Reinforcement Learning and Evolutionary Strategy for Control Tasks: A Comparison of Parallel Scalability

Shawn Anderson(saa108@sfu.ca)
Sethuraman Annamalai(sannamal@sfu.ca)
Namita Shah(namitas@sfu.ca)

I. INTRODUCTION

In recent years, a Big Data ecosystem has emerged. Industry and Science have seen growth in compounding factors, such as: size of datasets, distributed infrastructure, mass parallelism, and number of talented practitioners. In this embryonic landscape there has been a trend of resurgence for decades-old, biologically-inspired, optimization techniques that fall under the umbrella of Artificial Intelligence(AI). The trend started in 2009 when Deep Learning(DL) applications began winning international pattern recognition contests [14]. The trend now continues with Deep Reinforcement Learning(RL).

RL is an approach for solving control tasks such as robotics or game playing. RL techniques have recently reached super human performance in playing highly dimensional games such as Go and Atari [12]. Today, in 2018, we are seeing the resurgence of Evolutionary Strategy(ES) as an alternative solution for solving highly complex control tasks such as business optimizations at Uber [13]. Evolutionary strategy is proving to be competitive with RL for game playing, while being simpler to implement and easier to parallelize.

The purpose of this paper is to serve as a comparison of state of the the art RL and ES algorithms for benchmark control tasks. OpenAI hosts a suite of benchmark control tasks in their Gym environment library [1]. Gym is meant to be a standardized set of tasks for which RL researchers can use to show reproducible results of algorithms. In this paper we implement RL and ES solutions for the CartPole problem [11], which is hosted under the Classic Control section of Gym environments. We compare ES and RL in terms of implementation difficulty, parallelization difficulty, parallel scalability, performance, resource utilization, and power consumption. Tests are ran across varying numbers of CPU cores, with and without GPU utilization.

II. BACKGROUND

A. Deep Reinforcement Learning

Reinforcement Learning is a biologically inspired, dynamic programming method for control tasks, pioneered by Sutton and Barto in the 1980s [4]. The essential paradigm of reinforcement learning is that of an Agent within an Environment. At any point in time, the agent finds itself in a particular state within the environment. Given the state, the agent will take an action. Given the action, the environment will give a positive or negative reward to the agent and change to a new state, this action, reward, state change cycle is repeated in a loop over the course of an episode. The purpose of the agent is to maximize reward over an episode as it moves throughout states in the environment. In other words, the task of the agent is to choose actions that will maximize the reward received from the environment.

In 2015, Deepmind took Reinforcement Learning to a new level when it showed that reinforcement learning, combined with deep neural networks could be used to solve a general domain of problems with a single algorithm. Deepmind used Deep Reinforcement Learning to solve an array of Atari games, with no special domain knowledge for any particular game [4]. The agent only received state input in the form of screen pixels, and reward in the form of the game score. Deepmind continued to showcase the power of Deep RL when its Go playing AI AlphaGO defeated Grandmaster Lee Sedol in 2016 [2].

Deep Reinforcement Learning algorithms have traditionally been difficult to parallelize, and prone to diverge with subtle hyper-parameter or environment perturbations. There has been a lot of progress in recent years in finding robust and parallelizable approaches to using RL for control tasks. One successful, and widely adopted approach is Asynchronous Actor-Critic Agents(A3C) [3]. We investigate A3C as a solution to the CartPole environment.

B. Asynchronous Actor-Critic Agents

Until the recent past the most widely used technique to solve various reward based problems were multiple variations of DQNs(Deep Q-Networks) where an agent is trained on a copy of the problem instance utilizing a simple neural network with policy gradients [8]. So what if multiple agents (agents and workers will hereafter be used interchangeably) get trained in parallel and all the knowledge gained is effectively utilized to solve the problem? This scope along with the inability to parallelize traditional RL algorithms gave rise to A3C, which can fully take advantage of parallel computation technologies such as multi-core processing and GPUs.

In A3C, multiple worker agents are created (depending on the machine capability). Each agent has its own copy of the problem environment. These agents iteratively learn

Simon Fraser University

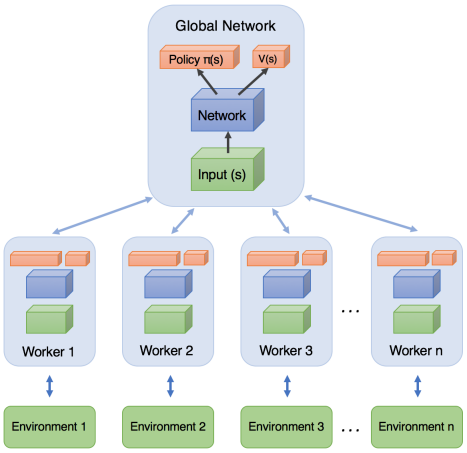

Fig. 1. Basic A3C architecture

through experience from their environments by taking random actions. The learning is primarily done by combining Q-Learning and policy gradients. All these experiences from different agents are combined together so that the overall training becomes as diverse as possible. Fig. 1. depicts a basic A3C architecture.

