Information Theoretic MPC Using Neural Network Dynamics

Grady Williams, Nolan Wagener, Brian Goldfain, Paul Drews, James M. Rehg,
Byron Boots, Evangelos A. Theodorou
Institute for Robotics and Intelligent Machines, School of Interactive Computing,
The Daniel Guggenheim School of Aerospace Engineering
Georgia Institute of Technology
Atlanta, GA, USA

Abstract

We introduce an information theoretic model predictive control (MPC) algorithm that is capable of controlling systems with dynamics represented by multi-layer neural networks and subject to complex cost criteria. The proposed approach is validated in two difficult simulation scenarios, a cart-pole swing up and quadrotor navigation task, and on real hardware on a 1/5th scale vehicle in an aggressive driving task.

1 Introduction

Many robotic control tasks can be phrased as reinforcement learning (RL) problems, where a robot seeks to optimize a cost function encoding a task by utilizing data collected from the system. The types of reinforcement learning problems encountered in robotic tasks are frequently continuous state-action and high dimensional [1], and the methods for solving these problems are often categorized into model-free and model-based approaches. Model-free approaches to RL have been successfully applied to many challenging tasks [2, 3, 4, 5, 6, 7, 8, 9]. These approaches typically require an expert demonstration to initialize the learning process, followed by many interactions with the actual robotic system. Unfortunately, model-free approaches often require a large amount of data from these interactions, which limits their applicability. Additionally, while optimization of the initial demonstrated policy leads to improved task performance, in the most popular gradient-based appraches the resulting solution remains within the envelope of the initially demonstrated policy. This limits the method's ability to discover novel optimal control behaviors. In the second paradigm, model-based RL approaches first learn a model of the system and then train a feedback control policy using the learned model [10, 11, 12]. The difficulty in model-based RL is handling modeling error, which can accumulate very quickly.

Despite all the progress on both model-based and model-free RL methods, generalization remains a primary challenge. Robotic systems that operate in changing and stochastic environments must adapt to many different situations and be equipped with fast decision making processes. In the standard RL framework this means obtaining a large training set of realistic scenarios for training a policy. This is often difficult or completely impractical in the context of robotics.

Model predictive control (MPC), or receding-horizon control, circumvents the problem of learning a policy by repeatedly computing an open-loop control law online for a short time horizon. This approach has several distinguishing features: the model is only required to be accurate over the given time horizon and model predictive controllers implicitly are feedback control policies. The idea we pursue in this work is to learn a model of the system using deep neural networks, and then solve reinforcement learning problems with MPC and the learned model.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

The theory for model predictive control of linear systems is well understood and has many successful applications in the process industries [13, 14], and, for non-linear systems, model predictive control is an increasingly active area of research in control theory [15]. However these methods focus on stabilization or trajectory tracking. To apply MPC to reinforcement learning problems, we need to be able to consider more complex cost criteria. One MPC approach capable of handling complex cost criteria and non-linear dynamics is based on sampling methods derived from the path integral control framework [16, 17]. These model predictive path integral (MPPI) control methods [18, 19, 20] sample thousands of possible trajectories from the system at every timestep in order to compute an optimal control sequence. Earlier approaches used physics-based methods to obtain approximate system models, which obey the condition that they are affine in the control input (i.e $\mathrm{d}\mathbf{x} = (\mathbf{f}(\mathbf{x}) + \mathbf{G}(\mathbf{x})\mathbf{u})\mathrm{d}t$), which is a necessary assumption for the derivation of path integral control. Unfortunately the control affine assumption limits the applicability of model predictive path integral control (MPPI) in the realm of reinforcement learning. This is because standard model learning approaches do not satisfy the control affine assumption. In the following, we show how the update law used in MPPI can be derived without making the control affine assumption. This contribution allows us to use MPPI for systems with dynamics represented by learned neural networks, and we demonstrate that this approach successfully performs difficult control tasks in both simulation and on real hardware.

2 Information Theoretic Control

In this section we introduce the theoretical basis for our sampling based MPC algorithm. The derivation relies on two important concepts from information theory: the KL divergence and free energy. The result of the derivation is an expression for an optimal control law in terms of a weighted average over sampled trajectories. This leads to a gradient-free update law which is highly parallelizable, making it ideal for online computation. Consider the discrete time stochastic dynamical system:

$$\mathbf{x}_{t+1} = \mathbf{F}(\mathbf{x}_t, \mathbf{v}_t) \tag{1}$$

The state vector is denoted $\mathbf{x}_t \in \mathbb{R}^n$, and $\mathbf{u}_t \in \mathbb{R}^m$ is the commanded control input to the system. We assume that the actual input is $\mathbf{v}_t \sim \mathcal{N}(\mathbf{u}_t, \Sigma)$. This is a reasonable noise assumption for many robotic systems where the commanded input has to pass through a lower control level before it reaches the actual system. A prototypical example is the steering and throttle inputs for a car which are then taken as inputs to low-level servo-controllers.

