Distributed Deep Reinforcement Learning on the Cloud for Autonomous Driving*

Mitchell Spryn
Microsoft Corporation
Redmond, WA, United States
mspryn@microsoft.com

Aditya Sharma
Microsoft Corporation
Redmond, WA, United States
adshar@microsoft.com

Dhawal Parkar
Microsoft Corporation
Redmond, WA, United States
dparkar@microsoft.com

Madhur Shrimal
Microsoft Corporation
Redmond, WA, United States
mashrima@microsoft.com

ABSTRACT

This paper proposes an architecture for leveraging cloud computing technology to reduce training time for deep reinforcement learning models for autonomous driving by distributing the training process across a pool of virtual machines. By parallelizing the training process, careful design of the reward function and use of techniques like transfer learning, we demonstrate a decrease in training time for our example autonomous driving problem from 140 hours to less than 1 hour. We go over our network architecture, job distribution paradigm, reward function design and report results from experiments on small sized cluster (1-6 training nodes) of machines. We also discuss the limitations of our approach when trying to scale up to massive clusters.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Robotics*; **Architectures**; *Cloud computing*; • **Theory of computation → Machine Learning Theory**; *Reinforcement Learning*;

KEYWORDS

Autonomous Driving, Deep Reinforcement Learning, Distributed Machine Learning, Cloud Computing, Simulation

ACM Reference Format:
Mitchell Spryn, Aditya Sharma, Dhawal Parkar, and Madhur Shrimal. 2018. Distributed Deep Reinforcement Learning on the Cloud for Autonomous Driving. In *SEFAIAS'18: SEFAIAS'18:IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems , May 28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3194085.3194088

*This work was carried out by the Deep Learning and Robotics Garage Chapter at Microsoft as part of the Distributed Reinforcement Learning tutorial in the Autonomous Driving Cookbook. The code for the work presented here can be found on GitHub by visiting this link: https://aka.ms/AutonomousDrivingCookbook

SEFAIAS'18, May 28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5739-5/18/05…$15.00
https://doi.org/10.1145/3194085.3194088

1 INTRODUCTION

Deep learning is one of the most important technologies that is powering the AI revolution. Models based around deep neural networks have been shown to dramatically outperform traditional models when tested on a wide range of applications, from image classification to text generation. While these models perform well, one of their main limitations is that they require massive amounts of labeled data for training. Collecting and maintaining these datasets is feasible for simple problems like image classification, but quickly becomes impractical for more complex problems like autonomous driving. Achieving levels 4 and 5 of autonomy in cars would require training on hundreds of millions and sometimes hundreds of billions of miles worth of data to demonstrate reliability, according to a report from the Rand Corporation [6]. The report further goes on to state that existing fleets of vehicles would take tens and sometimes hundreds of years to drive these miles. For solving complex problems of that scale, not only do we need smart ways to collect and use this data (simulated environments), it is also necessary to have algorithms that can learn from unlabeled data. One of the most popular classes of algorithms used for this purpose is deep reinforcement learning.

Deep reinforcement learning algorithms are inspired by the psychology of human and animal behavior. These models follow the paradigm of an agent interacting with its environment through a series of actions. The environment provides feedback to the agent via a reward signal, which the agent attempts to maximize. Over time, the agent learns how to select the actions that maximize the reward function, learning the desired behavior. Recently, the power of deep neural networks have been integrated into these frameworks, allowing AI agents to perform at or above parity with human experts on a wide range of difficult tasks such as mastering Atari video games [8] and the game of Go [14].

