Learning Robust Motion Planning for Large-Scale Deployment of Autonomous Driving Agents

Sahika Genc [*] **Sunil Mallya** [*] **Bharath Balaji** **Tao Sun** **Saurabh Gupta** **Vineet Khare** **Leo Dirac** **Eddie Calleja** **Brian Townsend** [1]

Abstract

Large-scale deployment and maintenance of industrial optimal control applications require expert guidance and maximally informative system identification in the design stage. We propose an alternative method to large-scale deployment of industrial control applications using robust deep reinforcement learning. Despite achieving almost human-level skills, the ability of deep reinforcement learning to outperform traditional advanced control approaches such as model predictive control at scale remains largely unproven. Our contribution is in three-folds: 1) Formulation and solution of a robust reinforcement learning algorithm, 2) narrowing the reality gap through joint perception and dynamics, and 3) distributed on-demand compute architecture for training optimal policies. Finally, we describe a self-driving vision-based robotic hardware and simulation environment, namely, Amazon Web Services DeepRacer and SageMaker Reinforcement Learning, respectively, to evaluate our contributions.

1. Introduction

Recently, it has been demonstrated that even one percent improvement in the performance of the industrial machinery at scale can result in a cumulative increase in throughput and savings. It is often difficult to achieve these operational improvements without optimal control and learning algorithms. However, developing, deployment, and maintenance of deep reinforcement learning algorithms *at scale* for industrial control applications is a challenging problem. Over the past decade, there have been significant advancements in the area of deep learning and reinforcement learning. Together, these advanced methods are capable of learning locomotion, perform motion planning, and learn to manipulate of robotic arms without human guidance. Despite achieving almost human-level skills, the ability of deep reinforcement learning to outperform traditional advanced control approaches such as model predictive control at scale remains largely unproven.

Most of the recent examples in deep reinforcement learning of autonomous control agents utilize realistic simulation environments. Yet, a controller learned in a simulation environment performs poorly when there is a mismatch in the simulation and real-world models. These discrepancies arise from a variety of reasons including but not limited to uncertainty in system parameters, unknown or unmodeled dynamics, and sensor measurement errors. While there are methods to improve the fidelity of the simulation environment and perception (see Section 2), the simulation-to-real experiments have mostly been successful only on a handful of platforms with limited variation in the physical and computational hardware.

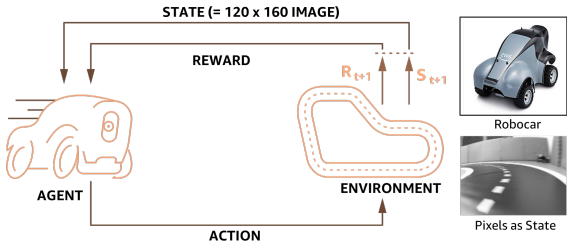

Figure 1. AWS DeepRacer robocar platform for large-scale deployment of deep reinforcement learning policies trained in simulation and transferred to the real world.

In this paper, we describe a large-scale deep reinforcement learning application to the control of robotic agents in the context of self-driving cars. We wish to promote adoption of deep reinforcement learning to the industrial controls, hence, we consider a real world controls application that mimic challenges in perception, unmodeled dynamics, and variability in mechanical and electrical components. Our vision-based robot car, AWS DeepRacer (Services, 2019c),

[*]Equal contribution [1]Amazon Web Services Artificial Intelligence Lab., Seattle, San Francisco, USA. Correspondence to: Sahika Genc <sahika@amazon.com>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

a 1:18th scale vision-based self-driving car as shown in Frigure 1, was introduced in late 2018. Since then, there have been hundreds of robot cars utilized across the world at more than 5 summits. More than 15 summits and events at conferences[1] are still to be completed.

To the best of our knowledge, AWS DeepRacer is the first successful *large-scale* deployment of deep reinforcement learning on a robotic control agent that uses only raw camera image as observed state and a model-free learning method to perform optimal path planning, and trained in simulation with no additional tuning in physical world. Our contribution is in three-folds: 1) Formulation and solution of a robust reinforcement learning algorithm, 2) narrowing the reality gap through joint perception and dynamics, and 3) distributed on-demand compute architecture for training optimal policies.