We define $V = (\mathbf{v}_0, \mathbf{v}_1, \dots \mathbf{v}_{T-1})$ as a sequence of inputs over some number of timesteps T. This sequence is itself a random variable defined as mapping $V : \Omega \to \Omega_V$ where Ω is the sample space and $\Omega_V = \mathbb{R}^m \times \{0, 1, \dots T-1\}$ the image of Ω. Note that changing the control input sequence $U = (\mathbf{u}_0, \mathbf{u}_1 \dots \mathbf{u}_{T-1})$ will change the probability distribution for V. There are three distinct distributions which we will be interested in. First, we denote $\mathbb{P}$ as the probability distribution of an input sequence in the uncontrolled system (i.e., $U \equiv 0$), next we denote $\mathbb{Q}$ as the distribution when the control input is an open-loop control sequence. Lastly, we denote $\mathbb{Q}^*$ as an abstract "optimal distribution" which we will define shortly. The probability density functions for these distributions are denoted as $\mathbf{p}$, $\mathbf{q}$, and $\mathbf{q}^*$ respectively. Note that the density functions $\mathbf{p}$ and $\mathbf{q}$ have simple analytic forms given by:

$$\mathbf{p}(V) = \prod_{t=0}^{T-1} \frac{1}{\sqrt{(2\pi)^m |\Sigma|}} \exp\left(-\frac{1}{2}\mathbf{v}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{v}_t\right) \tag{2}$$

$$\mathbf{q}(V) = \prod_{t=0}^{T-1} \frac{1}{\sqrt{(2\pi)^m |\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{v}_t - \mathbf{u}_t)^{\mathrm{T}} \Sigma^{-1} (\mathbf{v}_t - \mathbf{u}_t)\right) \tag{3}$$

$$\tag{4}$$

Given an initial condition $\mathbf{x}_0$ and an input sequence V, we can uniquely determine the corresponding system trajectory by recursively applying $\mathbf{F}$. We thus have a mapping from inputs V to trajectories, denoted as τ. Let $\Omega_\tau \subset \mathbb{R}^n \times \{0, \dots T-1\}$ be the space of all possible trajectories and define:

$$\mathcal{G}_{\mathbf{x}_0} : \Omega_V \to \Omega_\tau \tag{5}$$

as the function which maps the input sequences to trajectories for the given initial condition $\mathbf{x}_0$. Now consider a state-dependent cost function for trajectories:

$$C(\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_T) = \phi(\mathbf{x}_T) + \sum_{t=1}^{T-1} q(\mathbf{x}_t) \tag{6}$$

where $\phi(\cdot)$ is a terminal cost and $q(\cdot)$ is an instantaneous state cost. We can use this to create a cost function over input sequences by defining $S : \Omega_V \to \mathbb{R}^+$ as the composition:

$$S = C \circ \mathcal{G}_{\mathbf{x}_0} \tag{7}$$

Now let λ be a positive scalar variable. The free-energy of the control system $(\mathbf{F}, S, \lambda)$ is defined as:

$$\mathcal{F}(V) = -\lambda \log \left(\mathbb{E}_{\mathbb{P}} \left[\exp \left(-\frac{1}{\lambda} S(V) \right) \right] \right) \tag{8}$$

Next we switch the expectation to be with respect to $\mathbb{Q}$ by adding in the likelihood ratio $\frac{\mathbf{p}(V)}{\mathbf{q}(V)}$. This yields:

$$\mathcal{F}(V) = -\lambda \log \left(\mathbb{E}_{\mathbb{Q}} \left[\frac{\mathbf{p}(V)}{\mathbf{q}(V)} \exp \left(-\frac{1}{\lambda} S(V) \right) \right] \right) \tag{9}$$

Now apply Jensen's inequality:

$$\mathcal{F}(V) \leq -\lambda \mathbb{E}_{\mathbb{Q}} \left[\log \left(\frac{\mathbf{p}(V)}{\mathbf{q}(V)} \exp \left(-\frac{1}{\lambda} S(V) \right) \right) \right] \tag{10}$$