1.1 Deep Learning for Autonomous Driving

One of the most exciting AI challenges today is autonomous driving. Over the last couple of years, we have seen a shift in approach from tradition probabilistic robotics-based techniques to more end-to-end deep learning based techniques [2]. The goal of the problem is

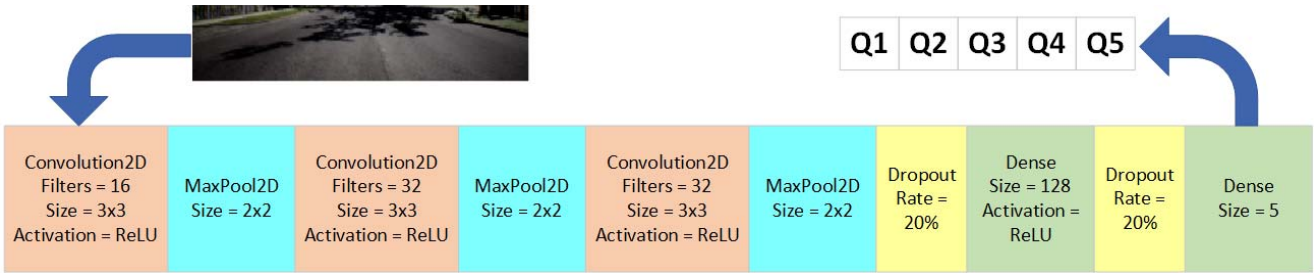

Figure 1: The architecture of our deep Q-network. The input is a cropped image from a front-facing web-cam on the car. The output is a set of five Q values corresponding to the five possible actions

to design an algorithm to generate the necessary control signals to drive a vehicle using only simple sensor data. While attempts to be made to solve this problem using supervised learning [2] [3], none have been totally successful, partially due to lack of an all-encompassing dataset. Deep reinforcement learning algorithms have shown promising results in some sub-problems related to autonomous driving such as lane assist [11] and highway navigation [13]. Although the ability to train on an unlabeled dataset is enticing, it comes at a steep cost. First, the learning process depends on the AI agent making costly mistakes. In the context of the autonomous driving problem, a mistake could lead to a crash. It is simply not feasible to be able to perform these tests in the real world. Instead, it is necessary to have a method of simulation in which the agent can safely make mistakes without incurring monetary cost or loss of life. Fortunately, high quality simulators are available [12], which help mitigate this issue.

Another big cost related to reinforcement learning is computation time. Reinforcement learning algorithms generally take much longer to train than their supervised learning counterparts. As an example, the original Deep Q-network (DQN) trained to play Atari games required millions of iterations before it converged, requiring run times of 12-14 days to train [8]. For a complex task like autonomous driving, the training can take much longer, even on the order of weeks or months. It is impractical to wait that long for a single model to train. The obvious solution is to attempt to somehow parallelize the process, cutting down the training time geometrically. It has been shown that with a simple parallelization strategy it is possible to decrease the training time for networks learning to play Atari games by up to a factor of five [10]. With cloud technology making massive virtual machine clusters widely available, this strategy can prove effective in decreasing training time and making deep reinforcement learning an effective strategy for solving the autonomous driving problem.

In this paper, we propose a solution for utilizing the cloud to improve the training time of a deep reinforcement learning model solving a simple problem related to autonomous driving. We describe a method of parallelizing the training work to decrease the training time, as well as some additional findings that served to further decrease the training time. We demonstrate that this approach achieves significant time savings over single-machine approaches, and also discuss some of the limitations to the scale-out model for increasing performance.

2 BACKGROUND AND METHOD

2.1 Deep Q-Network

Q-learning [16] [17] is one of the oldest and most popular reinforcement learning algorithms. In Q-learning, the world is divided into two parts: an agent and the surrounding environment. During each iteration of the training process, the agent is presented with a set of actions A, from which it selects one to perform. This action takes the agent from its current state S to a new state S'. As a result of selecting this action, the environment provides the agent with a reward $R(S, S', A)$. These actions are repeated until a terminal state is reached in which there are no further choices for actions which the agent can take. This marks the end of an *episode*, after which the agent is placed into a new random state and the training continues. For our problem, we define the state as a single RGB frame input from a front-facing web-cam on the car. Given this state information, the agent then takes the action of selecting a steering control signal from five possible values: *hard left*, *soft left*, *straight ahead*, *soft right* and *hard right*. Once the selection is made, the agent is then given a reward relative to its position in the environment. The details of the reward function are discussed in more detail in section 3.3.

