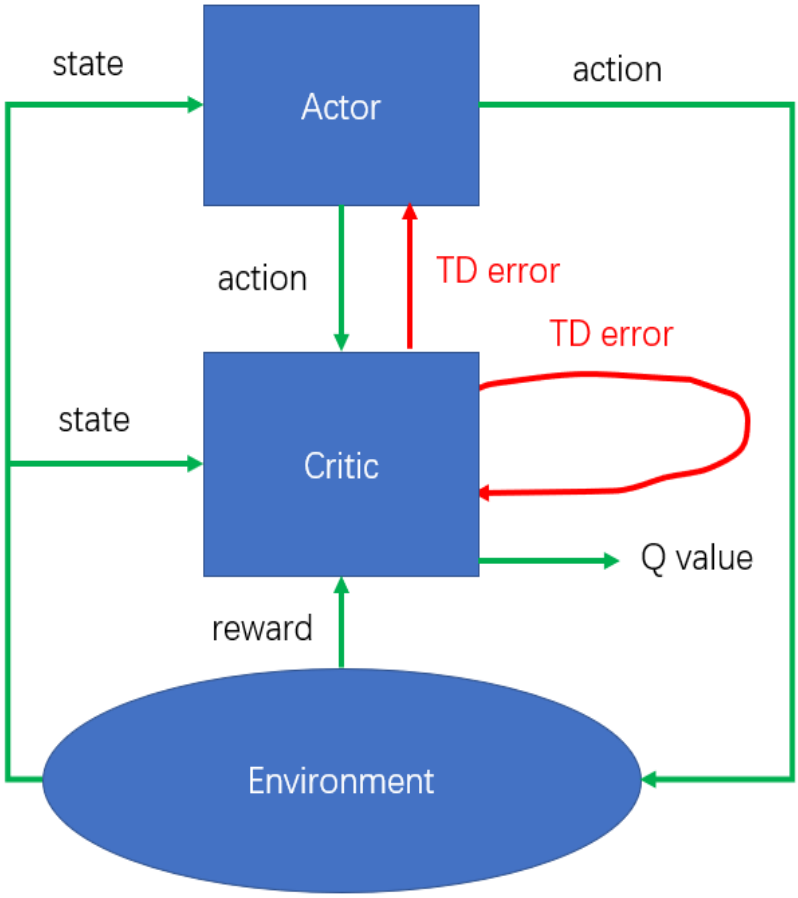
What is AC:

Actor: Policy Q-network, used to choose action.

Critic: Value Q-network, used to estimate action is good or not. And generates TD-error to **guide** Policy Q-network and Value Q-network updating.

Brief explanation: Actor, Critic are two neural networks. Actor uses policy gradient to update its parameters while Critic uses TD(0) . TD-error generated by Critic shows whether current gradient ascent is correct or not. If it's correct, it means that the neural networks can ascent more in this right direction; else the neural networks ascent less in current wrong direction.

Structure:



why AC:

AC is proposed by Google DeepMind and it aims to solve the problems in traditional value-based algorithms and policy-based algorithms. As we've mentioned before, DQN, Q-learning, SARSA/SARSA-λ .etc Value-based algorithms can't handle the **continuous action space problem**. And simple policy gradient algorithm is updated per-round, which results in **slow learning speed and low efficiency**.

AC algorithm combines the advantages of value-based algorithms and policy-based algorithms: it can handle continuous action space problem while keeping the single-step update mechanism, which benefits model's learning speed.

AC algorithm:

function QAC

Initialise s, θ

Sample $a \sim \pi_\theta$

for each step **do**

Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$

Sample action $a' \sim \pi_\theta(s', a')$

$\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$

$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$

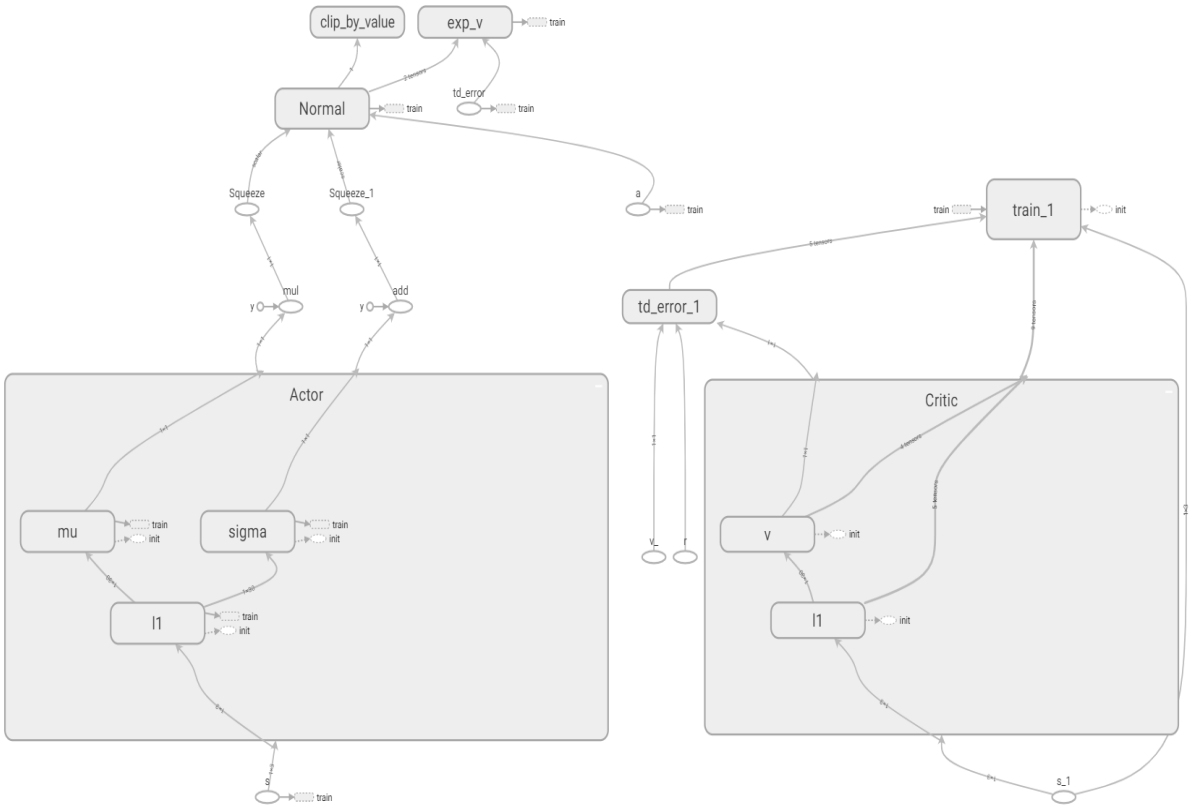
$w \leftarrow w + \beta \delta \phi(s, a)$

$a \leftarrow a', s \leftarrow s'$

end for

end function

Implements:



From the tensorboard graph we can see that Actor is composed by three parts:

```
with tf.variable_scope('Actor'):
    l1 = tf.layers.dense(inputs=self.s, units=30,
activation=tf.nn.relu,

kernel_initializer=tf.random_normal_initializer(0, 0.1),

bias_initializer=tf.constant_initializer(0.1), name='l1')
    mu = tf.layers.dense(inputs=l1, units=1, activation=tf.nn.tanh,

kernel_initializer=tf.random_normal_initializer(0, 0.1),

bias_initializer=tf.constant_initializer(0.1), name='mu')
    sigma = tf.layers.dense(inputs=l1, units=1,
activation=tf.nn.softplus,

 kernel_initializer=tf.random_normal_initializer(0, 0.1),

 bias_initializer=tf.constant_initializer(0.1), name='sigma')
```

where `mu,sigma` determine an action distribution. When actor needs to choose an action, it samples from the distribution, then the action value is clipped by Pendulum-V0's action bound.

Critic gets the sample action and state to generate Q-value and calculate TD-error. Critic network is constructed as below:

```
with tf.variable_scope('Critic'):
    l1 = tf.layers.dense(
        inputs=self.s,
        units=30,  # number of hidden units
        activation=tf.nn.relu,
        kernel_initializer=tf.random_normal_initializer(0., .1),  #
weights
        bias_initializer=tf.constant_initializer(0.1),  # biases
        name='l1'
    )
    self.v = tf.layers.dense(
        inputs=l1,
        units=1,  # output units
        activation=None,
        kernel_initializer=tf.random_normal_initializer(0., .1),  #
weights
```

```
            bias_initializer=tf.constant_initializer(0.1),  # biases
            name='v'
        )
```

The `TD-error` is calculated by:

```
with tf.variable_scope('squared_TD_error'):
    self.td_error = tf.reduce_mean(self.r + self.gamma * self.v_ -
self.v)
```

After that, we update the critic network and actor network by:

Critic:

```
with tf.variable_scope('train'):
    self.train_op =
tf.train.AdamOptimizer(self.lr).minimize(self.loss)
```

Actor:

```
with tf.name_scope('exp_v'):
    log_prob = self.normal_dist.log_prob(self.a)  # loss without
advantage
    self.exp_v = log_prob * self.td_error  # advantage (TD_error)
guided loss
    self.exp_v += 0.01 * self.normal_dist.entropy()  # Add cross
entropy cost to encourage exploration
with tf.name_scope('train'):
    self.train_op = tf.train.AdamOptimizer(self.lr).minimize(-
self.exp_v, global_step)  # min(v) = max(-v)
```

(1.2)

Drawbacks of AC:

It's hard for AC algorithm to converge. For the fact that model's convergence depends on Q-value estimated by Critic. However, the output of Critic depends on the sample action generated by Actor, which is updated continuously. Thus, the AC algorithm suffers the divergence problem.

