Sim2Real Transfer for Deep Reinforcement Learning with Stochastic State Transition Delays

Sandeep Singh Sandha[1], **Luis Garcia**[2], **Bharathan Balaji**[3], **Fatima M Anwar**[4], **Mani Srivastava**[1]
[1]University of California Los Angeles, United States
[2]USC ISI (Work done while at UCLA), United States
[3]Amazon, Seattle, United States
[4]University of Massachusetts, Amherst, United States
sandha@cs.ucla.edu, lgarcia@isi.edu
bhabalaj@amazon.com, fanwar@umass.edu, mbs@ucla.edu

Abstract:
Deep Reinforcement Learning (RL) has demonstrated to be useful for a wide variety of robotics applications. To address sample efficiency and safety during training, it is common to train Deep RL policies in a simulator and then deploy to the real world, a process called *Sim2Real* transfer. For robotics applications, the deployment heterogeneities and runtime compute stochasticity results in variable timing characteristics of sensor sampling rates and end-to-end delays from sensing to actuation. Prior works have used the technique of domain randomization to enable the successful transfer of policies across domains having different state transition delays. We show that variation in sampling rates and policy execution time leads to degradation in Deep RL policy performance, and that domain randomization is insufficient to overcome this limitation. We propose the Time-in-State RL (TSRL) approach, which includes delays and sampling rate as additional agent observations at training time to improve the robustness of Deep RL policies. We demonstrate the efficacy of TSRL on HalfCheetah, Ant, and car robot in simulation and on a real robot using a $1/18^{th}$ scale car.

1 Introduction

Deep Reinforcement Learning (RL) has shown promising results for a range of robotics applications, such as navigation [1], manipulation [2], and locomotion [3]. Deep RL policies are often trained with simulations due to cost, time to train, and safety concerns that arise when training on real robots [2]. Simulations are imperfect and difficult to calibrate. The resulting modeling discrepancies cause a *reality gap*, which makes the transfer of RL policies from simulation to the real-world (Sim2Real) a challenge [4]. Prior works have proposed domain randomization and adaptation techniques to address the reality gap in dynamics [2, 5] and image observations [6]. We study the reality gap introduced due to uncertainty in time between state transitions and the resultant impact on dynamics in agile robotic tasks such as locomotion and navigation.

RL agents make sequential decisions in a Markov Decision Process (MDP) in *discrete time steps*, where the input to the agent is the current state s_t of the environment, where t is the current time step, and output is the action a_t. The environment transitions to the next state s_{t+1} once the action is executed, and in turn used as the input for the next action a_{t+1}. If the states s_t and s_{t+1} were captured at world clock time τ and τ' respectively, the timing delay between state transitions is defined as $\Delta\tau = \tau - \tau'$. Note that t is a discrete time step in simulation while $\Delta\tau$ represents the actual passage of time on a robot. A common trend is to assume $\Delta\tau$ is fixed for Sim2Real transfer [1, 7]. However, $\Delta\tau$ varies in real robots due to variations in RL policy execution time, sensor sampling interval and communication delays. In Deep RL, policy execution time of neural networks dominates $\Delta\tau$ and shows significant variation due to various factors – uncertainties due to locally shared compute resources, asymmetric communication latencies with use of shared cloud resources [8], and delay variations due to processor throttling for thermal and energy constraints [9].

4th Conference on Robot Learning (CoRL 2020), Cambridge MA, USA.

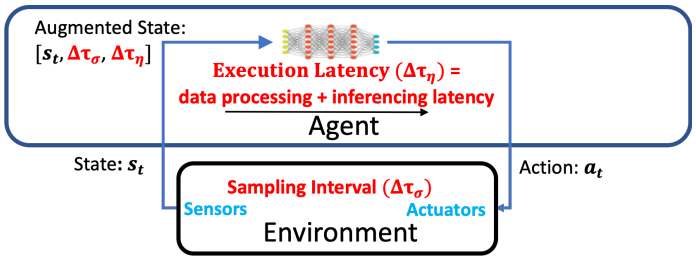


Figure 1: Delays for a typical sensing to actuation pipeline. TSRL augments the observed state with sampling interval and inferencing latency.

As the real robot operates in continuous world clock time, the state transitions observed by the agent will change with variations in $\Delta\tau$. If these variations are not captured by the simulator during training, it leads to poor Sim2Real transfer [10, 11]. Prior works randomized the state transition delays $\Delta\tau$ during training for a successful Sim2Real transfer [5]. We demonstrate that the policy performance degrades with variation in $\Delta\tau$ even with domain randomization. Another approach is to artificially extrapolate varying delays to the worst case $\Delta\tau$. For example, Molchanov et al. [7] use a fixed $\Delta\tau$ of 2ms for controlling a quadrotor while the neural network inference latency was only 0.8ms. With this approach, one is forced to pick a conservative $\Delta\tau$ that accounts for the *worst case delays* in the system or pick a small neural network to keep inference latency to a minimum. Large $\Delta\tau$ limits the applicability of Deep RL in agile tasks where fast response times are required [12], and small neural networks limit scalability for complex tasks with large state-action space.

We introduce Time-in-State RL (TSRL), a Deep RL approach that extends the observed state of the system by explicitly including time-delays, i.e., incorporating $\Delta\tau$ introduced by the sensor sampling interval and execution latency at training time. Even though the inferencing latency and sampling interval can vary for various reasons, they can be accurately measured by Deep RL agents at runtime. We test the following hypothesis: if the agent observes the factors that impact the $\Delta\tau$, and hence the state transitions, it helps the agent to learn a better policy compared to a policy that partially observes the impact of changing $\Delta\tau$ using domain randomization. We evaluate our approach on simulation-to-simulation (Sim2Sim) transfer on PyBullet HalfCheetah, PyBullet Ant [13] and DeepRacer [1]. We evaluate our approach on Sim2Real transfer using a $1/18^{th}$ scale car. We compare the TSRL policies with the policies trained using domain randomization (DR) of timing characteristics. Our results demonstrate that the TSRL policies are robust to the varying state transition delays and, as a result, transfer better across simulations and to real-world environments than the DR policies. The code of this paper can be found at `https://github.com/nesl/Time-in-State-RL/`.

Contributions. 1) We demonstrate that the performance of Deep RL policies degrades with variation in state-to-state transition delays–even with state-of-the-art domain randomization techniques. 2) We propose and evaluate TSRL, a a delay-aware Deep RL approach that extends the observed state by explicitly including the sensor sampling interval and inferencing latency in Sim2Sim and Sim2Real settings.

2 Background

RL agents learn to make sequential decisions in the environment to maximize the expected cumulative discounted reward. At each discrete time step t of an MDP, the agent takes action a_t based on state s_t, and the environment returns with scalar reward r_t and next state s_{t+1}. We consider episodic MDPs, where the environment is initialized with state s_0 and the interaction with the agent continues until the environment reaches the terminal state s_T. $p(s_{t+1}|s_t, a_t)$ is the probability of transition to state s_{t+1} given current state s_t and action a_t. Any changes to the real state transition time $\Delta\tau$ – as opposed to fixed discrete time steps in t – directly impacts the state transition probability $p(s_{t+1}|s_t, a_t)$. We further explain how changes in $\Delta\tau$ impact policy learning in Appendix A.

2.1 Variability in State Transition Time

The delays in a typical sensing to actuation pipeline for a Deep RL agent are shown in Figure 1. A typical state transition begins with sensing the current state s_t of the environment, executing the

agent action a_t on the environment, and again sensing the updated state s_{t+1}. Each of these steps can have variability in the real world as determined by the sampling interval of sensors $\Delta\tau_\sigma$, the execution latency $\Delta\tau_\eta$, and communication delays $\Delta\tau_m$. When multiple sensors are present, $\Delta\tau_\sigma$ is the maximum of individual sensor sampling intervals. Similar arguments are extended to $\Delta\tau_\eta$. We assume communication delays $\Delta\tau_m$ are small, and subsume them into execution latency $\Delta\tau_\eta$.

Execution Latency: Various factors can affect execution latency $\Delta\tau_\eta$ such as power management, computational resources, and complex operating system (OS) environments. Dynamic frequency scaling [9] is a commonly used technique to manage power dissipation [14]. Frequency over-clocking/under-clocking changes computation speed and lead to variable latencies. The OS scheduling processes result in scheduling noise that varies with system load and affects process latencies.