In all but the simplest problems, it does not suffice for an algorithm to optimize purely for the reward function $R(S, S', A)$. The algorithm loses the ability to make decisions that are disadvantageous in the short term, but more viable in the long term. As an example, consider the action of a vehicle swerving to avoid hitting a pothole. In general, a sharp swerve is an undesirable behavior, and should be punished by the environment. However, if the agent purely optimizes for immediate reward, it will not swerve until it is too late to avoid the pothole. Thus, the algorithm needs to not only consider the reward from taking the current action, but also the expected reward from future iterations resulting from taking the current action. In Q-learning, this is modeled by the Q values. Q values are defined as:

$$Q(S, S', A) = R(S, S', A) + \lambda * \max_{A'}(Q * (S', S'', A')) \quad (1)$$

λ is a scalar in the range $[0, 1]$ which controls the amount of weight given to future predictions. At each iteration, the agent simply computes the Q value for each of the possible actions, and selects the action corresponding the largest. It has been shown [7] that for any environment that can be modeled as a finite Markov Decision Process, this strategy is proven to maximize the reward

received by the agent. Unfortunately, in most real-world problems, the state space is too large to directly compute the Q values, so they must be estimated by some means. In our algorithm, we utilize a deep neural network that accepts the state information and outputs the Q values for each of the possible actions. The architecture of the deep neural network used to compute the Q values is a variant of the Deep Q-Network (DQN) proposed in [9]. This model is a convolutional neural network with three convolutional layers, each separated by a max pooling layer. The network ends with two dense layers, with the final layer having one neuron for each Q value. A visual representation of the network used during training can be seen in Figure 1.

During training, we utilized two optimizations discussed in [9]: replay memory and the target network hack. One of the issues with directly using the data generated from the RL model is that the labels for successive data points are highly correlated. For example, if the model is taking a sharp left turn at time t, it is very likely to be continuing that turn at time $t + 1$. As episodes are relatively short, this creates a label distribution that is fundamentally different from the underlying distribution of labels. This makes it difficult for the model to generalize, and it instead oscillates back and forth between the different skewed distributions, never converging to the correct values. Like [9], we solve this issue by utilizing a replay memory. The replay memory is simply a circular buffer that stores data points collected during the agent exploration process. After each episode, the data points collected are inserted into the replay memory, overwriting the oldest data points. Then, the training data is generated by randomly selecting data points from the replay memory. This breaks the correlation between successive data points, allowing the model to learn the underlying distribution of labels.

When examining the Bellman Equation, we notice that there is a bit of a chicken-and-egg problem: to compute the Q values for any given state, we must first be able to compute the $Q*$ values for all the successive states. As we do not have data points for the future actions that the agent did not select, we cannot compute this value exactly. Instead, we must use our model to predict these values. When we use the same model to predict the $Q*$ values that we are training to predict the Q values, the model fails to converge. This is because each training iteration changes the $Q*$ values drastically, so the model is chasing a moving target. The solution proposed in [9] is to use a second copy of the active network called the target network to predict the $Q*$ values. The target network is updated periodically by copying the weights from the actively trained active network. This process decreases the variance in the $Q*$ values, which allows the model to converge.

When training any deep learning algorithm, there are two strategies that can be taken to exploring the space of optimal values. First, the agent can attempt to make very small changes to the current strategy, iteratively making small improvements, called *exploitation*. Second, the agent can attempt a radically different strategy, called *exploration*. Both modes of operation are necessary for training a robust model. Without exploration, the agent can converge to a suboptimal strategy. Without exploitation, the model itself will never improve, as we will be training on decisions made at random. In our approach, we balance these two conflicting goals using linear annealing. At the start of training, we favor exploration, because our model has not trained on a sufficient number of samples, and the predictions are likely to be meaningless. Over time, as the model improves, we decrease the amount of exploration and increase the amount of exploitation as the model improves. Towards the end of training, we are almost exclusively using an exploitative strategy, but we still occasionally choose to explore in case the model has converged to a suboptimal strategy.