Note that if $\frac{\mathbf{p}(V)}{\mathbf{q}(V)} \propto \frac{1}{\exp\left(-\frac{1}{\lambda} S(V)\right)}$, then the term inside the logarithm will reduce to a constant, which means that Jensen's inequality becomes an equality. This will be important shortly. Now we continue by rewriting the right-hand side of (10) as:

$$- \lambda \mathbb{E}_{\mathbb{Q}} \left[\log \left(\frac{\mathbf{p}(V)}{\mathbf{q}(V)} \right) + \log \left(\exp \left(-\frac{1}{\lambda} S(V) \right) \right) \right] \tag{11}$$

$$= \mathbb{E}_{\mathbb{Q}}[S(V)] + \lambda \mathbb{E}_{\mathbb{Q}} \left[\log \left(\frac{\mathbf{p}(V)}{\mathbf{q}(V)} \right) \right] \tag{12}$$

$$= \mathbb{E}_{\mathbb{Q}} \left[S(V) + \lambda \sum_{t=0}^{T-1} \mathbf{u}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{u}_t \right] \tag{13}$$

where the final step is a consequence of (2 – 3) and the fact that $\mathbf{v}_t$ has mean $\mathbf{u}_t$ under the distribution $\mathbb{Q}$. So, mirroring the approach in [21], we have the cost of an optimal control problem bounded from below by the free energy of the system. Now we want to define an "optimal distribution" for which the bound is tight. We define $\mathbb{Q}^*$ through its density function:

$$\mathbf{q}^*(V) = \frac{1}{\eta} \exp \left(-\frac{1}{\lambda} S(V) \right) \mathbf{p}(V) \qquad \eta = \int_{\Omega_V} \exp \left(-\frac{1}{\lambda} S(V) \right) \mathbf{p}(V) \mathrm{d}V \tag{14}$$

It is easy to see that this distribution satisfies the condition that it is proportional to $1/\exp\left(-\frac{1}{\lambda} S(V)\right)$. So Jensen's inequality reduces to an equality for $\mathbb{Q}^*$, and the bound is tight. Now that we have an optimal distribution, we can follow the approach in [19] and compute the control to push the controlled distribution as close as possible to the optimal one. This corresponds to the optimization problem:

$$U^* = \underset{U}{\operatorname{argmin}} \, \mathbb{D}_{\mathrm{KL}} \left(\mathbb{Q}^* \parallel \mathbb{Q} \right) \tag{15}$$

Our goal is to derive an expression for the optimal controls defined in (15). Using the definition of KL divergence, we have $\mathbb{D}_{\mathrm{KL}} \left(\mathbb{Q}^* \parallel \mathbb{Q} \right)$ equal to:

$$\int_{\Omega_V} \mathbf{q}^*(V) \log \left(\frac{\mathbf{q}^*(V)}{\mathbf{q}(V)} \right) \mathrm{d}V = \int_{\Omega_V} \mathbf{q}^*(V) \log \left(\frac{\mathbf{q}^*(V)}{\mathbf{p}(V)} \frac{\mathbf{p}(V)}{\mathbf{q}(V)} \right) \mathrm{d}V$$

$$= \int_{\Omega_V} \underbrace{\mathbf{q}^*(V) \log \left(\frac{\mathbf{q}^*(V)}{\mathbf{p}(V)} \right)}_{\text{Independant of } U} - \mathbf{q}^*(V) \log \left(\frac{\mathbf{q}(V)}{\mathbf{p}(V)} \right) \mathrm{d}V$$

Removing the term which does not depend on U yields:

$$U^* = \underset{U}{\operatorname{argmax}} \int_{\Omega_V} \mathbf{q}^*(V) \log \left(\frac{\mathbf{q}(V)}{\mathbf{p}(V)} \right) \mathrm{d}V \tag{16}$$

Note that we have flipped the sign and changed the minimization to a maximization. It is easy to show that:

$$\frac{\mathbf{q}(V)}{\mathbf{p}(V)} = \exp \left(\sum_{t=0}^{T-1} -\frac{1}{2} \mathbf{u}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{u}_t + \mathbf{u}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{v}_t \right) \tag{17}$$

Inserting this into (16) yields:

$$\int \mathbf{q}^*(V) \left(\sum_{t=0}^{T-1} -\frac{1}{2} \mathbf{u}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{u}_t + \mathbf{u}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{v}_t \right) \mathrm{d}V \tag{18}$$

