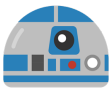

R2D2

Paper Link: RECURRENT EXPERIENCE REPLAY IN DISTRIBUTED REINFORCEMENT LEARNING

Key Features

- R2D2 is RNN-based RL algorithm from distributed prioritized experience replay.
- R2D2 alleviates the problems of representational drift and recurrent state staleness that can suffer when using RNNs for off-policy RL through burn-in and stored state strategies.
- RNN training potentially enables better representation learning, improving performance on domains that are partially observable.
- In addition, the paper found that the impact of RNN training goes beyond providing the agent with memory. RNN training improves performance even on domains that are fully observable and do not obviously require memory.

Background

Recurrent Reinforcement Learning

In order to achieve good performance in a partially observe environment, an RL agent requires a state representation that encodes information about its state-action trajectory. The most common way to achieve this is by using an RNN(typically LSTM) as part of the agent's state encoding. This paper empirically derive an improved training strategy for the problems of representational drift and recurrent state staleness that can be experienced when using RNNs for off-policy RL.

Distributed Reinforcement Learning

Recent reinforcement learning have achieved significantly improved performance by leveraging distributed training architectures which separate learning from acting, collecting data from many actors running in parallel on separate environment instances. Distributed replay allows the Ape-X agent(Horgan 2018) to decouple learning from acting, with actors feeding experience into the distributed replay buffer and the learner receiving training batches from it. Ape-X achieved high performance on Atari-57, significantly out-performing the best single-actor algorithms. In this paper, it is similar to Ape-X, but additionally uses the recurrent neural network mentioned above.

Method

Training recurrent RL Agent with experience replay

The paper Hausknecht & Stone(2015) compared two strategies of training LSTMs using sequential experiences:

- **The zero state training** strategy initializes the hidden state of LSTM to zero state at the beginning of sampled sequence.
- **Replaying whole episode trajectories**

The first strategy is very simple but prevents the network from using the temporal information. The second strategy, so it occurs a number of practical and computational issues. To fix these issues, this paper proposes following two strategies for training a recurrent neural network from randomly sampled replay sequences, that can be used individually or in combination:

- **stored state:** storing the recurrent state in replay and using it to initialize the hidden state of LSTM at training time.
- **burn-in:** Allow the network a ‘burn-in period’ by using a portion of the replay sequence only for unrolling the network and producing a start state, and update the network only on the remaining part of the sequence.

The burn-in strategy on its own partially mitigates the staleness problem on the initial part of replayed sequences, while not showing a significant effect on the Q-value discrepancy for later sequence states. The stored state strategy is overall much more effective at mitigating state staleness compared to the zero start state strategy, which also leads to clearer and more consistent improvement in empirical performance. The combination of both methods consistently yields the smallest discrepancy on the last sequence states and the most robust performance improvement.

Consequently, this paper demonstrates that both stored state and burn-in strategy provide substantial significant advantages over the naive zero state training strategy in terms of the effect in representation drift, recurrent state staleness, and empirical performance.

The recurrent replay distributed DQN agent

- Recurrent Replay Distributed DQN(R2D2) is most similar to Ape-X, built upon prioritized distributed replay and n-step double Q-learning (with n = 5), generating experience by a large number of actors (typically 256) and learning from batches of replayed experience by a single learner.
- Like Ape-X, R2D2 use the dueling network architecture, but provide an LSTM layer after the convolutional stack.
- Instead of regular $(s, a, r, s\prime)$ transition tuples, the paper store fixed-length (m = 80) sequences of (s, a, r)in replay, with adjacent sequences overlapping each other by 40 time steps, and never crossing episode boundaries.
- In addition, it uses an optional burn-in strategy of l = 40 or 20 steps.
- When training, it unrolls both online and target networks on the same sequence of states to generate value estimates and targets.
- Following the modified Ape-X version in Pohlen(2018), R2D2 do not clip rewards, but instead use an invertible value function rescaling of the form $h(x) = \text{sign}(x)(\sqrt{|x|+1}-1)+\epsilon x$ which results in the following n-step targets for the Q-value function:

$$\hat{y}_t = h\left(\sum_{k=0}^{n-1} r_{t+k}\gamma^k + \gamma^n h^{-1}\big(Q(s_{t+n}, a^*; \theta^-)\big)\right), \quad a^* = \arg\max_a Q(s_{t+n}, a; \theta)$$