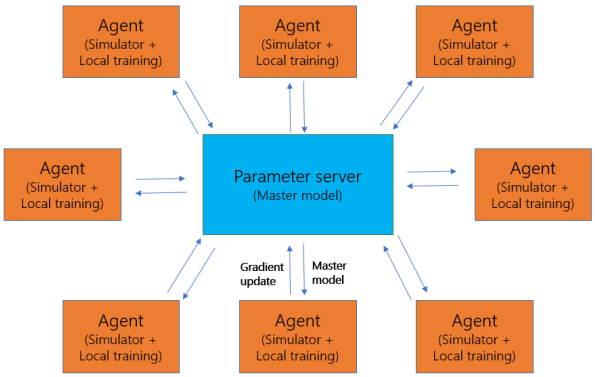

Figure 2: Our model distribution approach. Each agent runs a copy of the simulator as well as a local trainer. Periodically and asynchronously, agents send out gradient updates to the parameter server. The parameter server collects the incoming gradients, updates the copy of the master model and broadcasts it back to each agent.

2.2 Training Job Distribution

While reinforcement learning provides a huge advantage over supervised techniques in that it does not require labeled datasets, it poses another challenge: it needs a much larger amount data to train meaningful models. As we will see in section 3.2, although techniques like transfer learning [18] help improve the rate at which the model learns, accumulating close to 1 million iterations worth of training data still takes a lot of time. In addition, for more complex problems (like our autonomous driving use case), the number of training iterations needed increases dramatically, to the point at which it is impossible to train on a single node. To train the algorithm effectively, it is necessary to parallelize training across multiple machines. Thankfully with the availability of cloud resources, this is very achievable.

In our experiments, we tried a variant of the downpour stochastic gradient descent (downpour SGD) algorithm presented in [5]. Our job distribution paradigm is shown in Figure 2. We start with a pool of virtual machine nodes. At the start of training, one node is designated the *parameter server* node and all other nodes are designated as *agent* nodes. The parameter server is responsible for keeping the master copy of the model, accepting asynchronous updates from each of the agent nodes, and controlling the annealing rate. The agent nodes are responsible for running the simulator and performing local model training. After an agent completes an episode, it performs a training iteration on its local copy of the model with the collected data. Once the training completes, it then computes the change in weights of each of the layers of the model (the *gradient*). It then sends the gradient to the parameter server,