C. Evolutionary Strategy

Evolution Strategy (ES) is a search optimization paradigm that is inspired by the process of natural evolution[9]. The idea is to sample randomly about a current position in an optimization space, then move the current position towards those samples that appear closer to the optimization objective. Key concepts are that of Population, Mutation, Generation, Fitness, and Breeding. Given some point in an optimization space, a population can be created by randomly perturbing the point to create an array of new points, following some defined distribution. These random perturbations are known as mutation. Given a population, each point can be assessed for its fitness. Fitness is a notion of being closer to the objective. Each agent in the population will be assessed for its fitness, and those with satisfactory fitness will be selected to produce the next generation. The selection and combination phase can be described as Breeding. The process of mutation, selection, and breeding, is iteratively repeated until the optimization objective is satisfied.

ES is highly parallelizable due to it's simple nature of generating and evaluating samples. Samples need not even be broadcast to workers. If a worker knows the current central point of the population, it can generate it's own population through random mutation, evaluate the population, and select for the fittest agents, all independently. If we have 100 neural networks in our population and 100 processors, all of those networks can be evaluated at the same time [10].

D. NeuroEvolution

The particular type of ES that we evaluate is known as NeuroEvolution. NeuroEvolution is a method for training neural networks, alternative to backpropogation. In this case, the central point of our population, as described above, is a set of parameter values for an Artificial Neural Network(ANN). The mutation process involves adding a symmetrical, Gaussian sampled noise value to each of the parameters in the network. By sampling a population of mutations, we produce an array of new ANNs, all with the same topology as our original, but all with different parameter values. We then evaluate each new network in solving our control task. To breed the population we produce a new ANN with parameters equal to a weighted average of the parameters of our population, where the weight given to each agent in the population, is the fitness of that agent. This process is formalized by [6] in figure 3.

Algorithm 1 Evolution Strategies

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0
for $t = 0, 1, 2, \ldots$ **do**
  Sample $\epsilon_1, \ldots \epsilon_n \sim \mathcal{N}(0, I)$
  Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, \ldots, n$
  Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
end for

Fig. 2. Pseudo code for factored Gaussian NeuroEvolution[9].

E. OpenAI Gym and CartPole

Gym is an open source suite of simulation environments for testing RL algorithms with a python interface. Gym is made by OpenAI, a research organization which publishes work across the field of AI, including Deep Learning, Reinforcement Learning, and Evolution Strategy. Elon Musk is a founder, and the chairman of OpenAI. Musk advocates openness of research as essential to addressing any potential threat that Artificial General Intelligence(AGI) may pose to humanity.

CartPole is a classic control problem [11] in which a pole is balanced on a cart. The cart must balance the pole by moving along a single dimension. The cart must do this without exiting the allowed movement area. If the cart goes out of bounds or the pole falls beyond 15 degrees from vertical the episode is finished. The agent(the cart) receives +1 reward for every time step that the episode continues. Thus the goal of the agent is to maximize the number of time steps for which the pole is balanced. The environment state is composed of two real values, the position of the cart, and the angle of the pole. The agent chooses one of two actions for each time step, to move left or right. Solving the environment, or 'converging' is defined as achieving greater than 195 reward, averaged over 100 consecutive episodes. In our experiments, we run episodes with a max reward of 10000, such that agents have more room for learning, and to increase the duration of experiments.

F. Multiprocessing and GPU Parallelism

Since 2004 computational scaling has transitioned from vertical scaling, the production of more powerful processors, to horizantal scaling, the distribution of tasks across multiple processors. The trend of horizontal scaling began with the

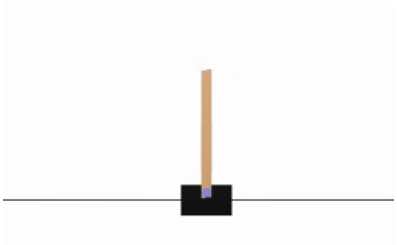

Fig. 3. A rendered image of the CartPole learning environment[9].

proliferation of multi-core CPUs, then progressed to mass parallelism with GPUs, and today continues in the realm of distributed systems. As this is the natural progression of parallelism, this is the approach that we took with our experiments. For each of NeuroEvolution, and A3C, we first implement solutions to the CartPole problem with single core implementations, moving on to multi-core, and then GPU implementations. Distributed implementations is left for future work.