2. Motivation and Related Work

We now examine in more detail the problems with the current simulation-to-real transfer methods and consider three main challenges to enable large-scale deployment of deep reinforcement learning. The main contributions of this paper are aligned to solve the following three challenges for a general class of systems and then use self-driving robot car application as a means to demonstrate capabilities for large-scale deployment of deep reinforcement learning algorithms on industrial control applications.

Challenge 1 - System identification: *Policies trained in one simulation environment with a specific hardware model configuration (e.g., friction constants, actuator latency) transfer poorly in the same simulation environment with a new configuration.* Policy optimization methods with parametrized policies provide the best results in simulation environments. These methods do not require modeling the dynamics of the system to determine the optimal policy. On the other hand, when the simulation fidelity is low and contain unmodeled dynamics, the policy trained in the simulation perform poorly in the real-world. There have been several approaches to solve the simulation-to-real problem for specific applications. In (Tan et al., 2018), improve the fidelity by modeling actuator dynamics and latency through system identification. In (Rusu et al., 2016), the authors train in simulation and tune in physical world. Improving the simulation fidelity using system identification is difficult because it requires measuring inertia or design of experiments to measure friction constants for the mechanical components.

Challenge 2 - Perception: *Policies trained in a specific simulation environment perform poorly in another environment with the same hardware model configuration.* The policies in simulation are trained without sensor models that mimic the measurement errors and uncertainty in the simulation environment. That is mainly because the parametrized stochastic policy does not depend on the plant dynamics which in traditional control theory will incorporate sensor models for state measurement errors. Therefore, the samples used to the train the policy should come from an environment with these discrepencies and uncertainty in the measurements already baked into the simulation environment. However, injecting noise, perturbing the system, using domain randomization results in longer training times. In (Tzeng et al., 2015), the authors apply domain adaptation at the feature level. In (Tobin et al., 2017) and (Mordatch et al., 2015) , the authors used domain and dynamics randomization, respectively. In [DBLP:conf/icml/HigginsPRMBPBBL17], the authors propose a new multi-stage RL agent, DARLA (DisentAngled Representation Learning Agent), which learns to see before learning to act. DARLAs vision is based on learning a disentangled representation of the observed environment.

Challenge 3 - Image as state: *Most traditional approaches to control robotic agents rely on extracting features from image or using non-image based sensors for state measurements instead of raw images as observed state measurements.* One of the hallmarks of the human-level intelligence is hand-eye coordination. Today, humans are able to drive cars with only visualizing their environment. While modern cars include proximity sensors, it is now common to include cameras in the perimeter of the car to assist backing out of a parking spot or driveway or alarming when the car is rearing outside of the lane lines. However, there are limited theoretical studies on designing control systems with raw image (pixels) as a state. The stability and robustness of these new types of controllers are mostly unknown for general class of systems. In (Bousmalis et al., 2017), the authors proposed a method for domain adaption for reinforcement learning models with observed state at pixel level.

In the next section, we provide the problem formulation for robust deep reinforcement learning and our solution approach to address Challenges 1 and 2. Due to limited space, we address our solution approach to Challenge 3 in a companion paper to be submitted to NeurIPS'19. We formulate our problem using concepts from well-understood robust control theory through an augmented reward function based on H_∞ control goal. We base our solution approach on likelihood ratio policy gradient method and derive the new gradient equations with respect to the augmented reward function and parametrized policy and environment. We assume the parameters of the policy and environment are independent which results in obtaining an equivalent of separation principle in classical control theory.

[1]Confirmed races at conference include AI Olympics at NeurIPS'19 Live Challenges in collaboration with Duckietown Organization (Paull et al., 2017) and IEEE CASE'19 Tutorials.

3. Problem Formulation

We consider a discrete-time formulation of reinforcement learning where $P(.|s,u)$ and $r(s,u)$ are the state transition distribution and reward, respectively, upon taking action u in state s. A policy $\pi = P(u|s)$ is a mapping from the states to a probability distribution over the set of actions. Suppose that the state transition distribution and policies are parametrized by vectors $\psi \in \mathbb{R}^k$ and $\theta \in \mathbb{R}^n$, respectively, where k, n are finite integers. The parameter vector ψ is related to unknown components of system dynamics or kinematics. In prior work, the authors [sim2real] apply system identification to determine these parameters for successful transfer of policies trained in simulation to the real world.