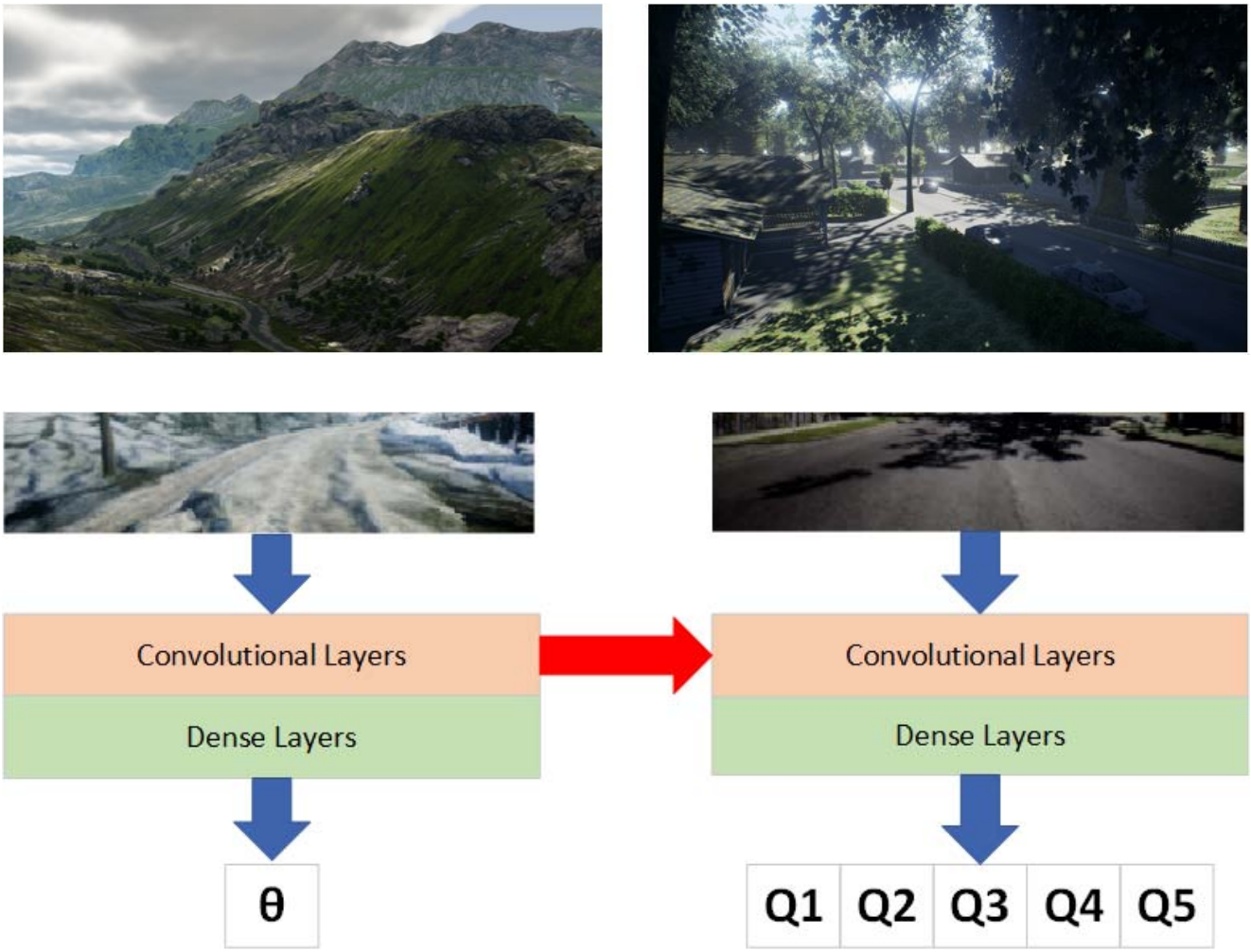

Figure 3: A graphical representation of the transfer learning process and the two environments involved. On the left is the mountain environment which was used for the supervised learning part. On the right is the neighborhood environment which the reinforcement learning model was trained on. You can see the vastly different textures, shapes and colors of the two roads. First, a model was trained on the mountain road in AirSim using labeled data collected by an expert human driver. Then, the weights from the convolutional layers were transfered to the reinforcement learning network and frozen. The model was then trained on roads in the neighborhood environment. The dense layers were trained from scratch.

and waits for a response. When the parameter server receives the gradient from the agent node, it adds the gradient to the master copy of the model, and then sends the updated model back to the agent node. This may be a different model than the agent currently has, as it may include gradients received from other nodes as well. This process repeats until the parameter server has received a set number of iterations.

3 EXPERIMENT DESIGN

3.1 Environment Details

We used Microsoft AirSim [12] as our simulator for the experiments presented here. In addition to having high-quality environments with realistic vehicle physics, it has a python API which allows for easy data extraction and control. This allowed us to perform all our modeling in a single language, simplifying implementation dramatically. We used the Keras [4] front-end with TensorFlow [1] back-end to architect and train our deep networks. The parameter server and agent communicate via HTTP requests, with the parameter server utilizing the Django framework to respond to asynchronous requests. Our experiments run on the Microsoft Azure cloud and utilize the NV-series virtual machines. These machines contain NVIDIA Tesla GPUs and are optimized for visualization tasks, which allow us to run our simulator to receive photo-realistic images for training. We used Azure Batch to manage the virtual machines and coordinate the distribution jobs.

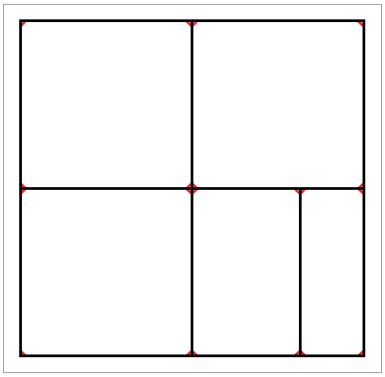

Figure 4: A schematic representation of the neighborhood map in AirSim. The black lines represent centers of the roads. In this representation, the car would be a point, and the reward function would be inversely proportional to the distance to the nearest line. All roads meet at 90° angles, which is difficult for the model to learn. After adding in the red "stubs" the vehicle was able to learn to navigate the 90° turns.