All experiments are done with CPU: Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz. This CPU has four cores, and two threads per core, resulting in a total of 8 threads. All experiments with GPU are done with: Nvidia GeForce GTX 1050 Ti. This GPU has 768 CUDA cores, and 4GB of memory.

III. Methodology

A. Asynchronous Actor-Critic Agents

The version of A3C used here is deployed on top of TensorFlow. We make use of the following fundamental building blocks of an A3C architecture to solve the CartPole problem in a parallel environment.

1) Environment: The Gym environment to solve the problem to achieve maximum reward. In this case, the CartPole environment.

2) A3C Network: Incorporates the components of the TensorFlow graph model. This topology allows the workers to communicate with the brain and vice versa. By communication, we mean the process of updating losses and gradients back and forth between the brain and the workers.

3) Agents: Multiple agents are deployed, each with a separate copy of the environment. Each agent is created on one physical thread on the machine being used. These agents then interact with their environments by taking random steps to maximize their reward getting potential. This is done through computations of basic Q-values of different states with respect to different actions, with which the "advantage" is calculated which acts as the basic update rule in the learner.

4) Brain: The brain essentially consists of two parts. A data queue and a simple neural network. The agents append their experiences to this queue in an asynchronous fashion. These experiences are then considered to be the new training data, compressed into batches and the network is iteratively optimized as new experiences get added to the queue.

5) Optimizer: The function of the optimizer is to recursively train the brain's network with the incoming samples of experiences from the workers. One simple Optimizer thread cannot keep up with the pace of the incoming samples. Hence, two optimizer threads are deployed where one thread pre-processes the data into the required format while the other thread computes the losses and the gradients needed to train the network either using a CPU or a GPU based on the TensorFlow implementation.

Also, the required hyper-parameters for the models were tuned using an efficient grid search model for various configurations (varying the number of threads and optimizers). These selected set of hyper-parameters were fixed for all the conducted experiments on A3C.

With brevity in mind, the detailed mathematics involved in the Q-Learning of the agents and the deep neural network of the brain is not discussed here. The essence of the paper is to bring out an effective methodology to solve these problems from a systems perspective.

B. Evolution Strategy

1) Neural Networks for Environment Navigation: We implement a simple version of Evolution Strategy under the class of Natural Evolution Strategy(NES), specifically, NeuroEvolution, as described in [Evolution Strategies]. In this algorithm, a neural network is used to select actions. The input to the network is the environment state, the output is a distribution over the action space of the agent. The environment state is described as a vector, in the case of CartPole, it is a vector of length two, a real scalar for the position of the cart, and a real scalar for the angle of the pole. The output of the network is a softmax activation. Softmax outputs a stochastic distribution over the action space. The action space for CartPole is the domain [0,1], where 0 represents moving left, and 1 represents moving right.

2) Action Selection: From the softmax output, there are two possible strategies for selecting which action to take, first is to simply select the argmax of the distribution, secondly is to sample from the distribution. For example, if the network output [0.3,0.7], an argmax implementation would take action 1, moving to the right, but a sample implementation would move to the right with 70% probability, and to the left with 30% probability. Action selection methods will have their own advantages and disadvantages for different problems. The sampling technique offers more exploration, and flexibility for complex, or adaptive environments. Whereas the argmax approach is better for simpler environments. For CartPole we use argmax.

3) Keras Implementation: Keras is a high level imperative interface for constructing neural networks [7]. It is a python library which uses graph computation backends such as Tensorflow or Theano. Keras makes it very easy to quickly implement and test neural network architectures. For this reason it was a natural choice for a NeuroEvolution implementation.

Indeed, a NeuroEvolution implementation with Keras proved to be very simple. A Keras model provides get_weights, and set_weights methods. For each generation of agents, we simply get the weights of our model, flatten them to a 1-D vector, generate 200 random noise vectors of equal length, and produce 200 mutations of our network by adding the weights to each random noise vector. For each mutation, we set weights of our model to the respective mutated weights and evaluate the fitness on a single episode of cartpole, where the fitness is the amount of reward achieved, or the number of consecutive steps that the agent succeeds in balancing the pole. Once we have evaluated the fitness of all of our mutations, we recombine them to a single set of weights by taking a weighted average of the mutations, weighted by fitness. This process is repeated for a certain amount of time, or until a convergence threshold is reached.

