Data-Driven Modeling and Control of an Autonomous Race Car

Ohiremen Dibua, Aman Sinha, and John Subosits
{odibua, amans, subosits}@stanford.edu

***Abstract*—This paper explores the application of machine learning techniques to the modeling and control of a race car. Regression models are used to simulate the vehicle's dynamics more accurately than traditional physics-based models, and we achieve the best generalization performance with bootstrap-aggregated regression trees. We incorporate this simulation model into an approximate dynamic programming framework that attempts to outperform a professional human driver. We demonstrate successful application of this approach on a 480-meter segment of a racetrack.**

1 Introduction

The application of machine learning techniques to the control of dynamical systems is an evolving field of artificial intelligence. Stanford's Dynamic Design Lab is currently developing control algorithms for an autonomous Audi TTS, nicknamed Shelley. In this paper, we apply machine learning and data-driven techniques to the control of this car over a closed course. The goal is to derive a control law that allows Shelley to outperform human drivers.

Data, Objectives, and Approach Data was recently recorded from a professional driver completing 15 laps of Thunderhill Raceway at pace in Shelley. Our data consists of measurements taken with a high precision GPS in conjunction with an IMU and signals taken directly from the car. Together, they supply roughly 420,000 data points for 25 continuous signals encompassing the vehicle states and the controls applied by the skilled human driver.

Our overall goal is to generate a minimum-time trajectory and associated control law over Thunderhill Raceway. One might first consider the approach of Abbeel et. al. [1], i.e. assume that the 15 laps are noisy versions of some desired trajectory, and proceed to infer this hidden trajectory. However, even the professional's fastest lap is imperfect, and trials are not equivalent noisy versions of an optimal trajectory. Thus, the apprenticeship approach of [1] is inapplicable.

Our approach is the following: we first build a simulation model of the car's dynamics and then apply reinforcement learning to optimize trajectories and control laws. For the first component, we apply standard regression techniques. For the second component, we apply approximate dynamic programming (ADP) [3] in a novel way to take advantage of the nearly-optimal human trajectories and avoid unnecessary exploration.

The remainder of this report is organized as follows: we discuss preprocessing methods and state-space choices in Section 2, followed by results for regression in Section 3 and path optimization in Section 4. We conclude with a discussion of our results, their implications, and open questions for further research.

2 Preprocessing/Feature Generation

Regression After smoothing the data to remove the effects of noise, we transform the data into the following state-space: body-frame velocity components (3), yaw and pitch angle (2), body-frame angular velocity components (3), body-frame accelerations (3), and the exogenous inputs of throttle, brake, and steering positions (3). Note that absolute position is not needed in the dynamics: any indirect effects of position on the vehicle were implicitly captured through other states (e.g. accelerations implicitly capture the effects of hills). Outputs are the differences in the 11 dynamical states between consecutive time steps.

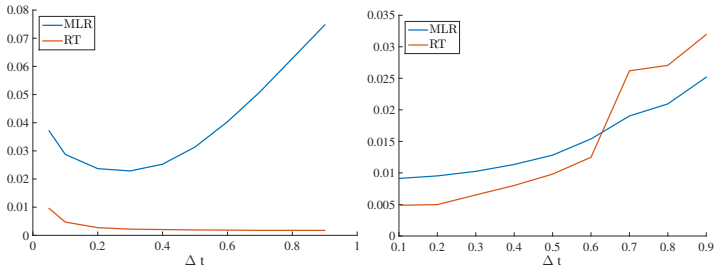
ADP Unlike with regression, position is relevant to ADP because we must ensure that the car stays on the road (between the lane boundaries) during simulation. Position along the manifold of the racetrack is maintained in curvilinear coordinates relative to the track's centerline.

3 Simulation Models

Rather than simply try a wide variety of methods, we adopt a more principled approach. Beginning with baseline methods, we iteratively consider sophistications to address key shortcomings of model performance. We use standardized mean square error (SMSE) as our error metric (MSE normalized by training data variance). Results for one-step prediction (regression) and multi-step predictions (simulation) are summarized in Table 3.1. For brevity, we only show results for predicting one-step changes in longitudinal velocity ($\Delta V_x/\Delta t$), and multi-step predictions of V_x. The normalization parameters are different for regression and simulation SMSE values: the former uses the variance of ΔV_x, whereas the latter uses the variance of V_x. Thus, relative error magnitudes *should not* be compared between regression and simulation.

3.1 Baselines

We start with multivariate linear regression (MLR) and regression trees (RT). We assume Markovian dynamics for these models (i.e. current outputs only depend on the current state and current inputs), so after computing the outputs as differ-



(a) Test regression SMSE (b) Simulation SMSE

Figure 1. (a) Errors in predicting $\Delta V_x/\Delta t$ vs. Δ_t for regression. (b) Errors in predicting V_x via simulation with baseline models.

ences between states in consecutive time steps, we randomly divide the data into 70% training and 30% test subsets.