3.2 Transfer Learning

One of the biggest issues encountered when applying reinforcement learning to a non-trivial problem is that the algorithm takes many iterations to converge. For example, the DQN trained in [9] to play Atari games trained for 1 million frames before reaching convergence. For a more complex task like autonomous driving, the algorithm could require an exponentially larger number of iterations before reaching an acceptable level of accuracy. The amount of computational power required to generate the samples and train the algorithm on this vast quantity of data quickly becomes a bottleneck, preventing fast algorithmic iteration and improvement.

One of the methods that has been shown to increase the speed of convergence is transfer learning [15] [18]. In this scenario, transfer learning refers to training a deep network for a related problem, transferring the weights (and hence the learned representations) for the convolutional layers to the current problem, and fine-tuning the final few layers through deep reinforcement learning. We began this experiment by setting up a related supervised learning problem. We recorded an expert human driver's inputs as well as the frames from the simulated front-facing web-cam to create a labeled data set. The expert driver drove in a different environment than the environment in which we would be training the DQN (Figure 3). Then, we trained a deep neural network using an identical architecture to the DQN, aside from the final dense layer, to predict the expert's steering angle. This network took 43 minutes to train to convergence on a single-node machine. Once converged, we transfered the weights of the convolutional layers to the DQN, and only trained the final two dense layers during training. This brought down the training time dramatically: Instead of potentially billions of examples, the algorithm only required approximately 1 million examples, or 32,000 minibatches, to converge. Figure 3 shows a graphical representation of the training process.

3.3 Reward Function

The design of the reward function is critical to the success of the model. For our experiments, we decided to make the reward a function of the distance of the car from the center of the road. We computed the reward using the equation:

$$R(S, S', A) = \exp(-\beta * ||x_c - x_r||) \tag{2}$$

where x_c is the position of the car, x_r is the position of the road, and β is a positive scaling constant that controlled the shape of the function. This reward function has the attractive property that it is in the range $[0, 1]$, making it easier for the model to learn than an unbounded function like the raw distance.

Initial experiments also showed that the model had difficulty learning sharp turns. The car would frequently drive full speed at a sharp turn, and attempt to turn at the last minute, resulting in a collision. To navigate these turns, the model needs to look many stages into the future to begin to plan for the turn. This is difficult for our formulation of Q-learning because it only considers the current and next state when computing the Q values. To make it easier for the model, we created short road "stubs" that were positioned diagonally across the sharp turns. In Figure 4 we show an overhead map of the neighborhood environment. The central lines of the roads are shown in black, and the added road stubs in red. The path outlined by the stubs more closely models the path that an expert driver would take when navigating these turns, giving the model a more realistic signal to learn from. Also, the function gives a more gradual reward for completing the turn, allowing the algorithm to learn how to navigate these turns.

4 RESULTS AND DISCUSSION

During training, the flow of each episode looks as follows: To begin the training episode, the agent node requests the latest version of the model from the parameter server. The car is initialized with the current version of the model file (as sent by the parameter server). While driving, the car's goal is to minimize its distance from the center of the road and hence maximize the reward in equation 2. An episode ends when the car collides or after 30 seconds of collisionless driving. The 30 second limit is enforced to make sure the car is not driving around for too long as that would make the local copy of the model outdated. It also helps to avoid accumulating too much data which might overwhelm the training process and cause further latency in model updates.

In this experiment, we tracked two metrics: *throughput* and *time to automation*. Throughput is the average number of examples or data points our system is able to digest per hour. It is calculated as follows:

$$Throughput = \frac{\#\ of\ minibatches * minibatch\ size}{training\ end\ time - training\ start\ time} \tag{3}$$

This will increase as the number of nodes in the system are increased, as we achieve higher degrees of parallelism.