Although Keras made for a very simple single process implementation of NeuroEvolution, it was a roadblock for a mult-threaded implementation. Any call to the Keras API within a python subprocess would result in our program hanging indefinitely with no error message, an indication of deadlock. In our research we found that this is a well known problem with Keras. Despite valiant effort, we could not find a solution to this problem, and had to abandon Keras in pursuit of a multi-threaded NeuroEvolution solution to CartPole. More on this issue with Keras is discussed in section VI.

4) Numpy and Minpy Implementations: We realized that our neural network architecture for this problem is simple enough that we could implement it by hand just using numpy. Numpy offers C-like contiguous arrays for python. The implementation is especially simple considering there is no need for gradients or backpropogation with NeuroEvolution. Only the feed-forward part of the network must be implemented. We soon found an open source implementation of ES that does just this[https://github.com/MorvanZhou/Evolutionary-Algorithm]. It was simple to modify this code for our experiments, running on multiple threads, and recording necessary metrics during training.

We attempted to modify our numpy ES implementation to run on GPU using minpy [5]. Minpy is a numpy interface that uses Apache MXNet [9] as a backend. MXNet serves as an open source interface to CUDA, allowing for GPU usage for numpy array and matrix processing. Unfortunately, due to time constraints, and limitations of the minpy API, we were not able to get experiment data for this version of our ES implementation. This is left for future work.

IV. Experiments

Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz, and Nvidia GeForce GTX 1050 Ti are used in experiments conducted to analyze the performance of various parallel configurations.

A. Evaluating A3C

Two types of experiments (using both CPU and GPU) were carried out to analyze the performance of A3C to solve the CartPole problem.

1) Varying Training Time: The training part of the A3C architecture can be divided into two parts. In the first part, all the agents start interacting with their environments to generate experiences and these experiences are appended to the training queue for the brain. In the second part, the data from training queue is taken and the brain is optimized. An important thing to note here is, both the parts are done asynchronously. The first experiment was varying this training time. Training for 10, 30 and 60 seconds were carried out.

2) Varying Number of Worker Threads: The implementation of A3C produces each worker on a separate physical thread of the machine. Hence, a machine with more number of cores has the capability to host numerous worker agents for the initial training. This would ultimately produce more sample experience with which the brain can be optimized.

Refer Fig. 4., Fig. 5. and Fig. 6. where the iterative reward obtained by different worker agents against increasing sequential number of episodes is plotted with respect to both the experiments stated above. In A3C implementation, the learning method employed by different worker agents involves taking random steps and learning from those steps by continuously interacting with the brain. Since considerable amount of randomness is involved, one run of any experiment would not paint an accurate picture of the observed metrics. Hence, for each of the configuration 10 runs were made and the averages were computed. In the plots, each point depicts the average reward obtained by one worker in a particular episode over 10 runs.

From these plots, the following things can be observed

1) Unsurprisingly, training the network for a longer time makes the worker agents to contribute a higher number of sample experiences for the brain
2) Similarly, having a higher number of threads also contributes to a higher number of sample experiences since each physical thread contributes to one worker.
3) The average reward attained by different workers (irrespective of the processing unit) is higher when the network is trained for a longer time or the number of worker agents in the architecture is high. The explanation behind this is quite straightforward, when there are either more threads or when the network is trained for a longer time, the number of samples created is large, as explained in the two points above. Since there are

Fig. 4. Episodes vs Reward - 10 Seconds training with 1, 2, 4 and 8 threads respectively

Fig. 5. Episodes vs Reward - 30 Seconds training with 1, 2, 4 and 8 threads respectively

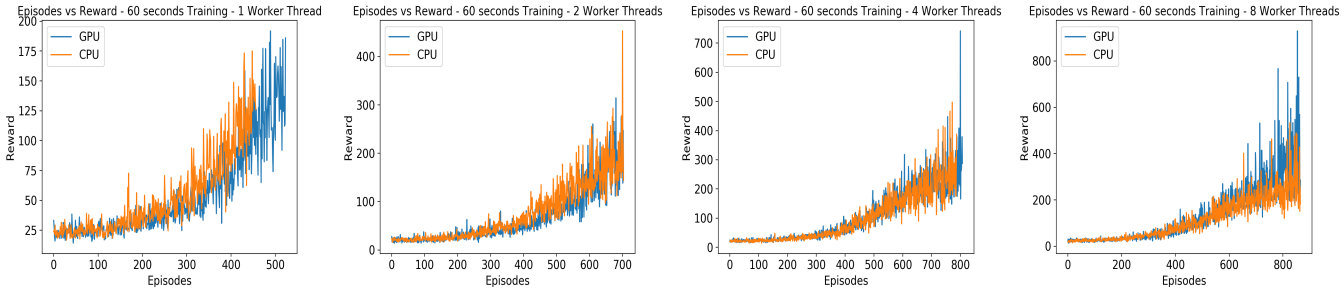

Fig. 6. Episodes vs Reward - 1 minute training with 1, 2, 4 and 8 threads respectively