- Here, θ^- denotes the target network parameters which are copied from the online network parameters θ every 2500 learner steps.
- R2D2 replay prioritization use a mixture of max and mean absolute n-step TD-errors δ_i over the sequence: $p = \eta \max i\delta_i + (1-\eta)\bar{\delta}$.
- Finally, compared to Ape-X, we used the slightly higher discount of $\gamma = 0.997$

Implementation on JORLDY

- R2D2 JORLDY Implementation

```
## R2D2 Network ##
# Use LSTM
# If there is no hidden state input, zero hidden state is used.


class R2D2(BaseNetwork):
    def __init__(self, D_in, D_out, D_hidden=512, head="mlp"):
        D_head_out = super(R2D2, self).__init__(D_in, D_hidden, head)
        self.D_hidden = D_hidden

        self.lstm = torch.nn.LSTM(
            input_size=D_head_out + D_out, hidden_size=D_hidden, batch_first=True
        )

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

        self.l1_a = torch.nn.Linear(D_hidden, D_hidden)
        self.l1_v = torch.nn.Linear(D_hidden, D_hidden)

        self.l2_a = torch.nn.Linear(D_hidden, D_out)
        self.l2_v = torch.nn.Linear(D_hidden, 1)

        orthogonal_init([self.l1_a, self.l1_v])
        orthogonal_init([self.l2_a, self.l2_v], "linear")

    def forward(self, x1, x2, hidden_in=None):
        x1 = super(R2D2, self).forward(x1)
        x = torch.cat([x1, x2], dim=-1)

        if hidden_in is None:
            hidden_in = (
                torch.zeros(1, x.size(0), self.D_hidden).to(x.device),
                torch.zeros(1, x.size(0), self.D_hidden).to(x.device),
            )

        x, hidden_out = self.lstm(x, hidden_in)

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

        x_a = F.relu(self.l1_a(x))
        x_v = F.relu(self.l1_v(x))

        # A stream : action advantage
        x_a = self.l2_a(x_a)  # [bs, seq, num_action]
        x_a -= x_a.mean(dim=2, keepdim=True)  # [bs, seq, num_action]

        # V stream : state value
        x_v = self.l2_v(x_v)  # [bs, seq, 1]

        out = x_a + x_v  # [bs, seq, num_action]
        return out, hidden_in, hidden_out
```

```
## R2D2 act ##
# Use hiden_state h', action a' of the previous step to get h, a


def act(self, state, training=True):
    self.network.train(training)
    epsilon = self.epsilon if training else self.epsilon_eval

    if self.prev_action is None:
        prev_action_onehot = torch.zeros(
            (state.shape[0], 1, self.action_size), device=self.device
        )
    else:
        prev_action_onehot = F.one_hot(
            torch.tensor(self.prev_action, dtype=torch.long, device=self.device),
            self.action_size,
        )

    q, hidden_in, hidden_out = self.network(
        self.as_tensor(np.expand_dims(state, axis=1)),
        prev_action_onehot,
        hidden_in=self.hidden,
    )

    if np.random.random() < epsilon:
        batch_size = (
            state[0].shape[0] if isinstance(state, list) else state.shape[0]
        )
        action = np.random.randint(0, self.action_size, size=(batch_size, 1))
    else:
        action = torch.argmax(q, -1).cpu().numpy()
    q = np.take(q.cpu().numpy()[:, -1], action)

    hidden_h = hidden_in[0].cpu().numpy()
    hidden_c = hidden_in[1].cpu().numpy()
    prev_action_onehot = prev_action_onehot.cpu().numpy()[:, -1]

    self.hidden = hidden_out
    self.prev_action = action

    return {
        "action": action,
        "prev_action_onehot": prev_action_onehot,
        "q": q,
        "hidden_h": hidden_h,
        "hidden_c": hidden_c,
    }
```

