Reinforcement Learning: Core Algorithms

Earl Wong

Core Algorithms

We examine the following algorithms:

- PPO
- Soft Actor Critic
- DDPG

PPO

- PPO is a model free, on policy based algorithm (-evolution of the base actor critic algorithm), succeeding TRPO.

- PPO prevents large, positive losses from being back propagated into the network - losses that could destabilize actor / policy training.

- This is achieved by computing the following ratio $r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$ between the current policy and the old policy, and clipping the ratio when it becomes too large.

- Under PPO, the policy improves gradually and steadily.

PPO

- Paper title: Proximal Policy Optimization Algorithms
- Authors: Schulman, et. al.

Foundational “Roots”

- PPO can be viewed as the “engineering” implementation / realization of a more theoretically grounded algorithm, TRPO.
- In TRPO, actor / policy updates are performed in a constrained optimization framework.
- A second order conjugate gradient optimizer is used to update the network parameters.
- In TRPO, a KL divergence constraint is applied to the old and current policies, ensuring that the two policies remain close to each other.

Foundational “Roots”

- PPO “trades off” the guarantees from TRPO, with ease of use and solid empirical results.
- In PPO, actor / policy updates are performed in an unconstrained optimization framework.
- The stochastic gradient descent optimizer is used to update the actor / policy network parameters.
- In PPO, the KL divergence constraint is enforced using a heuristic clipping function.

PPO

Algorithm 1 PPO, Actor-Critic Style

for iteration$=1, 2, \ldots$ **do**
  for actor$=1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

The Details

- The first main detail of the algorithm, is the surrogate loss function, used to train the actor / policy.
- In TRPO, the following constraint existed: $D_{\mathrm{KL}}^{\max}(\theta_{\mathrm{old}}, \theta) \leq \delta.$
- This constraint required the old policy and current policy to be “close” for every point in the state space.
- Since this constraint was not practical, it was relaxed in the TRPO paper to an “average” KL divergence.

The Details

- In PPO, clipping is applied to the policy ratio for every point in the state space.
- Specifically $\min\left(r_t(\theta)\,\hat{A}_t,\ \textbf{clip}\left(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon\right)\hat{A}_t\right)$
- The clip function discourages back propagating large, positive losses that can destabilize training.
- i.e. It is better to take small, positive steps to encourage actions with positive advantage, rather than taking a single, large, positive step.
- Note: One of the goals of TRPO, was to determine the largest permissible step that could be taken (i.e. trust region) while satisfying a closeness constraint between the old and new policies.
- Note: The surrogate function (shown above) does not protect against taking large, negative steps.

The Details

- The second main detail involves the advantage estimates $\hat{A}_t$ that are used to train the critic.

- In practice, generalized advantage estimates are used, to balance the bias - variance tradeoff.

- i.e. If the critic is updated using single time steps (or 1-step advantage estimates), the variance will be low, but the bias will be high.

- i.e. If the critic is updated using the complete monte carlo rollout, the variance will be high, but the bias will be low.

The Details

- One step advantage estimate: $\hat{A}_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
- Generalized advantage estimate: $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$

$$\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l}$$

- λ controls the bias - variance tradeoff.
- If λ = 1, we have the high variance and low bias scenario.
- If λ ~= 0, we have the low variance and high bias scenario.

The Details

- The final detail involves the introduction of an entropy term in the surrogate loss function, to prevent pre-mature convergence:

$$\mathscr{H}[\pi(\cdot \mid s_t)] = -\sum_a \pi(a \mid s_t) \log \pi(a \mid s_t)$$

- This term helps to keep exploration alive, when the surrogate loss is small.

- In practice, sample based entropy $\log \pi(a \mid s_t)$ is used to compute entropy.

- Note: As the confidence in a policy action increases, the entropy will naturally decrease.

The Details

- PPO can be applied to both continuous and discrete action spaces.
- For discrete action spaces, the logits are converted to action probabilities via the softmax function.
- For continuous action spaces, the network outputs mean and standard deviation values for every input state.
- A gaussian distribution is then created and sampled from, to obtain the continuous action associated with the input state.
- In practice, the reparameterization trick is used, to back propagate through the process to train the network.

Reparameterization Trick

- The reparameterization trick results from earlier work in variational auto encoders (VAE).
- There, a gaussian distribution was used.
- However, other distributions can also be used.
- i.e. The reparameterization trick can be applied, using other distributions.

Reparameterization Trick

In practice, gaussian distributions are a “fan favorite” because:

- Gaussian distributions occur naturally in nature.
- Gaussian distributions have nice properties, like smoothness / differentiability.
- Gaussian distributions are mathematically tractable / usually result in closed form, analytic solutions.
- Example: VAE’s minimize KL divergence. If both terms are gaussian distributions, a closed form expression results.