We analyzed the inference latencies of commonly used neural network architectures in Deep RL policies on several hardware platforms. The runtime latency depends on the complexity of neural network, hardware device, and multi-tenancy. On the GAP8 [15] microcontroller, the execution latency of a simple neural network increases from ∼7ms to ∼55ms when the number of CNN layers increases from 2 to 4. For deep learning accelerators like the Intel Neural Compute Stick 2 [16], the execution latency of a 2 layer CNN network is increased from ∼2ms to ∼16ms in presence of multiple inference tasks. We characterized the execution latency of the default Deep RL policy in DeepRacer [1], a $1/18^{th}$ scale autonomous car that comes with an Intel Atom processor and an integrated GPU. The execution latency varies from 15-20 ms on the GPU and goes up to 34 ms on the CPU. We include additional analysis on execution latency in Appendix B.

Sampling Interval: Stisen et al. [17] demonstrate that sampling interval of accelerometers can vary widely in smartphones depending on both software and hardware characteristics. We characterized the variation of the frame rate in the DeepRacer front facing camera. The variation in sampling interval was 20-45 ms in the 30Hz frame rate setting and 62-71ms in the 15Hz setting respectively.

2.2 Impact of Delay Variations on Deep RL Policies

The impact of state transition time variations on the RL policy depends on the relative value of sampling interval $\Delta\tau_\sigma$ to execution latency $\Delta\tau_\eta$.

(a) $\Delta\tau_\sigma \ll \Delta\tau_\eta$**:** When the sampling interval $\Delta\tau_\sigma$ is very small, the variations in execution latency $\Delta\tau_\eta$ dominate the impact on the policy. As $\Delta\tau_\eta$ varies, the *observed evolution of the environment state* in world clock time τ also varies correspondingly. Hence, the agent will observe stochasticity in state transitions $p(s_{t+1}|s_t, a_t)$ for the same state and action. If we ignore the variation in $\Delta\tau_\eta$ during training, there will be distribution mismatch from simulations to the real world, leading to poor transfer. Mahmood et al. [10] and Xie et al. [11] demonstrate that policies can break down with small changes (<100 ms) in latency for manipulation and locomotion respectively. On the other hand, if we introduce $\Delta\tau_\eta$ variations during training [2, 5], the additional state transition stochasticity introduces noise in the value function estimates and makes it difficult to converge to a good policy. We demonstrate this in both simulation and a real robot in Section 4.

(b) $\Delta\tau_\sigma \gg \Delta\tau_\eta$**:** In this case, variation in the sampling interval $\Delta\tau_\sigma$ dominates the impact on the RL policy. The agent needs to observe the effect of its action on the environment. When sampling interval is large or if changes in state are minor with a single action, it is common practice to repeat the agent action a fixed number of times [18, 19]. With variations in $\Delta\tau_\sigma$, the number of repeated actions need to be varied and the impact on the RL policy follows the same argument as above.

(c) $\Delta\tau_\sigma \sim \Delta\tau_\eta$**:** In this case, we want the agent to act for every sensed state [20]. When $\Delta\tau_\sigma$ and $\Delta\tau_\eta$ vary, there is a *phase shift* in each state transition depending on when the state gets sampled and when the action is executed. These phase shifts introduce noise in the state transition probabilities $p(s_{t+1}|s_t, a_t)$, and impact the performance of the RL policy. While phase shifts do occur in the other two cases, the impact on the policy is dominated by overall variation in either $\Delta\tau_\sigma$ or $\Delta\tau_\eta$.

3 Training DeepRL Policies with Delay Variations

A common technique in literature is to do domain randomization during training to account for uncertain and unmodeled environment dynamics [2]. Domain randomization makes sense for physical quantities such as friction and contact forces, as they are difficult to measure and calibrate across

real robots. However, state transition delay related variables such as execution time $\Delta\tau_\eta$ and sampling interval $\Delta\tau_\sigma$ are straightforward to measure both in simulation and real robots. We propose augmenting the agent state with these measurements: $\tilde{s} = [s, \Delta\tau_\eta, \Delta\tau_\sigma]$ where $\tilde{s}$ represents the augmented state. This simple trick enables the agent to distinguish between state transitions introduced by variations in delays. We refer to the augmented state policies as *Time in State (TS)* policies.

For low-dimensional state spaces, the execution time and sampling interval can be directly added as another state. For augmenting high-dimensional state spaces, such as images, the delay observations are fused in an intermediate layer of both policy and value networks [21]. To evaluate TSRL, we instrumented *HalfCheetahBulletEnv-v0* and *AntBulletEnv-v0* environments in the PyBullet simulator [22] to demonstrate Sim2Sim transfer on a low dimensional use case, as well as DeepRacer [1] to test both Sim2Sim and Sim2Real transfer on a high dimensional use case. We train our policies using the Proximal Policy Optimization (PPO) [23] algorithm as implemented in the OpenAI Baselines[1]. We use PPO as it has been widely used in robotics applications [1, 5, 7].

3.1 Low Dimensional Use Cases: HalfCheetah and Ant

The implementation for *HalfCheetahBulletEnv-v0* and *AntBulletEnv-v0* in PyBullet simulator evolve physics at a fixed time (Sim_{Time}) of 4.12 ms for each action. We modified the default environments and advance the simulation for multiple simulation steps per action to vary the execution latency and sampling interval. Thus, the granularity of the variation in the execution latency and sampling interval in our PyBullet experiments is Sim_{Time}.

Variation of Timing Characteristics. We vary the state transition delays in the simulator when training the policies and consider the setting where execution latency $\Delta\tau_\eta \leq$ sampling interval $\Delta\tau_\sigma$. For reactive systems, this setting is desired so that the agent can act for every sensed state [20]. The actuation of recent action is delayed by the execution latency $\Delta\tau_\eta$, and the next sensor sample is available after the sampling interval $\Delta\tau_\sigma$. We select the range of execution latencies $\Delta\tau_\eta$ between [0 - $10 * Sim_{Time}$] = [0 - 41.2 ms] and sampling interval $\Delta\tau_\sigma$ values between [Sim_{Time} - $10 * Sim_{Time}$] = [4.12 ms - 41.2 ms]. We varying $\Delta\tau_\eta$ and $\Delta\tau_\sigma$ for HalfCheetah and Ant within their respective ranges. Before the starting of episode, we fix $\Delta\tau_\eta$ and then decide $\Delta\tau_\sigma$ = max(4.12 ms , $\Delta\tau_\eta$). Between consecutive steps, we introduce random jitter of $\pm Sim_{Time}$ in both $\Delta\tau_\eta$ and $\Delta\tau_\sigma$.

Policy Training. The vanilla policy trained without varying state transition delays fails to work in presence of variable ($\Delta\tau_\eta$ and $\Delta\tau_\sigma$). We present analysis of the vanilla policy in Appendix G. We use domain randomization (DR) as our baseline algorithm, where the policy is trained by varying the state transition delays during training. We train the DR and TS policies by varying the $\Delta\tau_\eta$ and $\Delta\tau_\sigma$ as described above. When training DR policies, the default state from *HalfCheetahBulletEnv-v0* and *AntBulletEnv-v0* is used. We augment the state with $\Delta\tau_\eta$ and $\Delta\tau_\sigma$ for TS policies. We use 2-layer fully connected neural network with each layer having 64 nodes for policy and value function. We include additional details for reproducibility in Appendix C.

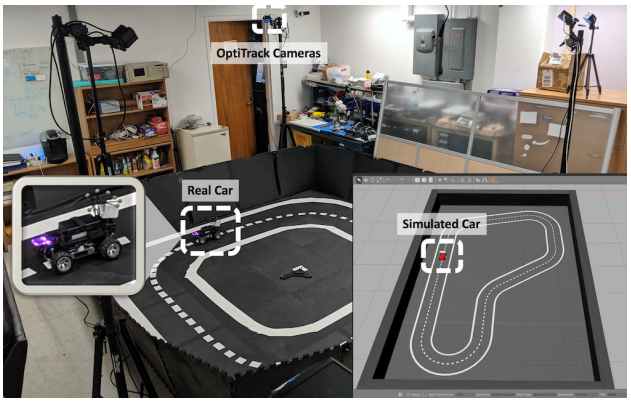
3.2 High-Dimensional Use Case: Autonomous Vehicle

We use Gazebo simulator[2] to train navigation policies for the DeepRacer car [1]. The simulator includes a robot model that is matched to the properties of the real car. The simulation advances in real-time. Images from the camera are used to navigate the car on the track. We modify the sampling rate of camera and runtime execution latency by adding controlled timing delays.

