CSE 525 Project Report

Yuchen Ma 112877697

Worst Case Policy Gradients

Yichuan Charlie Tang
Apple Inc.
yichuan tang@apple.com

Jian Zhang
Apple Inc.
jianz@apple.com

Ruslan Salakhutdinov
Apple Inc.
rsalakhutdinov@apple.com

- **Overview**

  In this report, I will mainly show the gist of the paper "Worst Case Policy Gradients" and the details about my implementation. The motivation part is intended to explain why the authors proposed this method and how this new network will tackle the existing problem. In the theory method part, I will show the algorithm and conceptual method that the paper brings in. In the third part, I will show the details on how I implement the method regarding the two primary parts (the update for Actor and Critic). The core codes will be shown as well. Unfortunately, I haven't made any progress on the experiment part, but I will still show some of the training results. At last, I will show my plan and the new techniques I want to try in the future.

- **Motivation**

  A policy giving actions with large rewards is considered as a satisfying policy. However, when learning policies for safety critical applications (such as driving), it is important to be sensitive to risks and avoid catastrophic events. To avoid the high-risk events, the authors propose an actor-critic framework which models the uncertainty of the future and learns a policy based on that uncertainty model. Specifically, given a distribution of the future return for any state and action, the method optimizes policies for varying levels of conditional Value-at-Risk. The learned policy can map the same state to different actions depending on the propensity for risk.

- **Theory Method**

  The method that the authors bring up is an extension of DDPG(Deep Deterministic Policy Gradients). They improve two main parts about the original method: modeling the distribution of rewards instead of the expectation, using CAaR as objective in the Actor. I will explain these two improvements in the following parts.

  - Distributional Critic

  As the motivation shows, the authors desire the stable and low-risk policies, which can be the low variance policies even with lower expectations. That requires to construct the distribution of the rewards with respect to the state and action pairs. In this way, the variances can be detected.

To learn the critic, the equivalent Bellman operator for distributions needs to be defined. Firstly, the distribution over total future return (which is gained by perform a policy until the terminal state is reached) is defined: $Z(s,a) = p^{\pi}(R \mid s,a)$, in which p^{π} is the transition operator. Then, the distribution $Z(s,a)$ is modeled as a Gaussian distribution and it is approximated to 2 statistics: Mean $\widehat{Q}^{\pi}(s,a)$ and Variance $\widehat{\mathrm{Y}}^{\pi}(s,a)$. The authors conduct close proof on the feasibility of the modeling, which is rather complex and I will not show this part. However, the final projection equation is essential. The projection for the Q^{π} is similar as in Q-learning:

$$Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{s'} p(s' \mid s,a)\, Q^{\pi}(s',a'). \quad \textbf{(1)}$$

The projection for the $\mathrm{Y}(s,a)$ is defined as:

$$\mathrm{Y}^{\pi}(s,a) = r(s,a)^2 + 2\gamma r(s.a) \sum_{s'} p(s' \mid s,a)\, Q^{\pi}(s',a') + \gamma^2 \sum_{s'} p(s' \mid s,a) \mathrm{Y}^{\pi}(s',a')$$

$$+ \gamma^2 \sum_{s'} p(s' \mid s,a)\, Q^{\pi}(s',a') - Q^{\pi}(s,a)^{\,2} \quad \textbf{(2)}$$

By using a network to approximate these two values, the mean and variance of the distribution is obtained. Then, the critic is learned by using TD (Temporal Difference) learning. Now, the loss function needs to be defined. The authors give two suggestions on choosing the metric. One is the KL-divergence and the other is the p-th Wasserstein metric. Note that the p-th Wasserstein metric is used to measure the distance between two distributions. I use the 2-Wasserstein metric.

- CVaR Actor

In this part, the novel criterion CVaR is proposed. As for risk-averse learning, it is desirable to minimize the expected worst cases performance rather than the average cases performance. The CVaR represents the expected return should we experience the bottom α-percentile of the possible outcomes. The α-percentile of distribution is captured by:

$$CVaR_{\alpha} = E_{p^{\pi}}[R \mid R \le pcntl(\alpha)]$$

where $pcntl(\alpha)$ is the α-percentile of p^{π} and the policy will focus on performing well in worse cases as $\alpha \to 0$.