Time to automation is the average time it takes for the car to start driving around in the neighborhood environment in AirSim without any collisions. The timer for this metric starts when the experiment is initialized and concludes when the node on the network has an episode of at least 30 seconds without encountering a collision with

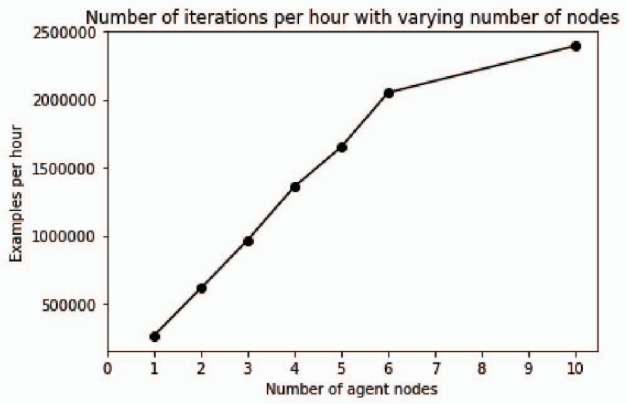

Figure 5: The first graphic shows the throughput of the system. At bottom graphic shows the time to automation for various learning rates and number of agent nodes. The 10 agent system never reached full automation, so it is omitted from this plot.

the environment. We utilized the transfer learning process described in section 3.2 to initialize our reinforcement learning training runs with learned representations from the mountain environment in AirSim. We ran experiments with varying the number of agent nodes from 1 to 6. In addition to varying the number of nodes, we also varied the annealing rate between a fast ($\Delta\epsilon$=-0.003) and a slow ($\Delta\epsilon$=-0.0003) value. We also performed experiments with 10 agent nodes at both the fast and slow annealing rates. For comparison, as a baseline benchmark, we also ran the code on a single agent system, but initializing all layers of the model randomly, that is, without utilizing the transfer learning process.

In Figure 5, we see the throughput and the time to automation as functions of the number of agent nodes. For clarity, we have left off the experiment without transfer learning from these graphics. We have also left out the experiment with 10 nodes from the second graphic as it never converged. The rate of increase of the throughput with respect to the number of nodes of our system is approximately linear. Intuitively, this makes sense - as all agent nodes are acting independently in our system, each agent will produce approximately the same amount of data per given time interval. The system does not scale indefinitely, however. We notice that once we start increasing the number of nodes beyond 7, the throughput tapers off. This is because our system only allows for a single parameter server node - it is simply getting overwhelmed with processing the vast amounts of incoming data produced by the agent nodes.

Even more surprising, however, is the fact that increasing the throughput of the system does not necessarily decrease training time. For the first few additional agent machines, we observe a dramatic drop in training time. However, surprisingly quickly we discover that adding additional machines increases the time to automation. We notice that when using a slower annealing rate, the number of machines required for optimal time to automation increases, but we still observe that arbitrarily increasing the number of agent machines leads to increased time to automation. In both cases, the 10 agent systems did not ever reach automation, even after 20 hours of training.

The reason for this divergence is that during the training process, each agent machine is working on a saved local copy of the latest model in memory. While the agent is training, it is possible that many updates are performed between the time that the agent starts an episode and the agent ends an episode. By the time the agent computes its gradient, the model's parameters may have changed dramatically. So, when the parameter server node integrates the gradient received from the agent, the gradient does not shift the model parameters in the proper direction because the model parameters in the latest model are so different from the model parameters used by the agent. For systems with a small number of agent nodes, it is unlikely that many iterations have been performed between the start and end of an episode, and the training time improvement from increased throughput is enough to overcome the training time regressions caused by using outdated models. As more nodes are added to the system however, the situation of an agent using outdated models becomes more likely, causing the training to diverge. The increased throughput is not enough to compensate for this issue, and the model takes longer to converge, if it converges at all. Potential solutions have been proposed [19] [10], but are yet to be verified for this specific scenario.