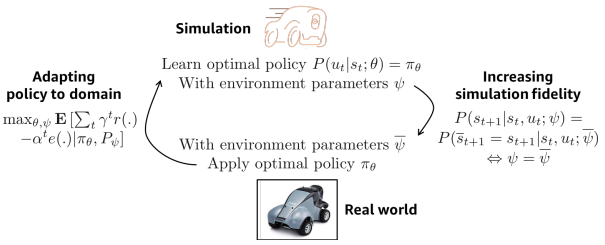

Figure 2. The cycle of robust reinforcement learning approach that accomodates for joint perception and dynamics learning to narrow the reality gap.

The basic goal in reinforcement learning is to maximize cumulative discounted reward over a time horizon. We assume a finite time horizon. A policy trained in simulation will more likely to transfer to the real world when the dynamics in simulation matches those in the real world (e.g., mass and friction of a vehicle) and the simulation environment is randomized to reflect sensor noise (e.g., texture of road) as illustrated in Figure 2. To improve robust performance in changing environments, we introduce the notion of worst error in state measurements in control theory to the reinforcement learning problem. This is similar to the notion of worst case disturbance in H_∞ in control theory and its formulation in the reinforcement learning paradigm (Morimoto & Doya, 2005).

Formally, we describe the reinforcement learning problem with augmented reward as

$$\max_{\theta,\psi} \mathbf{E}_{P(\tau;\theta)}\left[\sum_{t=0}^{T} \gamma^t r(s_t,u_t) - \alpha^t e(s_t,\overline{s}_t)|\pi_\theta, P_\psi\right] \quad (1)$$

where $e(s_t,\overline{s}_t)$ denotes the penalty for errors in observed states in simulation s versus real world $\overline{s}$ and $\gamma, \alpha \in (0,1]$ are discount factors. Note that the introduced penalty term is flexible enough to capture both system parameter estimation errors as well as measurement errors. Therefore, the formulation can be used to study to automatically learn the system parameters that results in a good transferable policy as well as train an optimal policy that generealizes to a class of systems with uncertainty in parameters. In addition, by introducing a discount factor α to the state estimation error, we can force the estimator to focus on the short-term or long-term errors.

4. Robust Reinforcement Learning with Augmented Rewards

To solve the augmented reward reinforcement learning problem for robust optimal policy optimization, we denote the expected return in Equation 1 as $U(\theta,\psi)$. To simplify, we overload the notation for the reward and error functions to obtain

$$U(\theta,\psi) = \sum_{\tau} P(\tau;\theta,\psi)R(\tau) - P(\overline{\tau}|\tau;\theta,\psi)E(\tau) \quad (2)$$

where $\tau = (s_0,u_0,s_1,u_1,\ldots,s_T,u_T)$ and $\overline{\tau} = (s_0,u_0,\overline{s}_1,\overline{u}_1,\ldots,\overline{s}_T,\overline{u}_T)$ is the discount factors and traces are rolled into $R(\tau)$ and $E(\tau)$.

There are two main methods to solve the above reinforcement learning problem; 1) value approximation and 2) policy approximation methods. In (Morimoto & Doya, 2005), the authors describe a method based on value function augmentation and then its optimization using a function approximator. We explore a policy gradient approach which is better equipped to handle the nonlinear nature of the dynamics for the vehicle model for autonomous racing application. However, the vehicle dynamics model can be linearized for a specific speed to derive a value approximator.

We consider likelihood ratio policy gradient method to perform a stochastic gradient ascent over the policy and state transition distribution parameter space $[\theta,\psi]$ to find a local optimum of $U(\theta,\psi)$. We derive the equations for the gradients with respect to θ and ψ following the derivations in [cite 6,7 REINFORCE]:

$$\begin{aligned}\nabla_{\theta,\psi}U(\theta,\psi) \quad &\approx \quad \frac{1}{m}\sum_{i=1}^{m}\Big(\nabla_{\theta,\psi}\log P(\tau^{(i)};\theta)R(\tau^{(i)}) \\ &- \quad \nabla_{\theta,\psi}\log P(\overline{\tau}^{(i)}|\tau^{(i)};\theta,\psi)E(\tau^{(i)})\Big)\end{aligned} \quad (3)$$

where m sample paths obtained from acting under policy π_θ. We derive the gradients using Markov assumption, conditional probability property, independence of the state transition distribution on θ as follows

$$\nabla_\theta \log P(\tau^{(i)};\theta,\psi) = \sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)}) \text{ and} \quad (4)$$

$$\nabla_\theta \log P(\overline{\tau}^{(i)}|\tau^{(i)};\theta,\psi) = 0 \text{ (see Section A)} \quad (5)$$

The gradient with respect to the second summation term in Equation 3 disappears because the actuator inputs are already known from the path τ (for derivation see Section A). Therefore, the problem reduces to the original likelihood ratio policy gradient when the simulation modeling error is ignored.