After integrating out the probability in the first term, this expands out to:

$$-\frac{1}{2} \sum_{t=0}^{T-1} \left(\mathbf{u}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{u}_t + \mathbf{u}_t^{\mathrm{T}} \int \mathbf{q}^*(V) \Sigma^{-1} \mathbf{v}_t \mathrm{d}V \right) \tag{19}$$

This is concave with respect to each $\mathbf{u}_t$ so we can find the maximum with respect to each $\mathbf{u}_t$ by taking the gradient and setting it to zero. Doing this yields:

$$\mathbf{u}_t^* = \int \mathbf{q}^*(V) \mathbf{v}_t \mathrm{d}V \tag{20}$$

If we could sample from the optimal distribution $\mathbb{Q}^*$, then we could compute $\mathbf{u}_t^*$ by drawing samples from $\mathbb{Q}^*$ and averaging them. However, we clearly cannot sample from $\mathbb{Q}^*$, so we need to be able to compute the integral by sampling from another distribution. Consider that (20) can be rewritten as:

$$\int \mathbf{q}(V) \underbrace{\frac{\mathbf{q}^*(V)}{\mathbf{p}(V)} \frac{\mathbf{p}(V)}{\mathbf{q}(V)}}_{w(V)} \mathbf{v}_t \mathrm{d}V \tag{21}$$

So the optimal controls can be rewritten as an expectation with respect to $\mathbb{Q}$:

$$\mathbf{u}_t^* = \mathbb{E}_{\mathbb{Q}}[w(V) \mathbf{v}_t] \tag{22}$$

where the importance sampling weight $w(V)$ is:

$$\frac{\mathbf{q}^*(V)}{\mathbf{p}(V)} \exp \left(\sum_{t=0}^{T-1} -\mathbf{v}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{u}_t + \frac{1}{2} \mathbf{u}_t^{\mathrm{T}} \Sigma \mathbf{u}_t \right) \quad = \exp \left(-\frac{1}{\lambda} S(V) + \sum_{t=0}^{T-1} -\mathbf{v}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{u}_t + \frac{1}{2} \mathbf{u}_t^{\mathrm{T}} \Sigma \mathbf{u}_t \right)$$

Lastly, we make a change of variables $\mathbf{u}_t + \epsilon_t = \mathbf{v}_t$, and denote the noise sequence as $\mathcal{E} = \{\epsilon_0, \epsilon_1 \dots \epsilon_{T-1}\}$. We then have:

$$w(\mathcal{E}) = \frac{1}{\eta} \exp \left(-\sum_{t=0}^{T-1} \frac{1}{\lambda} \mathbf{q}(\mathbf{x}_t) + \frac{1}{2} \mathbf{u}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{u}_t + \epsilon^{\mathrm{T}} \Sigma^{-1} \mathbf{u}_t \right) \tag{23}$$

Since $\mathbf{u}_t^{\mathrm{T}} \Sigma^{-1} \mathbf{u}_t$ does not depend on $\mathcal{E}$, we can remove it since it appears in the normalizing term as well. So $w(\mathcal{E})$ is equal to:

$$w(\mathcal{E}) = \frac{1}{\eta} \exp \left(-\sum_{t=0}^{T-1} \left(\frac{1}{\lambda} \mathbf{q}(\mathbf{x}_t) + \epsilon^{\mathrm{T}} \Sigma^{-1} \mathbf{u}_t \right) \right) \tag{24}$$

The Monte-Carlo iterative update law is then:

$$\mathbf{u}_t^{i+1} = \mathbf{u}_t^i + \sum_{n=0}^{N-1} w(\mathcal{E}_n) \epsilon_t^n \tag{25}$$

where N is the number of trajectory samples drawn from Q.

3 MPC with Neural Network Dynamics

To deploy (25) in a MPC setting, we need a model to sample from. In the model-based RL setting, this means learning a model from data. The kinematics for our systems of interest are trivial given the velocities, so we need only learn the dynamics of each system (i.e., its acceleration). Specifically, given that the state $\mathbf{x}$ is partitioned as $\mathbf{x} = (\mathbf{q}, \dot{\mathbf{q}})$, where $\mathbf{q}$ is the configuration of the system and $\dot{\mathbf{q}}$ is its time derivative, we seek a function $\mathbf{f}$ so that the full state transition is:

$$\mathbf{x}_{t+1} = \mathbf{F}(\mathbf{x}_t, \mathbf{u}_t) = \begin{bmatrix} \mathbf{q}_t + \dot{\mathbf{q}}_t \Delta t \\ \dot{\mathbf{q}}_t + \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t)\Delta t \end{bmatrix} \tag{26}$$

where Δt is a discrete time increment. We represent $\mathbf{f}$ with a neural network and train it on a dataset of state-action-acceleration triplets $\mathcal{D} = \{(\mathbf{x}_t, \mathbf{u}_t, (\dot{\mathbf{q}}_{t+1} - \dot{\mathbf{q}}_t)/\Delta t)\}_t$ using minibatch gradient descent with the RMSProp optimizer [22].

