APE-X

Paper Link: Distributed Prioritized Experience Replay

Key Features

- Ape-X proposes a distributed architecture for deep reinforcement learning
- Ape-X is trained using Distributed Stochastic Gradient Descent.
- Ape-X uses Prioritized Experience Replay and reduces variance through Distributed Importance Sampling.
- Each Actor can executes different policies to create experiences from variety of strategies.
- The general framework of the paper may be combined with other learning algorithms.

Background

Ape-X is a paper focusing on effective and scalable architecture for distributed reinforcement learning. The following three techniques are the main components.

Distributed Stochastic Gradient Descent

Distributed stochastic gradient descent is used to speed up training of deep neural network by parallelizing the computation of the gradients. The resulting parameter updates may be applied synchronously or asynchronously. Both approaches have proven effective and are becoming standard method for training of deep learning

Prioritized Experience Replay

Prioritized experience replay extends classic prioritized sweeping ideas to work with deep neural network function approximators. This approach uses a more general class of biased sampling procedures that focus learning on the most ‘surprising’ experiences. Biased sampling can be particularly helpful in reinforcement learning, since the reward signal may be sparse and the data distribution depends on the agent’s policy. In an ablation study conducted to investigate the relative importance of several algorithmic components (Hessel et al., 2017), prioritization was found to be the most important component contributing to the agent’s performance.

Distributed Importance Sampling

A complementary family of techniques for speeding up training is based on variance reduction by means of importance sampling. This has been shown to be useful in the context of neural networks. Sampling non-uniformly from a dataset and weighting updates according to the sampling probability in order to counteract the bias thereby introduced can increase the speed of convergence by reducing the variance of the gradients. In supervised learning, this approach has been successfully extended to the distributed setting (Alain et al., 2015).

Method

The Ape-X architecture

- Acotr
    - Multiple Actors, each has its own instance of the environment, generate experience.
    - Compute initial priorities of the generated experience and add it in the shared experience replay memory.
- Learner

    - The learner updates the network by sampling data from the shared experience replay memory.
    - And it updates priorities of the experience in the memory.
- The Actors' networks are periodically updated with the latest network parameters from the learner.

Algorithm

Algorithm 1 Actor

```
1: procedure ACTOR($B, T$)                                    ▷ Run agent in environment instance, storing experiences.
2:     $\theta_0 \leftarrow$ LEARNER.PARAMETERS( )                ▷ Remote call to obtain latest network parameters.
3:     $s_0 \leftarrow$ ENVIRONMENT.INITIALIZE( )                 ▷ Get initial state from environment.
4:     for $t = 1$ to $T$ do
5:         $a_{t-1} \leftarrow \pi_{\theta_{t-1}}(s_{t-1})$               ▷ Select an action using the current policy.
6:         $(r_t, \gamma_t, s_t) \leftarrow$ ENVIRONMENT.STEP($a_{t-1}$)   ▷ Apply the action in the environment.
7:         LOCALBUFFER.ADD($(s_{t-1}, a_{t-1}, r_t, \gamma_t)$)       ▷ Add data to local buffer.
8:         if LOCALBUFFER.SIZE( ) $\geq B$ then     ▷ In a background thread, periodically send data to replay.
9:             $\tau \leftarrow$ LOCALBUFFER.GET($B$)     ▷ Get buffered data (e.g. batch of multi-step transitions).
10:            $p \leftarrow$ COMPUTEPRIORITIES($\tau$)   ▷ Calculate priorities for experience (e.g. absolute TD error).
11:            REPLAY.ADD($\tau, p$)                      ▷ Remote call to add experience to replay memory.
12:        end if
13:        PERIODICALLY($\theta_t \leftarrow$ LEARNER.PARAMETERS())      ▷ Obtain latest network parameters.
14:    end for
15: end procedure
```