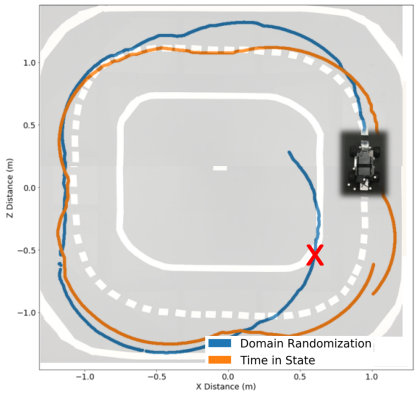
Variation of Timing Characteristics. During training in simulator, the execution latency $\Delta\tau_\eta$ is varied between the discrete set of values from 10 ms to 120 ms. The sampling interval $\Delta\tau_\sigma$ of 33 ms (30 Hz) is used when $\Delta\tau_\eta \leq \Delta\tau_\sigma$, otherwise $\Delta\tau_\sigma$ is matched to $\Delta\tau_\eta$. During policy training for each episode, we fix the value of $\Delta\tau_\eta$ and $\Delta\tau_\sigma$. The execution latency to do the policy network inference and image processing on the server machine is $\sim$10ms. The execution latency on the real car using integrated GPU is $\sim$20ms in the absence of other tasks. The sampling interval of 33 ms (30 Hz) corresponds to the supported camera sampling rate on the real car. The range of variations in $\Delta\tau_\eta$ and $\Delta\tau_\sigma$ are selected to benchmark the policy behavior of navigational policy in the presence of deployment variations of hardware, multi-tenancy, and communication delays.

[1] OpenAI Baselines: https://github.com/openai/baselines

[2] http://gazebosim.org/

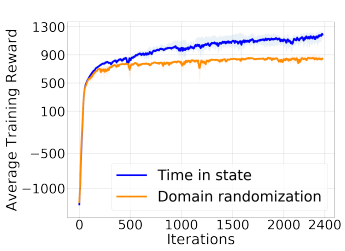


(a) The DeepRacer car on a real track and the simulated car in the Gazebo environment. The OptiTrack motion capture system is used to quantify the performance of policies on the real track.

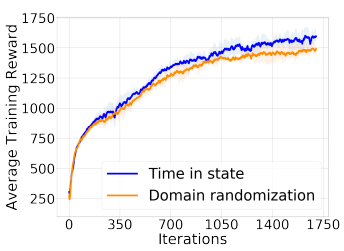


(b) A single instance of DR policy and TS policy tested on a real autonomous car in the presence of 60 ms execution latency.

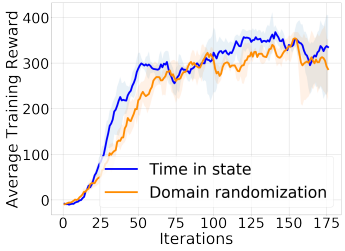
Figure 2: The real world environment setup and performance measurement using using OptiTrack.



(a) HalfCheetah task



(b) Ant task



(c) DeepRacer car

Figure 3: The learning curves for HalfCheetah, Ant and DeepRacer car for TS and DR policies.

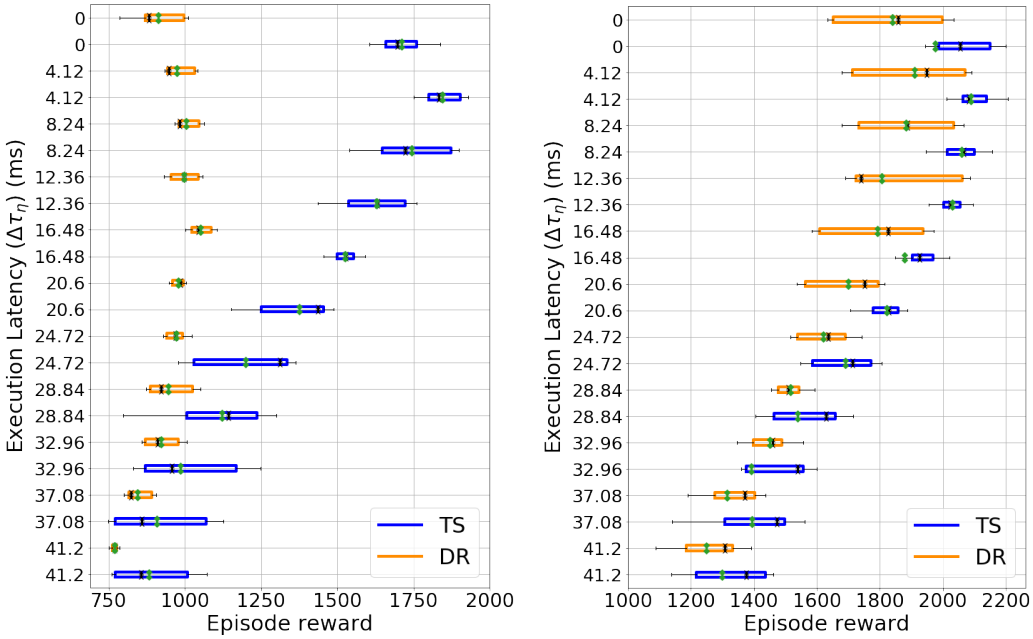
Policy Training. The DR and TS policies are trained by varying the state transition delays as described above. In addition, we use the recommended image augmentations [1] to enable the successful transfer of policy to the real car. Due to the image augmentation processing times and variations in simulation advancement, a jitter of 5 ms is present in both $\Delta\tau_\eta$ and $\Delta\tau_\sigma$. The simulation setting, along with the track and simulated car, is shown in Figure 2a. The simulated track has a centerline of the length of 17 meters and a track width of 0.44 meters. The policy's goal is to follow the centerline of the track by controlling the steering angle and speed. The highest reward of 1.1 is given when the center of the car matches the centerline, and the reward is scaled to zero as the car moves away from the centerline to offtrack. Each episode consists of 500 steps. The neural network is represented by 3 CNN layers followed by a 2 fully connected layers and an output layer. The DR policy uses only the images from the camera as input. For TS policy, we fuse images with the execution latency and sampling interval in the first fully connected layer after the CNN layers. We provide the details for reproducibility in Appendix D.

Real-world environment. We created a real track as shown in Figure 2a, with a center line distance of 7.3 meters and the track width of 0.52 meters. We compared the performance of policies by utilizing an OptiTrack motion capture system[3] shown in Figure 2a. We localized the location of the car with respect to the center line of the real track.

4 Results

We compare the time-in-state (TS) based policies and the domain randomization (DR) policies in simulation and on the real robot. We evaluate their robustness across varying sampling intervals and execution latencies. This section compares the performance of fully connected policies. The experiments with recurrent policies are discussed in Appendix F.

[3] https://optitrack.com/



(a) Sim2Sim for HalfCheetah task

(b) Sim2Sim for Ant task

Figure 4: Comparison of time-in-state (TS) and domain randomization (DR) policies for HalfCheetah and Ant tasks across different execution latencies ($\Delta\tau_\eta$). The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (4.12 ms , $\Delta\tau_\eta$), so that agent can act for each sensed state. The mean is shown in green, the black 'x' marker shows the median of IQR. For both tasks, TS policies achieve higher mean reward than DR policies.

Latency	20 ms	60 ms	100 ms
TS	20	17	13
DR	20	11	7

(a)

Latency	20 ms	60 ms	100 ms
TS	1.50 m/s	1.45 m/s	1.40 m/s
DR	1.45 m/s	1.44 m/s	1.45 m/s

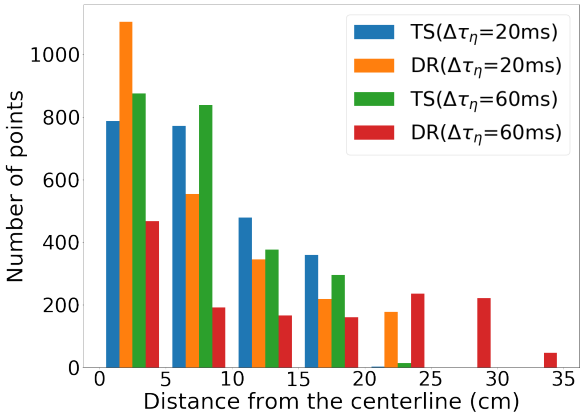
(b)

Table 1: (a) Comparison of time-in-state (TS) and domain randomization (DR) policies in completing laps on the real track out of 24 trials at different execution latencies. (b) The average speed used by TS and DR. TS adapts its speed with increase in execution latency.