more samples, the brain gets optimized with a higher amount of samples leading to a high quality learning. Also, the iterative learning of each worker involves interacting with the brain periodically. So, when the brain has better knowledge about the problem being solved, it benefits the agent and the consecutive steps taken by the agents become iteratively better.

4) In the lower configurations, i.e., lesser number of threads and shorter training time, the performance of CPU and GPU is quite comparable. In fact, in most cases, there is no clear winner, and it was observed that it was random to an extent.
5) In higher configurations, i.e., higher number of threads and longer training time, it can be seen that the performance (in terms of reward reaching potential) is higher in GPU.
6) The training and learning processes of both the brain and the workers consists of computations such as gradient losses, advantage, etc. All of these are predominantly matrix operations. When the TensorFlow model is deployed on the CPU, these operations are performed on the CPU and on the GPU otherwise. Matrix operations on GPU are generally multiple times faster when compared to the CPU. When the number of samples is less or the training time is shorter, this advantage is not quite seen. But when the number of training samples are high in the higher configurations the GPU is the clear winner because of this reason.
7) For example, when the network is trained for 60 seconds with 8 worker threads (the highest configuration), on an average close to 900 samples are created in both the processing units. But since the operations in the GPU are computed extremely fast, the number of optimize operations performed on the brain is going to be higher. This leads to a higher quality brain and ultimately better sample experiences from the workers. The computing power which enables a higher number of optimize operations is the real reason behind GPU

TABLE I

NUMBER OF OPTIMIZATIONS IN CPU

Training Time	1 Thread	2 Threads	4 Threads	8 Threads
10 Seconds	4190519	3315009	1843934	169917
30 Seconds	12408341	9468782	4900181	216279
60 Seconds	24511211	18729065	8828144	284966

TABLE II

NUMBER OF OPTIMIZATIONS IN GPU

Training Time	1 Thread	2 Threads	4 Threads	8 Threads
10 Seconds	4262052	3463418	1942255	282510
30 Seconds	12367608	9994129	5179248	371449
60 Seconds	25201837	19071461	9661038	502272

implementation of the A3C network performing better than the CPU version.

8) The number of optimizations to the brain determines the extent to which agents can claim rewards. From tables I and II it can be seen that the number of optimizations performed to the brain is high in GPU for every configuration. It can also be seen that the as the number of worker threads increases, the number of optimizations performed decreases. This can be attributed to the fact that the "Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz" has 4 cores and 8 physical threads. Apart from the worker threads, there are optimizer threads running on the machine, hence the interleaving of the threads decreases the number of optimizations.
9) For example, compare the 1 worker and 8 workers network trained for 60 seconds. Though the 1 worker implementation produces more than 5 times the optimization count as the 8 worker configuration, it still cannot beat the performance of the 8 thread implementation. This is because, in the 8 thread implementation, the number of experience sample produces is very high and diverse when compared to the 1 thread implementation.

B. *Evaluating NeuroEvolution*

To evaluate NeuroEvolution, we vary the number of cores, and give the algorithm a fixed amount of training time, to see what level of reward the agent can achieve in the allotted time. Results of this experiment can be seen in Figure 9 as a comparison to A3C. From this investigation we can see that NeuroEvolution achieves high performance even with a single core, and scales linearly with number of cores added. This result is rather impressive, and shows the potential of ES as a highly scalable, yet more simple alternative to backpropogation. The parallel performance gains can clearly be seen in figure 7 which shows that more threads directly results in faster generation iterations. It would be very interesting to continue testing this trend in a distributed setting with many more cores. We hypothesis that the linear gains would continue for very large numbers of threads due to the very small communication required by the parallelized ES algorithm.

