Soft Actor Critic

Used on the LunarLanderv2 Gym

Luigi Faticoso - 26 February 2019

Introduction

This report is about the work for the project of Reinforcement Learning based on the paper by Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel and Sergey Levine called Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.

I'm going to start off by saying that some of the foremost undefeated RL algorithms in recent years like Trust Region Policy optimisation (TRPO), proximal Policy optimisation (PPO) and Asynchronous Actor-Critic Agents (A3C) suffer from sample inefficiency. This is because they learn in an "on-policy" manner, they need utterly new samples when every policy update. In contrast, Q-learning based "off-policy" methods such as Deep Deterministic Policy Gradient (DDPG) and Twin Delayed Deep Deterministic Policy Gradient (TD3PG) are able to learn efficiently from past samples using experience replay buffers. However, the problem with these methods is that they are very sensitive to hyperparameters and require a lot of tuning to get them to converge. Soft Actor-Critic follows in the tradition of the latter type of algorithms and adds methods to combat the convergence brittleness.

State of the art

Soft actor-critic has been used even in non-simulation environments for applications in robotics and other domain such as the minotaur robot that learn in a really short time duration but also learns to generalize conditions that hasn't seen during training.

Environment

The environment used for the experiment is Lunar lander continuous v2 included in the box2d environments. Following is the description:

Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Action is two real values vector from -1 to +1. First controls main engine, -1..0 off, 0..+1 throttle from 50% to 100% power. Engine can't work with less than 50% power. Second value -1.0..-0.5 fire left engine, +0.5..+1.0 fire right engine, -0.5..0.5 off.

Theory behind soft actor-critic

Soft actor-critic is defined for reinforcement learning tasks involving continuous actions. The biggest feature of soft actor-crtic is that it uses a modified reinforcement learning objective function which instead of only seeking to maximize the lifetime rewards, it seeks to also maximize the entropy of the policy. Maximum entropy RL provides a robust framework that minimizes the need for hyperparameter tuning.

An algorithm is off-policy if we can reuse data collected for another task. In a typical scenario, we need to adjust parameters and shape the reward function when prototyping a new task, and use of an off-policy algorithm allows reusing the already collected data.

We want a high entropy in our policy also to explicitly encourage exploration, to encourage the policy to assign equal probabilities to actions that have same or nearly equal Q-values, and also to ensure that it does not collapse into repeatedly selecting a particular action that could exploit some inconsistency in the approximated Q function. Therefore, soft actor-critic overcomes the brittleness problem by encouraging the policy network to explore and not assign a very high probability to any one part of the range of actions.

How does soft actor-critic operates

Soft actor-critic operates by using three neural networks:

1. A state value function V parametrized by ψ.
2. A soft Q-function Q parametrized by θ.
3. A policy function π parameterized by ϕ.

Training the value network

For the Value network we want to minimise the following error:

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\frac{1}{2} \left(V_\psi(\mathbf{s}_t) - \mathbb{E}_{\mathbf{a}_t \sim \pi_\phi} \left[Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) \right] \right)^2 \right]$$

We want to decrease the squared difference between the prediction of our value network and expected prediction of the Q function plus the entropy of the policy function

Training the Q network

For the Q network we want to minimize the following error:

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \right)^2 \right]$$

Where
For all the state and action pairs the algorithm has experienced we want to minimise the

$$\hat{Q}(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \, \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[V_{\bar{\psi}}(\mathbf{s}_{t+1}) \right]$$

squared difference between the prediction of our Q function and the immediate reward plus the discounted expected Value of the next state. The Value comes with a bar that means it is the target value function.

Training the policy

For the Policy network we want to minimise the following error:
The DKL function is the Kullback-Leibler Divergence which easily explained computes

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\mathrm{D}_{\mathrm{KL}} \left(\pi_\phi(\cdot | \mathbf{s}_t) \, \middle\| \, \frac{\exp\left(Q_\theta(\mathbf{s}_t, \cdot)\right)}{Z_\theta(\mathbf{s}_t)} \right) \right]$$

how different two distributions are. So this function is basically trying to make the distribution of our Policy function look more like the distribution of the exponentiation of our Q function normalised by Z.
To minimize this objective we make use of the reparameterization trick which parameterises the policy as
Where the epsilon is a noise vector sampled from a Gaussian distribution

$$\mathbf{a}_t = f_\phi(\epsilon_t; \mathbf{s}_t)$$

Objective function

At the end we can express the objective function as

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}} \left[\log \pi_\phi(f_\phi(\epsilon_t; \mathbf{s}_t) | \mathbf{s}_t) - Q_\theta(\mathbf{s}_t, f_\phi(\epsilon_t; \mathbf{s}_t)) \right]$$