Remembering that we get the mean and variance from the critic, now the CVaR can be computed using these two:

$$CVaR_{\alpha} = Q^{\pi}(s,a) - (\phi(\alpha)/\Phi(\alpha))\sqrt{\mathrm{Y}^{\pi}(s,a)}$$

Where $\phi(\cdot)$ is standard normal distribution and $\Phi(\cdot)$ is its CDF. Now, for the objective CVaR, the equivalent deterministic policy gradient for the actor network can be computed in this way:

$$\nabla_{\theta} J_{\alpha} = \mathbb{E}_{s\sim\rho,\alpha}[\nabla_{\theta}\pi(a|s,\alpha)\nabla_{a}\Gamma^{\pi}(s,a,\alpha)]$$

$$= \mathbb{E}_{s\sim\rho,\alpha}\left[\nabla_{\theta}\pi(a|s,\alpha)\nabla_{a}\hat{Q}^{\pi}(s,a,\alpha) - (\phi(\alpha)/\Phi(\alpha))\nabla_{\theta}\pi(a|s,\alpha)\nabla_{a}\sqrt{\hat{\Upsilon}^{\pi}(s,a,\alpha)}\right]. \quad \textbf{(3)}$$

- Algorithm

  Using the equation **(1)**, **(2)** and **(3)**, the WCPG training algorithm can be performed. The algorithm proposed by the authors is as follows:

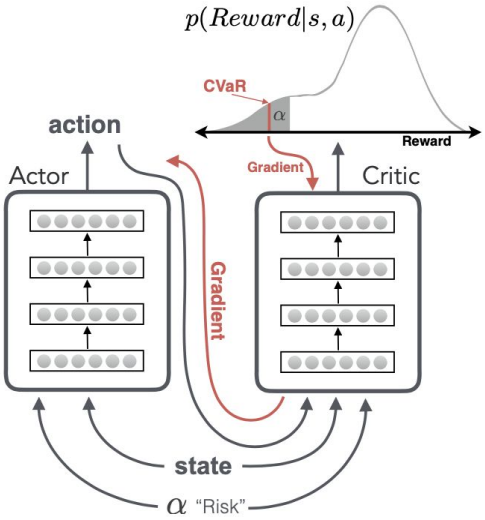
Algorithm 1: WCPG training algorithm

Input: Environment $\mathcal{E}$, actor-critic net hyperparameters, M episodes, T steps/episode, replay buffer $\mathcal{B}$.
Initialize: Randomly initialize actor π and critic, $\mathcal{B} \leftarrow \emptyset$.

for *episodes* $e = 0$ *to* M **do**
  $s_0 \leftarrow \mathcal{E}$.reset(), $\alpha \sim U(0, 1)$
  for *steps* $t = 0$ *to* T **do**
    action $a_t \leftarrow \pi_\alpha(s_t)$; add exploration noise.
    $\{s_{t+1}, r_t, done\} \leftarrow \mathcal{E}$.step$(a_t)$
    $\mathcal{B} \leftarrow \{s_t, a_t, r_t, s_{t+1}, \alpha\}$
    Randomly sample a minibatch from $\mathcal{B}$.
    Update critic
    Update actor with the CVaR objective
    if *done* **then** goto next episode.
  end
end

- **Implementation and Core Codes**

Firstly, I will show the architecture of the whole network:



The network is an actor-critic based on DDPG with two modifications.

The first is that an additional α CVaR input is added to both the actor and the critic.

The second is that instead of producing a scalar value for approximating $Q^\pi(s, a)$, the critic outputs parameters which govern the distribution of future returns.

- Actor net

  The size input of the Actor net is (*the size of the states* + 1), and the extra one input is the CVaR. The generation of the CVaR will be shown in the DDPG agent part. The size of output of the Actor is (*the size of the actions*).

```
class Actor(nn.Module):
   def __init__(self, nb_states, nb_actions, hidden1=400, hidden2=300,
init_w=3e-3):
       super(Actor, self).__init__()
       self.fc1 = nn.Linear(nb_states+1, hidden1)
       self.fc2 = nn.Linear(hidden1, hidden2)
       self.fc3 = nn.Linear(hidden2, nb_actions)
       self.relu = nn.ReLU()
       self.tanh = nn.Tanh()
       self.init_weights(init_w)

   def init_weights(self, init_w):
           … …

   def forward(self, x):
            … …
    out = self.tanh(out)
       return out
```

- Critic net

  The Critic net is parameterized by ω and is used for estimating the critic's mean and variance: $f_{critic}(s,a|\omega) \rightarrow \{\widehat{Q}^{\pi}(s,a), \widehat{\mathrm{Y}}^{\pi}(s,a)\}$. Thus, the size input of the Critic net is (*the size of the states*), and the size of output is 2, which represents for the estimated Mean and Variance of the distribution of future reward.

```
class Critic(nn.Module):
   def __init__(self, nb_states, nb_actions, hidden1=400, hidden2=300,
init_w=3e-3):
       super(Critic, self).__init__()
       self.fc1 = nn.Linear(nb_states, hidden1)
       self.fc2 = nn.Linear(hidden1 + nb_actions, hidden2)
       self.fc3 = nn.Linear(hidden2, 2)
       self.relu = nn.ReLU()
       self.init_weights(init_w)

   def init_weights(self, init_w):
               … …

   def forward(self, xs):
       x, a = xs
       out = self.fc1(x.float())
       out = self.relu(out)
       out = self.fc2(torch.cat([out, a], 1))
       out = self.relu(out)
       out = torch.sigmoid(self.fc3(out))

       q_values = out[:,0].unsqueeze(-1)
       variance_values = out[:,1].unsqueeze(-1)

       return q_values, variance_values
```

- DDPG agent

Intuitively, a different input α will have a different loss function. To train for all αs, we need uniformly sample α ~ Uniform(0, 1) during the *start* of an episode and fix α for the entirety of that episode. Here I write a method called "*update_alpha*" to achieve this.

