Basic concept

1)Policy-based method
probability of neural network to approximate a policy

2) hill climbling
The agent's goal is to maximize expected return J ,

$$J(\theta) = \sum_{\tau} P(\tau;\theta)R(\tau)$$

(from udacity)
pseudocode:

Hill Climbing

Initialize the weights θ in the policy arbitrarily.

Collect an episode with θ, and record the return G.

$\theta_{best} \leftarrow \theta, G_{best} \leftarrow G$

Repeat until environment solved:

Add a little bit of random noise to θ_{best}, to get a new set of weights θ_{new}.

Collect an episode with θ_{new}, and record the return G_{new}.

If $G_{new} > G_{best}$, then:
$\theta_{best} \leftarrow \theta_{new}, G_{best} \leftarrow G_{new}$

(from udacity)

G is a single episode return and is not good estimate value of expected return J
3)stochastic policy search
steepest ascent : help reduce risk of selecting a next policy. Choose the looks most promising. Have stuck of local optima
simulated annealing : control how to search strategies space by random and noise start.
adaptive noise : adapt to obverse change of strategies value

4)black-box optimization techniques
cross-entropy method :take average. Dont know solving problem meaning
evolution strategies: the optimization is a "guess and check" process.
RL injects noise in the action space and uses backpropagation to compute the parameter updates, while ES injects noise directly in the parameter space;weight sum of policy that get higher return(https://openai.com/blog/evolution-strategies/)

5)trajectory
state-action sequence

$$U(\theta) = \sum_{\tau} \mathbb{P}(\tau;\theta)R(\tau)$$

(from udacity)
for continuing tasks and instead of episode

6)drivation
likelihood radio trick(reinforce trick)

$$\nabla_\theta \log P(\tau;\theta) = \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)}$$

(from udacity)

a sample-based average

$$\nabla_\theta U(\theta) \approx \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log \mathbb{P}(\tau^{(i)};\theta) R(\tau^{(i)})$$

(from udacity)

7)Gradients are noisy
solution:sample more trajectories (in patallel),noise reduction

$$\left.\begin{array}{c} s_t^{(1)}, a_t^{(1)}, r_t^{(1)} \\ s_t^{(2)}, a_t^{(2)}, r_t^{(2)} \\ s_t^{(3)}, a_t^{(3)}, r_t^{(3)} \\ \vdots \end{array}\right\} \rightarrow g = \frac{1}{N}\sum_{i=1}^{N} R_i \sum_t \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})$$

(from udacity)

another bonus for running multiple trajectories: we can collect all the total rewards and get a sense of how they are distributed

$$R_i \leftarrow \frac{R_i - \mu}{\sigma} \qquad \mu = \frac{1}{N}\sum_i^N R_i \qquad \sigma = \sqrt{\frac{1}{N}\sum_i (R_i - \mu)^2}$$

(from udacity)

8)Policy Gradient

a better policy gradient would simply have the future reward as the coefficient. It turns out that mathematically, ignoring past rewards might change the gradient for each specific trajectory, but it doesn't change the averaged gradient.
a better policy gradient:

$$g = \sum_t R_t^{\text{future}} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

(from udacity)

Future reward; policy gradient;

9) sample

policy update in reinforce : current trajectories to compute gradient
important sampling : average of some quantity f(tau)

$$\sum_{\tau} \overbrace{P(\tau;\theta)}^{\substack{\text{sampling under} \\ \text{old policy } \pi_\theta}} \overbrace{\frac{P(\tau;\theta')}{P(\tau;\theta)}}^{\substack{\text{re-weighting} \\ \text{factor}}} f(\tau)$$

(from udacity)

sample: re-weighting factor

$$\frac{P(\tau;\theta')}{P(\tau;\theta)} = \frac{\pi_{\theta'}(a_1|s_1)\,\pi_{\theta'}(a_2|s_2)\,\pi_{\theta'}(a_3|s_3)\ldots}{\pi_\theta(a_1|s_1)\,\pi_\theta(a_2|s_2)\,\pi_\theta(a_2|s_2)\ldots}$$

(from udacity)

When some of policy gets close to zero, When this happens, the re-weighting trick becomes unreliable. So, In practice, we want to make sure the re-weighting factor is not too far from 1 when we utilize importance sampling

7)clipped surrogate function

$$g = \nabla_{\theta'} L_{\mathrm{sur}}(\theta', \theta)$$

$$L_{\mathrm{sur}}(\theta', \theta) = \sum_t \frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} R_t^{\mathrm{future}}$$

