REINFORCE

Paper Link: https://people.cs.umass.edu/~barto/courses/cs687/williams92simple.pdf

Key Features

- REINFORCE = MonteCarlo Policy Gradient Method
- Sample derived from REINFORCE has high variance
- REINFORCE is On-Policy Method

Background

MonteCarlo Policy Gradient Method

In applying the Policy Gradient, there is a problem that the expected value of return actually cannot be calculated directly. Therefore, REINFORCE algorithm use a method of extracting samples in units of episodes to estimate expected value of return and strengthening policies based on these estimates . The following shows the objective function of REINFORCE.

$$E_{\tau \sim p_\theta(\tau)}[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)] \approx \frac{1}{M} \sum_{m=1}^{M} [\sum_{t=0}^{T} \gamma^t r(s_t, a_t)]$$

Based on the above equation, the gradient of the objective function can be estimated approximately as follows.

$$\nabla_\theta J(\theta) \approx \frac{1}{M} \sum_{m=1}^{M} [\sum_{t=0}^{T} (\nabla_\theta log \pi_\theta (s_t^{(m)}, a_t^{(m)}) G_t^{(m)})] \tag{1}$$

$$= \nabla_\theta \frac{1}{M} \sum_{m=1}^{M} [\sum_{t=0}^{T} (log \pi_\theta (s_t^{(m)}, a_t^{(m)}) G_t^{(m)})] \tag{2}$$

Method

Loss Function of REINFORCE

The episode loss function is configured the policy π represented by θ. The $Loss$ is as follows.

$$Loss = -\sum_{t=0}^{T} (log \pi_\theta (s_t^{(m)}, a_t^{(m)}) G_t^{(m)})$$

This is an inversion of the derivative of the objective function. This expresses the inversely proportional nature of the loss decreasing as the value of the objective function expressed as the expectation of the Return G value increases.

It can be seen that this is a form obtained by multiplying the return for the cross-entropy equation in which the ratio of target is always 1. Therefore, as the size of the G increases, it affects the π more, and as the size of the G decreases, it affects the π less.

Algorithm

$$
\begin{aligned}
&function\ REINFORCE && (3)\\
&\quad Initialize\ \theta\ arbirarily && (4)\\
&\quad for\ each\ episode\ [s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T] \sim \pi_\theta\ do && (5)\\
&\quad\quad for\ t = 1\ to\ T-1\ do && (6)\\
&\quad\quad\quad \theta \leftarrow \theta + \alpha \nabla_\theta log \pi_\theta(s_t, a_t) G_t && (7)\\
&\quad\quad end\ for && (8)\\
&\quad end\ for && (9)\\
&\quad return\ \theta && (10)\\
&end\ function && (11)
\end{aligned}
$$

$$
\begin{aligned}
&function\ REINFORCE && (12)\\
&\quad Initialize\ \theta\ arbirarily && (13)\\
&\quad for\ each\ episode\ [s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T] \sim \pi_\theta\ do && (14)\\
&\quad\quad for\ t = 1\ to\ T-1\ do && (15)\\
&\quad\quad\quad \theta \leftarrow \theta + \alpha \nabla_\theta log \pi_\theta(s_t, a_t) G_t && (16)\\
&\quad\quad end\ for && (17)\\
&\quad end\ for && (18)\\
&\quad return\ \theta && (19)\\
&end\ function && (20)
\end{aligned}
$$

limitations

- **REINFORCE updates π based on sample of episode unit**
  In case of long episodes, neural network training time is prolonged.
- **The variance of REINFORCE's learning gradient is large**
  Due to the large variance of the G depending on the length of episode, neural network learning may be unstable overall and the training time may be longer.
- **REINFORCE algorithm is On-Policy algorithm**
  Since REINFORCE is On-Policy algorithm, which executes π to collect samples and

updates based on the obtained samples, bias occurs in samples. In addition, since the number of sample used for learning is small, data efficiency is poor.

Implementation on JORLDY

- REINFORCE JORLDY Implementation

```
### learning process function ###
# 1. save transition
# 2. learn
# 3. learning rate decay

def process(self, transitions, step):
        result = {}
        # Process per step
        self.memory.store(transitions)

        # Process per epi
        if transitions[0]["done"]:
            result = self.learn()
            self.learning_rate_decay(step)

        return result
```

```
### learn function ###
# 1. extract transitions
# 2. calculate loss
# 3. update
# 4. return result

def learn(self):
        transitions = self.memory.sample()

        state = transitions["state"]
        action = transitions["action"]
        reward = transitions["reward"]

        ret = np.copy(reward)
        for t in reversed(range(len(ret) - 1)):
            ret[t] += self.gamma * ret[t + 1]
        if self.use_standardization:
            ret = (ret - ret.mean()) / (ret.std() + 1e-7)

        state, action, ret = map(lambda x: self.as_tensor(x), [state, action, ret])
```

```
        if self.action_type == "continuous":
            mu, std = self.network(state)
            m = Normal(mu, std)
            z = torch.atanh(torch.clamp(action, -1 + 1e-7, 1 - 1e-7))
            log_prob = m.log_prob(z)
        else:
            pi = self.network(state)
            log_prob = torch.log(pi.gather(1, action.long()))
        loss = -(log_prob * ret).mean()

        self.optimizer.zero_grad(set_to_none=True)
        loss.backward()
        self.optimizer.step()

        result = {"loss": loss.item()}
        return result
```

References

Relevant papers

- Policy Gradient Methods For Reinforcement Learning With Approximation
  (R. S. Sutton et al, 1999)