Next, we consider the gradients with respect to ψ. The gradient for the first term in Equation 3

$$\nabla_\psi \log P(\tau^{(i)}; \theta, \psi) = \sum_{t=1}^{T} \nabla_\psi \log P(s_{t+1}^{(i)} | s_t^{(i)}, u_t^{(i)}) \quad (6)$$

To derive the gradient for the second term in the summation in Equation 3, we assume that with a probability p, the actual state $\overline{s}$ is equal to the simulation state s. Thus, the states in the actual and simulation differ with probability $1-p$. We ignore that the potential dependency of the probability to frequency domain, i.e., in general, the sensor noise manifests as higher magnitudes at higher frequencies. Then,

$$P(\overline{s}_{t+1} | s_t, u_t; \psi) = 1 - p + (2p-1) P(s_{t+1} | s_t, u_t; \psi) \quad (7)$$

as derived in Section A and the gradient with respect to the simulation parameters becomes

$$\nabla_\psi \log P(\overline{\tau}^{(i)} | \tau^{(i)}; \psi) = \sum_{t=1}^{T} (2p-1) \nabla_\psi \log P(s_{t+1}^{(i)} | s_t^{(i)}, u_t^{(i)}) \quad (8)$$

The augmented robust deep reinforcement learning algorithm is outlined in Algorithm 1. The algorithm alternates between optimizing the policy in the simulation environment given a specific system environment parameters in the simulation and optimizing the environment model given a sub-optimal policy in the real world. The algorithm is stopped when the error in the state estimation is within required limits for large-scale deployment. In the following, we evaluate the performance of the algorithm with respect to the current state-of-art based on system identification and domain adaptation.

Algorithm 1 Augmented Robust Deep Reinforcement Learning (ARDRL)

Initialize policy parameter θ, environment parameter ψ, baseline b
Optimize policy for a given environment
for iteration=$1, 2, \ldots$ **do**
  Collect trajectories by executing the current policy in simulation
  At each timestep in each trajectory, compute
    the augmented return $R_{aug} = \sum_{t=0}^{T} \gamma^t r(s_t, u_t) - \alpha^t e(s_t, \overline{s}_t)$
    the augmented advantage estimate
  Re-fit the augmented baseline by minimizing an L_2 norm
  Update the policy parameters, using a policy gradient estimate
end for
Optimize simulation for a given policy
for iteration=$1, 2, \ldots$ **do**
  Collect trajectories by executing the current policy in real world
    the augmented return
    the augmented advantage estimate
  Re-fit the augmented baseline by minimizing an L_2 norm
  Update the environment parameters, using a policy gradient estimate
end for

5. Robot Platform and Physics Simulation

The AWS DeepRacer is a 1/18th scale robocar (Services, 2019c). The robocar mechanical platform uses Ackermann drive for steering and Pulse Width Modulation (PWM) for steering and throttle. This is an affordable, low-cost ($399/robot). The platform was made available for preview at re:Invent in 2018 where thousands of people had access to hundreds of robocars without any hardware failures. For a summary of specification of the platform, see (Services, 2019c).

We use an Intel Atom processor with a built-in GPU to perform neural network inference locally using Intel's OpenVino. There is a separate battery and heat-sink to support the compute platform. The car chasses is based on an off-the-shelf toy car. Therefore, the variability of the mechanical components is higher than most industrial control systems. The motors are equipped with electronic speed controllers and driven by a separate servo control board. The car chassis has its own battery for locomotion. The 13600 mAh compute battery easily lasts more than 5 hours, allowing for convenient experimentation. The 1100 mAh drive battery last for 2 hours in typical experiments. The on-board WiFi chip enables convenient monitoring and remote programming. The camera is 4 MP camera with MJPEG but captures 160x120 pixel images for our self-driving application.