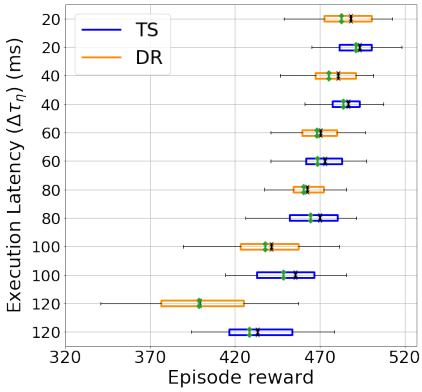
HalfCheetah and Ant Tasks. The policies are trained, as explained in Section 3.1. We train three models for each task, both for TS and DR, with different seeds. The learning curves for the policies are shown in Figure 3. For both tasks, TS achieves a better mean training reward than the DR policies. The evaluation of policies for both HalfCheetah and Ant tasks across the three trained models is shown in Figure 4. The spread of test reward is captured across 10 episodes for each model at a particular sampling interval ($\Delta\tau_\sigma$) and execution latency ($\Delta\tau_\eta$). The analysis shows that the TS policies perform better than the DR policies across state transition delay variations and maintains a higher mean reward. Figure 4 also shows that, in general, the performance of Deep RL policies degrades when exposed to higher sampling intervals and execution latencies. The degradation in performance is task-specific. Appendix E evaluates the policies with variable delays within an episode.

DeepRacer Simulator. Figure 3c shows the learning curve of TS and DR policies trained using DeepRacer simulator. We train 3 models for each policy. Figure 5b shows the Sim2Sim of the policies across 3 set of models using the DeepRacer simulator. We evaluate each model for 16 episodes (500 steps on track for each episode). We test the policies across a spread of different sampling intervals ($\Delta\tau_\sigma$) and the execution latencies ($\Delta\tau_\eta$). The results also show that as $\Delta\tau_\eta$ and $\Delta\tau_\sigma$ are increased, the performance of the Deep RL policy degrades in general. However, in comparison to DR, the TS policies have better performance.

Sim2Real transfer. We compare the performance of the TS and DR policies on the DeepRacer robot using the real track. The results for 24 trials in both the directions for TS and DR policies across the 3 trained models are shown in Table 1a. The results shows the performance gain of the



(a) The distance of the real car from the centerline captured using OptiTrack cameras. The number of points plotted is 2400, except the DR ($\Delta\tau_\eta$=60ms), which has 1657 points. The onboard camera of car was running at 30Hz.



(b) Analysis of TS and DR policies across different execution latencies ($\Delta\tau_\eta$) in DeepRacer simulator. The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (33 ms , $\Delta\tau_\eta$). The green color shows mean, the black 'x' marker shows the median of IQR.

Figure 5: Evaluation of time-in-state (TS) and domain randomization (DR) policies using DeepRacer car across different sampling intervals ($\Delta\tau_\sigma$) and execution latencies ($\Delta\tau_\eta$).

TS policies is transferred to real robot. DR and TS policies work very well on the real track by successfully completing higher number of laps for $\Delta\tau_\eta$ = 20ms. As the $\Delta\tau_\eta$ is increased to 60ms and 100ms, the performance of DR policies significantly degrade in comparison to the TS policies. Table 1b shows the average action speed of policies in the simulator track. TS adapts its speed with increasing execution latency by taking slower actions whereas DR does not change its action speed. In our supplementary video, we show that the TS policy speed adaptation occurs primarily on the curved regions of the track, and helps it achieve robust navigation. The $\Delta\tau_\eta$ of neural network policy using GPU of Car is within 15-20ms. We introduce extra delay and fix the $\Delta\tau_\eta$ to 20ms, 60ms and 100ms respectively to generate the comparison of TS and DR policies. The measured sampling interval $\Delta\tau_\sigma$ was directly given as input to the TS policies. The camera was running at the sampling rate of 30 Hz, which was measured to have variable sampling interval from 25-45ms at runtime. Figure 2b shows an instance of the real run captured using OptiTrack setup comparing TS and DR based policies for the sampling rate of 30 Hz and $\Delta\tau_\eta$ of 60ms. The TS policies have more stable performance on the real track as compared to the DR policies. We analyzed the distance from the centerline maintained by both TS and DR policies on the real track. The distance is captured using OptiTrack setup. The distance maintained is shown in Figure 5a. TS policies maintain a closer distance to the centerline. The DR policies have more oscillating behavior around the centerline. We believe the oscillating behavior is the reason for higher number of points within [0-5 cm] for DR at $\Delta\tau_\eta$ to 20ms.

5 Related Work

State Transition Delays in Reinforcement Learning. Handling of state transition delay variations in a robot has been identified as crucial for successful Sim2Real transfer by many prior works [11, 10, 2, 5]. Mahmood et al. [10] examine the design decisions for deployment of Deep RL policies for manipulation. They highlight the lack of guidelines to pick state transition times, or to implement asynchronous mechanisms to reduce state transition delays. They also demonstrate that different types of state transition delay variations can drastically impact the performance of the RL policy. Similarly, Xie et al. [11] note that their bipedal robot quickly falls with a sensing delay of 10ms. However, they do not propose any solutions to this problem.

Peng et al. [24] and Andrychowicz et al. [5] randomize the state transition delay each time step according to an exponential distribution for manipulation of an arm and a hand respectively. They also use an LSTM layer that summarizes the history of state transitions and accounts for unmodeled dynamics compared to just using the current state as input to a feed forward network. In an ablation study, Peng et al. show that removing state transition delay randomization during training reduces

Sim2Real success from 89% to just 29%. Tan et al. [25] uniformly randomize the state transition delay across episodes to enable successful Sim2Real transfer. We borrow from these solutions and propose augmenting the state with the measured state transition delay. Our evaluation results show that our approach improves upon the domain randomization solution.

Prior works consider action or observation delays that occur across multiple time steps [12, 26, 27, 28]. These works suggest augmenting the state space with past actions to account for the delay. In contrast, we study the impact of continuous time delays that occur in a state transition, and augment the state with the time delay observed. Schuitema et al. [29] propose combining past actions to account for state transition delays in RL policies. While this approach works for continuous actions, its not clear how the actions can be combined in the discrete action case. Their solution also does not account for phase shifts due to sensing delays. There exist several works that propose RL algorithms for continuous time and space [30, 31]. While these works have a rich set of theoretical results, the application of these algorithms have been limited to small scale problems in simulation and have not yet been applied to real robots. Chebotar et al. [32] adapt the simulation parameters using real world roll-outs that can be used to match a fixed delay between the simulator and real robot. However, it is not clear how to extend this approach to variable delays within and across deployments.

Control System Approaches. The presented experiments in this paper could be reformulated as classical control problems. Control systems approaches typically model finite-dimensional systems that may require linearization. The goal in these contexts is to develop robust controllers by approximating the worst case time-delays [33] and sampling variation [34], or by compensating for delays using damping components. For the latter case, energy-based controllers [35, 36] or Lyapunov-based controllers [33] rely on state estimation that is essentially extracting features of the plant (environment). For control problems with lower-dimensional inputs, these approaches can be used to develop robust controllers. However, we focus on Deep RL that is end-to-end and can handle inputs with high dimensionality, e.g., images. Because TSRL is augmenting state space with measured time-delays, one may hypothesize that the adaptive policies are mimicking the classical control approaches. However, the state-estimation and associated control transfer function properties reside in the latent space of the deep neural network, which are notoriously uninterpretable[4]. We extend the discussion of the control system approaches in Appendix H.

6 Conclusion

We introduced Time-in-State RL (TSRL), a delay-aware deep reinforcement learning approach that incorporates sampling interval and execution latency into its state space. By utilizing domain randomization with time in the state, TSRL's policies are robust against varying execution latencies and sampling rates for both Sim2Sim and Sim2Real transfer. The application performance characterization of TSRL can be exploited by policies to conserve platform resources by acting slowly along with staying within the desired reward budget. The performance characterization can also help developers make informed decisions in selecting appropriate compute hardware and deployment settings. The evaluation of time in state policies show that the policies are able to maintain higher rewards across a range of timing characteristics and, thus, can be used in presence of deployment uncertainties impacting the timing characteristics at runtime. Through a range of choices of latencies and sampling intervals, our study also shows different tasks can work reasonably only up to to a certain latency and after that suffers significant degradation in performance. We hope the concepts and results introduced in this paper will motivate the development of Deep RL policies that are robust to runtime uncertainties.

Acknowledgements. The first, second, and last authors would like to acknowledge the support of their research from the National Science Foundation (NSF) under award # CNS-1329755, and from the U.S. Army Research Laboratory under Agreement Number W911NF-17-2-0196. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, NSF, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright the notation here on.