How to solve:

1. We can apply the **Fixed Q-network and Experience Replay** strategy taken by DQN in Actor-Critic, which forms the famous DDPG algorithm.
2. The lower the correlation of parameters in updating, the higher the convergence of the algorithm. We can reduce the correlations between parameters by applying the "**Centralized Update, Decentralized Execution**" strategy, which forms A3C algorithm.

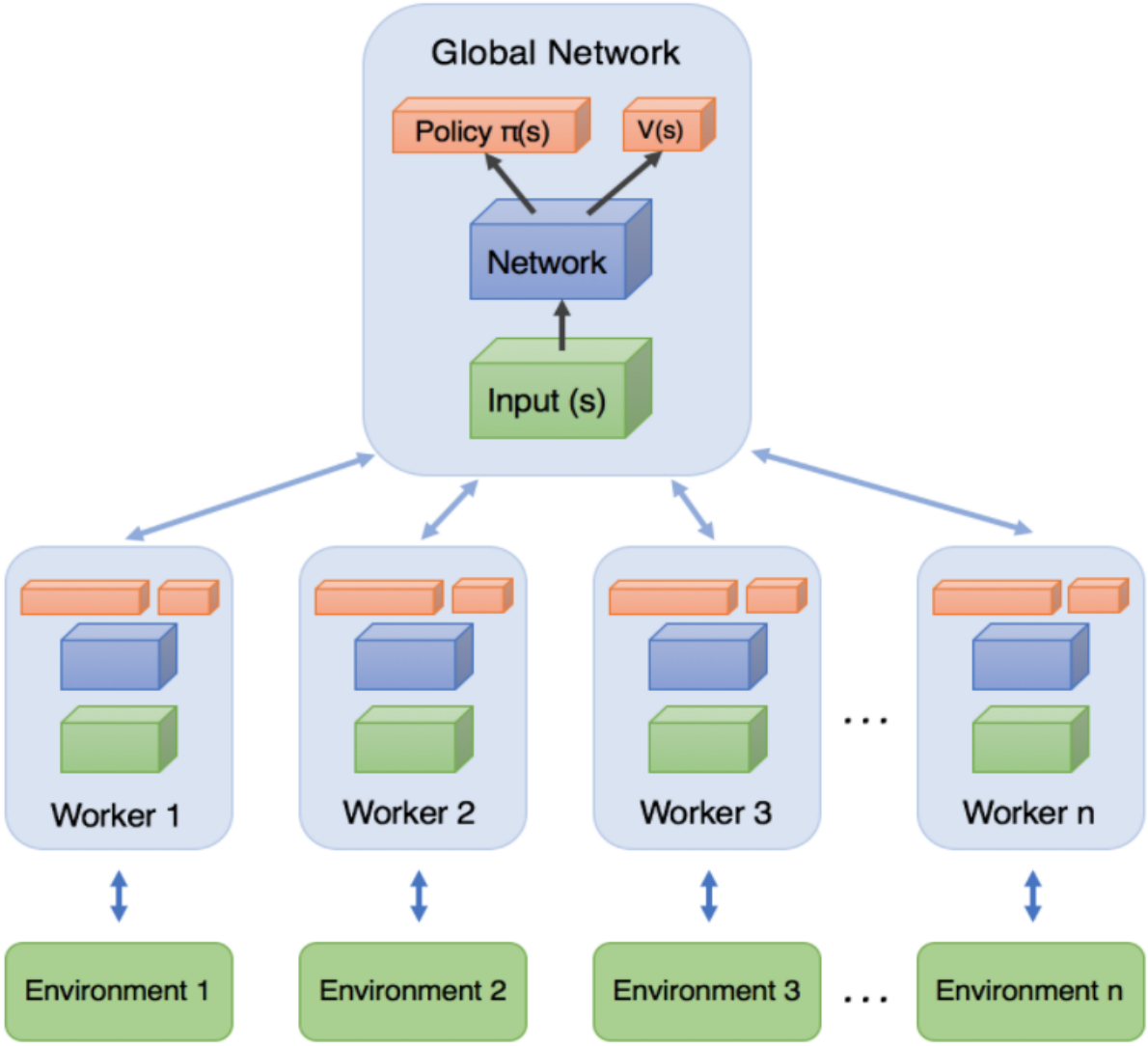
2 . Asynchronous Advantage Actor-Critic (A3C)

(2.1)

What is A3C

Asynchronous + Advantage + Actor-Critic. A3C implements "**Centralized Update, Decentralized Execution**" by introducing a **Global Network** and **Worker Network**. Global network stores parameters while worker networks calculate gradients and store experiences separately. The global network calls `pull()` function to collect newly calculated gradients from worker networks and put them together to update parameters. Then the global network calls `push()` function to synchronize updated parameters to all workers. This process is "Centralized Update" . And asynchronous means put AC algorithm into multi-threads to train independently, which is known as "Decentralized Execution".

Structure:



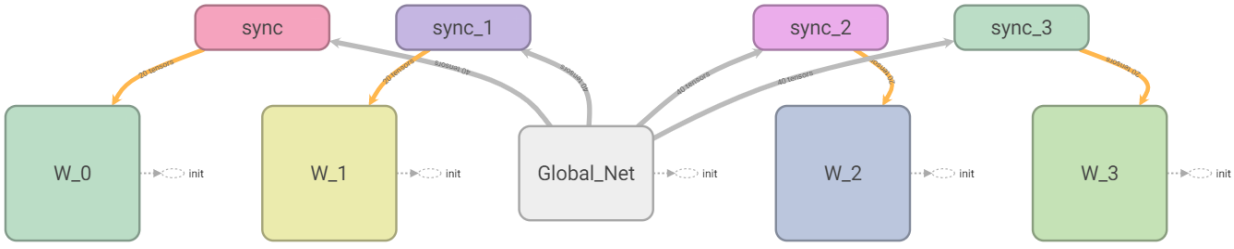
A3C algorithm:

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

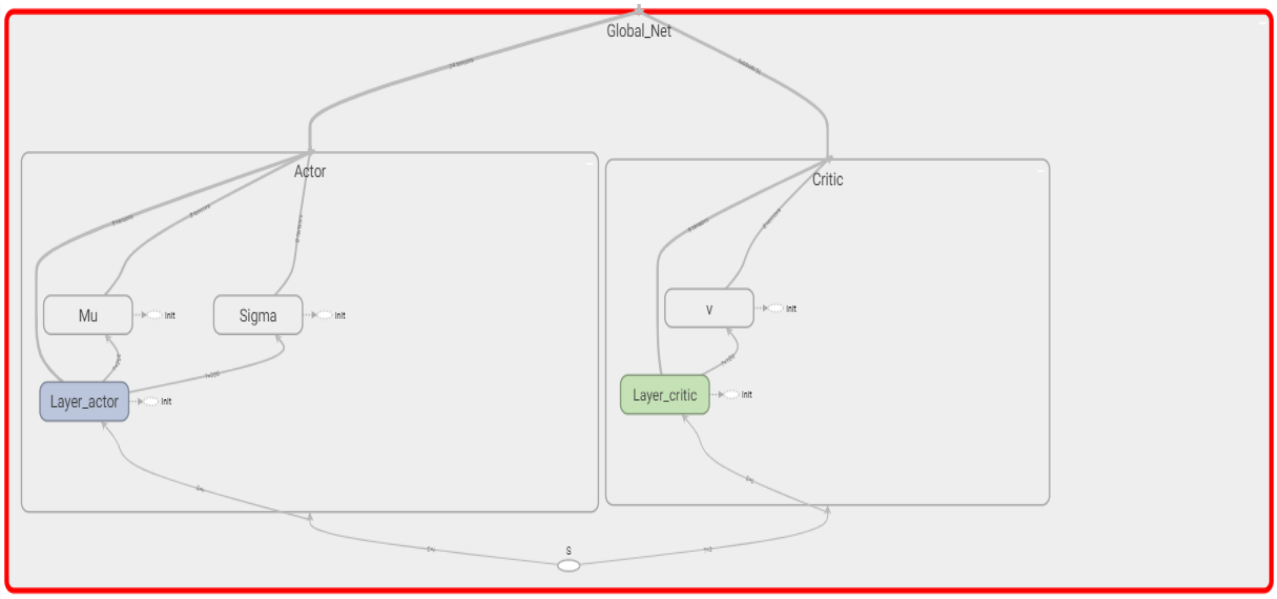
// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t+1$
    $T \leftarrow T+1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Implements:

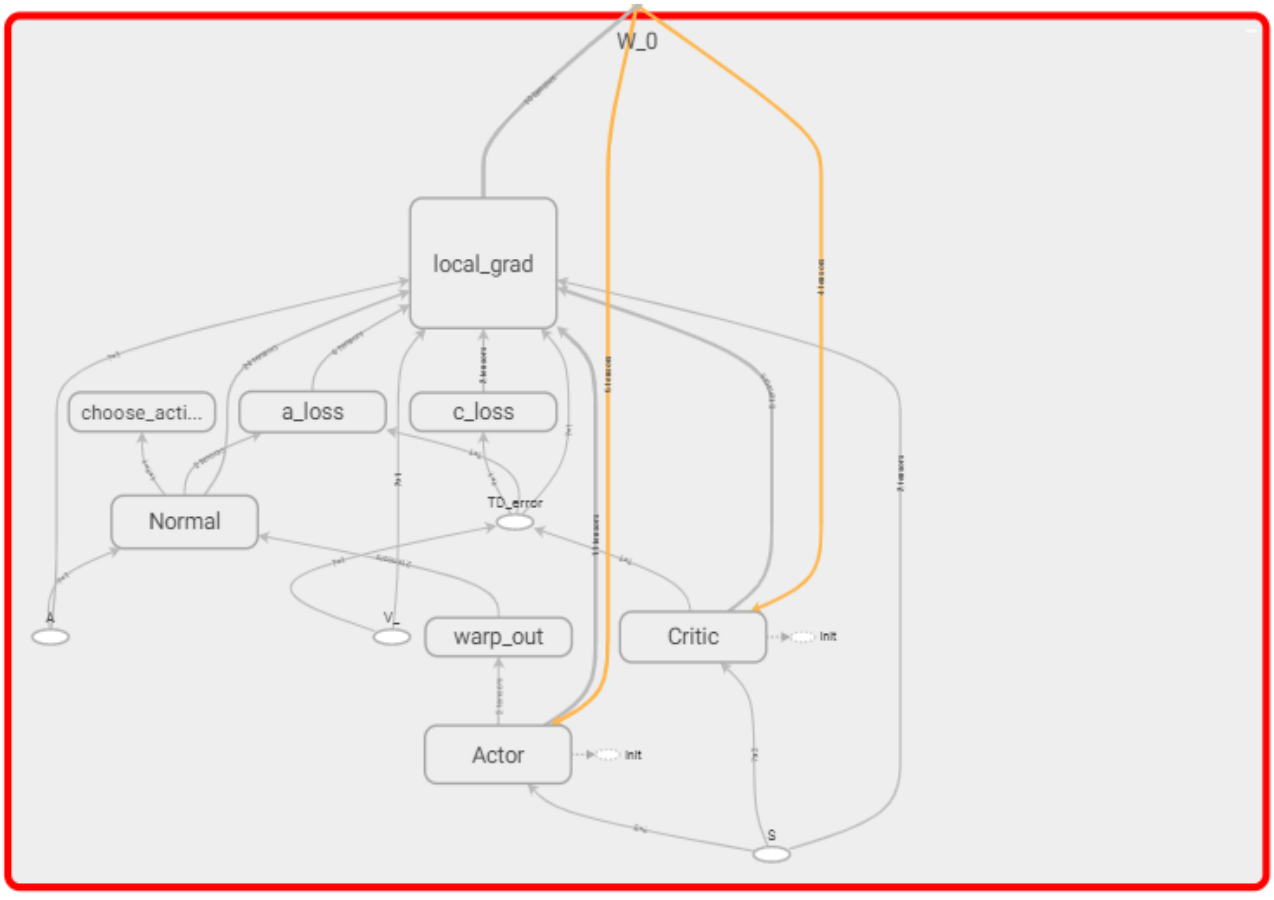
Model structure (4 workers):



Global Network structure :



Worker Network structure :



From the worker network structure graph and global network structure graph we can known that they are the duplicates of the same AC network. According to the model structure graph we can see that all the worker networks are connected to the global network by `sync` component and there is no connection among the worker networks, which means they are trained independently. With the help of `multiprocessing` library, we implement "**Centralized Update, Decentralized Execution**" as below:

```
with tf.device("/cpu:0"):
    OPT_A = tf.train.RMSPropOptimizer(0.0001, name='RMSPropA')
    OPT_C = tf.train.RMSPropOptimizer(0.001, name='RMSPropC')
    GLOBAL_AC = ACNet('Global_Net')  # we only need its params
    workers = []
    num_workers = multiprocessing.cpu_count()
    for i in range(num_workers):
        i_name = 'W_%i' % i  # worker name
        workers.append(Worker(i_name, GLOBAL_AC))
```

We initialize class `ACNet` to get the global network object and pass this object as an argument to all the workers (get by initialize class `Worker`). The number of workers is set as the number of threads in current CPU device (`16` in my case). For each worker, we set the `work()` function as the job for the corresponding thread and use `COORD.join()` function to control parallel execution. The implement code is shown as below:

```
worker_threads = []
for worker in workers:
    job = lambda: worker.work()
    t = threading.Thread(target=job)
    t.start()
    worker_threads.append(t)
COORD.join(worker_threads)
```

(2.2)

Advantages:

1. Increase the training speed. For multi-threading is used to perform training task in parallel.
2. Reduce the correlation of parameter in updating, which improves convergence speed.

3 . Proximal Policy Optimization (PPO)

(3.1)

What is PPO:

PPO is proposed by OpenAI in 2017. After that, it's widely used in complex RL settings and it's also the default Rl algorithm in OpenAI newly published papers. As one of the most advanced RL algorithms, PPO does well in many traditional RL problems and it's easy to implement and debug. The predecessor of PPO is TRPO (Trust Region Policy Optimization), which is also published by OpenAI in 2015. There is another algorithm called ACER (Actor Critic with Experience Replay) that has the similar performance with PPO but is much more complex in implementation. What's more, PPO has a **better generalization performance** than TRPO and ACER. Compared to A3C, DDPG, the advantage of PPO lies in **better sample efficiency** and **higher robustness**.

PPO uses adaptive `KL-divergence` to constrain the update scope of new policy. This constraint reduces the hypter-parameter sensitiveness of model and enhance the robustness.

Why PPO:

Policy-based algorithms (e.g. Policy Gradient, AC, A3C, DDPG ...) are sensitive to hyper-parameters (e.g. learning rate, step size). Therefore, their robustness is unsatisfactory. What's more, the sample efficiency of these policy-based algorithms is usually bad, which means that millions of iterations are needed to train a single model in complex problem settings. It will spend a lot of time to train and it needs high computational power.

PPO algorithm:

Algorithm 1 Proximal Policy Optimization (adapted from [8])

for $i \in \{1, \cdots, N\}$ **do**
  Run policy π_θ for T timesteps, collecting $\{s_t, a_t, r_t\}$
  Estimate advantages $\hat{A}_t = \sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t)$
  $\pi_{\text{old}} \leftarrow \pi_\theta$
  for $j \in \{1, \cdots, M\}$ **do**
    $J_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t - \lambda \mathrm{KL}[\pi_{old}|\pi_\theta]$
    Update θ by a gradient method w.r.t. $J_{PPO}(\theta)$
  end for
  for $j \in \{1, \cdots, B\}$ **do**
    $L_{BL}(\phi) = -\sum_{t=1}^{T} (\sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t))^2$
    Update ϕ by a gradient method w.r.t. $L_{BL}(\phi)$
  end for
  if $\mathrm{KL}[\pi_{old}|\pi_\theta] > \beta_{high} \mathrm{KL}_{target}$ **then**
    $\lambda \leftarrow \alpha\lambda$
  else if $\mathrm{KL}[\pi_{old}|\pi_\theta] < \beta_{low} \mathrm{KL}_{target}$ **then**
    $\lambda \leftarrow \lambda/\alpha$
  end if
end for

What is DPPO:

DPPO is Distributed PPO, which is proposed by Google DeepMind in 2017. DPPO introduces "**Centralized Update, Decentralized Execution**" strategy in PPO to reduce the relevance of network parameters. Property of distribution enables DPPO to gain a higher robustness and faster convergence speed. Data collection, gradient calculation are distributed over workers. When the global network calls `pull()` function, it averages gradients from workers to update its policy. When it calls `push()` function, the updated policy (NN parameters) is assigned to all workers synchronously.