(from udacity)

comparing different policies,reusing old trajectories and updating policy ,it may cause approximation invalid that at some point the new policy might become different enough from the old one

some point hit cliff, gradient is zero and update stop.We want to make sure the two policy is similar, or that the ratio is close to 1. So we choose a small ϵ(typically 0.1 or 0.2), and apply the clip function to force the ratio to be within the interval $[1-\epsilon,1+\epsilon]$:

$$L_{\mathrm{sur}}^{\mathrm{clip}}(\theta', \theta) = \sum_t \min\left\{ \frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} R_t^{\mathrm{future}}, \mathrm{clip}_\epsilon\left(\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \right) R_t^{\mathrm{future}} \right\}$$

(from udacity)

8)Baselines and Critics

monte carlo estimate has high variance ans no bias, TD estimate has low variance but low bias. Use TD estimate to train critic for reducing variants thus improving convergence properties and speeding up learning. Monte carlo baseline

9)N-step Bootstrapping

wait some step for faster convergence, TD is onw-step bootstrapping.monte carlo estimate is an infinite step boostrapping.

10)DDPG
DQN : copy same weights
DDPG : mix in 0.01% weight to target network weight

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

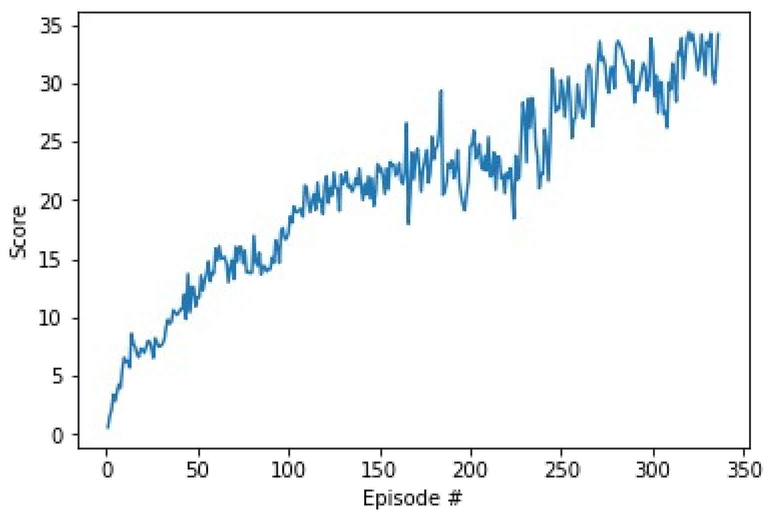
(paper: CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING)

code

```
def learn(self, experiences, gamma):
    """Update policy and value parameters using given batch of experience tuples.
    Q_targets = r + γ * critic_target(next_state, actor_target(next_state))
    where:
        actor_target(state) -> action
        critic_target(state, action) -> Q-value

    Params
    ======
        experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
        gamma (float): discount factor
    """
    states, actions, rewards, next_states, dones = experiences

    # ---------------------------- update critic ---------------------------- #
    # Get predicted next-state actions and Q values from target models
    actions_next = self.actor_target(next_states)
    Q_targets_next = self.critic_target(next_states, actions_next)
    # Compute Q targets for current states (y_i)
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
    # Compute critic loss
```

```
        Q_expected = self.critic_local(states, actions)
        critic_loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        torch.nn.utils.clip_grad_norm_(self.critic_local.parameters(), 1)####
        self.critic_optimizer.step()

        # ---------------------------- update actor ---------------------------- #
        # Compute actor loss
        actions_pred = self.actor_local(states)
        actor_loss = -self.critic_local(states, actions_pred).mean()
        # Minimize the loss
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

        # ----------------------- update target networks ----------------------- #
        self.soft_update(self.critic_local, self.critic_target, TAU)
        self.soft_update(self.actor_local, self.actor_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        θ_target = τ*θ_local + (1 - τ)*θ_target

        Params
        ======
            local_model: PyTorch model (weights will be copied from)
            target_model: PyTorch model (weights will be copied to)
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
            target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

the result change figure:



(score result)

parameter

1)BUFFER_SIZE = int(2e6) # replay buffer size
2)BATCH_SIZE = 64   # minibatch size: if number is 128 or 256 ,the learning speed will slow down.
3)GAMMA = 0.99   # discount factor
4)TAU = 1e-3   # for soft update of target parameters
5)LR_ACTOR = 1e-4   # learning rate of the actor : in some extent , the small one will let actor learning slower than large one
6)LR_CRITIC = 3e-4   # learning rate of the critic
7)layer size :
size will effect on the convergence. The number is smaller ,like actor 256 ,the average reward will fast to 29 ,then value to drop down. The layer number is influence in model gasp feature of state and action.
actor : 300
critic : 400→300→ 128

Future

different learning task may need different skills to change RL model