When trained without using transfer learning, the single agent system took 140 hours to reach full automation. When trained with transfer learning, it took 5 hours, a dramatic improvement. This is interesting because the training dataset used for the transfer learning experiment was generated from a fundamentally different environment than the environment used for training the reinforcement learning model. As can be seen in Figure 3, the road in the mountain environment has a very different shape and color from the road in the neighborhood environment used for reinforcement learning. The surrounding scenery is also different. The fact that the weights in early layers can be effectively re-used suggests that during the supervised learning process, the model has learned a very general representation of the input that can be applied to a variety of environments. This also suggests a process for training models when data is only available for a subset of the regions in which they will operate - train a supervised learning model on the

labeled data, and then use reinforcement learning to fine-tune these models to operate in the other regions. This balances the manual time needed to collect labeled data with the computational time needed to train reinforcement learning algorithms to ultimately minimize the model development time.

5 CONCLUSION

In this paper, we proposed an architecture for leveraging cloud computing technologies to decrease the training time for deep reinforcement learning models for autonomous driving. We showed that by parallelizing the training process, one can achieve a massive decrease in the training time. In addition, we showed that by carefully designing the reward function and by utilizing transfer learning, we are able to further decrease the training time (from 140 hours to automation on a single machine using random weights to less than 1 hour on a 5-agent cluster with transfer learning). This demonstrates that application of these powerful models to real world problems is more feasible now than ever before. While our proposed system drastically decreases training time, we also showed that it does not scale infinitely, and further architectural work is needed to allow the system to scale to massive clusters. Solving the problem of training massive machine learning models at scale is a necessary prerequisite to solving the Autonomous Driving problem, and the proposed framework brings us one step closer to having the ability to train an algorithm to drive a fully autonomous vehicle.

ACKNOWLEDGMENTS

The authors would like to thank everyone who helped bring these ideas to life through discussions and feedback. The authors would also specifically like to thank Kurt Niebuhr and the Microsoft Azure Batch team for providing them with the computational resources needed to run these experiments.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. *CoRR* abs/1604.07316 (2016). arXiv:1604.07316 http://arxiv.org/abs/1604.07316

[3] Chenyi Chen, Ari Seff, Alain L. Kornhauser, and Jianxiong Xiao. 2015. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. *2015 IEEE International Conference on Computer Vision (ICCV)* (2015), 2722–2730.

[4] François Chollet et al. 2015. Keras. https://github.com/keras-team/keras. (2015).

[5] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1223–1231. http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[6] Nidhi Kalra and Susan M. Paddock. 2016. Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability? (2016). https://www.rand.org/pubs/research_reports/RR1478.html

[7] Francisco S Melo. [n. d.]. Convergence of Q-learning: A simple proof. ([n. d.]).

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari With Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (Feb. 2015), 529–533. http://dx.doi.org/10.1038/nature14236

[10] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. 2015. Massively Parallel Methods for Deep Reinforcement Learning. *CoRR* abs/1507.04296 (2015). arXiv:1507.04296 http://arxiv.org/abs/1507.04296

[11] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani. 2016. End-to-end deep reinforcement learning for lane keeping assist. (2016). https://arxiv.org/pdf/1612.04340.pdf

[12] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. 2017. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. In *Field and Service Robotics*. arXiv:arXiv:1705.05065 https://arxiv.org/abs/1705.05065

[13] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. 2016. Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving. *CoRR* abs/1610.03295 (2016). arXiv:1610.03295 http://arxiv.org/abs/1610.03295

[14] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. 550 (10 2017), 354–359.

[15] Matthew E. Taylor and Peter Stone. 2009. Transfer Learning for Reinforcement Learning Domains: A Survey. *J. Mach. Learn. Res.* 10 (Dec. 2009), 1633–1685. http://dl.acm.org/citation.cfm?id=1577069.1755839

[16] Christopher John Cornish Hellaby Watkins. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation. King's College, Cambridge, UK. http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

[17] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. In *Machine Learning*. 279–292.

[18] Karl Weiss, Taghi M. Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *Journal of Big Data* 3, 1 (2016), 1–40. https://doi.org/10.1186/s40537-016-0043-6

[19] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2015. Staleness-aware Async-SGD for Distributed Deep Learning. *CoRR* abs/1511.05950 (2015). http://arxiv.org/abs/1511.05950