Rainbow DQN

Paper Link: Rainbow: Combining Improvements in Deep Reinforcement Learning

Key Features

- Rainbow DQN is an value based off-policy RL algorithm.
- Rainbow DQN applies six DQN variants (Double Q-learning, Prioritized replay, Dueling network, Multi-step, Distributional RL, Noisy Nets).
- Rainbow DQN shows better performance than the 7 algorithms which consist of Rainbow DQN.

Background

Deep Reinforcement Learning and DQN

In DQN, deep neural networks and reinforcement learning are successfully combined by using a convolutional neural network to approximate the action values for a given state S_t. At each step, based on the current state, the agent selects an action using ε-greedy. According to the action, agent conducts the action in the environment and adds a transition $(S_t, A_t, R_{t+1}, \gamma_{t+1}, S_{t+1})$ to a replay memory buffer. The parameters of the neural network are optimized by using stochastic gradient descent to minimize the following loss

$$\left(R_{t+1} + \gamma_{t+1} \max_{a'} Q_{\bar{\theta}}(S_{t+1}, a') - Q_{\theta}(S_t, A_t)\right)^2. \quad (1)$$

The gradient of the loss is back-propagated only into the parameters θ of the online network. The term $\bar{\theta}$ represents the parameters of a target network. The target network has same structure with online network. However, it is not directly optimized, so it

periodically copies the parameters of the online network. The use of experience replay and target network enables relatively stable learning of Q values.

Double Q-learning

Conventional Q-learning is affected by an overestimation bias due to the maximization step in Equation(1). It can harm the performance of training. Double Q-learning, addresses this overestimation by decoupling. In the maximization performed for the bootstrap target, the action selection is decoupled from evaluation with two different networks of DQN (online network, target network). For the Double DQN, the online network is used for action selection and the target network is used for evaluation. The loss of Double DQN is as follows.

$$\left(R_{t+1} + \gamma_{t+1} Q_{\bar{\theta}}(S_{t+1}, \arg\max_{a'} Q_\theta(S_{t+1}, a')) - Q_\theta(S_t, A_t)\right)^2.$$

This change reduces harmful overestimation of DQN.

Prioritized experience replay

DQN samples the transition data uniformly from the replay buffer. But ideally, it's efficient to sample more frequently those transitions from that there is much to learn. For efficient sampling, prioritized experience replay samples transitions with probability which is proportional to the absolute TD error:

$$p_t \propto \left| R_{t+1} + \gamma_{t+1} \max_{a'} Q_{\bar{\theta}}(S_{t+1}, a') - Q_\theta(S_t, A_t) \right|^w,$$

where ω is a hyperparameter that determines the shape of the distribution. New transitions are inserted into the replay buffer with maximum priority, providing a bias towards recent transitions.

Dueling network

The dueling network is a neural network architecture designed for value based RL. It estimates the state-action value function Q using two streams which consists of the

state value $V(s)$ and action advantage functions $A(s,a)$. The dueling network has one common convolution network and separates fully connected layers for obtaining state value and action advantage, respectively. The advantage function subtracts the value of the state from the Q function to obtain a relative measure of the importance of each action. $A(s,a) = Q(s,a) - V(s)$. However, dueling network uses average value of the action advantage as a value of the state. Therefore, the equation for state-action value Q is as follows.

$$Q_\theta(s,a) = V\eta\big(f_\xi(s)\big) + A_\phi\big(f_\xi(s),a\big) - \frac{\sum_{a'} A_\phi\big(f_\xi(s),a'\big)}{N_{actions}},$$

where ξ, η, and ϕ are, respectively, the parameters of the shared encoder f_ξ , the value stream $V\eta$, and the advantage stream a_ϕ. $\theta = \{\xi, \eta, \phi\}$ is their concatenation.

Multi-step learning

Generally, Q-learning accumulates a single reward and the maximum state-action value of the next step to bootstrap. As an alternative, forward-view multi-step can be used. multistep defines the truncated n-step return from a given state S_t as follows.

$$R_t^n \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}$$

A multi-step variant of DQN is then defined by minimizing the alternative loss,

$$\big(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} Q_{\bar\theta}(S_{t+n},a') - Q_\theta(S_t,A_t)\big)^2.$$

If the hyperparameter n is tuned appropriately for multi-step target, it often leads to faster learning.

