Fast Population-Based Reinforcement Learning on a Single Machine

Arthur Flajolet, Claire Bizon Monroc, Karim Beguir, Thomas Pierrot

Presenter: Guillaume Richard

ICML 2022

InstaDeep™

Special thanks to Thomas Hirtz for streamlining the implementation

Population-based Reinforcement Learning

Recent work has shown that training multiple RL agents concurrently can be beneficial:

- for hyperparameter tuning [1, 2, 3],
- to generate diverse behaviors for efficient exploration [4, 5] or fast adaptation in damage-recovery applications [6].

Even simply training the same agent many times in independent runs is crucial to be able to draw conclusions [7].

However, training a population of agents:

- is much slower than training a single one if the agents are trained sequentially on a single hardware accelerator,
- requires a lot of resources if a full hardware accelerator is dedicated to each agent.

Observation: in many practical cases, the neural networks used to parametrize the agents are small enough that training a single agent does not fully leverage the vectorization capabilities of modern hardware accelerators.

Idea: vectorize "training-related" computations over the population similarly to what is done for the batch size.

1: Jaderberg et al. (2017) - Population-based training of neural networks. arXiv:1711.09846.
2: Vinyals et al. (2019) - Grandmaster level in starcraft II using multi-agent reinforcement learning. Nature, 575 (7782):350–354.
3: Jaderberg et al. (2019) - Human-level performance in 3d multiplayer games with population-based reinforcement learning. Science, 364(6443):859–865.
4: Pierrot et al. (2022) - Diversity policy gradient for sample efficient quality-diversity optimization. arXiv:2006.08505.
5: Pourchot, A. and Sigaud, O (2019) - CEM-RL: Combining evolutionary and gradient-based methods for policy search. ICLR 2019.
6: Eysenbach et al. (2019) - Diversity is all you need: Learning skills without a reward function. ICLR 2019.
7: Agarwal et al. (2021) - Deep reinforcement learning at the edge of the statistical precipice. NeurIPS 2021.

Population-based RL Training on a Single GPU

We first study a simplistic setting:

- a single hardware accelerator is available,
- the agents are trained independently from one another,
- training data is available without delay.

We compare multiple implementations of a population-wide update step based on the runtime per step:

- sequential,
- parallel,
- vectorized,

for three standard off-policy RL algorithms:

- Twin Delayed Deep Deterministic Policy Gradient (**TD3**) with fully-connected neural networks on a MuJoCo locomotion environment,
- Soft Actor Critic (**SAC**) with fully-connected neural networks on a MuJoCo locomotion environment,
- Deep Q-Learning (**DQN**) with convolutional neural networks on an Atari 2600 environment,

using two automatic differentiation frameworks:

- PyTorch,
- JAX with Just-In-Time compilation.

InstaDeep™

Implementations of a Population-wide Update Step

Sequential implementation: Iterate over all agents and carry out one update step each time.

```
all_agents = [MyAgent() for _ in range(population_size)]
training_batch_iterator = MyDataset()

while True:
  for agent in all_agents:
    training_batch = next(training_batch_iterator)
    agent.update_step(training_batch)
```

Advantages

- trivial to implement
- trivial to extend to more complex settings where some losses / parameters are shared across agents
- low memory usage

Disadvantages

- inefficient if training a single agent does not fully utilize the accelerator

InstaDeep™

Implementations of a Population-wide Update Step

Parallel implementation: Spawn one process per agent and share the accelerator across processes.

```
from multiprocessing import Process

def train_one_agent():
    agent = MyAgent()
    training_batch_iterator = MyDataset()
    while True:
        training_batch = next(training_batch_iterator)
        agent.update_step(training_batch)

all_processes = []
for _ in range(population_size):
    process = Process(target=train_one_agent)
    process.start()
    all_processes.append(process)
```

Advantages

- trivial to implement but training data will need to flow between processes
- (potentially) more efficient use of the GPU parallelization capabilities than the sequential approach

Disadvantages

- memory fragmentation
- still unable to fully leverage the GPU parallelization capabilities
- requires efficient inter-process communication tools in more complex settings where some losses / parameters are shared across agents

InstaDeep™

Implementations of a Population-wide Update Step

Vectorized implementation: Concatenate neural network weights as well as training data across the population and write vectorized versions of the neural networks / losses in **PyTorch** (or use the **vmap** primitive in **JAX**).

```
import torch

class VectorizedLinearLayer(torch.nn.Module):
    """Vectorized version of torch.nn.Linear."""

    def __init__(self, population_size: int, in_features: int, out_features: int):
        super().__init__()

        self._population_size = population_size
        self._weight = torch.nn.Parameter(torch.empty(population_size, in_features, out_features), requires_grad=True)
        self._bias = torch.nn.Parameter(torch.empty(population_size, 1, out_features), requires_grad=True)

        # Initialization of the weights
        ...

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        assert x.shape[0] == self._population_size
        return x.matmul(self._weight) + self._bias
```