Alpha is the temperature that control the trade off between the expected return (first summand) and expected entropy (second summand)

Approach

For the implementation I have used pytorch

```
env = NormalizedActions(gym.make("LunarLanderContinuous-v2"))

action_dim = env.action_space.shape[0]
state_dim  = env.observation_space.shape[0]
```

Here we set how many features we want in our hidden layers

```
hidden_dim = 256
```

Now we initialize the three networks we want to train along with a target V network.

We are using two soft q networks to solve the problems of overestimation of Q values by using the minimum of the two networks

```
value_net        = ValueNetwork(state_dim, hidden_dim).to(device)
target_value_net = ValueNetwork(state_dim, hidden_dim).to(device)
soft_q_net1 = SoftQNetwork(state_dim, action_dim, hidden_dim).to(device)
soft_q_net2 = SoftQNetwork(state_dim, action_dim, hidden_dim).to(device)
policy_net = PolicyNetwork(state_dim, action_dim, hidden_dim).to(device)
```

We use Polyak averaging because every network depends on the others which makes the training unstable.

```
for target_param, param in zip(target_value_net.parameters(),
value_net.parameters()):
    target_param.data.copy_(param.data)
```

Minimize squared error loss function

```
value_criterion  = nn.MSELoss()
soft_q_criterion1 = nn.MSELoss()
soft_q_criterion2 = nn.MSELoss()
```

Learning rates

```
value_lr  = 3e-4
soft_q_lr = 3e-4
policy_lr = 3e-4
```

Optimizers

```
value_optimizer  = optim.Adam(value_net.parameters(), lr=value_lr)
soft_q_optimizer1 = optim.Adam(soft_q_net1.parameters(), lr=soft_q_lr)
soft_q_optimizer2 = optim.Adam(soft_q_net2.parameters(), lr=soft_q_lr)
policy_optimizer = optim.Adam(policy_net.parameters(), lr=policy_lr)
```

Here we instantiate a replay buffer which contains all the observations

```
replay_buffer_size = 1000000
replay_buffer = ReplayBuffer(replay_buffer_size)
```

Here we set the hyper parameter for the training which are the maximum frames, the maximum steps for each frame, the starting frame, an array containing the rewards and the batch size indicating the values to grab in the replay buffer to update the networks

```
max_frames  = 30000
max_steps   = 500
i   = 0
rewards     = []
batch_size  = 128

for i in tqdm(range(max_frames)):
    state = env.reset()
    episode_reward = 0
    for step in range(max_steps):
Here we have a choice of exploration or exploitation
        if i > 2000:
            action = policy_net.get_action(state).detach()
            next_state, reward, done, _ = env.step(action.numpy())
        else:
            action = env.action_space.sample()
            next_state, reward, done, _ = env.step(action)
        replay_buffer.push(state, action, reward, next_state, done)

        state = next_state
        episode_reward += reward
        i += 1
```

```
        if len(replay_buffer) > batch_size:
            update(batch_size)
        if i % 10000 == 0:
            if i > 10000:
                plot(i, rewards)
        if done:
            break
    rewards.append(episode_reward)
```

Now we see how the networks are implemented. We take the case of the policyNetwork as the other networks are pretty standard. The policy has two output: the mean and the log standard deviation. We use log standard deviations since their exponential always gives a positive number.

```
class PolicyNetwork(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size, init_w=3e-3,
log_std_min=-20, log_std_max=2):
        super(PolicyNetwork, self).__init__()

        self.log_std_min = log_std_min
        self.log_std_max = log_std_max

        self.linear1 = nn.Linear(num_inputs, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)

        self.mean_linear = nn.Linear(hidden_size, num_actions)
        self.mean_linear.weight.data.uniform_(-init_w, init_w)
        self.mean_linear.bias.data.uniform_(-init_w, init_w)
        self.log_std_linear = nn.Linear(hidden_size, num_actions)
        self.log_std_linear.weight.data.uniform_(-init_w, init_w)
        self.log_std_linear.bias.data.uniform_(-init_w, init_w)

def forward(self, state):
        x = F.relu(self.linear1(state))
        x = F.relu(self.linear2(x))

        mean    = self.mean_linear(x)
        log_std = self.log_std_linear(x)
```