Distributional RL

The conventional RL algorithms usually learn Q-value as a scalar. However, Distributional RL is a reinforcement learning mechanism that learns Q-value as a

distribution. To learn the probability masses, this algorithm proposes a distributional Bellman equation as follows.

$$Z(S_t, A_t) \overset{D}{=} R_{t+1} + \gamma Z(S_{t+1}, A_{t+1})$$

For a given state S_t and action A_t, the distribution of the returns under the optimal policy π^* should match a target distribution which is defined by taking the distribution for the next state S_{t+1} and action $a^*_{t+1} = \pi^*(S_{t+1})$. Then the target distribution is contracted towards zero according to the discount, and shifted by the reward. A distributional variant of Q-learning is derived by constructing a new support for the target distribution, and then minimizing the Kullbeck-Leibler divergence between the distribution d_t and the target distribution

$$d'_t \equiv \big(R_{t+1} + \gamma_{t+1} z,\ p_{\bar{\theta}}(S_{t+1}, \bar{a}^*_{t+1})\big),\ D_{KL}(\Phi_z d\prime_t || d_t). \tag{2}$$

where Φ_z is a L2-projection of the target distribution onto the fixed support z, and $\bar{a}^*_{t+1} = \arg\max_a Q_{\bar{\theta}}(S_{t+1}, a)$ is the greedy action with respect to the mean action values $Q_{\bar{\theta}}(S_{t+1}, a) = z \top p_\theta(S_{t+1}, a)$ in state S_{t+1}.

Noisy Nets

The limitations of exploring using ε-greedy policies are clear in games such as Montezuma's Revenge, where many actions must be executed to collect the first positive reward. Noisy Nets propose a noisy linear layer that combines a deterministic and noisy stream,

$$y = (b + Wx) + (b_{noisy} \odot \varepsilon^b + (W_{noisy} \odot \varepsilon^w)x),$$

where ε^b and ε^w are random variables, and $\odot$ denotes the element-wise product. This transformation can be used in place of the standard linear $y = b + Wx$. Over time, the network is trained to ignore the noisy stream, but it is occurred at different rates in different parts of the state space, allowing state-conditional exploration with a form of self-annealing.

Method

Rainbow DQN integrates all the aforementioned components into a single integrated agent.

First, the paper replaces the 1-step distributional loss Equation (2) with a multi-step variant. Construct the target distribution by contracting the value distribution in S_{t+n} according to the cumulative discount and shifting it by the truncated n-step discounted return. Therefore, the target distribution can be defined as follows

$$d_t^{(n)} = \left(R_t^{(n)} + \gamma_t^{(n)} z,\ p_{\bar{\theta}}(S_{t+n}, a_{t+n}^*)\right).$$

The loss is

$$D_{KL}(\Phi_z d_t^{(n)} || d_t).$$

Second, the paper combines the multi-step distributional loss with double Q-learning by using the greedy action in s_{t+n} selected according to the online network as the bootstrap action a_{t+n}, and evaluating such action using the target network.

In the standard proportional prioritized replay, the absolute TD error is used to prioritize the transitions. However, in the paper, all distributional Rainbow variants prioritize the transitions by the KL loss instead of the TD error. Therefore, the priority can be defined as follows:

$$p_t \propto \left(D_{KL}(\Phi_z d_t^{(n)} || d_t\right)^w.$$

Finally, the network architecture is a dueling network architecture adapted to use with return distributions. The network has a shared representation $f_\xi(s)$, which is fed into a value stream $V\eta$ with N_{atoms} outputs, and advantage stream A_ξ with N_{atoms} × $N_{actions}$ outputs, where $A_\xi^i(f_\xi(s), a)$ denotes the output corresponding to atom i and action a. The value and advantage streams are aggregated, as in dueling DQN, and then passed through a softmax layer to obtain the normalized parametric distributions used to estimate the distributions of the returns:

$$p_\theta^i(s, a) = \frac{\exp\left(V_\eta^i(\phi) + A_\psi^i(\phi, a) - \bar{A}_\psi^i(s)\right)}{\sum_j \exp\left(V_\eta^j(\phi) + A_\psi^j(\phi, a) - \bar{A}_\psi^j(s)\right)},$$

where $\varphi = f_\xi(s)$ and $\bar{A}^i_\psi(s) = \frac{1}{N_{actions}} \sum_{a'} \bar{A}^i_\psi(\phi, a')$.