To create a dataset for learning the model, we follow a two-phase approach. In the first phase, we collect system identification data and then train the neural network. For collecting system ID data in simulation the MPPI controller with the ground truth model is run, whereas a human driver is used to gather data in real-world experiments. The ability to collect a bootstrapping dataset in this manner is one of the main benefits of model-based RL approaches: they can use data collected from *any* interaction with the system since the dynamics do not usually depend on the algorithm controlling the system or the task being performed. In the second phase, we repeatedly run the MPPI algorithm with the neural network model, augment the dataset from the system interactions, and re-train the neural network using the augmented dataset. In some cases, the initial dataset is enough to adequately perform the task. We perform this procedure using a range of network sizes in order to determine the effect of network configuration on control performance. For each simulated system we tried three networks, each with two hidden layers of sizes 16-16, 32-32, and 64-64, respectively. For the experiments on the autonomous driving platform we used a neural network with two hidden layers of 32 neurons each. For all neural networks, we used the hyperbolic tangent nonlinearity.

Algorithm 1: MPPI

Given : $\mathbf{F}$: Transition Model;
K: Number of samples;
T: Number of timesteps;
$(\mathbf{u}_0, \mathbf{u}_1, ...\mathbf{u}_{T-1})$: Initial control sequence;
Σ, ϕ, q, λ: Control hyper-parameters;
while *task not completed* **do**
  $\mathbf{x}_0 \leftarrow$ GetStateEstimate();
  for $k \leftarrow 0$ **to** $K-1$ **do**
    $\mathbf{x} \leftarrow \mathbf{x}_0$;
    Sample $\mathcal{E}^k = \{\epsilon_0^k, \epsilon_1^k, \dots \epsilon_{T-1}^k\}$;
    for $t \leftarrow 1$ **to** T **do**
      $\mathbf{x}_t \leftarrow \mathbf{F}(\mathbf{x}_{t-1}, \mathbf{u}_{t-1} + \epsilon_{t-1}^k)$;
      $S(\mathcal{E}^k)$ += $\mathbf{q}(\mathbf{x}_t) + \lambda \mathbf{u}_{t-1}^{\mathrm{T}} \Sigma^{-1} \epsilon_{t-1}^k$;
    $S(\mathcal{E}^k)$ += $\phi(\mathbf{x}_T)$;
  $\beta \leftarrow \min_k(S(\mathcal{E}^k))$;
  $\eta \leftarrow \sum_{k=0}^{K-1} \exp\left(-\frac{1}{\lambda}(S(\mathcal{E}^k) - \beta)\right)$;
  for $k \leftarrow 0$ **to** $K-1$ **do**
    $w(\mathcal{E}^k) \leftarrow \frac{1}{\eta} \exp\left(-\frac{1}{\lambda}(S(\mathcal{E}^k) - \beta)\right)$;
  for $t \leftarrow 0$ **to** $T-1$ **do**
    $\mathbf{u}_t$ += $\sum_{k=1}^{K} w(\mathcal{E}^k)\epsilon_t^k$;
  SendToActuators($\mathbf{u}_0$);
  for $t \leftarrow 1$ **to** $T-1$ **do**
    $\mathbf{u}_{t-1} \leftarrow \mathbf{u}_t$;
  $\mathbf{u}_{T-1} \leftarrow$ Intialize($\mathbf{u}_{T-1}$);

Once we have a trained neural network, we can apply our iterative update law in an MPC context. In this setting optimization and execution take place simultaneously: a control sequence is computed, and then the first element of the sequence is executed. This process is repeated using the unexecuted portion of the previous control sequence as the importance sampling trajectory for the next iteration. In order to ensure that at least one trajectory has non-zero mass (i.e., at least one trajectory has low cost), we subtract the minimum cost of all the sampled trajectories from the cost function. Note that subtracting by a constant has no effect on the location of the minimum.

The key requirement for sampling-based MPC is to produce a large number of samples in real time. As in [19], we perform sampling in parallel on a graphics processing unit (GPU) using Nvidia's CUDA architecture.