DPPO algorithm:

Algorithm 2 Distributed Proximal Policy Optimization (chief)

for $i \in \{1, \cdots, N\}$ **do**
  for $j \in \{1, \cdots, M\}$ **do**
    Wait until at least $W - D$ gradients wrt. θ are available
    average gradients and update global θ
  end for
  for $j \in \{1, \cdots, B\}$ **do**
    Wait until at least $W - D$ gradients wrt. ϕ are available
    average gradients and update global ϕ
  end for
end for

Algorithm 3 Distributed Proximal Policy Optimization (worker)

for $i \in \{1, \cdots, N\}$ **do**
  for $w \in \{1, \cdots T/K\}$ **do**
    Run policy π_θ for K timesteps, collecting $\{s_t, a_t, r_t\}$ for $t \in \{(i-1)K, \ldots, iK-1\}$
    Estimate return $\hat{R}_t = \sum_{t=(i-1)K}^{iK-1} \gamma^{t-(i-1)K} r_t + \gamma^K V_\phi(s_{iK})$
    Estimate advantages $\hat{A}_t = \hat{R}_t - V_\phi(s_t)$
    Store partial trajectory information
  end for
  $\pi_{\text{old}} \leftarrow \pi_\theta$
  for $m \in \{1, \cdots, M\}$ **do**
    $J_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t - \lambda \text{KL}[\pi_{old}|\pi_\theta] - \xi \max(0, \text{KL}[\pi_{old}|\pi_\theta] - 2\text{KL}_{target})^2$
    if $\text{KL}[\pi_{old}|\pi_\theta > 4\text{KL}_{target}$ **then**
      break and continue with next outer iteration $i+1$
    end if
    Compute $\nabla_\theta J_{PPO}$
    send gradient wrt. to θ to chief
    wait until gradient accepted or dropped; update parameters
  end for
  for $b \in \{1, \cdots, B\}$ **do**
    $L_{BL}(\phi) = -\sum_{t=1}^{T} (\hat{R}_t - V_\phi(s_t))^2$
    Compute $\nabla_\phi L_{BL}$
    send gradient wrt. to ϕ to chief
    wait until gradient accepted or dropped; update parameters
  end for
  if $\text{KL}[\pi_{old}|\pi_\theta] > \beta_{high}\text{KL}_{target}$ **then**
    $\lambda \leftarrow \tilde{\alpha}\lambda$
  else if $\text{KL}[\pi_{old}|\pi_\theta] < \beta_{low}\text{KL}_{target}$ **then**
    $\lambda \leftarrow \lambda/\tilde{\alpha}$
  end if
end for

Note that DPPO introduces more training tricks to workers than PPO. Such as **early stopping** and **parallel executing**. These tricks makes DPPO train a better model.

Implements:

Critic:

```
def build_critic_net(self):
    l1 = tf.layers.dense(inputs=self.s, units=100,
activation=tf.nn.relu,

kernel_initializer=tf.random_normal_initializer(0, 0.1),

bias_initializer=tf.constant_initializer(0.1), name='critic_l1')
    v = tf.layers.dense(inputs=l1, units=1, activation=None,

 kernel_initializer=tf.random_normal_initializer(0, 0.1),

 bias_initializer=tf.constant_initializer(0.1), name='v')
    return v
```

Actor:

```
def build_action_net(self, scope, trainable):
    with tf.variable_scope(scope):
        l1 = tf.layers.dense(inputs=self.s, units=100,
activation=tf.nn.relu,

kernel_initializer=tf.random_normal_initializer(0, 0.1),

bias_initializer=tf.constant_initializer(0.1), name='actor_l1',
trainable=trainable)
        mu = 2 * tf.layers.dense(inputs=l1, units=self.a_dim,
activation=tf.nn.tanh,  # Note: here has :  2* ...

kernel_initializer=tf.random_normal_initializer(0, 0.1),

bias_initializer=tf.constant_initializer(0.1), name='mu',
trainable=trainable)
        sigma = tf.layers.dense(inputs=l1, units=self.a_dim,
activation=tf.nn.softplus,

 kernel_initializer=tf.random_normal_initializer(0, 0.1),

 bias_initializer=tf.constant_initializer(0.1), name='sigma',
trainable=trainable)
        norm_dist = tf.distributions.Normal(loc=mu, scale=sigma)
    a_params = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
scope=scope)
```

```
        return norm_dist, a_params
```

One of the most significant things in implementation is about how to calculate loss. The loss of Critic (`c_loss`) can be treated as the square of TD-error. However, it's more complex to calculate the loss of Actor (`a_loss`). Generally speaking, there exists two different ways to calculate `a_loss`:

1. KL-penalty:

$$L^{KL}(\theta) = \hat{E}[\frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)}\hat{A}_t - \lambda KL[\pi_{old}|\pi_\theta]]$$

2. Clipped surrogate objective:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta) * \hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon) * \hat{A}_t)]$$

Both implementation of `a_loss` can work well in real test. But as for me, I prefer Clipped method. Because compared to KL-penalty method, it's more simple to implement and it's compatible with Gradient Descent. So in experiment part, I use Clipped method to implement PPO algorithm. The code is shown below:

c_loss:

```
with tf.variable_scope('c_loss'):
    c_loss = tf.reduce_mean(tf.square(self.advantage))
```

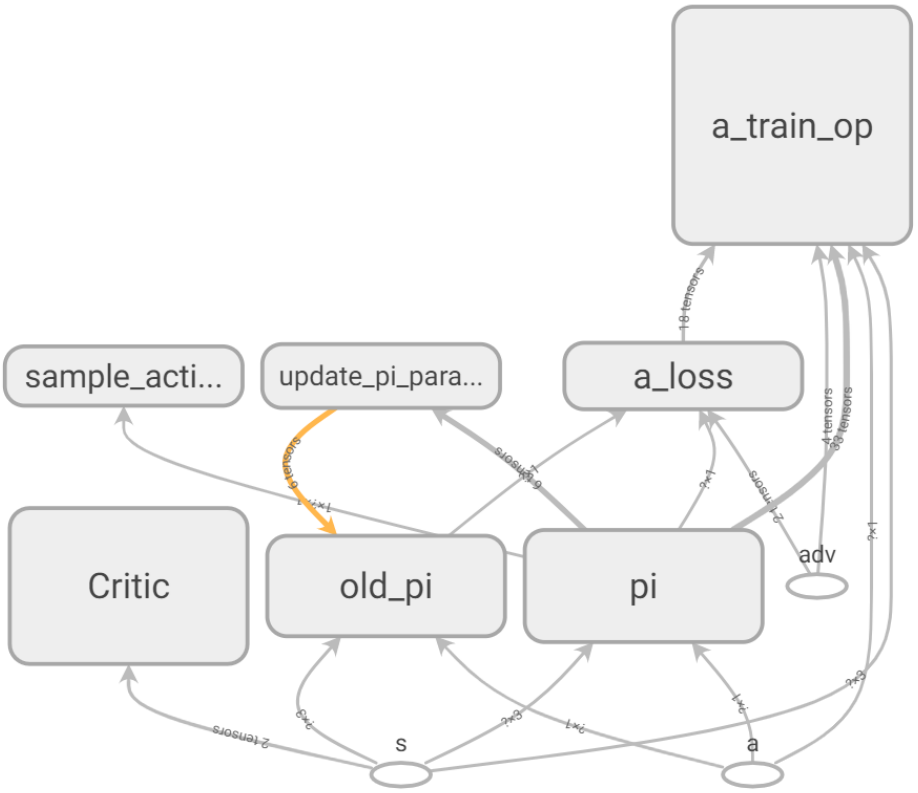
a_loss:

```
with tf.variable_scope('a_loss'):
    with tf.variable_scope('surrogate'):  # Note that here we add a
surrogate scope, to have better coding
        ratio = pi.prob(self.a) / (old_pi.prob(self.a) + 1e-5)
        surrogate = ratio * self.adv
    if self.method['name'] == 'kl_penalty':
```

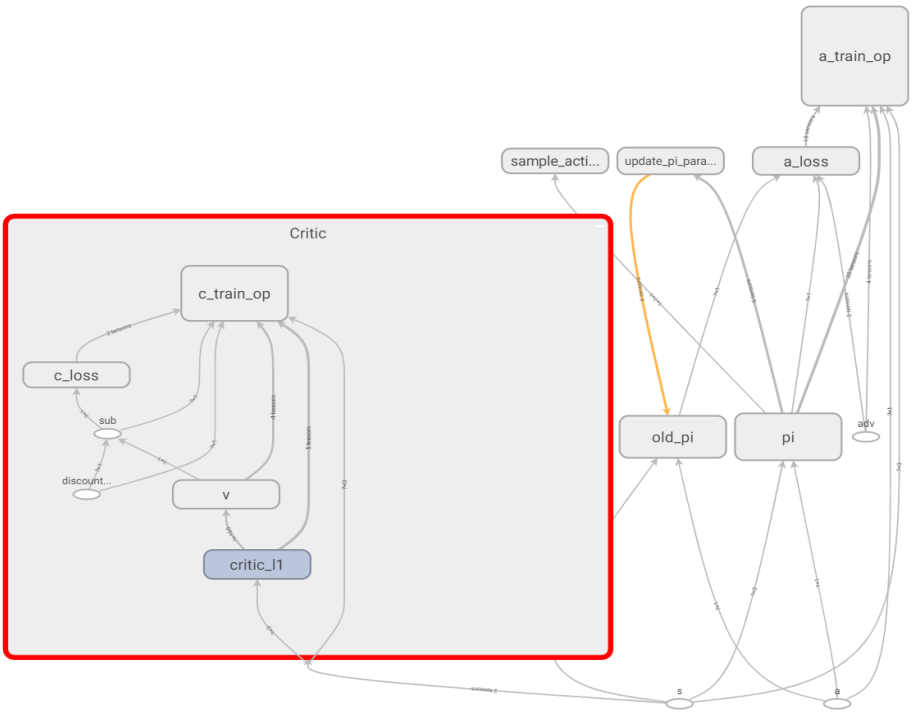
```
        kl = tf.distributions.kl_divergence(old_pi, pi)
        self.kl_mean = tf.reduce_mean(kl)
        self.lam = tf.placeholder(tf.float32, None, name='lam')
        a_loss = -tf.reduce_mean(surrogate - self.lam * kl)  #
Note: need to add '-' here
    else:
        J_theta = tf.minimum(surrogate,
                             tf.clip_by_value(ratio, 1. -
self.method['epsilon'],
                                              1. +
self.method['epsilon']) * self.adv)
        a_loss = -tf.reduce_mean(J_theta)  # No need to define
a_loss as class variable
```