- Generating a recurrent experience in Actor. Distributed Actors code see Ape-X.

```
## Transition post-processing after env.step of Actor. ##
# Updates the current transition for n-steps using the sequence transitions stored in the tmp_buffer.
# Apply burn-in and store start strategies

def interact_callback(self, transition):
```

```
    _transition = {}
    self.tmp_buffer.append(transition)

    if (self.store_start or self.store_period_stamp == self.store_period) and (
        (self.zero_padding and len(self.tmp_buffer) >= self.n_step + 1)
        or (
            not self.zero_padding and len(self.tmp_buffer) == self.tmp_buffer.maxlen
        )
    ):

        _transition["hidden_h"] = self.tmp_buffer[0]["hidden_h"]
        _transition["hidden_c"] = self.tmp_buffer[0]["hidden_c"]
        _transition["next_hidden_h"] = self.tmp_buffer[self.n_step]["hidden_h"]
        _transition["next_hidden_c"] = self.tmp_buffer[self.n_step]["hidden_c"]

        for key in self.tmp_buffer[0].keys():
            # if key not in ['action', 'hidden_h', 'hidden_c', 'next_state']:
            if key not in ["hidden_h", "hidden_c", "next_state"]:
                if key in ["q", "state", "prev_action_onehot"]:
                    _transition[key] = np.stack(
                        [t[key] for t in self.tmp_buffer], axis=1
                    )
                else:
                    _transition[key] = np.stack(
                        [t[key] for t in self.tmp_buffer][:-1], axis=1
                    )

        # state sequence zero padding
        if self.zero_padding and len(self.tmp_buffer) < self.tmp_buffer.maxlen:
            lack_dims = self.tmp_buffer.maxlen - len(self.tmp_buffer)
            zero_state = np.zeros((1, lack_dims, *transition["state"].shape[1:]))
            _transition["state"] = np.concatenate(
                (zero_state, _transition["state"]), axis=1
            )
            zero_prev_action_onehot = np.zeros(
                (1, lack_dims, *transition["prev_action_onehot"].shape[1:])
            )
            _transition["prev_action_onehot"] = np.concatenate(
                (zero_prev_action_onehot, _transition["prev_action_onehot"]), axis=1
            )
            zero_action = np.zeros((1, lack_dims, *transition["action"].shape[1:]))
            _transition["action"] = np.concatenate(
                (zero_action, _transition["action"]), axis=1
            )
            zero_reward = np.zeros((1, lack_dims, *transition["reward"].shape[1:]))
            _transition["reward"] = np.concatenate(
                (zero_reward, _transition["reward"]), axis=1
            )
            zero_done = np.zeros((1, lack_dims, *transition["done"].shape[1:]))
            _transition["done"] = np.concatenate(
                (zero_done, _transition["done"]), axis=1
            )
            zero_q = np.zeros((1, lack_dims, *transition["q"].shape[1:]))
            _transition["q"] = np.concatenate((zero_q, _transition["q"]), axis=1)

            if lack_dims > self.n_step:
                _transition["next_hidden_h"] = self.tmp_buffer[0]["hidden_h"]
                _transition["next_hidden_c"] = self.tmp_buffer[0]["hidden_c"]
            else:
```

```
                _transition["next_hidden_h"] = self.tmp_buffer[
                    self.n_step - lack_dims
                ]["hidden_h"]
                _transition["next_hidden_c"] = self.tmp_buffer[
                    self.n_step - lack_dims
                ]["hidden_c"]

        target_q = self.inv_val_rescale(
            _transition["q"][:, self.n_burn_in + self.n_step :]
        )
        for i in reversed(range(self.n_step)):
            target_q = (
                _transition["reward"][:, i + self.n_burn_in : i + self.seq_len]
                + (
                    1
                    - _transition["done"][:, i + self.n_burn_in : i + self.seq_len]
                )
                * self.gamma
                * target_q
            )

        target_q = self.val_rescale(target_q)
        td_error = abs(
            target_q - _transition["q"][:, self.n_burn_in : self.seq_len]
        )
        priority = self.eta * np.max(td_error, axis=1) + (1 - self.eta) * np.mean(
            td_error, axis=1
        )
        _transition["priority"] = priority
        del _transition["q"]

        self.store_start = False
        self.store_period_stamp = 0

    if (
        len(self.tmp_buffer) > self.n_step
        and self.tmp_buffer[-self.n_step - 1]["done"]
    ):
        self.store_start = True
        self.tmp_buffer = deque(
            islice(self.tmp_buffer, len(self.tmp_buffer) - self.n_step, None),
            maxlen=self.tmp_buffer.maxlen,
        )

    self.store_period_stamp += 1
    if transition["done"]:
        self.hidden = None
        self.prev_action = None

    return _transition
```