[4]For simpler neural networks such as the one used in the HalfCheetah and Ant tasks, one may be able to generate an approximate representation for stability analysis. However, this is not scalable for higher-dimensional inputs.

References

[1] B. Balaji, S. Mallya, S. Genc, S. Gupta, L. Dirac, V. Khare, G. Roy, T. Sun, Y. Tao, B. Townsend, et al. Deepracer: Educational autonomous racing platform for experimentation with sim2real reinforcement learning. *arXiv preprint arXiv:1911.01562*, 2019.

[2] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.

[3] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, 2019.

[4] S. Koos, J.-B. Mouret, and S. Doncieux. Crossing the reality gap in evolutionary robotics by promoting transferable controllers. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 119–126, 2010.

[5] O. M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

[6] F. Sadeghi and S. Levine. Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*, 2016.

[7] A. Molchanov, T. Chen, W. Hönig, J. A. Preiss, N. Ayanian, and G. S. Sukhatme. Sim-to-(multi)-real: Transfer of low-level robust control policies to multiple quadrotors. *arXiv preprint arXiv:1903.04628*, 2019.

[8] S. Chinchali, A. Sharma, J. Harrison, A. Elhafsi, D. Kang, E. Pergament, E. Cidon, S. Katti, and M. Pavone. Network offloading policies for cloud robotics: A learning-based approach. In *Proceedings of Robotics: Science and Systems*, FreiburgimBreisgau, Germany, June 2019. doi:10.15607/RSS.2019.XV.063.

[9] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Mobile Computing*, pages 449–471. Springer, 1994.

[10] A. R. Mahmood, D. Korenkevych, B. J. Komer, and J. Bergstra. Setting up a reinforcement learning task with a real-world robot. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4635–4640. IEEE, 2018.

[11] Z. Xie, G. Berseth, P. Clary, J. Hurst, and M. van de Panne. Feedback control for cassie with deep reinforcement learning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1241–1246. IEEE, 2018.

[12] T. J. Walsh, A. Nouri, L. Li, and M. L. Littman. Learning and planning in environments with delayed feedback. *Autonomous Agents and Multi-Agent Systems*, 18(1):83, 2009.

[13] E. Coumans and Y. Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2019.

[14] A. Tang, S. Sethumadhavan, and S. Stolfo. {CLKSCREW}: exposing the perils of security-oblivious energy management. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1057–1074, 2017.

[15] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini. Gap-8: A risc-v soc for ai at the edge of the iot. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–4. IEEE, 2018.

[16] Intel. Intel neural compute stick 2 product specifications. URL `https://ark.intel.com/content/www/us/en/ark/products/140109/intel-neural-compute-stick-2.html`.

[17] A. Stisen, H. Blunck, S. Bhattacharya, T. S. Prentow, M. B. Kjærgaard, A. Dey, T. Sonne, and M. M. Jensen. Smart devices are different: Assessing and mitigatingmobile sensing heterogeneities for activity recognition. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 127–140, 2015.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[19] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[20] N. Halbwachs. *Synchronous programming of reactive systems*, volume 215. Springer Science & Business Media, 2013.

[21] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng. Multimodal deep learning. 2011.

[22] E. Coumans and Y. Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. 2016.

[23] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[24] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018.

[25] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

[26] K. V. Katsikopoulos and S. E. Engelbrecht. Markov decision processes with delays and asynchronous cost collection. *IEEE transactions on automatic control*, 48(4):568–574, 2003.

[27] S. Ramstedt and C. Pal. Real-time reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 3073–3082, 2019.

[28] B. Chen, M. Xu, L. Li, and D. Zhao. Delay-aware model-based reinforcement learning for continuous control. *arXiv preprint arXiv:2005.05440*, 2020.

[29] E. Schuitema, L. Buşoniu, R. Babuška, and P. Jonker. Control delay in reinforcement learning for real-time dynamic systems: a memoryless approach. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3226–3231. IEEE, 2010.

[30] K. Doya. Reinforcement learning in continuous time and space. *Neural computation*, 12(1): 219–245, 2000.

[31] J. Y. Lee and R. S. Sutton. Integral policy iterations for reinforcement learning problems in continuous time and space. *arXiv preprint arXiv:1705.03520*, 2017.

[32] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8973–8979. IEEE, 2019.

[33] B. W. Bequette. *Process control: modeling, design, and simulation*. Prentice Hall Professional, 2003.

[34] K. Worthmann, M. Reble, L. Grune, and F. Allgower. The role of sampling for stability and performance in unconstrained nonlinear model predictive control. *SIAM Journal on Control and Optimization*, 52(1):581–605, 2014.

[35] M. De Stefano, L. Vezzadini, and C. Secchi. Time-delay compensation using energy tank for satellite dynamics robotic simulators. In *IEEE International Conference on Intelligent Robots and Systems*, 2019.

[36] J.-H. Ryu, D.-S. Kwon, and B. Hannaford. Stability guaranteed control: Time domain passivity approach. *IEEE Transactions on Control Systems Technology*, 12(6):860–868, 2004.

Appendix

A Variable State Transition Time in Deep RL

RL agents learn to make sequential decisions in the environment to maximize the expected cumulative discounted reward. At each discrete time step t of an MDP, the agent takes action a_t based on state s_t, and the environment returns with scalar reward r_t and next state s_{t+1}. We consider episodic MDPs, where the environment is initialized with state s_0 and the interaction continues until the environment reaches the terminal state s_T. $p(s_{t+1}|s_t,a_t)$ is the probability of transition to state s_{t+1} given current state s_t and action a_t, and $\pi(a|s)$ represents the probability of taking action a given state s by policy π. Any changes to the real state transition time $\Delta\tau$ – as opposed to fixed discrete time steps in t – directly impacts the state transition probability $p(s_{t+1}|s_t,a_t)$.

The objective of the agent is to learn a policy π that maximizes:

$$J = \mathop{\mathbb{E}}_{\substack{\pi(a|s) \\ p(s_{t+1}|s_t,a_t)}} \left[\sum_{t=0}^{T} \gamma^{t-1} r_t \right] \tag{1}$$

where $\gamma \in [0,1]$ discounts future rewards, T is the episode length, $p(s_{t+1}|s_t,a_t)$ is the probability of transition to state s_{t+1} given current state s_t and action a_t, and $\pi(a|s)$ represents the probability of taking action a given state s by policy π. Any changes to the real state transition time $\Delta\tau$ – as opposed to fixed discrete time steps in t – directly impacts the state transition probability $p(s_{t+1}|s_t,a_t)$.

We focus on model free RL algorithms, where the state transition probabilities $p(s_{t+1}|s_t,a_t)$ are unknown to the agent and are inferred indirectly through environment interactions. One of the simplest Deep RL algorithms is REINFORCE, where the policy is represented by a neural network with parameters θ. The policy is learned using gradient ascent on the objective function.

$$\nabla_\theta J(\theta) = \sum_{n=0}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t=0}^{T} \gamma^{t-1} R(s_t,a_t) \tag{2}$$

where $R(s_t,a_t)$ is the reward function and data from N episodes is used for the update. REINFORCE has high variance as the gradient update depends on the total discounted reward collected during each episode. To reduce variance, the advantage function $A(s_t,a_t)$ is used for the gradient update:

$$\nabla_\theta J(\theta) = \sum_{n=0}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t,a_t) \tag{3}$$

$$A(s_t,a_t) = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{4}$$

where $V_\phi(s_t)$ estimates the cumulative discounted reward from state s_t using a separate value network with parameters ϕ. Intuitively, advantage estimates the relative benefit of taking action a_t compared to other possible actions in state s_t. The value network is trained with a mean squared error loss function:

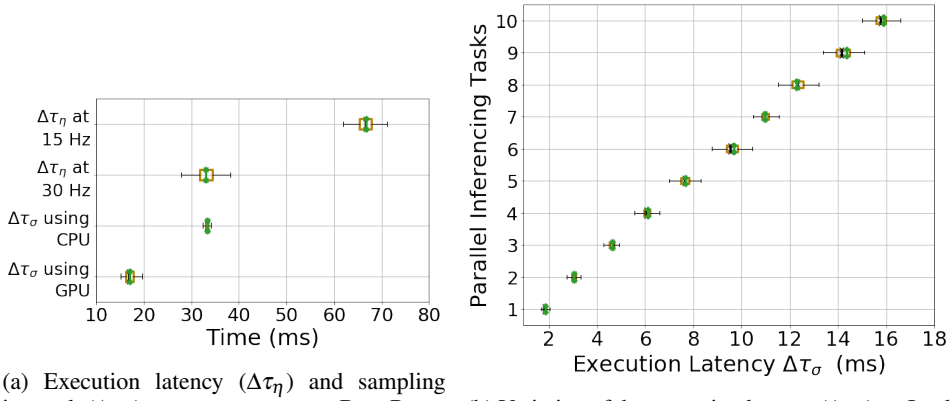
$$L_\phi = \frac{1}{2} \left\| \sum_t V_\phi(s_t) - \big(r_t + \gamma V_\phi(s_{t+1})\big) \right\|^2 \tag{5}$$

Due to variable state transition delay $\Delta\tau$, the agent observes stochasticity in state transitions $p(s_{t+1}|s_t,a_t)$ for the same state and action. Ignoring the variations in $\Delta\tau$ during training results in a distribution mismatch from simulations to the real world, leading to poor transfer. On the other hand, if we introduce domain randomization in time by considering variable $\Delta\tau$ during training, the additional state transition stochasticity introduces noise in the value function estimates $V_\phi(s)$ estimates in Equation 5 and makes it difficult to converge to a good policy.