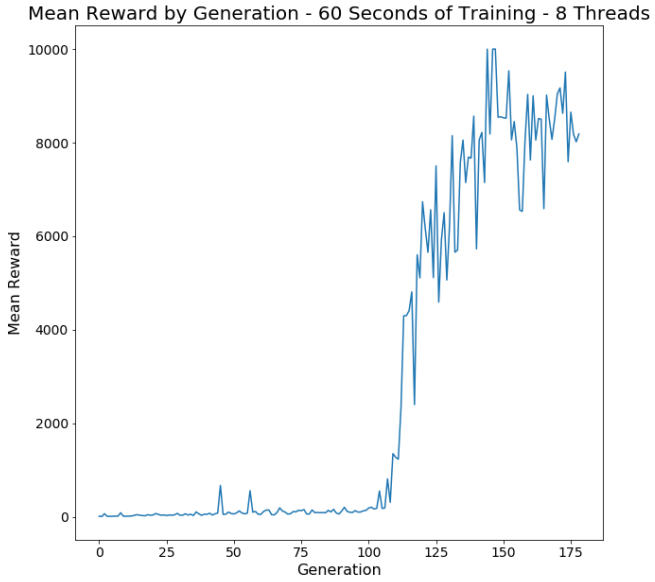

Fig. 7. A display of the learning capability of NeuroEvolution on the CartPole problem. There is a component of momentum in the learning process of evolution, once stepping stones are discovered, they compound progress towards the objective.

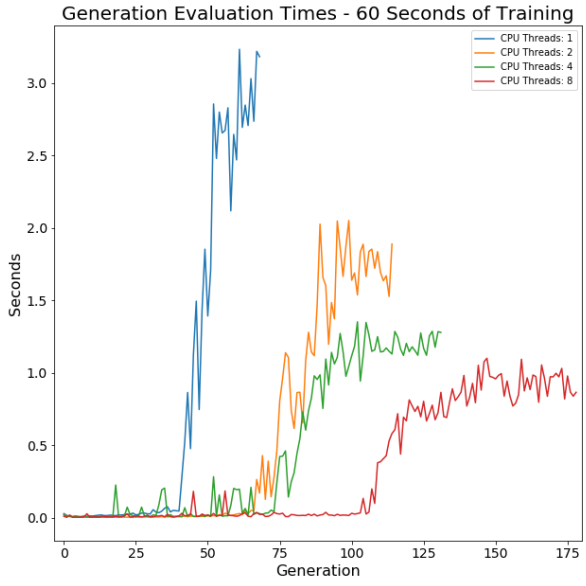

Fig. 8. NeuroEvolution: The time it takes to evaluate and breed a generation of network mutations by number of threads. Later generations take longer because episodes last longer as the agent gets better at balancing the pole. We can see that with more threads, generations are processed faster. This highlights the parallel scalability of NeuroEvolution.