We built the car software on top of Robot Operating System (ROS) (Quigley et al., 2009). We load the trained models either using a USB drive or over the network. At a high level, the car performs neural network inference with camera images as input and control the motors that drive the car. We use Intel OpenVino to convert our Tensorflow models to an optimized binary for fast inference time with the GPU. The fast inference is critical in our application as the goal is

to complete a lap faster than other competitors.

To allow stable control at faster speeds, the sampling rate of the controller needs to be high enough to be able to take corrective actions. In classical control theory, the rule of thumb is to have observations 3-5 times faster than the controller sampling rate. This is a significant requirement in our case because in some ways our neural network will also be performing state estimation from raw images. Any lag in estimation will create instability or poor performance in policy transfer. The camera images are fed to the OpenVino inference engine and the web server that allows the user to view real-time video feed from a browser. The model inference results are converted to motor control commands based on the calibration and action space mapping. The action probabilities of the model inference are also shown on the web browser interface. Lastly, the browser has an interface for manual joystick like control.

We built the car assets including camera, actuators, and Ackerman driving models in Gazebo (Koenig & Howard, 2004). Gazebo consists of several components that work together to create a high fidelity robotics simulation. A *robot model* describes each component of the DeepRacer car - the chassis, wheels, camera - their dimensions, how they link with each other, their properties such as mass, field of view of the camera. We specify the model using the Unified Robot Description Format (URDF) and our model is inspired from the MIT Racecar (Karaman et al., 2017). We create our tracks and background environment using a 3D modeling software, and import these assets into Gazebo.

We use the ROS for communication between the simulation and the agent. ROS is a collection of tools and libraries for robotics applications. It provides a language agnostic publish-subscribe messaging mechanism that enables communication between various components in a robot. Hence, we can communicate between the Gazebo simulation plugins written in C++ with our agent model written in Python. The agent uses ROS to place the car in the track at the beginning of an episode, get images from the camera module, get the car's position, velocity, and send throttle, steering commands to control the car. Our message sizes are small and ROS messaging is extremely fast (Maruyama et al., 2016), hence we ignore any latencies incurred by ROS.

6. Distributed Reinforcement Learning Computational Platform

Our robocar platform uses AWS SageMaker RL to train the neural network model and AWS Robomaker (Services, 2019a) to simulate the environment. AWS SageMaker is a platform to train machine learning models in the cloud. RoboMaker is a cloud service to develop, test and deploy robot software. We create our simulation environment, i.e. track, car, etc., in RoboMaker. We initialize our agent, i.e. our neural network model, in AWS SageMaker and store a persistent copy in Amazon S3, an object store service. We initialize our environment in RoboMaker and copy the agent model from S3. The agent interacts with the environment and we store the experience data collected in Redis, an in-memory database. AWS SageMaker trains the neural network with data collected in Redis and updates the neural network models in S3. RoboMaker copies the model from S3 and creates more experience data. The cycle continues until training stops. The same setup allows us to launch multiple simulations to train faster or train the agent to traverse multiple tracks simultaneously. We next describe AWS SageMaker and RoboMaker in more detail.

AWS SageMaker is a platform to train and deploy machine learning models at scale using Jupyter Notebook as interface. It integrates with popular libraries such as scikit-learn (Pedregosa et al., 2011), frameworks such as TensorFlow (Abadi et al., 2015) and eases development with tools for hyper-parameter optimization and web service based hosting of trained models. AWS SageMaker integrates RL algorithms like PPO using Intel RL Coach (Caspi et al., 2017) and Ray RLlib (Liang et al., 2018) libraries that build on top of existing deep learning frameworks. The algorithms integrate with environments using the Open AI Gym (Brockman et al., 2016) interface which specifies a standard way to define the state, action, reward and episode so that agent can interact with the environment. A number of simulators such as PyBullet for robotics, EnergyPlus (Crawley et al., 2001) for buildings and Matlab for industrial control are supported by AWS SageMaker. For DeepRacer, we use the RL Coach library that implements the PPO algorithm. We develop our simulation environment with RoboMaker and use OpenAI Gym to interface with the Coach library.