Implementation on JORLDY

- Rainbow DQN JORLDY Implementation
- Rainbow DQN agent init method

```
def __init__( ... )

    ...

    # MultiStep
    self.n_step = n_step
    self.tmp_buffer = deque(maxlen=n_step)

    # PER
    self.alpha = alpha
    self.beta = beta
    self.learn_period = learn_period
    self.learn_period_stamp = 0
    self.uniform_sample_prob = uniform_sample_prob
    self.beta_add = (1 - beta) / run_step

    # C51
    self.v_min = v_min
    self.v_max = v_max
    self.num_support = num_support

    # MultiStep
    self.memory = PERBuffer(buffer_size, uniform_sample_prob)

    # C51
    self.delta_z = (v_max - v_min) / (num_support - 1)
    self.z = torch.linspace(v_min, v_max, num_support, device=self.device).view(
        1, -1
    )
```

- Rainbow DQN agent learn method

```
def learn(self):
        transitions, weights, indices, sampled_p, mean_p = self.memory.sample(
            self.beta, self.batch_size
        )
        for key in transitions.keys():
```

```
            transitions[key] = self.as_tensor(transitions[key])

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

        logit = self.network(state, True)
        p_logit, q_action = self.logits2Q(logit)

        action_eye = torch.eye(self.action_size).to(self.device)
        action_onehot = action_eye[action.long()]

        p_action = torch.squeeze(action_onehot @ p_logit, 1)

        target_dist = torch.zeros(
            self.batch_size, self.num_support, device=self.device, requires_grad=False
        )
        with torch.no_grad():
            # Double
            _, next_q_action = self.logits2Q(self.network(next_state, True))

            target_p_logit, _ = self.logits2Q(self.target_network(next_state, True))

            target_action = torch.argmax(next_q_action, -1, keepdim=True)
            target_action_onehot = action_eye[target_action.long()]
            target_p_action = torch.squeeze(target_action_onehot @ target_p_logit, 1)

            Tz = self.z
            for i in reversed(range(self.n_step)):
                Tz = (
                    reward[:, i].expand(-1, self.num_support)
                    + (1 - done[:, i]) * self.gamma * Tz
                )

            b = torch.clamp(Tz - self.v_min, 0, self.v_max - self.v_min) / self.delta_z
            l = torch.floor(b).long()
            u = torch.ceil(b).long()

            support_eye = torch.eye(self.num_support, device=self.device)
            l_support_onehot = support_eye[l]
            u_support_onehot = support_eye[u]

            l_support_binary = torch.unsqueeze(u - b, -1)
            u_support_binary = torch.unsqueeze(b - l, -1)
            target_p_action_binary = torch.unsqueeze(target_p_action, -1)

            lluu = (
                l_support_onehot * l_support_binary
                + u_support_onehot * u_support_binary
            )

            target_dist += done[:, 0, :] * torch.mean(
```

```
                l_support_onehot * u_support_onehot + lluu, 1
            )
            target_dist += (1 - done[:, 0, :]) * torch.sum(
                target_p_action_binary * lluu, 1
            )
            target_dist /= torch.clamp(
                torch.sum(target_dist, 1, keepdim=True), min=1e-8
            )

        max_Q = torch.max(q_action).item()
        max_logit = torch.max(logit).item()
        min_logit = torch.min(logit).item()

        # PER
        KL = -(target_dist * torch.clamp(p_action, min=1e-8).log()).sum(-1)
        p_j = torch.pow(KL, self.alpha)

        for i, p in zip(indices, p_j):
            self.memory.update_priority(p.item(), i)

        weights = torch.unsqueeze(torch.FloatTensor(weights).to(self.device), -1)

        loss = (weights * KL).mean()

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

        self.num_learn += 1

        result = {
            "loss": loss.item(),
            "beta": self.beta,
            "max_Q": max_Q,
            "max_logit": max_logit,
            "min_logit": min_logit,
            "sampled_p": sampled_p,
            "mean_p": mean_p,
        }

        return result
```