Algorithm 2 Learner

```
1: procedure LEARNER($T$)                                     ▷ Update network using batches sampled from memory.
2:     $\theta_0 \leftarrow$ INITIALIZENETWORK( )
3:     for $t = 1$ to $T$ do                                      ▷ Update the parameters $T$ times.
4:         $id, \tau \leftarrow$ REPLAY.SAMPLE( )   ▷ Sample a prioritized batch of transitions (in a background thread).
5:         $l_t \leftarrow$ COMPUTELOSS($\tau; \theta_t$)          ▷ Apply learning rule; e.g. double Q-learning or DDPG
6:         $\theta_{t+1} \leftarrow$ UPDATEPARAMETERS($l_t; \theta_t$)
7:         $p \leftarrow$ COMPUTEPRIORITIES( )     ▷ Calculate priorities for experience, (e.g. absolute TD error).
8:         REPLAY.SETPRIORITY($id, p$)                            ▷ Remote call to update priorities.
9:         PERIODICALLY(REPLAY.REMOVETOFIT())              ▷ Remove old experience from replay memory.
10:    end for
11: end procedure
```

The general framework described above may be combined with different algorithms. The paper describes the following two methods.

Ape-X DQN

Ape-X DQN uses double Q-learning and multi-step bootstrap targets as a learning algorithm, and a dueling network architecture as the function approximator. This results in computing for all elements in the batch the loss $l_t(\theta) = \frac{1}{2}\left(G_t - q(S_t, A_t, \theta)\right)^2$ with

$$G_t = \underbrace{R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n \overbrace{q(S_{t+n}, \operatorname*{argmax}_a q(S_{t+n}, a, \boldsymbol{\theta}), \boldsymbol{\theta}^-)}^{\text{double-Q bootstrap value}}}_{\text{multi-step return}},$$

In principle, Q-learning variants are off-policy methods, so we are free to choose the policies for generating data. However, in practice, the choice of behaviour policy does affect both exploration and the quality of function approximation. In Ape-X DQN, each actor executes a different policy, and this allows experience to be generated from a variety of strategies, relying on the prioritization mechanism to select the most effective experiences. In Ape-X DQN, the actors use ε-greedy policies with different values of ε.

Ape-X DPG

The Ape-X DPG setup is similar to Ape-X DQN, but the actor's policy is now represented explicitly by a separate policy network, in addition to the Q-network. The two networks are optimized separately by minimizing different losses on the sampled experience. The Q-network outputs an action-value estimate $q(s, a, \psi)$ for a given state s and multi-dimensional action $a \in \mathbb{R}^m$. It is updated using temporal-difference learning with a multi-step bootstrap target. The Q-network loss can be written as $l_t(\psi) = \frac{1}{2}\left(G_t - q(S_t, A_t, \psi)\right)^2$, where

$$G_t = \underbrace{R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n q(S_{t+n}, \pi(S_{t+n}, \phi^-), \psi^-)}_{\text{multi-step return}}.$$

The policy network outputs an action $A_t = \pi(S_t, \phi) \in \mathbb{R}^m$. The parameters of policy network are updated using policy gradient ascent on the estimated Q-value, using gradient $\nabla_\phi q(S_t, \pi(S_t, \phi), \psi)$ —note that this depends on the policy parameters ϕ only through the action $A_t = \pi(S_t, \phi)$ that is input to the critic network.

Implementation on JORLDY

JORLDY distributed processing Implementation

- In case of synchronously distributed processing
- Learner

```
# Initialize synchronously Actors
# Learner Algorithm

def sync_distributed_train(config_path, unknown):

    ...

    try:
        # Create Actors in DistributedManager
        distributed_manager = DistributedManager(
            Env,
            config.env,
            Agent,
            {"device": "cpu", **agent_config},
            config.train.num_workers,
            "sync",
        )

        ## Procedure Learner(T) ##
        # θ <- Initialize network
        agent = Agent(**agent_config)
        assert agent.action_type == env.action_type
        if config.train.load_path:
            agent.load(config.train.load_path)

        save_path = path_queue.get()
        step, print_stamp, save_stamp = 0, 0, 0
        # for t=1 to T do
        while step < config.train.run_step:
            # run Actors and receive transitions
            transitions = distributed_manager.run(config.train.update_period)
            step += config.train.update_period
            print_stamp += config.train.update_period
            save_stamp += config.train.update_period
            # transitions store replay buffer and sample, network update
            result = agent.process(transitions, step)
            # send latest network parameters
            distributed_manager.sync(agent.sync_out())

    ...
```

