Harvard John A. Paulson
School of Engineering
and Applied Sciences

Optimizing Data Generation for Reinforcement Learning

Rudra Barua, Charlie Harrington, Fiona Henry, Michael Krumdick

Problem

Deep reinforcement learning has been used to train agents to play games and bring trained robotics simulations to the real world

- Cutting edge experiments requires trillions of state transitions

Scaling deep reinforcement learning experiments have typically involved large distributed systems and upscaling hardware

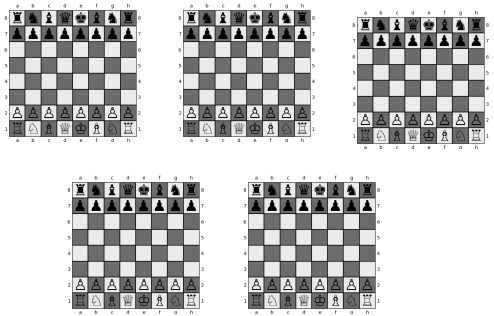
- Expensive and limits access
- Distributed system approaches underutilize resources on individual machines

Reinforcement learning workloads are primarily dominated by environment simulation, model inference, and back propagation computations

- In prior works, processes need to wait for each other to generate the data needed for the next computation
- The overall throughput is often dominated by this bottleneck and processes idle waiting for input

Sequential Data Generation

Environment 1



Dataset of 5 Moves

15 seconds

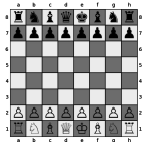
Reinforcement Learning Agent

.1 seconds

Total Time: 15.1 seconds

Parallelized Data Generation

Environment 1



Environment 2

Environment 3

Environment 4

Environment 5

Dataset of 5 Moves

3 seconds

Reinforcement Learning Agent

.1 seconds

Total Time: 3.1 seconds

The total runtime is:

# of batches required — Informed by the sample complexity of the model and hyperparameters, **built upon decades of research**

X

Time to update a model on each batch — Informed by the underlying deep learning library and compute hardware, **built upon decades of research and millions of R&D dollars**

X

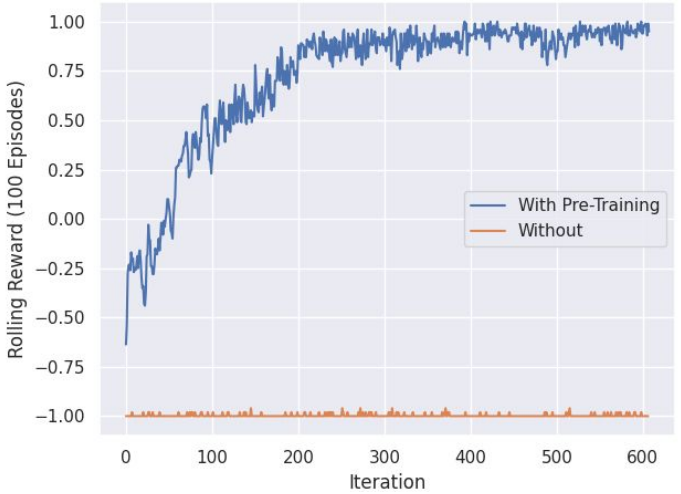
Time to generate a batch — Usually one-off code written by a practitioner, **built upon one afternoon reading "multiprocessing in python" tutorials**

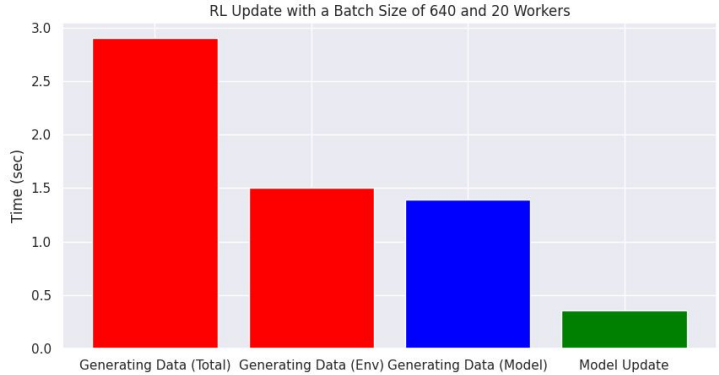
Benchmarking on Chess DRL Agent

We have a pre-existing DRL system we can use to benchmark increases in performance

Used 20 workers running 32 steps to generate a batch of 640 examples

We can compute the estimated speed up without re-running the learning algorithm





Plan for Parallelization

Construct table similar to matrix multiplication speed for environment generation

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62.806	2.7	40.45

Plan for Parallelization

Construct table similar to matrix multiplication speed for environment generation

Version	Implementation	Environment Generation Speed-up	Estimated Impact on RL Performance
1	Python	-	-
2	Parallelize Python	-	-
3	C++	-	-
4	C++ with OpenMP	-	-
5	C++ with MPI	-	-

Questions?