Structures:

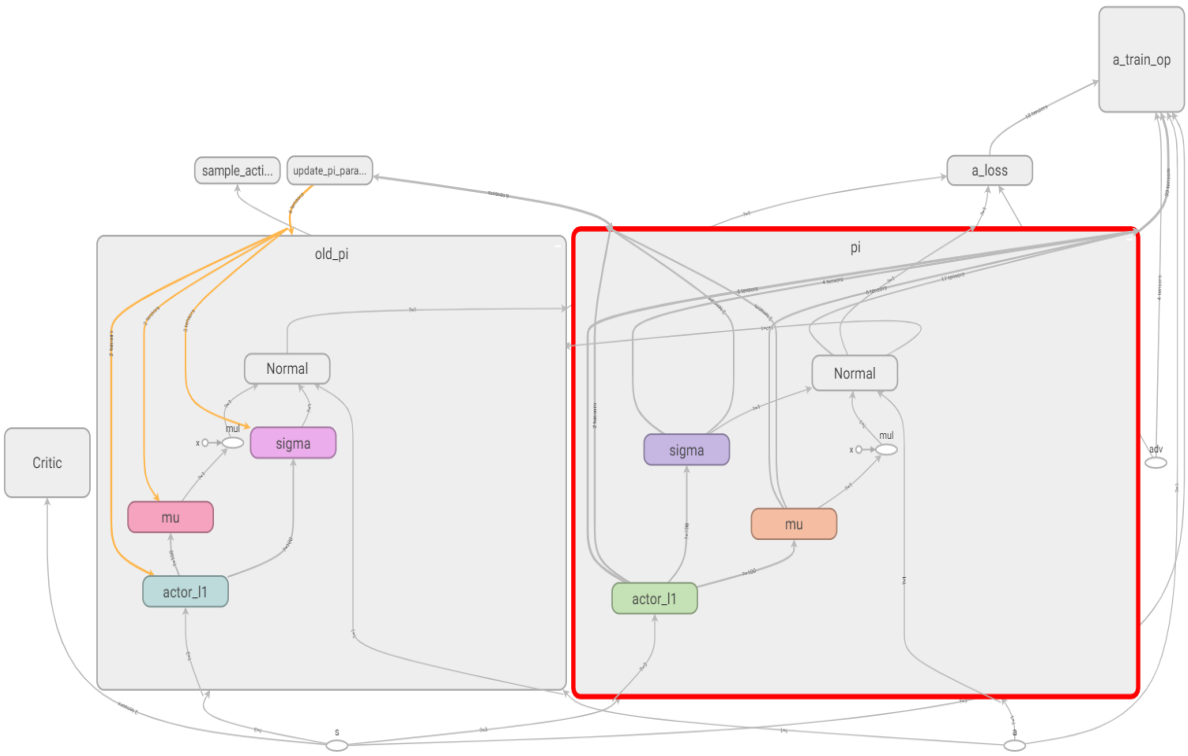
Model structure:



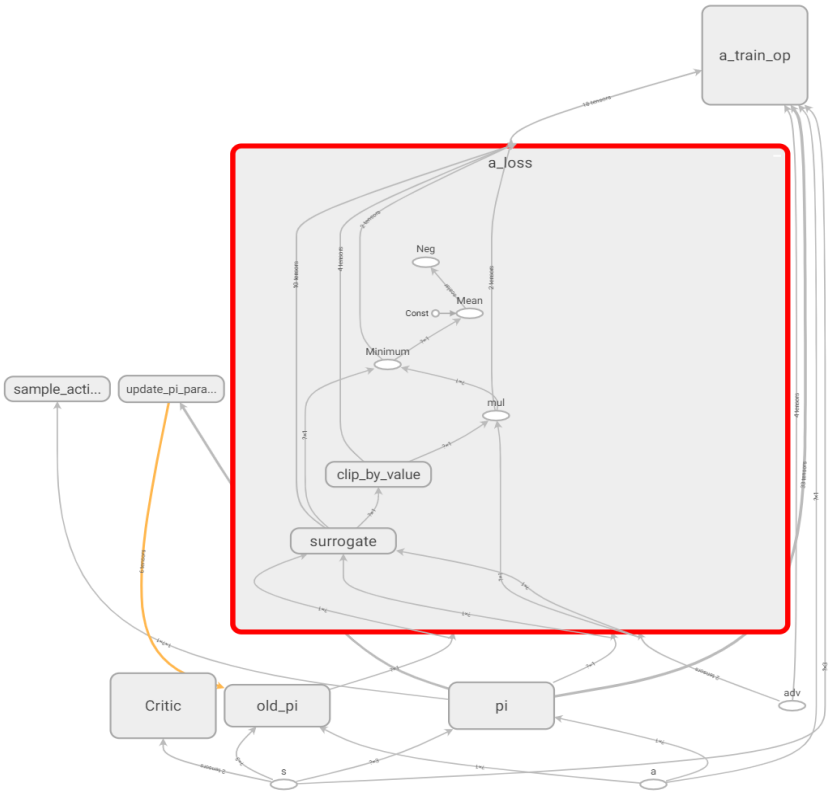
Critic structure:



Actor structure:



a_loss structure:



(3.2)

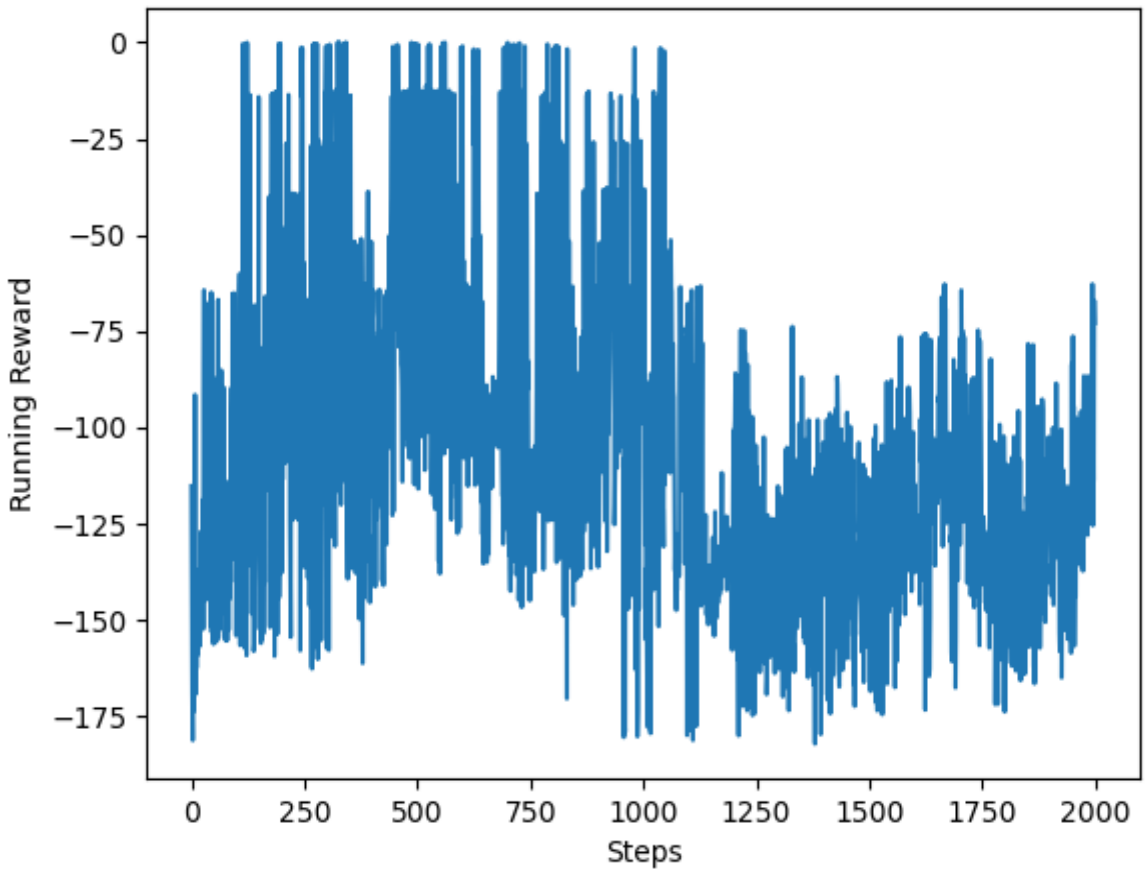
Advantages:

- Compared with ACER, TRPO:
    a. More general
    b. Simpler to implement
- Compared with A3C, DDPG:
    a. Better sample complexity
    b. Robustness