- Actor

```
# Create Actors

class DistributedManager:
    def __init__(self, Env, env_config, Agent, agent_config, num_workers, mode):

        ...

        self.actors = [
            Actor.remote(Env, env_config, agent, i) for i in range(self.num_workers)
        ]

        ...

    # run Actors synchronously
    def run(self, step=1):
        assert step > 0
        if self.mode == "sync":
            items = ray.get([actor.run.remote(step) for actor in self.actors])
            transitions = reduce(lambda x, y: x + y, [item[1] for item in items])


# Actor Algorithm

@ray.remote
class Actor:
    def __init__(self, Env, env_config, agent, id):
        self.id = id
        # Initialize Environment
        self.env = Env(id=id + 1, **env_config)
        self.agent = agent.set_distributed(id)
        self.state = self.env.reset()

    def run(self, step):
        transitions = []
        for t in range(step):
            # a <- π(s) (select action using the current policy)
            action_dict = self.agent.act(self.state, training=True)
            # (r, s) <- Environment.step(a)
            next_state, reward, done = self.env.step(action_dict["action"])
            # LocalBuffer.Add(s, a, r)
            transition = {
                "state": self.state,
                "next_state": next_state,
                "reward": reward,
                "done": done,
            }
            transition.update(action_dict)
            # p <- compute priorities(τ)
            transition = self.agent.interact_callback(transition)
            if transition:
                transitions.append(transition)
            self.state = next_state if not done else self.env.reset()
        # Replay.Add(τ, p) (send transitions to Learner buffer)
        return self.id, transitions

    # θ <- Learner.Parameter
    def sync(self, sync_item):
        self.agent.sync_in(**sync_item)
```

- In case of asynchronously distributed processing
- Learner

```
# Initialize asynchronously Actors
# Learner Algorithm

def async_distributed_train(config_path, unknown):

    ...

    interact = mp.Process(
        target=interact_process,
        args=(
            DistributedManager,
            distributed_manager_config,
            trans_queue,
            interact_sync_queue,
            config.train.run_step,
            config.train.update_period,
        ),
    )
    manage.start()
    # Create Actors in interact.distributed_manager
    interact.start()
    try:
        ## Procedure Learner(T) ##
        # θ <- Initialize network
        agent = Agent(**agent_config)
        assert agent.action_type == env.action_type
        if config.train.load_path:
            agent.load(config.train.load_path)
        save_path = path_queue.get()
        step, _step, print_stamp, save_stamp = 0, 0, 0, 0
        # for t = 1 t
        while step < config.train.run_step:
            transitions = []
            while (_step == 0 or not trans_queue.empty()) and (
                _step - step < config.train.update_period
            ):
                # receive Actors transitions
                _step, _transitions = trans_queue.get()
                transitions += _transitions
            delta_t = _step - step
            print_stamp += delta_t
            save_stamp += delta_t
            step = _step
            # transitions store replay buffer and sample, network update
            result = agent.process(transitions, step)
            try:
                interact_sync_queue.get_nowait()
            except:
                pass
            # send latest network parameter
            interact_sync_queue.put(agent.sync_out())

    ...
```