Advantages

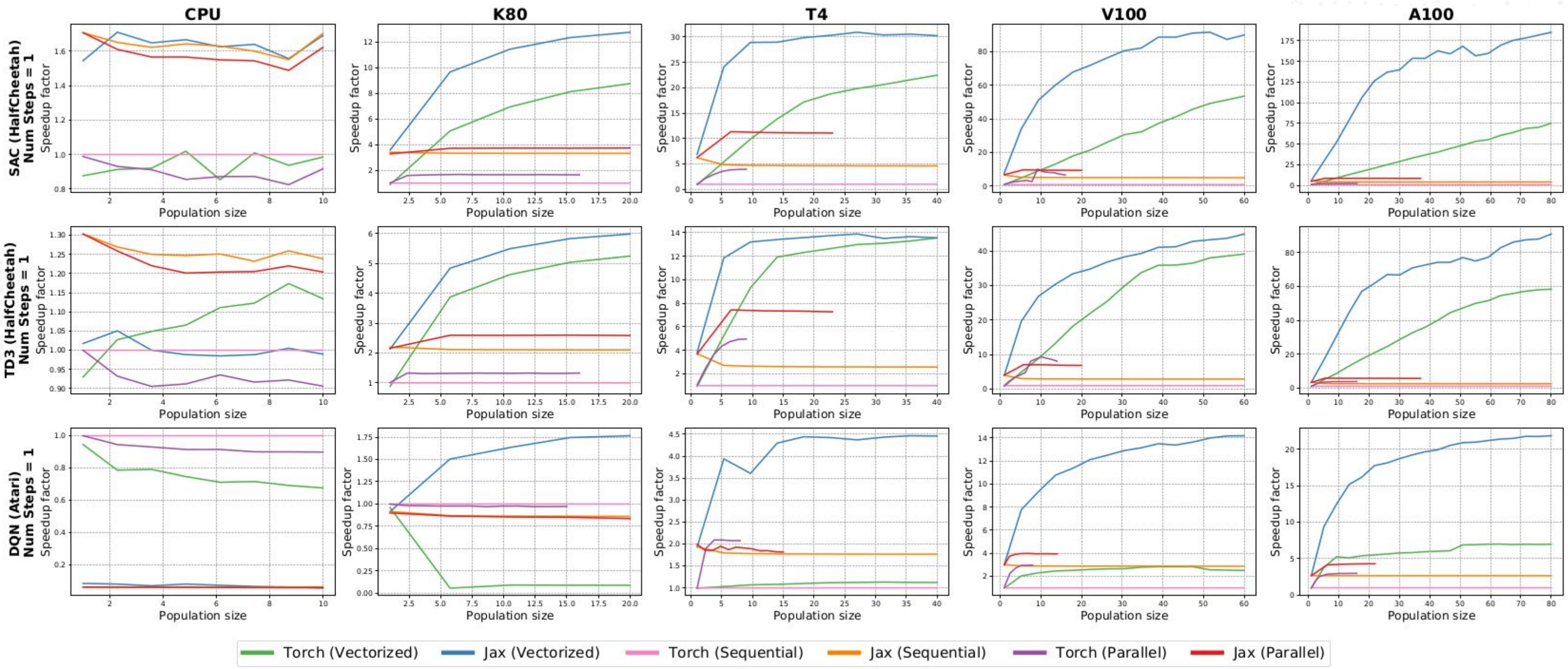
- can fully leverage GPU parallelization capabilities
- easy to extend to more complex settings where some losses / parameters are shared across agents
- efficient memory utilization compared to the parallel implementation
- easy to implement with automated vectorization frameworks (such as JAX)

Disadvantages

- higher memory utilization compared to the sequential approach
- sometimes require to vectorize “by hand” to get the best results

Implementations of a Population-wide Update Step

Speedup factor of the population-wide update step w.r.t. the sequential implementation in PyTorch



- Vectorized implementations can be up to two orders of magnitude faster than sequential ones for population sizes up to 80.
- Parallel implementations can be much faster than sequential ones but memory fragmentation severely limits the population size.
- Compiling the static graph of computations can readily yield significant speedups (up to 14x in our experiments).
- Vectorizing computations does not help on CPUs.

InstaDeep™

Case Studies

We revisit three population-based RL studies from previous works:

- Hyperparameter tuning with Population-Based Training (**PBT**) [1].
- Off-policy RL mixed with the Cross Entropy Method (**CEM-RL**) [2], where some of the neural network weights are shared across the population.
- Diversity via Determinants (**DvD**) [3], where the loss includes a term that involves the neural network weights of all agents in the population.

using:

- vectorized update steps implemented in JAX,
- simple tools based on the built-in multiprocessing python library for efficient data collection.