Reparameterization Trick

- Now, what is the trick, and why is it needed?

- First, let's address the need.

  Flow:

- For a given state input, the network outputs a mean value (say b) and std value (say a).

- A normal random variable y is created, using these output values:

$$y \sim N(b, a^2)$$

- The random variable is then sampled -corresponding to an action.

Reparameterization Trick

The problem:

- Gradients cannot be back propagated through a sampling process.
- What is the workaround?
- The reparameterization trick.

Reparameterization Trick

How does it work?

- Define a new function / random variable: $z = \mu + \sigma \cdot \epsilon$
- Where $\epsilon \sim N(0,1)$
- By construction, z is a differentiable function with respect to the mean and standard deviation.
- Now, back propagation can occur.
- Note: Sampling now results from the epsilon random variable.

Reparameterization Trick

- Does the new random variable z, have the same distribution as the normal distribution y, created earlier?
- Yes.
- Why?
- We have applied a linear transformation aX + b, to a N(0,1) random variable with mean b and standard deviation a.
- The resulting random variable z, has the same distribution as $y \sim N(b, a^2)$

Soft Actor Critic

- Soft actor critic is a model free, off policy algorithm (-evolution of the base actor critic algotithm), designed for continuous action spaces.

- Soft actor critic employs a stochastic policy - usually gaussian.

- Soft actor critic introduces entropy directly into the objective function (vs the loss function).

- As a result, the objective changes from reward maximization to BOTH reward maximization and entropy maximization.

- Soft actor critic employs two critics to reduce over estimation biases.

Soft Actor Critic

- Paper title: Soft Actor-Critic: Off Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor
- Authors: Haarnoja, Zhou, Abbeel, Levine

Foundational “Roots”

- In PPO, entropy was introduced into the LOSS FUNCTION, to keep training alive / prevent early convergence to a local minima.
- In SAC, entropy is directly infused into the TRAINING OBJECTIVE, to encourage BOTH high reward and high exploration.
- In SAC, entropy is also introduced into the two target Q networks, used to train the two main Q networks (critics).
- The latter entropy helps keep training alive / prevents early convergence of the two main Q networks (critics), to a local minima.

Foundational “Roots"

- In SAC, continuous actions are obtained by sampling a gaussian distribution.
- The gaussian distribution is created from u and std vector outputs from a neural network, for a given state vector input.
- In order to successfully perform back propagation through this structure, the reparameterization trick is employed.
- The reparameterization trick was first introduced during the training of variational auto encoders.

Foundational “Roots”

- Whenever $max_a Q(s, a)$ appears in a policy, over estimation biases result.
- This is because of the inherent noise present, in the action value function output.
- i.e. By taking the max of noisy expected returns, over estimation results.
- Over estimation bias was first addressed by Van Hasselt in 2010 for the tabular data case, with the introduction of two Q-tables.
- Soft Actor Critic addresses overestimation bias by 1) estimating two Q networks (two Q critics) and 2) taking the min of the values produced.

Foundational “Roots”

- Like DQN, training stability of the Q network was of paramount importance.

- With DQN, a target network was introduced, storing a “delayed” version of the main Q network parameters, to enhance learning stability.

- In Soft Actor Critic, two target Q networks are introduced alongside the two main Q networks (critics).

- In Soft Actor Critic, polyak averaging is used to update the parameters of the two target Q networks.

Foundational “Roots”

- Since SAC is an offline algorithm, SAC samples from and writes to (=updates the inputs in) a replay buffer.
- This activity parallels the activity described in DQN algorithm training.
- i.e. Experience replay and sampling from a replay buffer.

Soft Actor Critic - original paper

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$
  end for
end for

Soft Actor Critic - from openAI

Pseudocode

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

The Details

- Soft Actor Critic maximizes both return and exploration.

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta \left(\tilde{a}_\theta(s) | \, s \right) \Big),$$

- Soft Actor Critic trains two main Q networks (critics), using the squared error loss.

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s,a) - y(r,s',d) \right)^2$$

The Details

- In the first equation, entropy is specifically added to the training objective.

