PER(Prioritized Experience Replay)

Paper Link: Prioritized Experience Replay

Key Features

- PER uses priorities to make experience replay more efficient than replay memory which samples transitions uniformly.
- PER introduces a stochastic sampling method that interpolates between greedy prioritization and uniform random sampling.
- PER uses importance sampling to correct the bias that occurs in prioritized replay.

Background

There were two problems with the existing online reinforcement learning agents that incrementally update parameters while they observe a stream of experience. First, it break the i.i.d. assumption of many popular stochastic gradient-based algorithms. The second problem is rapid forgetting of possibly rare experiences that would be useful later on.

Experience replay(Lin, 1992) addresses both issues. With experience stored in a replay memory, it is possible to break the temporal correlations by mixing old and new experience for the updates. Also, the rare experience is used for more than just a single update. Prioritized Experience Replay introduces stochastic sampling which uses priorities for the transitions of experience replay. Therefore, it is more efficient than uniform random sampling.

Method

Prioritizing with TD-error

The central component of prioritized replay is that the importance of each transition can be measured. The importance would be the amount the RL agent learns from the specific transition in replay memory, but it cannot be directly obtained. Instead, it is measured by the magnitude of a transition's TD error δ, which indicates how 'surprise' or 'unexpected' the transition is. TD error is calculated as how far the value is from the next-step bootstrap estimate. For the Double DQN used in the paper, the TD error is:

$$\delta_t = R_t + \gamma Q_{target}(S_t, \arg\max_a Q(S_t, a)) - Q(S_{t-1}, A_{t-1})$$

The 'greedy TD-error prioritization' algorithm samples the transition in memory with the largest absolute TD error.

Stochastic Prioritization

However, greedy TD-error prioritization has several issues.

1. To avoid expensive sweeps over the entire replay memory, TD errors are only updated for the transitions that are replayed.
2. it is sensitive to noise spikes, which can be exacerbated by bootstrapping, where approximation errors appear as another source of noise.
3. greedy prioritization focuses on a small subset of the experience. This makes the system prone to overfitting.

To overcome these issues, this paper uses a probabilistic sampling method that interpolates between greedy prioritization and uniform random sampling as follows:

$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

where $p_i > 0$ is the priority of transition i. The exponent α determines how much the prioritization is applied. With $\alpha = 0$, it corresponds to the uniform case., and $\alpha = 1$ to the fully prioritized case. p_i can be expressed in two ways.

1. Direct, proportional prioritization : $p_i = |\delta_i| + \varepsilon$, where ε is a small positive constant that prevents the edge-case of transitions not being revisited if their error is close to zero.
2. indirect, rank-based prioritization : $p_i = \frac{1}{rank(i)}$, where $rank(i)$ is the rank of transition i when the replay memory is sorted according to $|\delta_i|$.

Annealing the bias

Prioritized replay is biased due to the difference between the distribution of the total replay memory and the distribution of the actual extracted results. Prioritized experience replay corrects this bias by using importance sampling weights:

$$w_i = \left(\frac{1}{N} \times \frac{1}{P(i)}\right)^\beta$$

that fully compensates for the non-uniform probabilities $P(i)$ if $\beta = 1$. It is used to update using $w_i \delta_i$ instead of δ_i in Q-learning as a weighted IS method, Mahmood(2014). the weights are always normalized by 1/w for stability.

In typical reinforcement learning scenarios, the unbiased nature of the updates is most important near convergence at the end of training. This paper hypothesize that a small bias can be ignored in this context: thus exploiting the flexibility of annealing the importance sampling correction over time by defining a schedule for the exponent β that only reaches 1 at the end of learning.

Algorithm

The following is the algorithm when Prioritized Experience Replay is applied to Double DQN.