Specifically, we compare the total reward achieved as a function of the total walltime elapsed since the start of the experiment (which includes the overhead due to data collection and environment interactions) when:

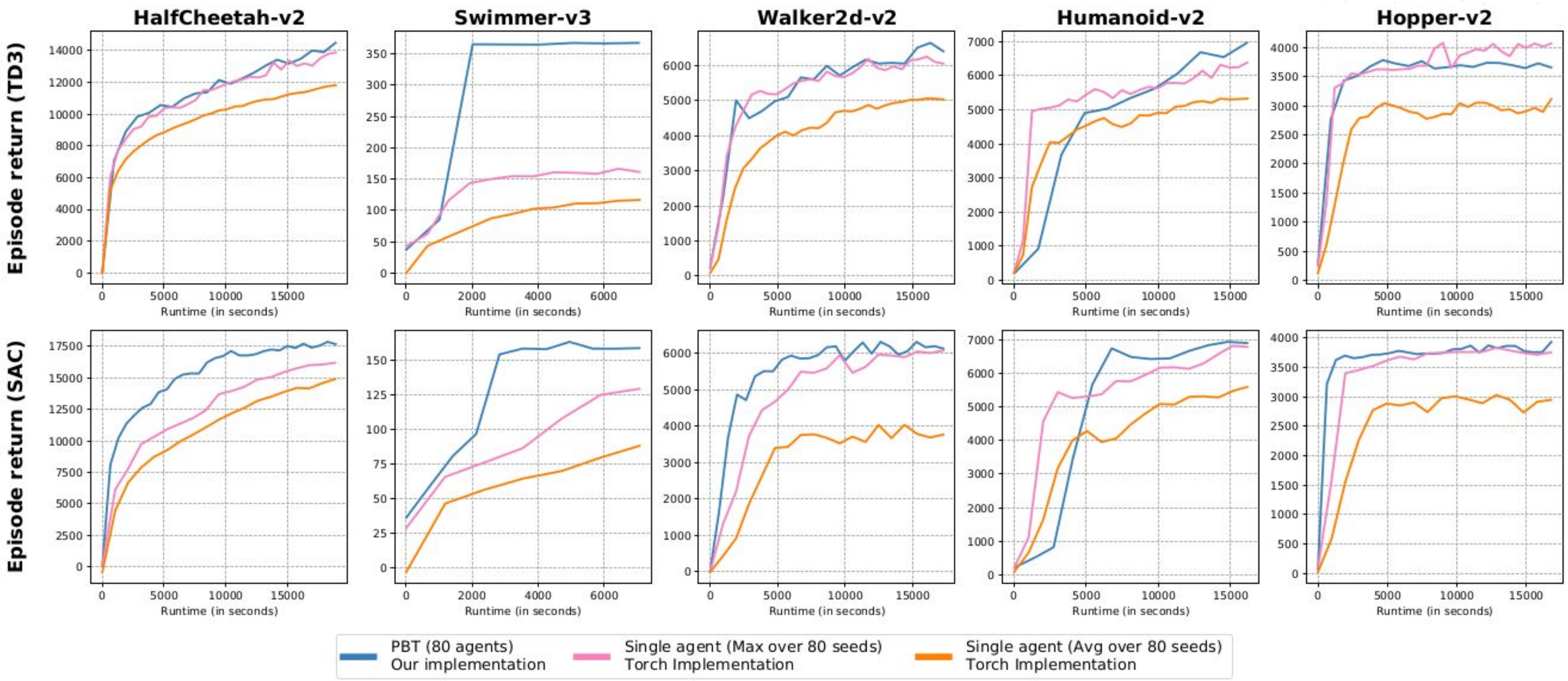
- training a population of agents using the vectorized approach with JAX,
- training a single agent with PyTorch.

1: Jaderberg et al. (2017) - Population-based training of neural networks. arXiv:1711.09846.

2: Pourchot, A. and Sigaud, O (2019) - CEM-RL: Combining evolutionary and gradient-based methods for policy search. ICLR 2019.

3: Parker-Holder et al. (2020) - Effective diversity in population based reinforcement learning. NeurIPS 2020.

Case Study: Hyperparameter Tuning of SAC and TD3 with PBT



Performance (in terms of mean episode returns) achieved as a function of total time elapsed since the beginning of the training run for various implementations and Gym locomotion environments. All experiments are run on a single machine with 4 T4 accelerators and 40 CPU cores.

InstaDeep™

Fast Population-Based Reinforcement Learning on a Single Machine

Thank you!

Code: https://github.com/instadeepai/fastpbrl

Paper: https://arxiv.org/abs/2206.08888

Contact: a.flajolet@instadeep.com and t.pierrot@instadeep.com

InstaDeep™