4 . Experiments

AC:

Running time: `2995.78s`



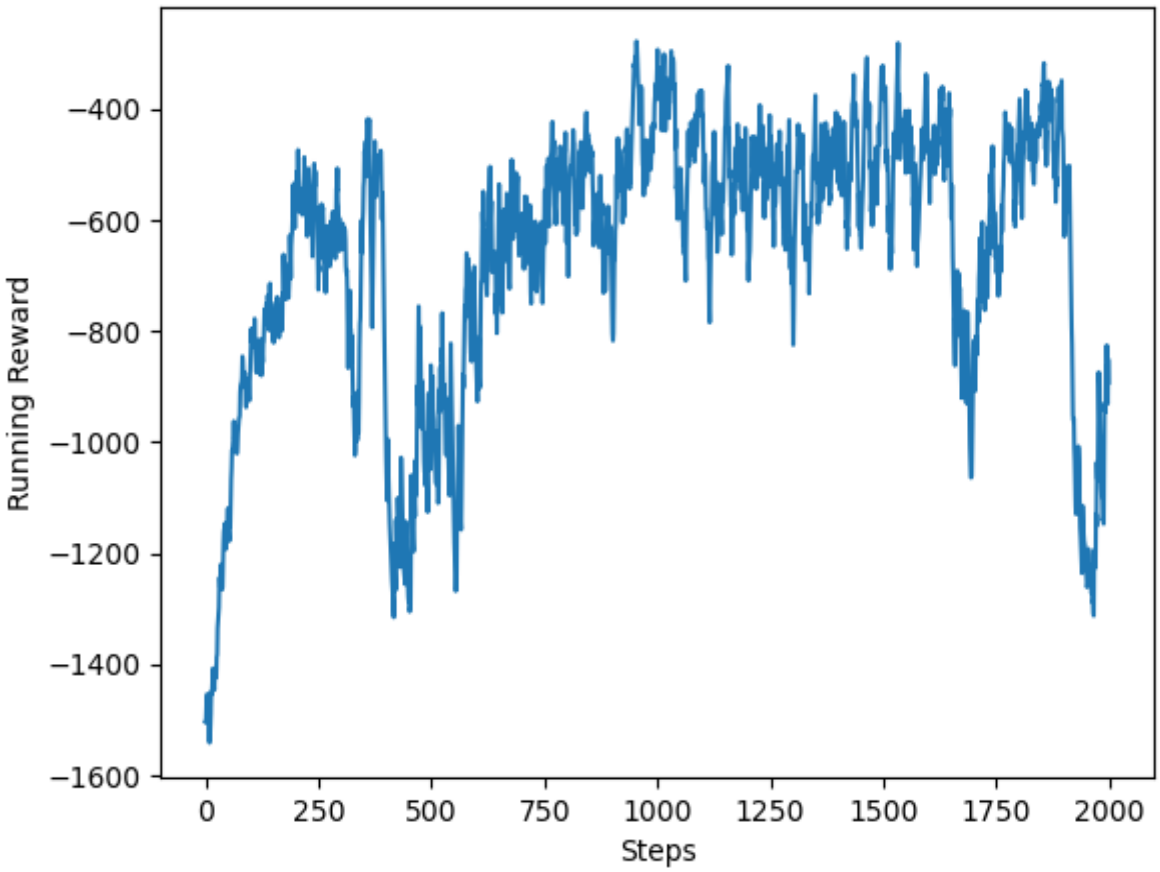
Analysis:

1. From the reward graph we can see that in the beginning the model converges to a relatively good optimal. However, after `1000` steps the model starts to diverge. We can see that the reward begins to decrease and reaches anther local optimal. This phenomenon shows the drawbacks of AC that it's hard for AC algorithm to converge. For the fact that model's convergence depends on Q-value estimated by Critic. However, the output of Critic depends on the sample action generated by Actor, which is updated continuously. Thus, the AC algorithm suffers the divergence problem.
2. Another important information we can get from the training reward graph is that the model can't converge smoothly to a local optimal. As we can see that the line oscillates up and down between `1500` and `2000` steps. We can use **Learning rate Decay** to alleviate this problem.
3. From the running time we can know that it takes relatively a long time to train a AC model. As we've mentioned above, the **sample efficiency** of policy-based algorithms is usually bad, which means that thousands of iterations are needed to train a single model. It will spend a lot of time to train and it needs high computational power.

A3C:

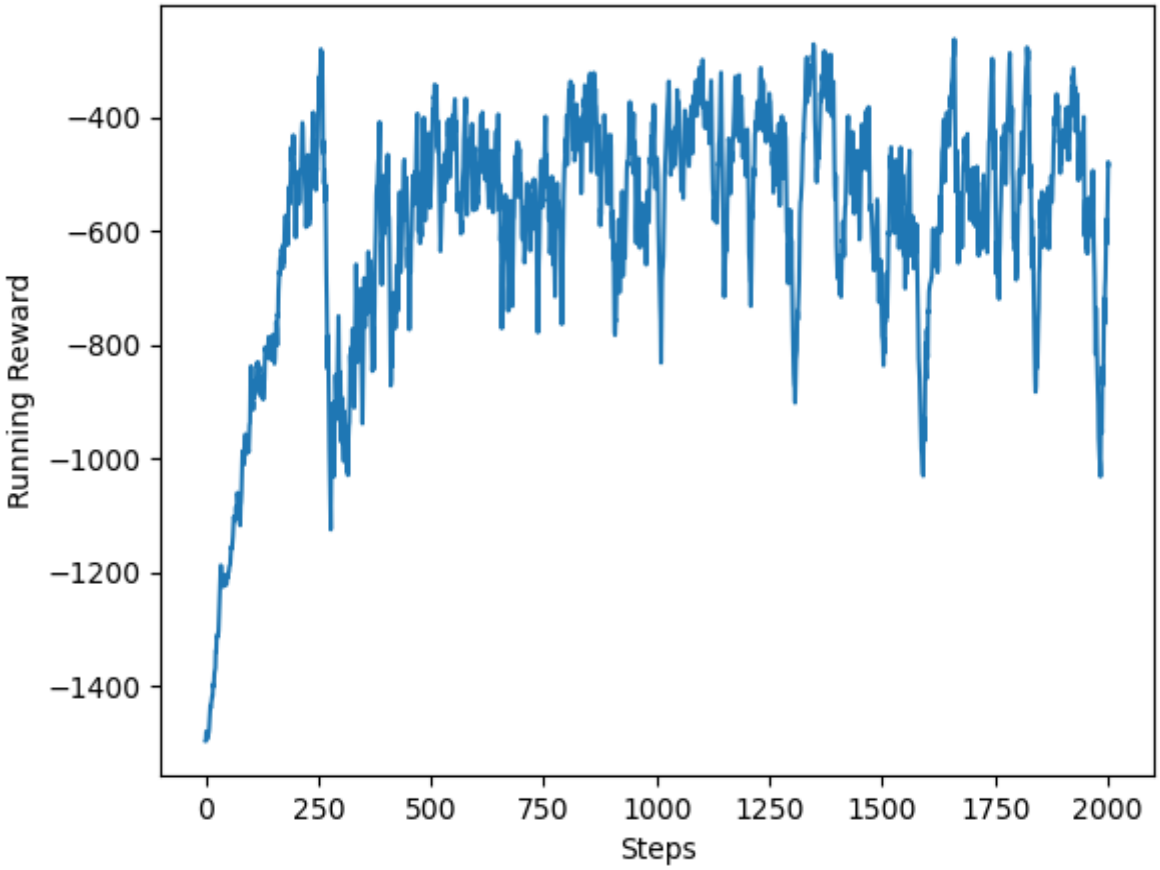
number of workers = `1`:

Running time: `2278.68s`



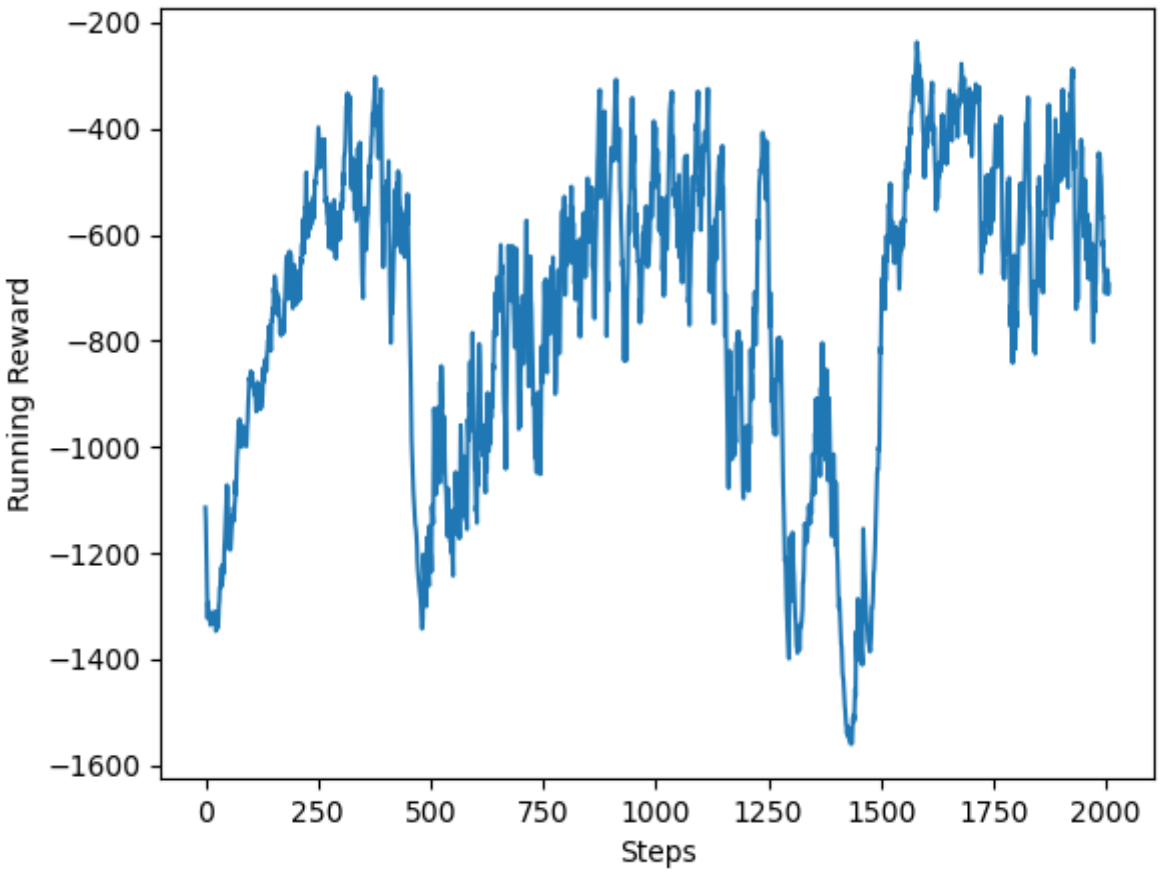
number of workers = `4`:

Running time: `124.02s`



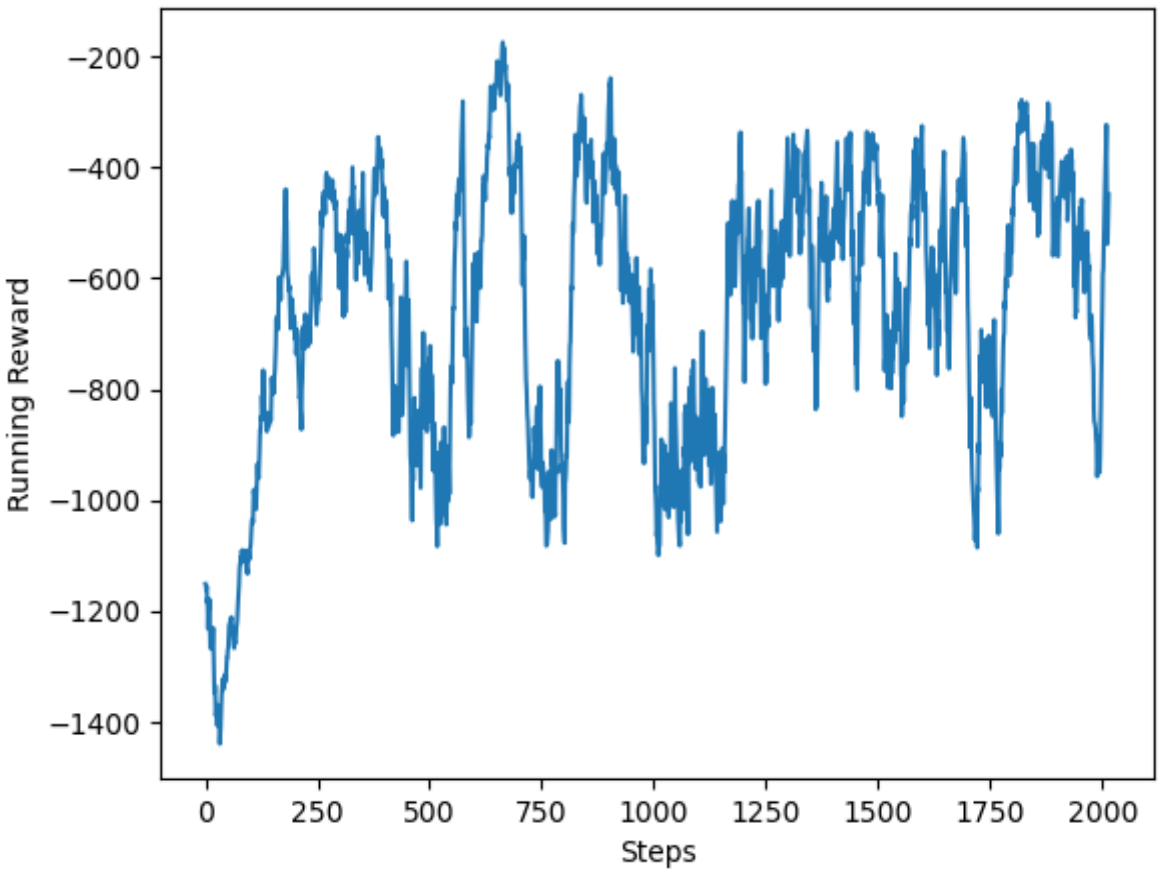
number of workers = `8`:

Running time: `126.84s`



number of workers = `16`:

Running time: `135.10s`



Analysis:

1. From the results above we can see that all the A3C experiments converged, except for the setting of `number of  worker = 1`, which is actually AC (we can see that after `1800` steps it starts to diverge). The contrast shows that A3C can improve divergence problem by reducing the relevance of parameter in updating.
2. We can also find that the final convergence optimal is different in these settings. The more the number of workers , the better the local optimal we can reach.
    - **number of workers** = `4`**:** approximately `-400`
    - **number of workers** = `8`**:** approximately `-300`
    - **number of workers** = `16`**:** approximately `-200`

    There are mainly two reasons for bringing about this result:

    a. The global network uses `pull` function to collect gradients from workers so as to update its policy. If more information can be obtained from those workers, the updated policy will be more stable. This leads to a better performance.

    b. The workers work independently to explore the environment and submit their distinctive experience (data collection, gradient calculation) back to the global network. So increasing the number of

workers will increase the exploration ability of the A3C model, which helps to find a better local optimal.

3. The difference of running time: the more the number of workers, the higher the running time.

    - **number of workers** = `4`**:**

        Running time: `124.02s`

    - **number of workers** = `8`**:**

        Running time: `126.84s`

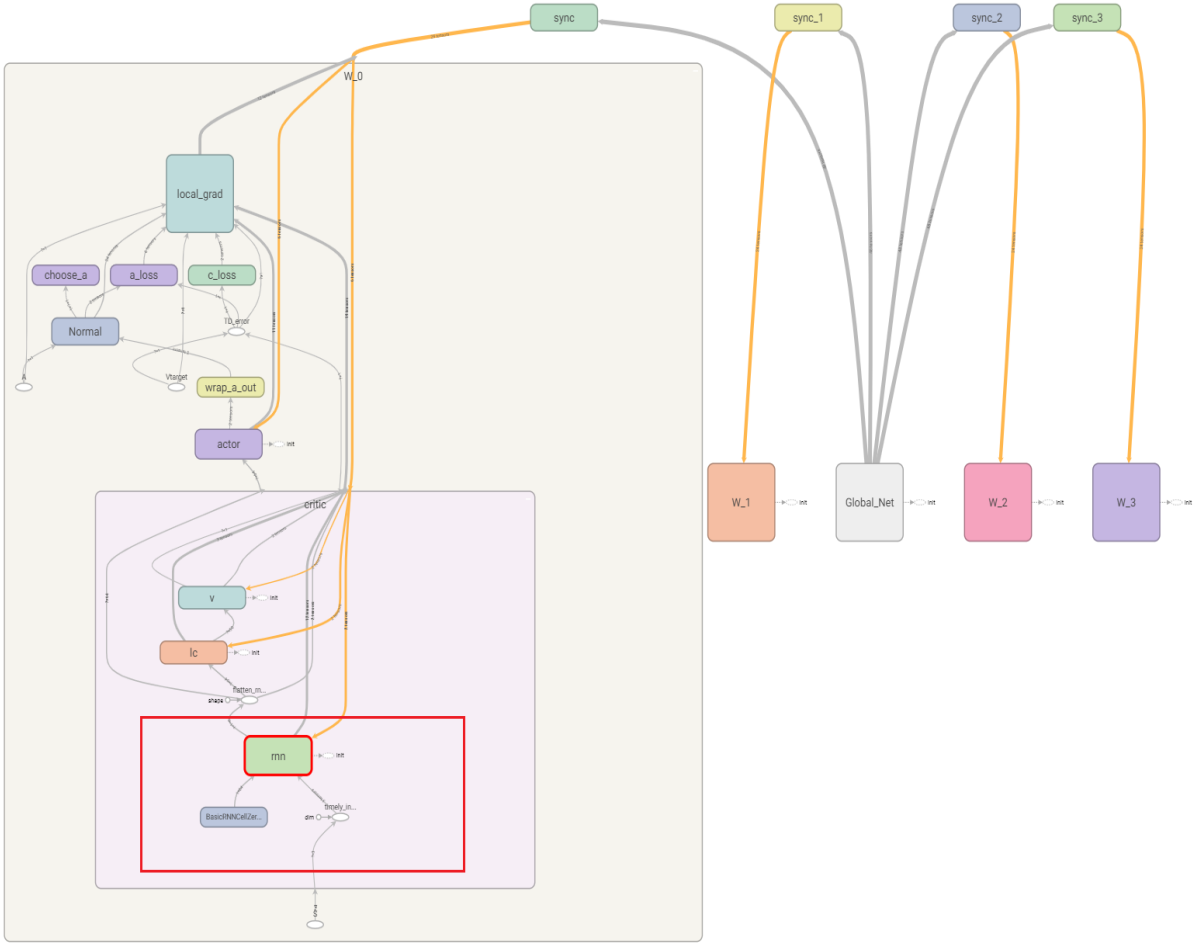
    - **number of workers** = `16`**:**

        Running time: `135.10s`

    When the global network wants to update, all worker threads must be suspended until the main thread completes its task. After that, the updated policy needs to be synchronously assigned to all the workers. Because the main process costs more time in **thread synchronization** and **assigning** updated policy to worker threads, it's reasonable that "the more the number of workers, the higher the running time".