- Recall (previous):

$$\pi_* = argmax_\pi E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t, a_t)\right]$$

- Now:

$$\pi_* = argmax_\pi E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t (R(s_t, a_t) + \alpha H(\pi(\,\cdot\,|\,s_t)))\right]$$

The Details

- The state value function now becomes the “soft” version:

$$V_\pi(s) = E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t(R(s_t, a_t) + \alpha H(\pi(\,\cdot\,|\,s_t)))\,|\,s_0 = s\right]$$

- The action value function now becomes the “soft” version:

$$Q_\pi(s, a) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_t, a_t, s_{t+1}) + \alpha H\big(\pi(\,\cdot\mid s_t)\big)\right)\,\middle|\, s_0 = s, a_0 = a\right]$$

- Linking state and action value functions via entropy:

$$V_\pi(s) = E_{a\sim\pi}[Q_\pi(s, a)] + \alpha H(\pi(\,\cdot\,|\,s))$$

The Details

- With the addition of entropy, the Bellman Equation for the Q network now becomes:

$$Q_\pi(s,a) = \mathbb{E}_{\substack{s' \sim P(\cdot \mid s,a) \\ a' \sim \pi(\cdot \mid s')}} \left[R(s,a,s') + \gamma \left(Q_\pi(s',a') + \alpha H\big(\pi(\cdot \mid s')\big) \right) \right]$$

- Or:

$$Q_\pi(s,a) = \mathbb{E}_{\substack{s' \sim P(\cdot \mid s,a) \\ a' \sim \pi(\cdot \mid s')}} \left[R(s,a,s') + \gamma V_\pi(s') \right]$$

The Details

- Soft Actor Critic samples from a gaussian distribution, and uses the reparameterization trick to back propagate through the network.

- In python code:

```python
def sample_action_manual(mu, std):
    epsilon = torch.randn_like(std)            # Sample from N(0, 1)
    action = mu + std * epsilon                # Reparameterized sample
    return action
```

```python
def sample_action_rsample(mu, std):
    dist = torch.distributions.Normal(mu, std)
    action = dist.rsample()
    return action
```

```python
mu, std = policy(state)

# Manual
action_manual = sample_action_manual(mu, std)

# One-liner
action_rsample = sample_action_rsample(mu, std)
```

The Details

- Soft Actor Critic reduces over estimation bias by utilizing two target Q networks, and selecting the action that results in the minimum return.

$$y(r, s', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

- Soft Actor Critic introduces entropy into the target network used for main Q network training, to reduce the chance of early convergence to a local minima.

- Soft Actor Critic updates the target network using polyak averaging between the main and target network parameters.

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1-\rho)\phi_i$$

DDPG

- Deep deterministic Policy Gradient (DDPG) is a model free, off policy based actor-critic algorithm.
- DDPG extends DQN, by producing a policy with continuous actions, enabling the algorithm to be applied to physical control tasks.
- DDPG learns policies with deterministic actions.

DDPG

- Paper title: Continuous Control with Deep Reinforcement Learning
- Authors: Lillicrap, et. al.

Foundational “Roots”

- DDPG extends DQN to continuous action spaces.
- DDPG uses the same two tricks to train DQN: 1) target network and 2) replay buffer with temporally uncorrelated / near IID inputs.
- DDPG applies normalization across input features, much like classical machine learning approaches.

DDPG

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

The Details

- Like REINFORCE in vanilla policy gradient, DDPG wants to find the network weights that will maximize the return.

$$J(\theta) = E_{\tau \sim u}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right]$$

- However, instead of maximizing the return over a trajectory using a stochastic policy, the return is now maximized over a deterministic policy based on a state visitation distribution.

$$J(\theta) = E_{s \sim \rho^u}\left[Q^u(s, u_\theta(s))\right]$$

- $\rho^u(s)$ Is used to denote the state visitation distribution.

The Details

- It is formally defined as: $$\rho^u(s) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s \mid u_\theta)$$
- The objective function can now be written as:

$$J(\theta) = \int \rho^u(s) r(s, u_\theta(s)) ds = E_{s \sim \rho^u} \left[Q_u(s, u_\theta(s)) \right]$$

- Differentiating the objective, we obtain:

$$\nabla_\theta J(\theta) = \int \rho^u(s) \nabla_\theta Q_u(s, u_\theta(s)) ds$$

$$\nabla_\theta J(\theta) = \int \rho^u(s) \nabla_a Q_u(s, a) |_{a=u_\theta(s)} \nabla_\theta u_\theta(s)) ds$$

$$\nabla_\theta J(\theta) = E_{s \sim \rho^u} \left[\nabla_a Q_u(s, a) |_{a=u_\theta(s)} \nabla_\theta u_\theta(s)) \right]$$

The Details

- Note: $\rho^u(s)$ does depend on the network parameters.
- However, the author has chosen to ignore this “indirect” dependency, and the corresponding chain rule term associated with it.

The Details

- DDPG encourages exploration by adding a noise term to the deterministic action produced by the policy.

$$a_t = u(s_t \mid \theta^u) + N_t$$

The Details

- DDPG uses two target networks -one for the Q function and one for the target policy:

$$\theta^{Q'} \leftarrow \theta^{Q}, \theta^{u'} \leftarrow \theta^{u}$$

- DDPG updates the network parameters in the target Q function and target policy using weighted averaging:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$

$$\theta^{u'} \leftarrow \tau\theta^{u} + (1-\tau)\theta^{u'}$$