The use of neural networks as models makes sampling in real time considerably more difficult because forward propagation of the network can be expensive, and this operation must be performed $T \times K$ times. To make this tractable we take advantage of the parallel nature of neural networks and further

parallelize the algorithm by using multiple CUDA threads per trajectory. This was only done for the driving task where used eight CUDA threads per trajectory sample.

4 Experimental Results

We tested our approach in simulation on a cart-pole swing-up and quadrotor navigation tasks and on real hardware in an aggressive driving scenario. In the simulated scenarios, a convenient benchmark is the MPPI algorithm with access to the ground-truth model used for the simulation. This provides a metric for how much performance is lost by using an approximate model to the system.

4.1 Cart-Pole Swing Up

In this task, the controller has to swing and hold a pendulum upright using only actuation on the attached cart, starting with the pendulum oriented downwards. The state-dependent cost function has the form:

$$10x^2 + 500(\cos(\theta) + 1)^2 + \dot{x}^2 + 15\dot{\theta}^2 \tag{27}$$

The system noise is set at 0.9 and the temperature λ is set equal to 1. The bootstrapping dataset for the cart-pole comes from 5 minutes of multiple MPPI demonstrations using known dynamics but a different cost function for the swing-up task. These system identification trajectories show the cart-pole's behavior when the pole is upright, but they don't exhaust enough of the state-action space for the MPPI controller to act correctly immediately. This results in the first few iterations going to unseen parts of the state-action space and rapid improvement once this data is added into the training set. The cart-pole is of low enough dimensionality that no bootstrap dataset is required to perform the task, though at the cost of more training iterations.

The relative trajectory costs are shown in Fig. 1, where each iteration consists of one 10 second trial. Smaller networks (especially with bootstrapping) perform best in terms of both convergence speed and consistency. A network with 64 neurons per layer overfits the dynamics early on and causes the system to take costly trajectories, even with bootstrapping. We also note that all MPPI controllers with neural network dynamics converge to nearly the same cost as the ideal MPPI controller, which results in successful swing-up and balancing of the pole.

4.2 Quadrotor Navigation

For this task, a quadrotor must fly from one corner of a field to the other while avoiding circular obstacles. The cost function is composed of a quadratic cost on distance from the target state, and a very large cost for hitting obstacles. This large cost has the effect of annihilating any trajectories which collide with an obstacle. Since the quadrotor has twelve state and four action dimensions, bootstrapping the neural network dynamics becomes necessary. Running the algorithm without a bootstrapped neural network results in no learning progress being made as the quadrotor repeatedly crashes into the ground. We bootstrap the neural network with 30 minutes of an MPPI demonstration with known dynamics and a moving target but no obstacles. This allows the quadrotor to cover enough of the state-action space to know the dynamics of maintaining altitude and flying around, but not necessarily be familiar with the maneuvers required for avoiding obstacles.

All three neural network models yield similar results, as shown in Figure 1. The bootstrap data is enough for the MPPI controller with the medium-sized network to navigate the field. However, the smallest and largest networks require an extra iteration to become competent at the task. After one iteration, the algorithm achieves the same level of performance regardless of which network is being used. This suggests that the size of the network employed does not have a significant effect on the MPPI controller performance, as long as it accurately models the system dynamics. An example trajectory successfully navigating the field is also shown in Figure 1.

4.3 Autonomous Racing

We apply our approach to the task of aggressively driving around a dirt track with a one-fifth scale AutoRally vehicle [19]. In prior work, MPPI was successfully applied to this task using a physics-inspired model. In these experiments, a neural network is used in place of this hand-designed model. To train an initial model, we collected a system identification dataset of approximately 30 minutes of

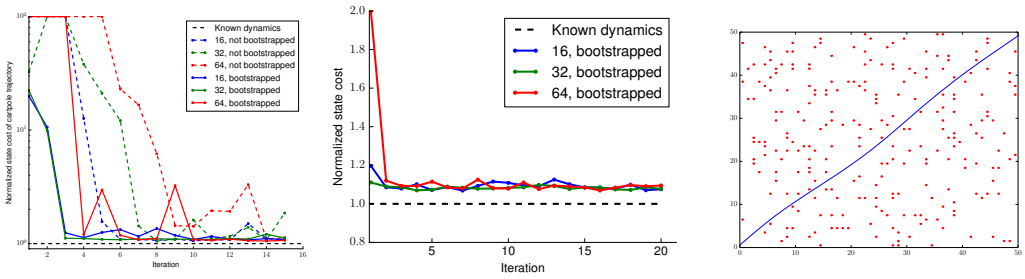

Figure 1: Left: Cart-Pole swing up results. Middle: Quadrotor navigation results. Right: Example path taken through obstacle field.