7. Evaluation and Discussion

The AWS DeepRacer robocar and training console is available to general public. The racers have been heavily utilizing the platform and sharing accompanying videos[2]. Links to the most recent videos from racers around the world are provided in (Services, 2019b). The simulated environments and deep reinforcement learning algorithms are open-source (see (**?**) and (**?**)). The racers can either use the AWS DeepRacer console or AWS SageMaker notebooks to train deep reinforcement learning models.

In the AWS DeepRacer console, the racers can edit the action space for the robocar and engineer their own reward function. In addition, in the AWS SageMaker RL Jupyter notebook, the racers can edit the gym environment file which provides more flexibility such as accessing the

[2]Search for #AWS DeepRacer on digital media

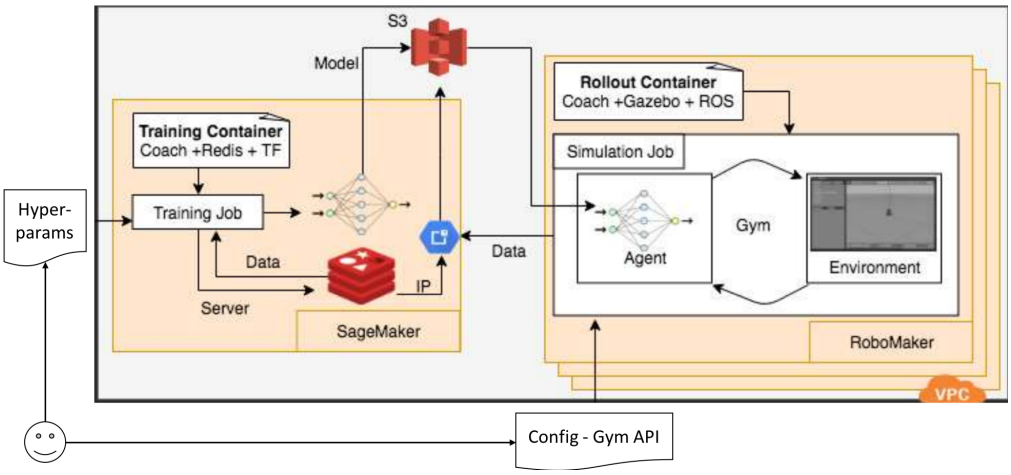

Figure 3. AWS DeepRacer computational platform integrating AWS RoboMaker as the simulation services and AWS SageMaker as the computational training platform.

Gazebo assets to create new variables for reward function or add probabilities. In addition, the racers can change the neural network architectures for the policy and value function approximators. The default is a five layer network with dense input and output embedders and a 3-layer convolutional neural network forming the middle layer. The Jupyter notebook points to the AWS RoboMaker application. For racers who desire to edit the Gazebo assets by implementing their own car models and tracks can do so as long as the gym environment variables to create the reward function and calculate state transitions remain consistent. The training computational resources are allocated to achieve 90% lap completion with models trained for an hour with default settings which uses a "c4.large" EC2 instance.

When testing the robocar on the real world, the racers can increase or decrease the speed of the car through the web console. There is no additional fine-tuning on the physical system. The deep reinforcement learning models downloaded from the AWS SageMaker RL are uploaded to the robocar via a USB stick and optimized using OpenVino only when selected for autonomous driving. We evaluated the average lap-time across to be approximately 20 seconds, resulting in 0.9 meters per second. However, the top five racers at the last 5 events across U.S., Asia, Australia, and other locations of the recent 5 summits achieved less than 9 second lap- time over the approximately 18- meter track, resulting in average velocity of 2 meters per second. This is close to the lap-time when model predictive control is used with dynamic or kinematic difference equations.

We now describe our initial experiments prior to the large-scale deployment of our platform, and compare our robust reinforcement learning algorithm to the current state-of-the-art. First, we performed experiments to narrow the reality gap via system identification (e.g., mass, friction) and randomization in the environment (e.g., carpet versus wood floor as track background). We used proximate policy optimization to train our model with this higher-fidelity simulation environment. Second, we implemented our robust reinforcement learning algorithm in Algorithm [] in RL Coach in a lower-fidelity simulation environment with varying frictional constants and intertia. Finally, to examine the improvements to the robustness of new policies, we compare two sets of metrics: 1) A continuous measure to quantify the success rate, i.e., expected return from simulation and real world experiments, and 2) the number of iterations to reach a specific expected return with domain and parameter randomization versus robust learning approach.