- Learner and Actors interact asynchronously

```
def interact_process(
    DistributedManager,
    distributed_manager_config,
    trans_queue,
    sync_queue,
    run_step,
    update_period,
):
    distributed_manager = DistributedManager(*distributed_manager_config)
    num_workers = distributed_manager.num_workers
    step = 0
    try:
        while step < run_step:
            # run Actors and receive transitions asynchronously
            transitions = distributed_manager.run(update_period)
            delta_t = len(transitions) / num_workers
            step += delta_t
            trans_queue.put((int(step), transitions))
            if sync_queue.full():
                distributed_manager.sync(sync_queue.get())
            while trans_queue.full():
                time.sleep(0.1)
    except Exception as e:
        traceback.print_exc()
    finally:
        distributed_manager.terminate()


class DistributedManager:

    ...

    def run(self, step=1):
        assert step > 0
        if self.mode == "sync":

          ...

        # run Actors asynchronously
        else:
            if len(self.running_ids) == 0:
                self.running_ids = [actor.run.remote(step) for actor in self.actors]

            done_ids = []
            while len(done_ids) == 0:
                done_ids, self.running_ids = ray.wait(
                    self.running_ids, num_returns=self.num_workers, timeout=0.1
                )

            items = ray.get(done_ids)
            transitions = reduce(lambda x, y: x + y, [item[1] for item in items])
            runned_ids = [item[0] for item in items]

            if self.sync_item is not None:
                ray.get(
                    [self.actors[id].sync.remote(self.sync_item) for id in runned_ids]
                )
            self.running_ids += [self.actors[id].run.remote(step) for id in runned_ids]
```

APE-X DQN JORLDY Implementation

- Currently, only Ape-X DQN is implemented in JORLDY.

```
class ApeX(DQN):

    ...

    def learn(self):
        # Sample a prioritized batch of transitions in PER buffer
        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"]

        eye = torch.eye(self.action_size).to(self.device)
        one_hot_action = eye[action.view(-1).long()]
        q = (self.network(state) * one_hot_action).sum(1, keepdims=True)

        with torch.no_grad():
            max_Q = torch.max(q).item()
            # calculate double q-network q(s, argmaxπ(s))
            next_q = self.network(next_state)
            max_a = torch.argmax(next_q, axis=1)
            max_one_hot_action = eye[max_a.long()]

            next_target_q = self.target_network(next_state)

            target_q = (next_target_q * max_one_hot_action).sum(1, keepdims=True)
            # y = multi-step + double q
            for i in reversed(range(self.n_step)):
                target_q = reward[:, i] + (1 - done[:, i]) * self.gamma * target_q

        # Update sum tree
        td_error = abs(target_q - q)
        p_j = torch.pow(td_error, 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 * (td_error**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()

    ...

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

        # Process per step
        delta_t = step - self.time_t
        self.memory.store(transitions)
        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.buffer_counter >= self.batch_size
            and self.time_t >= self.start_train_step
        ):
            result = self.learn()
            self.learning_rate_decay(step)
            self.learn_period_stamp = 0

        # Process per step if train start
        if self.num_learn > 0 and self.target_update_stamp >= self.target_update_period:
            self.update_target()
            self.target_update_stamp = 0

        return result

    def interact_callback(self, transition):
        _transition = {}
        self.tmp_buffer.append(transition)
        if len(self.tmp_buffer) == self.tmp_buffer.maxlen:
            _transition["state"] = self.tmp_buffer[0]["state"]
            _transition["action"] = self.tmp_buffer[0]["action"]
            _transition["next_state"] = self.tmp_buffer[-1]["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][:-1], axis=1
                    )

            target_q = self.tmp_buffer[-1]["q"]
            for i in reversed(range(self.n_step)):
                target_q = (
                    self.tmp_buffer[i]["reward"]
                    + (1 - self.tmp_buffer[i]["done"]) * self.gamma * target_q
                )
            priority = abs(target_q - self.tmp_buffer[0]["q"])

            _transition["priority"] = priority
            del _transition["q"]

        return _transition
```

References

Relevant papers

- Prioritized Experience Replay
- Rainbow: Combining Improvements in Deep Reinforcement Learning
- Online Batch Selection for Faster Training of Neural Networks