To address variations in the state transition delay, we propose augmenting the agent state with execution time $\Delta\tau_\eta$ and sampling interval $\Delta\tau_\sigma$ measurements: $\tilde{s} = [s, \Delta\tau_\eta, \Delta\tau_\sigma]$ where $\tilde{s}$ represents the augmented state. This simple trick enables the agent to distinguish between state transitions introduced by variations in delays. We train the agent with the augmented state and introduce delay variations in the simulator. As the delays are explicitly represented in state, it becomes much easier to estimate the value function $V_\phi(s)$ using Equation 5. Since the delay measurements are directly fed as input to both policy and value networks, the agent learns to generalize beyond the exact numbers seen during training.

Num. of CNN layers	2	3	4
Network parameters	54k	157k	267k
Execution Latency	7.5ms	19.75ms	55.85ms

Table 2: Execution latency ($\Delta\tau_\eta$) on GAP8 increases the increase in the number of CNN layers in the neural network.



(a) Execution latency ($\Delta\tau_\eta$) and sampling interval ($\Delta\tau_\sigma$) measurements on DeepRacer robotic car.

(b) Variation of the execution latency ($\Delta\tau_\eta$) on Intel neural compute with parallel inferencing tasks.

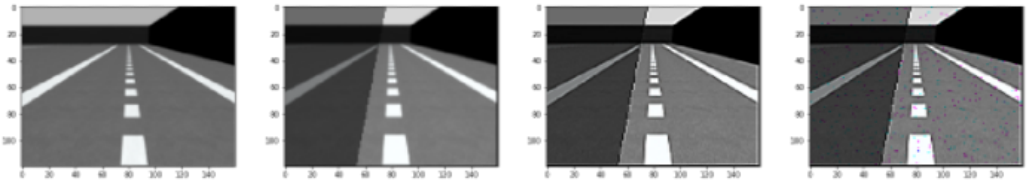
Figure 6: Delay measurements on deepracer and Intel neural compute stick. The mean is shown in green. The back 'x' marker shows the median of IQR.

B Delay Measurements on Different Hardware Platforms

Figure 6a shows the delay measurements of the navigational policy using the Deepracer robotic car. The execution latency ($\Delta\tau_\eta$) is dependent on the choice of the hardware resource at runtime (GPU vs. CPU). Deepracer camera supports a sampling rate of 15Hz and 30Hz. The sampling interval ($\Delta\tau_\sigma$) is 20-45 ms in the 30Hz frame rate setting and 62-71ms in the 15Hz setting, respectively. We also measure the execution latency when the complexity of the neural network is increased by adding more CNN layers and in the presence of other inference tasks. The complexity of the neural network and the required compute requirements vary with additional CNN layers that can happen when an application selects a more complex network to achieve better inference accuracy. The case for computing in the presence of other inference tasks can show up in two ways: (i) multiple models need to run for the same robotic application; (ii) an edge server acting like a machine learning model server for several applications.

We analyze the execution latency of shallow neural networks on a low power microcontroller device (GAP8) and edge accelerator (Intel neural compute stick 2). GAP8 is a milli-watt range micro-controller device having a dedicated CNN accelerator enabling battery-operated edge devices with rich analytics capabilities. We analyze the execution latency of the neural network trained using the MNIST dataset on GAP8 as the number of CNN layers is increased from 2 to 4. The network consists of the CNN layers followed by an output layer. Table 2 shows the increase in inferencing latency from 7.5ms to 55.90ms on increasing CNN layers from 2 to 4. Average results for 10 runs are reported. The variations across individual runs are very small (few microseconds).

Intel neural compute stick 2 (NCS2) is a deep learning processor on a USB stick that provides faster neural network inference capabilities to Raspberry Pi like edge devices. We analyze the variation in execution latency of a neural network with 2 CNN layers and an output layer trained using the MNIST dataset in the presence of other inference tasks on NCS2. We simulation parallel inference tasks by hosting the same neural network multiple times, all of which are running parallel. Figure 6b shows the results. As seen, when more parallel tasks are using the same hardware resource, in this case, the NCS2, the execution latency can increase up to 10x times.



(a) Original Image (b) Random Shadow (c) Random Shadow + Sharpen (d) Random Shadow+ Sharpen+Random noise

Figure 7: Image augmentations applied to enable successful Sim2Real transfer.

C Additional Details on HalfCheetah and Ant Tasks

We vary the state transition delays ($\Delta\tau_\sigma$ and $\Delta\tau_\eta$) in the simulator when training the policies for HalfCheetah and Ant Tasks. For HalfCheetah and Ant Tasks, PyBullet simulator evolves physics at a fixed time (Sim_{Time}) of 4.12 ms for each action. When the agent acts, the simulation is advanced by applying the past action for $\frac{\Delta\tau_\eta}{Sim_{Time}}$ simulation steps, and the most recent action for $\frac{\Delta\tau_\sigma - \Delta\tau_\eta}{Sim_{Time}}$ simulation steps.

State space. The default state of *HalfCheetahBulletEnv-v0* and *AntBulletEnv-v0* is used for domain randomization policies. To train a policy with time in the state, $\Delta\tau_\sigma$ and $\Delta\tau_\eta$ are directly added as another state thereby increasing the input dimensions by 2.

Actions. The actions available in default environments for *HalfCheetahBulletEnv-v0* and *AntBulletEnv-v0* is used.

Reward function. For every agent's action, the simulation is advanced for multiple simulation steps depending on the $\Delta\tau_\eta$ and $\Delta\tau_\sigma$. This results in multiple reward calculation for each simulation step. For every agent's action, we take the average of the rewards from all simulation steps. This ensures the reward is on the same scale as the default environments.

Hyperparameters. For training, we use the default hyperparameters from OpenAI Baselines PPO implementation other than making the following changes. We used a learning rate of $3 * 10^{-4}$ and 10,000 steps between policy update.

D Additional Details on Autonomous Vehicle Task

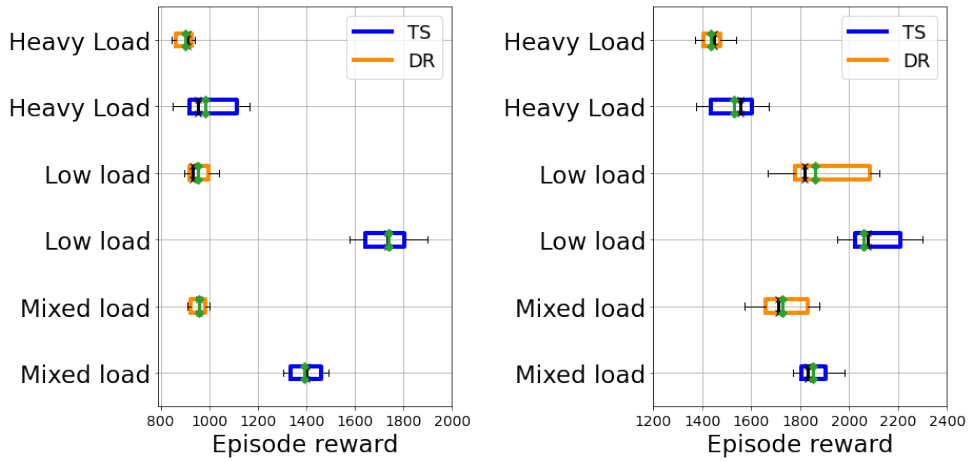
State space. The images (160x120) are directly used to train the domain randomization (DR) policy. For time in state (TS) policy, the sampling interval and execution latency are fused with images using the approach of multimodal fusion.