Clamping of the log

```
        log_std = torch.clamp(log_std, self.log_std_min, self.log_std_max)
```

```
        return mean, log_std

    def evaluate(self, state, epsilon=1e-6):
        mean, log_std = self.forward(state)
        std = log_std.exp()

        normal = Normal(0, 1)
        z      = normal.sample()
        action = torch.tanh(mean+ std*z.to(device))
        log_prob = Normal(mean, std).log_prob(mean+ std*z.to(device)) -
torch.log(1 - action.pow(2) + epsilon)
        return action, log_prob, z, mean, log_std
```

To get the action we use the reparameterization trick

$$\tilde{a}_\theta(s,\xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \odot \xi\right), \qquad \xi \sim \mathcal{N}(0, I)$$

```
    def get_action(self, state):

        state = torch.FloatTensor(state).unsqueeze(0).to(device)
        mean, log_std = self.forward(state)
        std = log_std.exp()
```

We sample a noise from the standard normal distribution and multiply it with our standard deviation and then add the result to the mean. At the end the log probability is calculated using an approximate of the log likelihood of tanh(mean + std * z)

```
        normal = Normal(0, 1)
        z      = normal.sample().to(device)
        action = torch.tanh(mean + std*z)
        action  = action.cpu()
        return action[0]
```

My results

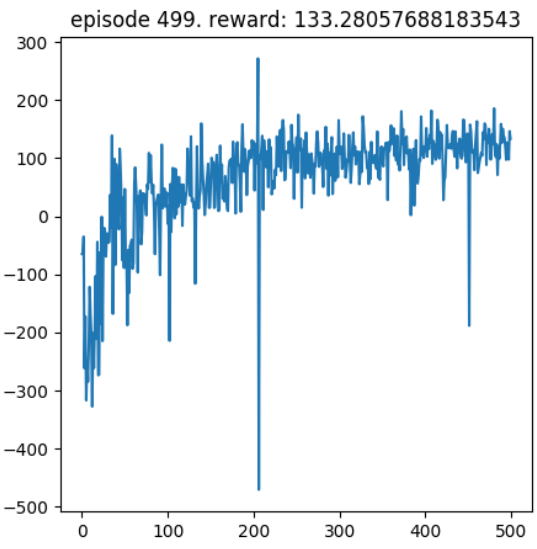
This is my first result using the following parameters:

Episodes	500
Max steps per episode	500
Batch size	128
Value learning rate	0,003
Soft q learning rate	0,003
Policy learning rate	0,003
Exploration stopped after	1000 steps

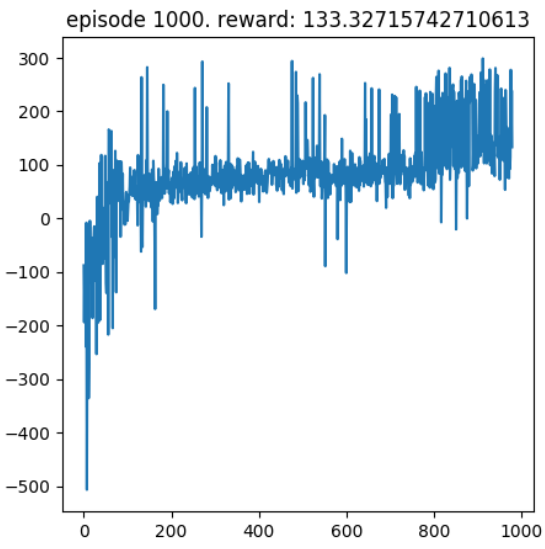
And those are my second experiment results tuning the parameter to have more exploration

Episodes	1000
Max steps per episode	500
Batch size	128
Value learning rate	0,003
Soft q learning rate	0,003
Policy learning rate	0,003
Exploration stopped after	1500 steps

1st experiment



2nd experiment



Conclusions

As further improvements, there is a new version of the algorithm that uses only a Q function and disposes of the V function, it also adds automatic discovery of the weight of the entropy term called temperature. Since it is extremely new I haven't found any source code to understand and to implement it. As a conclusion I can say that soft actor-critic is great algorithm that can be applied most of all to learn robot skills in challenging real-world settings. Since the policies are learned directly in the real world, they exhibit robustness to variations in the environment, which can be difficult to obtain otherwise. There is still some work to be done to adopt deep RL for more complex real-world tasks in the future.