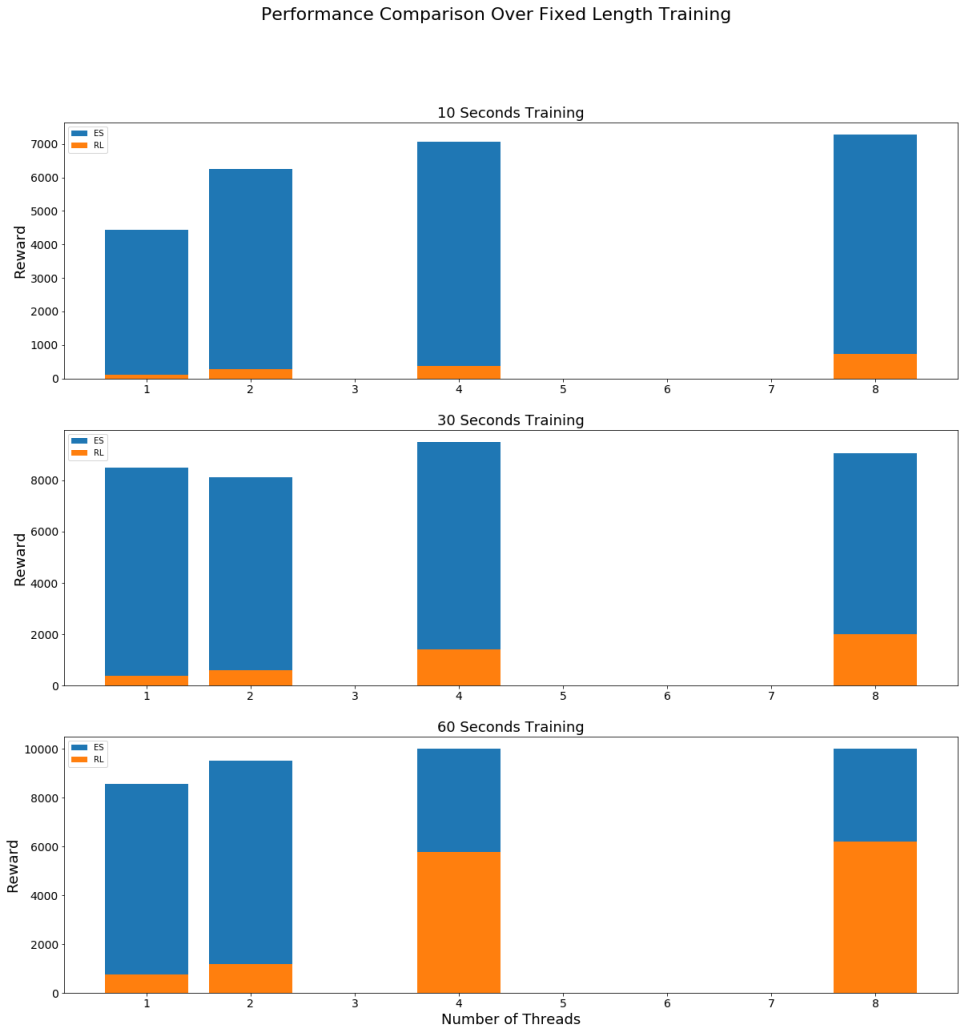

Fig. 9. A comparison of performance scalability between RL and ES. We can see that ES has much higher performance in the case of a single thread, but as training time increases, and number of threads increases, A3C begins to approach the performance of NeuroEvolution. It would be interesting to see how this trend continues in a distributed setting with tens or hundreds of cores.

C. Comparison

Both algorithms have made significant contributions to the space of control tasks. We feel that neither algorithm can be strictly proclaimed as superior, although the simplicity and high initial performance of ES is highly appealing. We have highlighted two essential concepts to consider when choosing an algorithm as a solution to a control task: 1. Complexity, we believe that higher complexity problems may demand higher complexity solutions. 2. Resources available, A3C benefits highly from GPU, do to it's use of traditional backpropogation for training it's brain. With ES, additional CPU cores are more important than access to GPU, as no backpropogation is required. A comparison of performance scalability between the algorithms can be seen in figure 9.

In terms of CPU utilization, ES seems to scale perfectly up to 8 cores, with no overhead introduced by adding more cores. This is due to the lack of data sharing requirements of workers. This high CPU utilization is not necessarily a strict advantage. A3C appears to have high performance scaling, while have low CPU utilization scaling. At large scale this could potentially result in comparable performance to ES while having lower power consumption. Comparisons of resource utilization and power consumption can be seen in figure 10.

V. Learnings

A. Reinforcement Learning and Evolution Strategy

Two of our group members had zero experience with Reinforcement Learning, and none of us had any experience with Evolution Strategy. We read a lot of papers and blog posts about how these things work. We learned the most

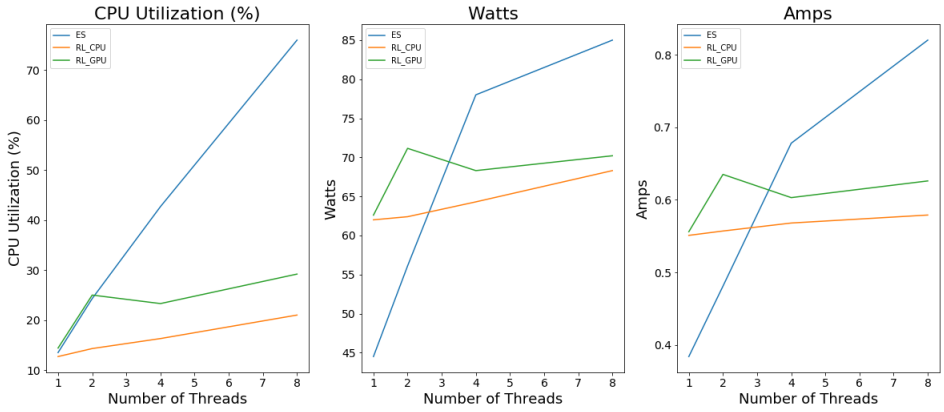

Fig. 10. A comparison of resource utilization and power consumption between A3C and NeuroEvolution. We can see that CPU utilization begins similar for both algorithms, but power consumption is higher for A3C. Also, CPU utilization scales much better for NeuroEvolution. These two observations imply that A3C has much higher communication overhead than NeuroEvolution, and indeed this is true.

however, from reading and using implementations found on the web, and in the case of ES, making our own implementations. Running experiments over various parameters of the algorithms, and with different resources gave a deep understanding of the mechanisms at play in these algorithms, especially from a systems perspective.