Image augmentations. We apply augmentations to the image by modifying its brightness randomly, adding random shadows, sharpen and random noises during the training of both DR and TS policies so as to enable successful transfer to the real track in the presence of sensor noises. Without image augmentations, we observe an inferior Sim2Real transfer. Figure 7 visualizes the image augmentations.

Actions. The action space of the agent consists of speed and steering angle. The agent is given a choice of 15 actions which consists of 3 different speeds (1.2m/s,1.5m/s, 1.8m/s) and 5 steering angels which are (-30,-15,0,15,30) in degrees.

Reward function. The reward signal is calculated based on the distance of the car from the centerline of the track. The highest reward of 1.1 is given when the center of the car matches the centerline, which is scaled to zero when the distance from the centerline makes the car close to offtrack. The agent is rewarded more for high-speed actions. A negative reward of -30 is given to the agent when the car goes off the track.

Hyperparameters. We use the default hyperparameters from OpenAI Baselines PPO implementation other than making the following changes. We fixed 7,000 Steps between policy update and an entropy coefficient of 0.001.



(a) HalfCheetah task

(b) Ant task

Figure 8: Comparison of time-in-state (TS) and domain randomization (DR) policies for HalfCheetah and Ant tasks across different multitenancy settings. The mean is shown in green. The back 'x' marker shows the median of IQR.

E Experiments with Variable Delays within an Episode

The experiments at different delays shown in Figure 4 and Figure 5 highlight Time-in-state policy's superior performance across a wide range of delay variations, along with quantifying the drop in performance as the delay magnitude is increased across episodes. Next, we expose the trained fully connected policies of Halfcheeth and Ant tasks to variable delay in a single episode.

We consider three different multitenancy settings: (i) low load, (ii) heavy load, and (iii) mixed load, exposing policies to a range of delay variations. These experiments simulate the arrival of multiple parallel tasks on the same hardware running the deep RL policy. In the low load setting, the execution latency is selected between [0-2 $*$ Sim_{Time}] randomly for each step during the episode. For both HalfCheetah and Ant Task, the Sim_{Time} = 4.12ms, and each episode consists of 1000 steps. For heavy load, the execution latency is varied randomly between [6 $*$ Sim_{Time}-10 $*$ Sim_{Time}]. The execution latency is varied randomly between [0-2 $*$ Sim_{Time}] for the first 333 steps, [3 $*$ Sim_{Time}-5 $*$ Sim_{Time}] for the next 333 steps, and [6 $*$ Sim_{Time}-10 $*$ Sim_{Time}] for the remaining 334 steps in the case of mixed load setting. The sampling interval is selected to be a maximum of (Sim_{Time}, execution latency). A random jitter of $\pm Sim_{Time}$ is added to the execution latency and the sampling interval during each step to account for the measurement noises.

As shown in Figure 8, Time-in-State policies have superior performance as compared to the domain randomization policies. The spread of reward is captured across 10 episodes for each model at a particular multitenancy setting. The variation in Figure 8 follows the same behavior shown in Figure 4. For example, at a low execution latency (in Figure 4) for the HalfCheetah task, the difference in the performance of Time-in-state policy and domain randomization policy is significant. A similar significant performance difference is observed in the low load setting for the HalfCheetah task in Figure 8. The heavy load and mixed load also follow the performance difference observed in Figure 4.

F Experiments with Recurrent Policies

The recurrent policies are known to perform better than the fully connected policies for tasks where the partial state is observed. We evaluate TS and DR policies with recurrent architectures for the HalfCheetah task. The state space, actions, and reward function used are the same as the one discussed in Appendix C for fully connected policies. We modified the open-source code available from

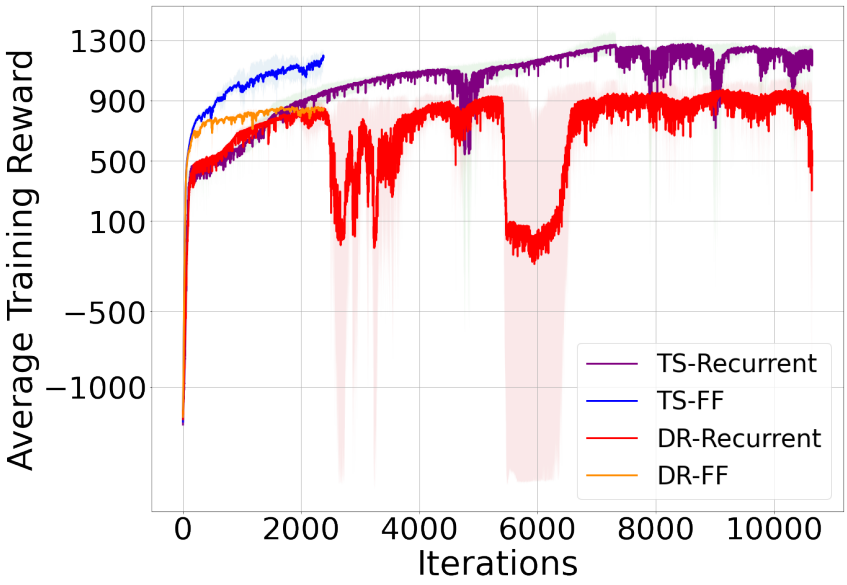


Figure 9: Learning curves of time-in-state recurrent (TS-Recurrent), time-in-state fully connected (TS-FF), domain randomization recurrent (DR-Recurrent), and domain randomization fully connected (DR-FF) policies for HalfCheetah task. The fully connected policies are trained for $\sim$ 2400 iterations whereas the fully connected policies are trained for $\sim$ 10000 iterations. TS policies achieve higher training reward than the DR policies.

Hafner et al.[5] to train recurrent policies. The variation of timing characteristics during the training was done as discussed in Section 3.1. The network architecture consists of a fully connected layer with 64 nodes, followed by a GRU layer with 64 units, and an output layer.

We train 3 models each for recurrent TS and recurrent DR, with different seeds. Figure 9 compares the learning curves of recurrent and fully connected policies. The learning curves for fully connected policies, also shown in Figure 3a, are added here for comparison. We stopped the training of fully connected policies after $\sim$ 2400 iterations. The recurrent policies require a significantly larger number of iterations ($\sim$ 10000) to train. The recurrent TS and recurrent DR achieve higher training rewards as compared to the fully connected TS and fully connected DR respectively. The max average training reward achieved by fully connected TS and fully connected DR is 1199 and 857 respectively. The recurrent TS and recurrent DR achieves max average training reward of 1278 and 974 respectively. We observe that the fully connected TS achieves significantly higher training reward than the recurrent DR, suggesting that adding time to the state is a better approach than training a recurrent DR policy.

Figure 10 shows the comparison of recurrent policies and fully connected across three trained models. The Sim2sim comparison of fully connected policies is added from Figure 4a for comparison. The spread of test reward is captured across 10 episodes for each model at a particular sampling interval ($\Delta\tau_\sigma$) and execution latency ($\Delta\tau_\eta$). The TS policies perform better than the DR policies across a range of state transition delay variations, maintaining a higher mean test reward.

G Vanilla Policy without Varying Timing Characteristics

The vanilla fully connected and vanilla recurrent policies for the HalfCheetah task are trained without varying the $\Delta\tau_\sigma$ and $\Delta\tau_\eta$. The policies are trained by using the default *HalfCheetahBulletEnv-v0* environment in which the simulation is advanced for a fixed time (Sim_{Time}) of 4.12 ms for each action. By default, the execution latency ($\Delta\tau_\eta$) of 0 is present, as the simulation is paused when deciding action (by doing neural network inference and other processing). Since updated state is

[5]Hafner, Danijar, James Davidson, and Vincent Vanhoucke. "Tensorflow agents: Efficient batched reinforcement learning in tensorflow." arXiv preprint arXiv:1709.02878 (2017).