In the class of DDPG, I write two methods for *action_selection*: random and ε*-greedy*. Similar to the DPG, I implement the target network and replay buffer. In the "*upadate_policy*", I use the method proposed by the authors to compute the loss.

```
class DDPG(object):
   def __init__(self, nb_states, nb_actions):

       self.actor = Actor(self.nb_states, self.nb_actions)
       self.actor_target = Actor(self.nb_states, self.nb_actions)
       self.actor_optim = Adam(self.actor.parameters(), lr=prate)

       self.critic = Critic(self.nb_states, self.nb_actions)
       self.critic_target = Critic(self.nb_states, self.nb_actions)
       self.critic_optim = Adam(self.critic.parameters(), lr=rate)

       # Create replay buffer
       self.memory = SequentialMemory()
       self.random_process = OrnsteinUhlenbeckProcess()

   def update_alpha(self):
       self.alpha = np.random.uniform(0.0, 1.0, size=1)[0]

   def update_policy(self):

       self.memory.sample_and_split(self.batch_size)

       # Critic update
       next_q_values, next_variance_values = self.critic_target()

      # refers to the equation (1)
       target_q_batch = to_tensor(reward_batch) + \
         self.discount * to_tensor(terminal_batch.astype(np.float)) * next_q_values

       # refers to the equation (2)
       target_variance_batch = to_tensor(reward_batch) * to_tensor(reward_batch)\
                             + 2 * self.discount * to_tensor(reward_batch)*     \
                              to_tensor(terminal_batch.astype(np.float)) *      \
                              next_q_values + self.discount * self.discount *   \
             next_variance_values * to_tensor(terminal_batch.astype(np.float))+ \
                                       self.discount * self.discount *          \
   to_tensor(terminal_batch.astype(np.float)) * next_q_values * next_q_values - \
                      target_q_batch * target_q_batch



       q_batch , variance_batch = self.critic()

       dist_current = generate_gaussian(q_batch, variance_batch)
       dist_future = generate_gaussian(target_q_batch, target_variance_batch)

       value_loss = caculate_loss(dist_current, dist_future)
       value_loss = value_loss.mean()
       value_loss.backward()


       # Actor update
```

```
        std_term = norm.pdf(self.alpha) / norm.cdf(self.alpha)


        state_batch_a = np.hstack((state_batch, temp))
        q_est, v_est = self.critic()

        # refers to the equation (3)
        policy_loss = -(q_est - to_tensor(np.array([std_term])) * torch.sqrt(v_est))
        policy_loss = policy_loss.mean()
        policy_loss.backward()

        # Target update
        soft_update(self.actor_target, self.actor, self.tau)
        soft_update(self.critic_target, self.critic, self.tau)


    def random_action(self):

    def select_action(self, s_t, decay_epsilon=True):

    def eval(self):

    def observe(self, r_t, s_t1, done):

    def reset(self, obs):

    def load_model(self, output):

    def save_model(self,num):
```

- Distributional Critic

The idea to update the Critic from the paper is :

I. Obtain the mean and the variance of the distribution of the future reward from the neutral network.
II. Use the two values to generate the gaussian distribution.
III. Compute the 2-Wasserstein distance and the critic will try to minimize the distance by back propagating the gradient.

```
def generate_gaussian(mean, variance):
    np.random.normal(m_temp, v_temp, 10000))))
    return to_tensor(dist)

def caculate_loss(dist1, dist2):
    stats.wasserstein_distance(dist_1, dist_2)
    return to_tensor(diff).unsqueeze(-1)

dist_current = generate_gaussian(q_batch, variance_batch)
dist_future = generate_gaussian(target_q_batch, target_variance_batch)
value_loss = caculate_loss(dist_current, dist_future)
```

- **Experiment**

  I try to train the model in the *gym* environment `"Pendulum-v0"`, which is with continuous action and state space. Due to the high computation and the lack of training techniques, I failed to train the agent well. At last, I set the *total_episode_number* to 1000 and the *max_steps* in each episode to 500. The result doesn't show any convergence. The *episode_rewards* is basically the same as the result from the random policy.

- **To Do**

  Firstly, I plan to finish the training on the `"Pendulum-v0"`. In addition, considering both the WCPG algorithm and D4PG are distributional reinforcement learning methods, I want to train the D4PG agent in the same environment and compare the performance of both agents. In addition, I really want to try this algorithm on self-driving simulated environments. I plan to find more scenarios where the prevention of the worst cases is crucial.