B. Scientific Process and Measurement Tools

One of the most interesting and challenging aspects of the project is the process of experiment design. Often we found that we did not truly understand what questions we were asking until we began collecting data to answer those questions. Results of one experiment would lead to deeper questions to be answered by another experiment. We did not foresee measuring power consumption until very late in our work, but doing so lead to very interesting findings, and align with theoretical principles of the algorithms.

C. Tensorflow Locking Mechanisms Prevent Multiprocessing

It is a known, and unsolved issue with Keras that it will hang when being used in a subprocess or thread. This is due to locking mechanisms which are used when accessing the model weights. To solve this we tried using alternative backends such as Tensorflow and Theano, we tried moving around Keras imports and instantiations, such as only within subprocesses themselves, we investigated disabling tensorflow locking, all with no success. We found many references around the web with very few claiming success. We tried the approaches proposed by those who claimed success, but we were unsuccessful in reproducing their results. Eventually we opted to not use keras for network creation, using numpy instead. This was simple enough as the network is small, and there is no need for back propagation in Evolution Strategy.

D. Sparse Rewards in the MountainCar Environment

The original environment that we chose to run our experiments on is the MountainCar environment under the Classic Control section of Gym environments. In this problem, a car has found itself at the bottom of a valley, and does not have enough power to drive up out of the valley. The solution is for the car to drive back and fourth, using momentum to escape the valley. No positive reward is ever given to the agent until it actually escapes the valley. This problem can be labeled as a Sparse Reward learning problem because the agent never receives any reward until it solves the task. This requires an agent to first solve the task through random exploration with no notion of fitness, before it can gain reward and begin learning.

The sparse reward nature of this problem proved to make it extremely difficult to solve, and solutions had extreme variability in their time to find a solution. For this reasons we were forced to abandon the MountainCar problem as a good benchmark for our experiments.

VI. Future Work

There are many interesting frameworks available for ES implementations in python which we did not get a chance to investigate in this work. In particular, DEAP, is a distributed, multiprocessing library for evolutionary algorithms in python. Elephas is a distributed Keras extension built with SPARK. Using Elephas could be a way around the multiprocessing challenges that we face from Keras's intrinsic locking mechanisms. Using such frameworks, we would like to continue our experiments with many more number of cores in a distributed environment.

We would like to test these algorithms in more complex and highly dimensional learning environments such as the MuJoCo physics simulator. We have also discovered the PyGame Learning Environment, which offers many reinforcement learning environments as alternatives to Gym environments.

VII. Conclusions

Both algorithms are excellent for control tasks and offer efficient scalability. ES is much simpler in design, it is simpler to implement and simpler to parallelize. For this

reason we suggest ES as a preferable solution to simple problems such as the CartPole problem. However, we have a feeling that more complex tasks may demand more complex solutions. A3C appears to be approaching the performance of ES as running time and number of threads is increased. Also, A3C appears to scale better in terms of power consumption. We believe there may be a critical point of task complexity in which A3C surpasses ES in terms of scalable performance. We would like to see future work done to investigate this hypothesis.

VIII. References

*References

[1] 1606.01540.pdf. https://arxiv.org/pdf/1606.01540.pdf.

[2] Alphago — deepmind. https://deepmind.com/research/alphago/.

[3] Asynchronous methods for deep reinforcement learning. https://arxiv.org/pdf/1602.01783.pdf.

[4] bookdraft2017nov5.pdf. http://incompleteideas.net/book/bookdraft2017nov5.pdf.

[5] dmlc/minpy: Numpy interface with mixed backend execution. https://github.com/dmlc/minpy. (Accessed on 04/14/2018).

[6] Evolution strategies as a scalable alternative to reinforcement learning. https://arxiv.org/pdf/1703.03864.pdf.

[7] keras-team/keras: Deep learning for humans. https://github.com/keras-team/keras.

[8] Lets make an a3c: Implementation — . https://jaromiru.com/2017/03/26/lets-make-an-a3c-implementation/.

[9] Mxnet: A scalable deep learning framework. https://mxnet.apache.org/.

[10] Neuroevolution: A different kind of deep learning - o'reilly media. https://www.oreilly.com/ideas/neuroevolution-a-different-kind-of-deep-learning.

[11] Openai gym. https://gym.openai.com/envs/CartPole-v0/.

[12] Playing atari with deep reinforcement learning — deepmind. https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning/. (Accessed on 04/14/2018).

[13] Welcoming the era of deep neuroevolution — uber engineering blog. https://eng.uber.com/deep-neuroevolution/.

[14] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.