- Leaner

```
## R2D2 learn ##
```

```
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"][:, : self.seq_len]
    action = transitions["action"][:, : self.seq_len]
    prev_action_onehot = transitions["prev_action_onehot"][:, : self.seq_len]
    reward = transitions["reward"]
    next_state = transitions["state"][:, self.n_step :]
    next_prev_action_onehot = transitions["prev_action_onehot"][:, self.n_step :]
    done = transitions["done"]
    hidden_h = transitions["hidden_h"].transpose(0, 1).contiguous()
    hidden_c = transitions["hidden_c"].transpose(0, 1).contiguous()
    next_hidden_h = transitions["next_hidden_h"].transpose(0, 1).contiguous()
    next_hidden_c = transitions["next_hidden_c"].transpose(0, 1).contiguous()
    hidden = (hidden_h, hidden_c)
    next_hidden = (next_hidden_h, next_hidden_c)

    eye = torch.eye(self.action_size).to(self.device)
    # one_hot_action = eye[action.view(-1,self.seq_len).long()]
    one_hot_action = eye[action.view(-1, self.seq_len).long()][:, self.n_burn_in :]

    q_pred = self.get_q(state, prev_action_onehot, hidden, self.network)
    q = (q_pred * one_hot_action).sum(-1, keepdims=True)
    with torch.no_grad():
        max_Q = torch.max(q).item()
        next_q = self.get_q(
            next_state, next_prev_action_onehot, next_hidden, self.network
        )
        max_a = torch.argmax(next_q, axis=-1)
        max_one_hot_action = eye[max_a.long()]

        next_target_q = self.get_q(
            next_state, next_prev_action_onehot, next_hidden, self.target_network
        )
        target_q = (next_target_q * max_one_hot_action).sum(-1, keepdims=True)
        target_q = self.inv_val_rescale(target_q)

        for i in reversed(range(self.n_step)):
            target_q = (
                reward[:, i + self.n_burn_in : i + self.seq_len]
                + (1 - done[:, i + self.n_burn_in : i + self.seq_len])
                * self.gamma
                * target_q
            )

        target_q = self.val_rescale(target_q)

    # Update sum tree
    td_error = abs(target_q - q)
    priority = self.eta * torch.max(td_error, axis=1).values + (
        1 - self.eta
    ) * torch.mean(td_error, axis=1)
    p_j = torch.pow(priority, self.alpha)
    for i, p in zip(indices, p_j):
        self.memory.update_priority(p.item(), i)
```

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

#         weights = torch.FloatTensor(weights[..., np.newaxis, np.newaxis]).to(self.device)
#         loss = (weights * (td_error**2)).mean()

weights = torch.FloatTensor(weights[..., np.newaxis]).to(self.device)
loss = (weights * (td_error[:, -1] ** 2)).mean()

self.optimizer.zero_grad(set_to_none=True)
loss.backward()
torch.nn.utils.clip_grad_norm_(self.network.parameters(), self.clip_grad_norm)
self.optimizer.step()

...
```

References

Relevant papers

- **Deep Recurrent Q-Learning for Partially Observable MDPs**
- **Distributed Prioritized Experience Replay**
- **Observe and Look Further: Achieving Consistent Performance on Atari**