Challenge 1 - System identification. Many published work on model predictive control of self-driving cars use dynamic vehicle models based on a simplistic bicycle model. The past research in vehicle dynamics recommend a dynamic model with inertial and frictional constants to perform system identification. However, in a recent paper [], the authors compared the dynamic bicycle model to the kinematic one, and discovered that the kinematic model can replace the dynamic one at lower speeds for actual cars. Since our car is a 1:18 scale one, we focused on the parameters of the kinematic model for system identification. The parameters of the kinematic model are smaller, hence, enabling a quick turnout. Our extensive experiments combined with the intuition of using the kinematic model, led us to determine wheel/ground interaction via frictional constants instead of developing a linear tire model with unmodeled dynamics. We tested more than 100 policies trained in the simulation environment with varying reward functions and action space, and none of these transferred policies worked with the wrong frictional constants. *We hypothesize that our robust reinforcement learning approach will be able*

to identify the frictional parameters to adjust for optimal policy transfer to the real world.

Challenge 2 - Perception. To improve the perception, we focus on the middle layer neural network architectures. To determine the depth, convolution size, and other hyper-parameters of the neural network, we conducted a set of experiments. In our experiments, we trained models with varying reward functions that encode behaviors such as stay away from white lines versus follow yellow dotted line. Then, we collected images from the real track while the robocar run with the corresponding model. We used GradCAM (Selvaraju et al., 2017) on the real images from the physical world to compare the perception to control objective. The basic idea for GradCAM is that since the last convolution layer conveys high-level semantics and detailed spatial information, the weighted sum of gradient information of the feature maps from the last convolution layer illustrates which parts of the input image "activate" the major concept in the output. A GradCAM view of the deep reinforcement learning modeled trained with reward function to keep the car within 5 cm of the centerline of the track is shown in Figure 4.

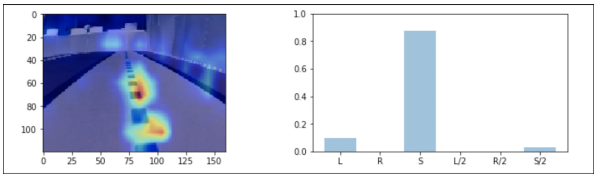

Figure 4. (Left)The GradCAM on the real images from the reward function that promotes keeping the car within 5 cm of the centerline of the track. (Right) The action probabilities generated by the model given the real image on the left.

Robust Deep Reinforcement Learning. In progress and to be completed by May 15th for systems other than self-driving.

8. Conclusion

The focus of this paper is on development of robust deep reinforcement learning algorithms for large-scale deployment of optimal industrial control applications. We demonstrated best practices in control system design and used the corresponding results as baseline for our robust learning algorithm. We described flexible end-to-end platform to build, train, and deploy deep reinforcement learning models on a vision-based robocar to hundreds of racers.

Acknowledgements

The authors would like to thank Edo Liberty, Ph.D. for discussions on theory and application.

A. Supplementary Material

We do not have dependency on the policy because the policy result is already in the trace given from the simulation. Mathematically, through Markov property and conditional probability expansions we derive:

$$P(\overline{s}_T\overline{u}_T \ldots | s_T u_T \ldots; \theta, \psi) = \prod_{t=1}^{T} P(\overline{s}_{t+1}|s_t, u_t) \quad (9)$$

To see this result consider $T = 1$,

$$\begin{aligned} & P(\overline{s_1 u_1} s_0 u_0 | s_1 u_1 s_0 u_0) = P(\overline{s_1 u_1} | s_0 u_0 s_1 u_1 s_0 u_0) \\ = \; & P(\overline{s_1} | \overline{u_1} s_1 u_1 s_0 u_0).\, P(\overline{u_1} | s_1 u_1 s_0 u_0) \\ = \; & P(\overline{s_1} | s_0 u_0) \end{aligned}$$

and iterate for $T > 1$. The logarithm operation turns the product to summation

$$\log P(\overline{\tau}^{(i)} | \tau^{(i)}; \theta, \psi) = \sum_{t=1}^{T} \log P(\overline{s}_{t+1} | s_t, u_t; \psi) \quad (10)$$

Derivation of Equation 7

$$\begin{aligned} P(\overline{s}_{t+1}|s_t, u_t; \psi) \quad &= \quad pP(s_{t+1}|s_t, u_t; \psi) + \\ & \quad (1-p)(1 - P(s_{t+1}|s_t, u_t; \psi)) \\ &= \quad (1-p) + (2p-1)P(s_{t+1}|s_t, u_t; \psi) \end{aligned}$$

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Bousmalis, K., Irpan, A., Wohlhart, P., Bai, Y., Kelcey, M., Kalakrishnan, M., Downs, L., Ibarz, J., Pastor, P., Konolige, K., Levine, S., and Vanhoucke, V. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. *CoRR*, abs/1709.07857, 2017. URL http://arxiv.org/abs/1709.07857.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI Gym, 2016.