human-controlled driving at speeds varying between 4 and 10 m/s. The driving was broken into five distinct behaviors: (1) normal driving at low speeds (4–6 m/s), (2) zig-zag maneuvers performed at low speeds (4–6 m/s), (3) linear acceleration maneuvers which consist of accelerating the vehicle as much as possible in a straight line, and then braking before starting to turn, (4) sliding maneuvers, where the pilot attempts to slide the vehicle as much as possible, and (5) high speed driving where the pilot simply tries to drive the vehicle around the track as fast as possible. Each one of these maneuvers was performed for three minutes while moving around the track clockwise and for another three minutes moving counter-clockwise.

The experimental setup (Fig. 2) consists of an elliptical dirt track approximately 3 meters wide and 30 meters across at its furthest point. The cost function consists of a desired speed, maximum allowable

Figure 2: Left: Experimental set-up for AutoRally tests. Right: Drifting around a turn with the MPPI controller and a neural network model.

side-slip angle, and a penalty for being off the track. The MPPI controller is provided with a global map of the track in the form of a cost-map grid. This cost-map is a smoothed occupancy grid with values of zero corresponding to the center of the track, and values of one corresponding to terrain that is not a part of the track. The cost-map grid has a 10 centimeter resolution and is stored in CUDA texture memory for efficient look-ups inside the optimization kernel. We use a neural network with 2 hidden layers of 32 neurons each and hyperbolic tangent non-linearities. The MPPI controller uses a time horizon of 2.5 seconds, a control frequency of 40 Hz, and performs 1200 samples every time-step. This corresponds to 4.8 million forward passes through the neural network every second. On-board computation is performed using an Nvidia GTX 750 Ti GPU, which has 640 CUDA cores.

We trained the model using the same bootstrapping method as for the simulation tasks, during the initial training runs we set the desired speed to 9 m/s and the maximum slip angle to 0.275 radians. This results is relatively conservative behavior. Adding new data into the initial training set and re-training the neural network model did not noticeably improve the performance of the algorithm. One explanation for this is that the initial dataset was deliberately collected for system identification, and it consists of a variety of maneuvers meant to excite various modes of the dynamics. This is in contrast to the simulated experiments where the initial dataset consists of a *single* task repeatedly performed by an expert controller.

After the initial training runs, we tested the limits of the controller, using the model from the final training iteration. Specifically, we increased the threshold for penalized slip angle to 0.375 radians and increased the desired speed. We started with the desired speed at 10 m/s and gradually increased it to 13 m/s.

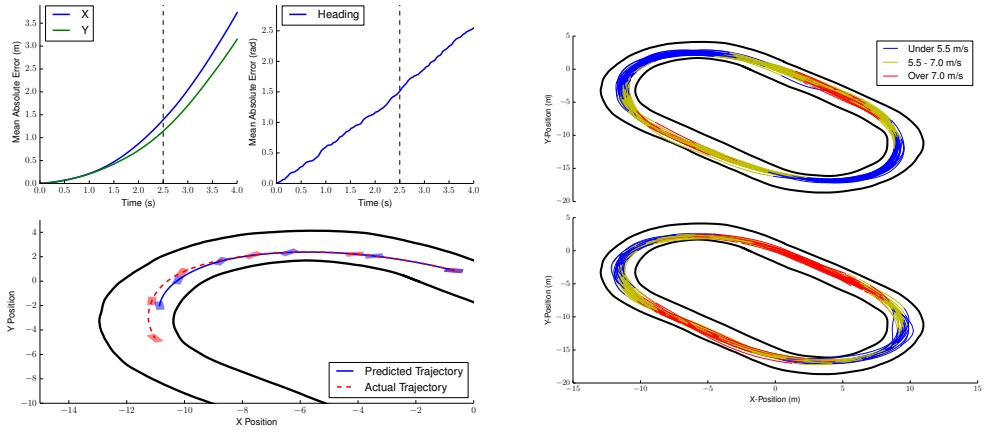

Figure 3: Top Left: Multi-step prediction error for car dynamics. Bottom Left: Actual vs. predicted trajectory over 2.5 second long time horizon. Top Right: Trajectory paths taken during training runs. Bottom Right: Trajectories taken during test runs.