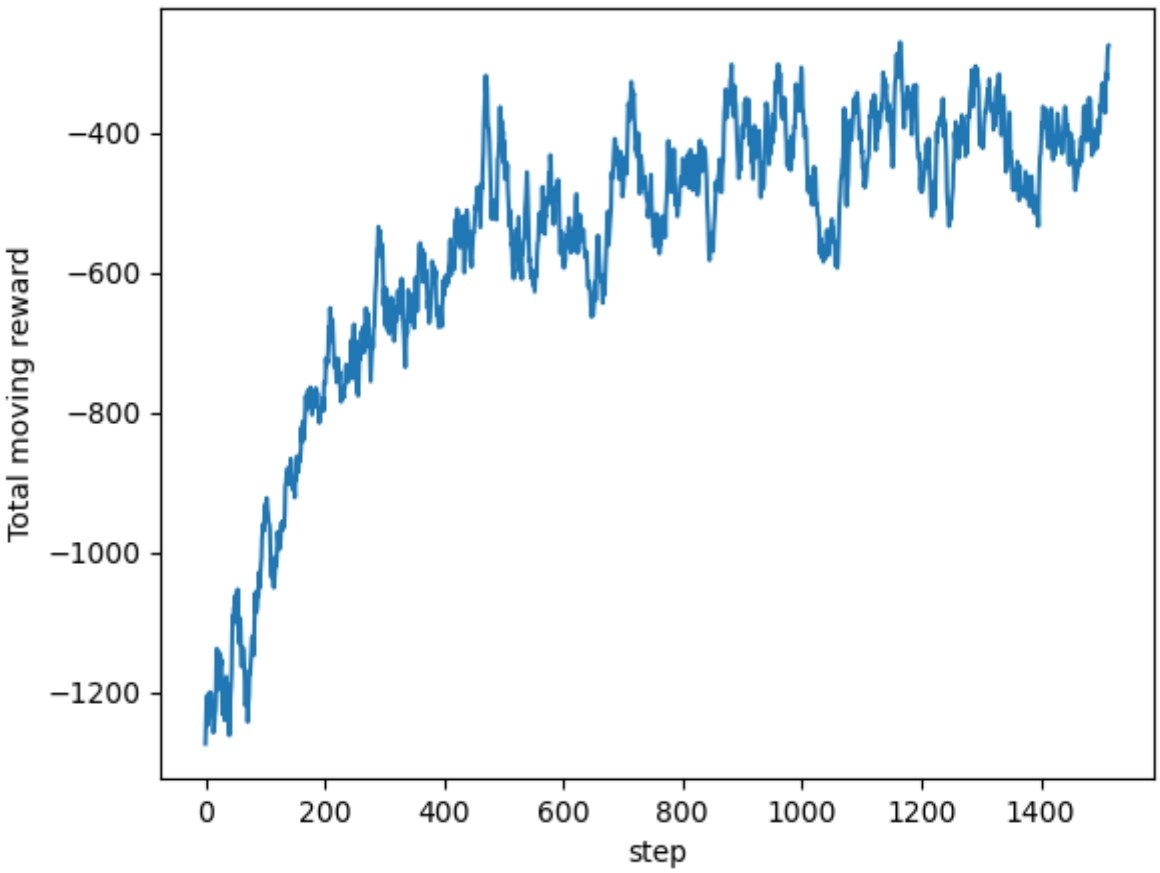
A3C + RNN (number of workers = 16):

Network Structure:



We can see from the network structure graph that the only difference between A3C model and A3C+RNN model is that the critic in A3C+RNN model uses `RNN` layer to extract the features. All other structures are the same. Here are the results:

Running time: `161.20s`

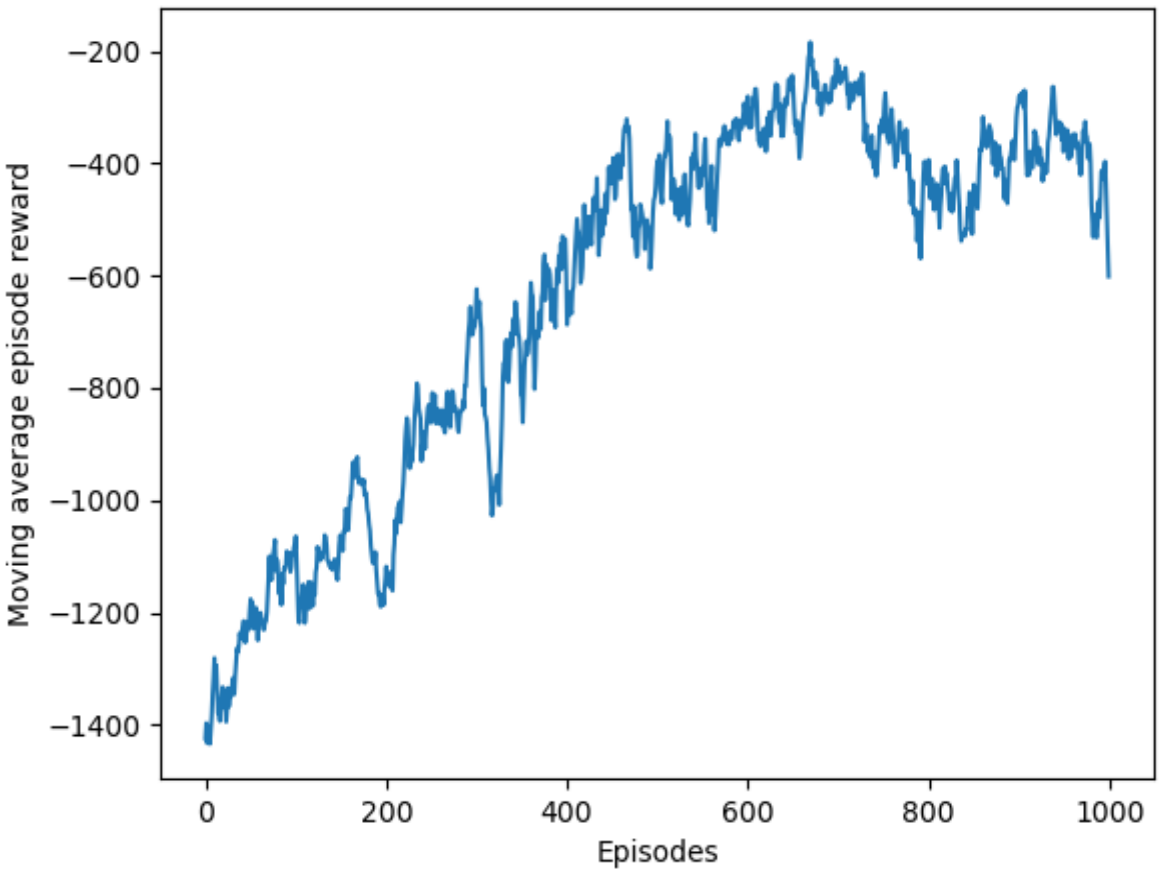


Analysis:

1. From the result we can see that the model converged after `600` steps. Compared with A3C model (number of workers = 16), though A3C+RNN model didn't find such good local optimal as AC (it can only reach about `-400`), it converged more stably in the end.
2. The training time is longer (A3C+RNN: `161.20s`; A3C: `135.10s`), because of the high calculation time for RNN units.

PPO:

Running time: `315.78s`



Analysis:

1. The training time is longer, because there is no multi-threading (If DPPO is used, it will save a lot of time) . However, compared with A3C (number of worker = 1), the training time is significantly shorter, and episodes are less, because the sampling efficiency of PPO is higher.
2. Good performance:
    a. the reward of convergence is almost the highest;
    b. Stability after convergence
3. The implementation is simple, without multi-threading code, PPO is easy to debug (One difficulty of multi-threading code is that it is difficult to debug)

Considering comprehensively, I think PPO is the best model for this assignment settings.