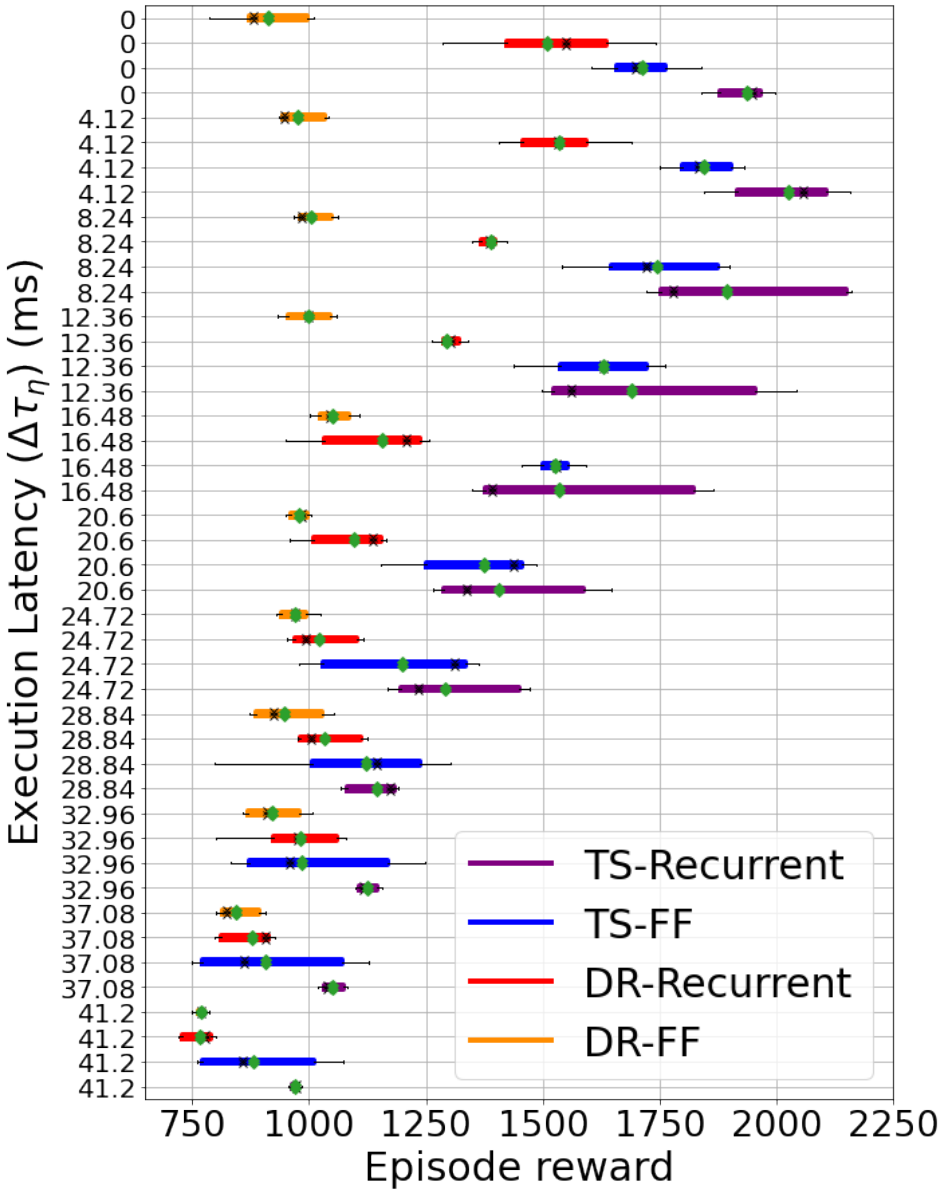
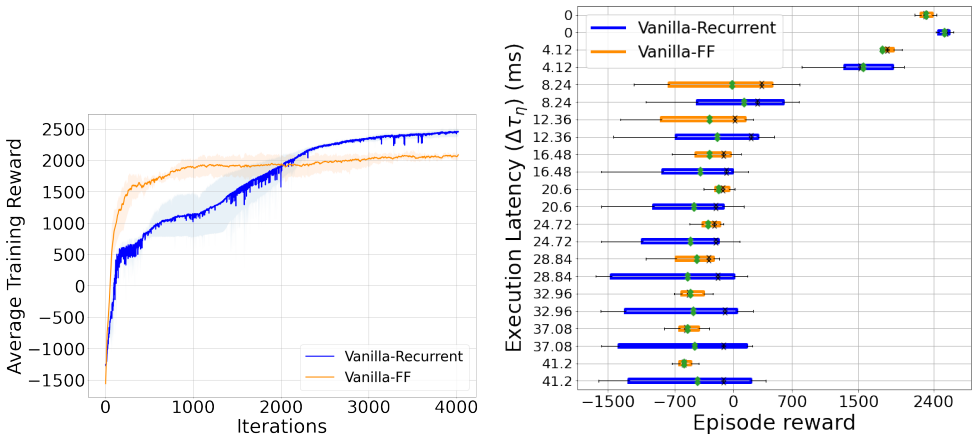


Figure 10: Sim2sim comparison of time-in-state recurrent (TS-Recurrent), time-in-state fully connected (TS-FF), domain randomization recurrent (DR-Recurrent), and domain randomization fully connected (DR-FF) policies for HalfCheetah task. The comparison is done across different execution latencies ($\Delta\tau_\eta$). The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (4.12 ms , $\Delta\tau_\eta$), so that agent can act for each sensed state. The mean is shown in green, the black 'x' marker shows the median of IQR. TS policies achieve higher mean reward than DR policies.

available every Sim_{Time}, it has a fixed sampling interval ($\Delta\tau_\sigma = Sim_{Time}$). We train three models, both for vanilla recurrent and vanilla fully connected with different seeds. The learning curves of the vanilla fully connected and vanilla recurrent policies are shown in Figure 11a. The vanilla recurrent policies achieve higher training reward than the vanilla fully connected policies. The mean test reward achieved by the vanilla recurrent policies is higher than the vanilla fully connected policies for the settings that are same as the training settings ($\Delta\tau_\eta = 0$, $\Delta\tau_\sigma = Sim_{Time}$) as shown in Figure 11b. However, both vanilla recurrent policies and vanilla fully connected policies fails to work for execution latencies ($\Delta\tau_\eta$) $\geq$ 8.24 ms, which is twice the Sim_{Time}. This motivates the need to have variable state transition delays ($\Delta\tau_\sigma$ and $\Delta\tau_\eta$) during training to have policies robust to variable timing characteristics.



(a) (b)

Figure 11: (a) The learning curves for vanilla fully connected (Vanilla-FF) and vanilla recurrent policies (Vanilla-Recurrent) for the HalfCheetah task. (b) The evaluation of vanilla policies across different execution latencies ($\Delta\tau_\eta$). The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (4.12ms , $\Delta\tau_\eta$). The mean is shown in green. The back 'x' marker shows the median of IQR.

H Extended Discussion on Control System Approaches

Multiple researchers have investigated the design of classical controllers in the presence of delays. We present a summary in Section 5. Here, we include more relevant literature. Traditionally, the delay is modeled and incorporated in the design of optimal controllers to compensate for it [6, 7, 8, 9, 10, 11, 12]. Wittenmark et al.[6] models the delays showing complicated patterns in nested communication loops suggesting that it is important to consider delays in the design of the controller. Hespanha et al.[7] and Lian et al.[8] discuss the presence of variable delays in networked controlled systems. The different approaches to handle network delay include network protocols guaranteeing constant delay, usage of buffers to transform variable delays to the constant delay, and the assumption of worst-case delay.

Nilsson et al.[9], Åström et al.[10] and Liberzon[11] discusses the design of optimal controllers in presence of delays using Lyapunov functions. Sharon et al.[12] discuss the design of networked control systems in the presence of small delays. Unlike controller designed via analytical means, the DNN-based controller trained via RL is a black box. There are no known mechanisms to compensate for delays and, thus, the overall system performance degrades when the delay varies between training and deployment.

[6] Wittenmark, Björn, Ben Bastian, and Johan Nilsson. "Analysis of time delays in synchronous and asynchronous control loops." Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No. 98CH36171). Vol. 1. IEEE, 1998.

[7] Hespanha, Joo P., Payam Naghshtabrizi, and Yonggang Xu. "A survey of recent results in networked control systems." Proceedings of the IEEE 95.1 (2007): 138-162.

[8] Lian, Feng-Li, James Moyne, and Dawn Tilbury. "Time delay modeling and sample time selection for networked control systems." Proceedings of ASME-DSC. Vol. 20. New York: DCS, 2001.

[9] Nilsson, Johan. "Real-time control systems with delays." (1998).

[10] Åström, Karl J., and Björn Wittenmark. Computer-controlled systems: theory and design. Courier Corporation, 2013.

[11] Liberzon, Daniel. "Quantization, time delays, and nonlinear stabilization." IEEE Transactions on Automatic Control 51.7 (2006): 1190-1195.

[12] Sharon, Yoav, and Daniel Liberzon. "Stabilization of linear systems under coarse quantization and time delays." IFAC Proceedings Volumes 43.19 (2010): 31-36.