Caspi, I., Leibovich, G., Novik, G., and Endrawis, S. Reinforcement Learning Coach, December 2017. URL https://doi.org/10.5281/zenodo.1134899.

Crawley, D. B., Lawrie, L. K., Winkelmann, F. C., Buhl, W. F., Huang, Y. J., Pedersen, C. O., Strand, R. K., Liesen, R. J., Fisher, D. E., Witte, M. J., et al. EnergyPlus: creating a new-generation building energy simulation program. *Energy and buildings*, 33(4):319–331, 2001.

Karaman, S., Anders, A., Boulet, M., Connor, J., Gregson, K., Guerra, W., Guldner, O., Mohamoud, M., Plancher, B., Shin, R., et al. Project-based, collaborative, algorithmic robotics for high school students: Programming self-driving race cars at mit. In *2017 IEEE Integrated STEM Education Conference (ISEC)*, pp. 195–203. IEEE, 2017.

Koenig, N. and Howard, A. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2149–2154, Sendai, Japan, Sep 2004.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., and Stoica, I. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

Maruyama, Y., Kato, S., and Azumi, T. Exploring the performance of ros2. In *Proceedings of the 13th International Conference on Embedded Software*, pp. 5. ACM, 2016.

Mordatch, I., Lowrey, K., and Todorov, E. Ensemble-cio: Full-body dynamic motion planning that transfers to physical humanoids. In *IROS*, pp. 5307–5314. IEEE, 2015. ISBN 978-1-4799-9994-1. URL http://dblp.uni-trier.de/db/conf/iros/iros2015.html#MordatchLT15.

Morimoto, J. and Doya, K. Robust reinforcement learning. *Neural Comput.*, 17(2):335–359, February 2005. ISSN 0899-7667. doi: 10.1162/0899766053011528. URL http://dx.doi.org/10.1162/0899766053011528.

Paull, L., Tani, J., Ahn, H., Alonso-Mora, J., Carlone, L., Cap, M., Chen, Y. F., Choi, C., Dusek, J., Fang, Y., et al. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1497–1504. IEEE, 2017.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, pp. 5. Kobe, Japan, 2009.

Rusu, A. A., Vecerik, M., Rothörl, T., Heess, N., Pascanu, R., and Hadsell, R. Sim-to-real robot learning from pixels with progressive nets. *CoRR*, abs/1610.04286, 2016. URL http://arxiv.org/abs/1610.04286.

Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., and Batra, D. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 618–626, 2017.

Services, A. W. AWS Robomaker. https://aws.amazon.com/robomaker/, May. 2019a. Accessed: 2019-05-05.

Services, A. W. AWS Deepracer: Virtual league. https://aws.amazon.com/deepracer/league/#, May. 2019b. Accessed: 2019-05-05.

Services, A. W. AWS Deepracer: getting started documentation. https://aws.amazon.com/deepracer/getting-started/, Mar. 2019c. Accessed: 2019-03-08.

Tan, J., Zhang, T., Coumans, E., Iscen, A., Bai, Y., Hafner, D., Bohez, S., and Vanhoucke, V. Sim-to-real: Learning agile locomotion for quadruped robots. *CoRR*, abs/1804.10332, 2018. URL http://arxiv.org/abs/1804.10332.

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. Domain randomization for transferring deep neural networks from simulation to the real world. *CoRR*, abs/1703.06907, 2017. URL http://arxiv.org/abs/1703.06907.

Tzeng, E., Devin, C., Hoffman, J., Finn, C., Peng, X., Levine, S., Saenko, K., and Darrell, T. Towards adapting deep visuomotor representations from simulated to real environments. *CoRR*, abs/1511.07111, 2015. URL http://arxiv.org/abs/1511.07111.