- Rainbow DQN agent interact_callback method

```
# multi-step prcoess

def interact_callback(self, transition):
        _transition = {}
        self.tmp_buffer.append(transition)
        if len(self.tmp_buffer) == self.n_step:
            _transition["state"] = self.tmp_buffer[0]["state"]
```

```
            _transition["action"] = self.tmp_buffer[0]["action"]
            _transition["next_state"] = self.tmp_buffer[-1]["next_state"]

            for key in self.tmp_buffer[0].keys():
                if key not in ["state", "action", "next_state"]:
                    _transition[key] = np.stack(
                        [t[key] for t in self.tmp_buffer], axis=1
                    )

        return _transition
```

- Rainbow DQN network rainbow.py

```
class Rainbow(BaseNetwork):
    def __init__(
        self, D_in, D_out, N_atom, noise_type="factorized", D_hidden=512, head="mlp"
    ):
        D_head_out = super(Rainbow, self).__init__(D_in, D_hidden, head)
        self.D_out = D_out
        self.N_atom = N_atom
        self.noise_type = noise_type

        self.l = torch.nn.Linear(D_head_out, D_hidden)

        self.mu_w_a1, self.sig_w_a1, self.mu_b_a1, self.sig_b_a1 = init_weights(
            (D_hidden, D_hidden), noise_type
        )
        self.mu_w_v1, self.sig_w_v1, self.mu_b_v1, self.sig_b_v1 = init_weights(
            (D_hidden, D_hidden), noise_type
        )

        self.mu_w_a2, self.sig_w_a2, self.mu_b_a2, self.sig_b_a2 = init_weights(
            (D_hidden, N_atom * self.D_out), noise_type
        )
        self.mu_w_v2, self.sig_w_v2, self.mu_b_v2, self.sig_b_v2 = init_weights(
            (D_hidden, N_atom), noise_type
        )

        orthogonal_init(self.l)

    def forward(self, x, is_train):
        x = super(Rainbow, self).forward(x)

        x = F.relu(self.l(x))

        x_a = F.relu(
            noisy_l(
                x,
                self.mu_w_a1,
                self.sig_w_a1,
                self.mu_b_a1,
```

```
                self.sig_b_a1,
                self.noise_type,
                is_train,
            )
        )
        x_v = F.relu(
            noisy_l(
                x,
                self.mu_w_v1,
                self.sig_w_v1,
                self.mu_b_v1,
                self.sig_b_v1,
                self.noise_type,
                is_train,
            )
        )

        # A stream : action advantage
        x_a = noisy_l(
            x_a,
            self.mu_w_a2,
            self.sig_w_a2,
            self.mu_b_a2,
            self.sig_b_a2,
            self.noise_type,
            is_train,
        )  # [bs, num_action * N_atom]
        x_a = torch.reshape(
            x_a, (-1, self.D_out, self.N_atom)
        )  # [bs, num_action, N_atom]
        x_a_mean = x_a.mean(dim=1).unsqueeze(1)  # [bs, 1, N_atom]
        x_a = x_a - x_a_mean.repeat(1, self.D_out, 1)  # [bs, num_action, N_atom]

        # V stream : state value
        x_v = noisy_l(
            x_v,
            self.mu_w_v2,
            self.sig_w_v2,
            self.mu_b_v2,
            self.sig_b_v2,
            self.noise_type,
            is_train,
        )  # [bs, N_atom]
        x_v = torch.reshape(x_v, (-1, 1, self.N_atom))  # [bs, 1, N_atom]
        x_v = x_v.repeat(1, self.D_out, 1)  # [bs, num_action, N_atom]

        out = x_a + x_v  # [bs, num_action, N_atom]

        return out
```

References

Relevant papers

- Human-level control through deep reinforcement learning
- Double Q-learning
- Prioritized Experience Replay
- Dueling Network Architectures for Deep Reinforcement Learning
- Learning to predict by the methods of temporal differences
- A Distributional Perspective on Reinforcement Learning
- Noisy Networks for Exploration