Table 1: Test run statistics

Target Speed	Avg. Lap (s)	Best Lap (s)	Top Speed (m/s)	Max. Slip
10 m/s	10.34	9.93	8.05	38.68
11 m/s	9.97	9.43	8.71	34.65
12 m/s	9.88	9.47	8.63	43.72
13 m/s	9.74	9.36	8.44	48.70

Figure 3 shows the paths taken by the controller along with their velocity profiles. In both cases, the vehicle slows down coming into turns and accelerates out of them, eventually reaching a high speed along the middle of the straight. However, in the more aggressive test runs the vehicle carries considerably more velocity into turns and reaches a higher peak velocity along the straights.

5 Conclusion

We have derived an information theoretic version of model predictive path integral control which generalizes previous interpretations by allowing for non-affine dynamics. We exploited this generalization by applying the MPPI algorithm in the context of model-based reinforcement learning and used multi-layer neural networks to learn a dynamics model. In two challenging simulation tasks, the controller with the learned neural network model achieves performance within 10% of what is obtained with a perfect ground-truth model. The scalability and practicality of the algorithm was demonstrated on real hardware in an aggressive driving scenario, where the algorithm was able to race a one-fifth scale rally car around a 30 meter long track at speeds over 8 m/s.

This type of model-based reinforcement learning that we propose, combining generalized model predictive control with neural networks for learning dynamics, is a promising new direction for solving the challenging problems that arise in robotics. The key tools in this approach are the information theoretic concepts of free energy and the KL divergence, scalable machine learning algorithms for learning dynamics, and intensive parallel computation for online optimization.

References

[1] Jens Kober and Jan Peters. Reinforcement learning in robotics: A survey. In *Reinforcement Learning*, pages 579–610. Springer, 2012.

[2] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2219–2225. IEEE, 2006.

[3] Hamid Benbrahim and Judy A Franklin. Biped dynamic walking using reinforcement learning. *Robotics and Autonomous Systems*, 22(3):283–302, 1997.

[4] Jens Kober and Jan R Peters. Policy search for motor primitives in robotics. In *Advances in neural information processing systems*, pages 849–856, 2009.

[5] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Robot motor skill coordination with EM-based reinforcement learning. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 3232–3237. IEEE, 2010.

[6] J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21(4):682–97, 2008.

[7] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. A generalized path integral control approach to reinforcement learning. *The Journal of Machine Learning Research*, 9999:3137–3181, 2010.

[8] J. Buchli, E. A. Theodorou, F. Stulp, and S. Schaal. Variable impedance control - a reinforcement learning approach. In *Robotics: Science and Systems Conference*, 2010.

[9] F. Stulp, E.A. Theodorou, and S. Schaal. Reinforcement learning with sequences of motion primitives for robust manipulation. *IEEE Transactions on Robotics*, 28(6):1360–1370, 2012.

[10] Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 2010.

[11] Stefan Schaal et al. Learning from demonstration. In *Advances in neural information processing systems*, 1997.

[12] Christopher G Atkeson and Juan Carlos Santamaria. A comparison of direct and model-based reinforcement learning. In *In International Conference on Robotics and Automation*. Citeseer, 1997.

[13] S Joe Qin and Thomas A Badgwell. A survey of industrial model predictive control technology. *Control engineering practice*, 11(7):733–764, 2003.

[14] David Q Mayne, James B Rawlings, Christopher V Rao, and Pierre OM Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 36(6):789–814, 2000.

[15] David Q Mayne. Model predictive control: Recent developments and future promise. *Automatica*, 50(12):2967–2986, 2014.

[16] H. J. Kappen. Path integrals and symmetry breaking for optimal control theory. *Journal of Statistical Mechanics: Theory and Experiment*, 11:P11011, 2005.

[17] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. Reinforcement learning of motor skills in high dimensions: A path integral approach. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2397–2403. IEEE, 2010.

[18] Vicenç Gómez, Sep Thijssen, Hilbert J Kappen, Stephen Hailes, and Andrew Symington. Real-time stochastic optimal control for multi-agent quadrotor swarms. *RSS Workshop, 2015. arXiv:1502.04548*, 2015.

[19] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou. Aggressive driving with model predictive path integral control. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1433–1440, May 2016.

[20] G.Williams, A.Aldrich, and E.A.Theodorou. Model predictive path integral control: From theory to parallel computation. *AIAA Journal of Guidance, Control and Dynamics*, page To appear, 2016.

[21] Evangelos A Theodorou and Emo Todorov. Relative entropy and free energy dualities: Connections to path integral and KL control. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 1466–1473. IEEE, 2012.

[22] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5 - RMSProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.