3.1.1 Time-Step and Model Selection

Our first task in model selection is determining the time step over which we compute changes in states. The original data was recorded at $200Hz$, but simulating at this level of detail was deemed to be too costly given our time constraints as well as unnecessary to fully capture the dynamics of the car's motion. Instead, we test the regression models at time steps ranging from 0.1 to 0.9 seconds in 0.1 second increments, hypothesizing that the characteristic time scale for the car's dynamics lies in the 0.1 to 0.4 second range.

Two major observations should be noted about the regression error, i.e. the error in predicting the change in vehicle states over a single time step (Figure 1(a)). The first is that MLR's errors are worse than that of RT, indicating that we want to use the latter. The second is that in the case of the regression tree, the regression errors decrease with increasing time steps. Importantly, this *does not* imply that the largest time steps should be used for simulation, as we show next.

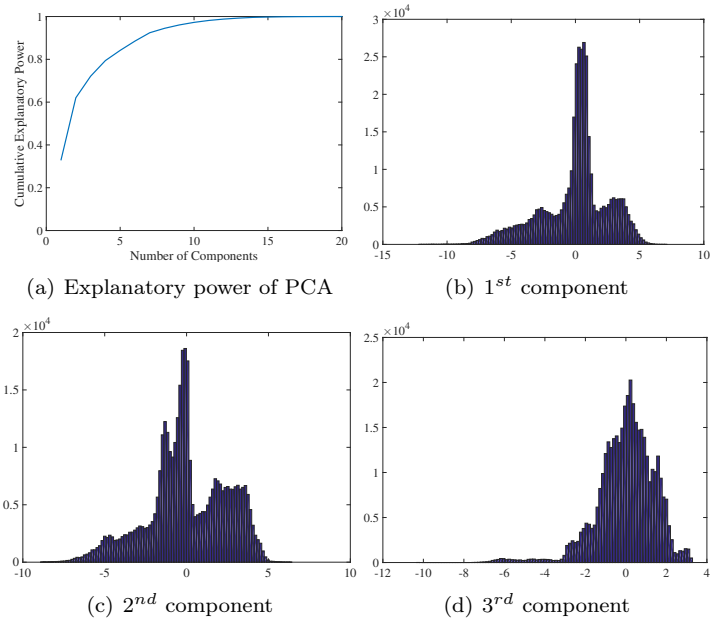
We corroborate our regression analysis with simulation, i.e. multi-step prediction, to make a more definitive conclusion. We simulate the behavior of Shelley over the course of one lap in 2-second intervals. In contrast with the regression errors, the simulation errors *increase* monotonically with time step size, as shown in Figure 1(b). According to the simulation error results, we choose $\Delta t = 0.1$, and all models in Table 3.1 use this value. We can explain the dichotomy between regression and simulation trends for RT as follows: the models trained on large time steps ignore the fast dynamics of the car, and as a result are able to easily predict the low-pass filtered data for one-step prediction. However, ignoring these dynamics in multistep predictions obviously yields drastic errors during simulation. Regression and simulation results indicate that we select RT over MLR for further analysis.

3.1.2 Turning Issues

We found that the regression tree has exceptionally poor simulation performance on turns, and we also noticed high model variance (overfitting) in key states such as lateral velocity *solely* for turning regimes. Attempts at forward/backward feature selection, specifically by adding features like tire slips and removing noisy features like vertical velocity, yielded little

Table 3.1. SMSE Errors for $\Delta V_x/\Delta t$ and V_x

Model	Regression $(\Delta V_x/\Delta t)$		Simulation
	Train	Test	(V_x)
MLR	0.02838	0.02865	0.00913
RT	0.00342	0.00471	0.00487
NARX RNN	0.00147	0.00168	0.00173
MultRT	0.00279	0.00973	0.00012
BagRT-5	1.277e-5	3.622e-5	2.773e-5
BagRT-20	1.057e-5	3.283e-5	2.528e-5
BagRT-35	1.030e-5	3.177e-5	*1.919e-5*
BagRT-100	*9.257e-6*	*2.7623e-5*	2.119e-5



(a) Explanatory power of PCA (b) 1^{st} component

(c) 2^{nd} component (d) 3^{rd} component

Figure 2. PCA on standardized data. (a) Explanatory power of components given by a normalized cumulative sum. The first 6 components are sufficiently powerful. (b), (c) and (d) are histograms of standardized data projected onto the 1^{st}, 2^{nd}, and 3^{rd} components. A positive (negative) projection onto the 1^{st} (2^{nd}) component roughly corresponds to left turns, while a vanishing projection corresponds to a straightaway. A large negative projection onto the 3^{rd} component corresponds to strong deceleration.

improvement. This indicates that the poor simulation performance in turning regimes is largely due to other effects associated with turns. Specifically, training data bias leads to high model variance in turns, while high error autocorrelations lead to poor simulations. We now analyze these issues.

3.2 Training Data Bias

We suspected that the training data was biased towards regions devoid of turning, which would account for the relatively high model variance in turning regimes compared to other regions. Unsupervised learning algorithms verify this intuition.