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Implementation on JORLDY

- Prioritized Experience Replay Buffer JORLDY Implementation

```
## In JORLDY, PERBuffer was created based on the sum_tree. ##

class PERBuffer(ReplayBuffer):
    def __init__(self, buffer_size, uniform_sample_prob=1e-3):
        super(PERBuffer, self).__init__(buffer_size)
        self.tree_size = (self.buffer_size * 2) - 1
        self.first_leaf_index = self.buffer_size - 1

        self.sum_tree = np.zeros(self.tree_size)
        self.tree_index = self.first_leaf_index

        self.max_priority = 1.0
        self.uniform_sample_prob = uniform_sample_prob

    def store(self, transitions):
        if self.first_store:
            self.check_dim(transitions[0])

        for transition in transitions:
            self.buffer[self.buffer_index] = transition
            new_priority = (
                transition["priority"]
```

```
                if "priority" in transition
                else self.max_priority
            )
            self.add_tree_data(new_priority)

            self.buffer_counter = min(self.buffer_counter + 1, self.buffer_size)
            self.buffer_index = (self.buffer_index + 1) % self.buffer_size

    ...

    def update_priority(self, new_priority, index):
        ex_priority = self.sum_tree[index]
        delta_priority = new_priority - ex_priority
        self.sum_tree[index] = new_priority
        self.update_tree(index, delta_priority)

        self.max_priority = max(self.max_priority, new_priority)

    ...

    def sample(self, beta, batch_size):
        assert self.sum_tree[0] > 0.0
        uniform_sampling = np.random.uniform(size=batch_size) < self.uniform_sample_prob
        uniform_size = np.sum(uniform_sampling)
        prioritized_size = batch_size - uniform_size

        uniform_indices = list(
            np.random.randint(self.buffer_counter, size=uniform_size)
            + self.first_leaf_index
        )

        targets = np.random.uniform(size=prioritized_size) * self.sum_tree[0]
        prioritized_indices = [self.search_tree(target) for target in targets]

        indices = np.asarray(uniform_indices + prioritized_indices)
        priorities = np.asarray([self.sum_tree[index] for index in indices])
        assert len(indices) == len(priorities) == batch_size

        uniform_probs = np.asarray(1.0 / self.buffer_counter)
        prioritized_probs = priorities / self.sum_tree[0]

        # Calculate the IS weight.
        usp = self.uniform_sample_prob
        sample_probs = (1.0 - usp) * prioritized_probs + usp * uniform_probs
        weights = (uniform_probs / sample_probs) ** beta
        weights /= np.max(weights)
        batch = [self.buffer[idx] for idx in indices - self.first_leaf_index]

        transitions = self.stack_transition(batch)

        sampled_p = np.mean(priorities)
        mean_p = self.sum_tree[0] / self.buffer_counter
        return transitions, weights, indices, sampled_p, mean_p
```

- PER JORLDY Implementation

```
class PER(DQN):
    def __init__(
        self,
        alpha=0.6,
        beta=0.4,
        learn_period=16,
        uniform_sample_prob=1e-3,
        run_step=1e6,
        **kwargs
    ):
        super(PER, self).__init__(run_step=run_step, **kwargs)
        self.memory = PERBuffer(self.buffer_size, uniform_sample_prob)
        self.alpha = alpha
        self.beta = beta
        self.beta_add = (1 - beta) / run_step

    ...


    def process(self, transitions, step):
        result = {}

        # Process per step
        self.memory.store(transitions)
        delta_t = step - self.time_t
        self.time_t = step
        self.target_update_stamp += delta_t
        self.learn_period_stamp += delta_t

        # Annealing beta
        self.beta = min(1.0, self.beta + (self.beta_add * delta_t))

        if (
            self.learn_period_stamp >= self.learn_period
            and self.memory.size >= self.batch_size
            and self.time_t >= self.start_train_step
        ):
            result = self.learn()
            self.learning_rate_decay(step)
            self.learn_period_stamp -= self.learn_period

    ...


    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])

        # Calculate td_error with double DQN algorithm.
        ...

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

        # The IS weight is multiplied by the gradient.
        loss = (weights * (td_error**2)).mean()
        self.optimizer.zero_grad(set_to_none=True)
        loss.backward()
        self.optimizer.step()

        self.num_learn += 1

        result = {
            "loss": loss.item(),
            "epsilon": self.epsilon,
            "beta": self.beta,
            "max_Q": max_Q,
            "sampled_p": sampled_p,
            "mean_p": mean_p,
        }
        return result
```

References

Relevant papers

- Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching
- Deep Reinforcement Learning with Double Q-learning