Principal component analysis on standardized data indicates that most of the data's variation lies in a 6-dimensional subspace of input space (Figure 2). The two dominant principal components loosely correspond to turning dynamics, with strong and opposing magnitudes for lateral acceleration, yaw

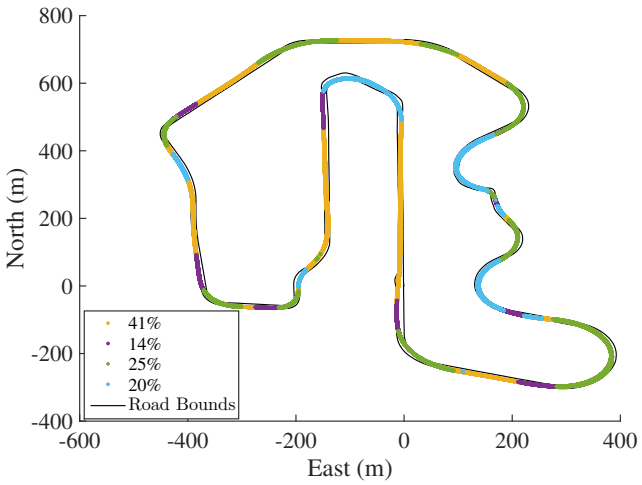


Figure 3. Data points assigned to the four clusters displayed on a map of the car's trajectory through the racetrack. Relative sizes of clusters are displayed in the legend.

rate, and steering angle. The third component distinguishes strongly decelerating regions from accelerating regions, i.e. regions just before turns vs. everything else. The remaining 3 dominant components constitute slightly perturbed forms of the first three components. Thus, we constrain ourselves to the first three components for further analysis.

The three dominant principal components signify that there are roughly 4 regions of data: right turns, left turns, straights, and regions just before turns. Clustering the reduced data reveals exactly this partitioning. We use the k-medoids algorithm, a variant of k-means that is more robust to outliers. Figure 3 shows a map of the racetrack with data points along the trajectory colored according to their respective clusters. Most importantly, we see that training data is indeed biased. Regions just before turns account for only 14% of the data, and right/left turns account about for $20-25\%$ each. Thus, generalization performance suffers in turns and regions just before turns due to a lack of training data, and the resulting high test errors propagate during simulation.

We also compared results of two nonlinear reduction algorithms with those of PCA: kernel PCA with polynomial and Gaussian kernels as well as maximum variance unfolding. Neither of these advanced methods provided further insights.

3.3 Error Autocorrelations

High error autocorrelations between consecutive state predictions lead to error propagation during simulation. Such effects are most evident when the error magnitudes are high (i.e. along turns). These effects definitely violate the assumption of error independence of models such as MLR, further motivating our choice of RT. However, even RT shows severe autocorrelation in delays of up to 15 time steps (Figure 4).

There are three ways to combat error autocorrelation: increase model complexity, reduce the magnitudes of errors to minimize correlation impact, and model and manually compensate for autocorrelated errors. We chose to attempt the first and second of these methods with improved models.

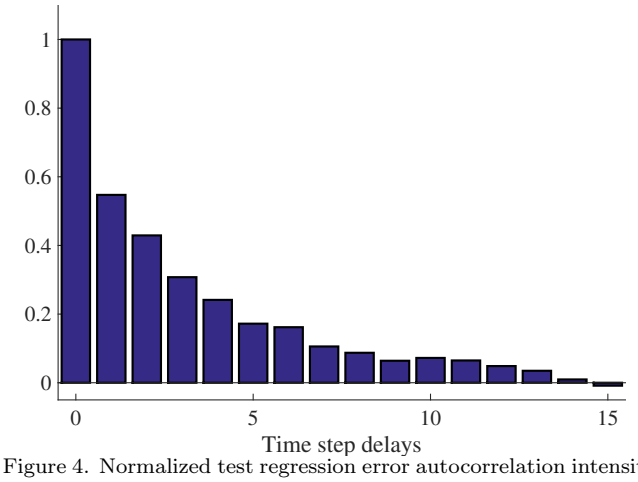


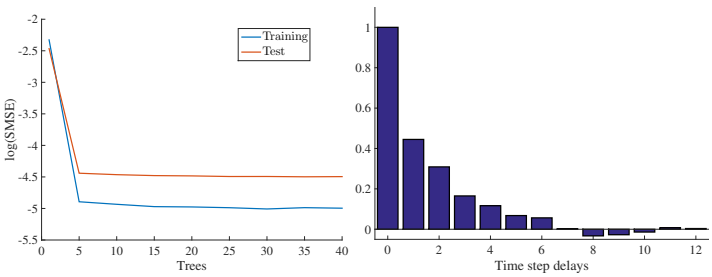
Figure 4. Normalized test regression error autocorrelation intensity for RT vs. time step delays.

3.4 Improved Models

NARX RNN In order to address issues of training bias and error autocorrelation, we first consider a more complex model: a nonlinear autoregressive model with exogenous inputs (NARX model) in the form of a recurrent neural network (RNN). The motivation for this approach is that including autoregressive components for the input can potentially reduce error correlations. We consider an RNN with two hidden layers of 30 and 10 nodes respectively as well as 5 delays for both autoregressive and exogenous inputs (i.e. feedback and input delays). As indicated in Table 3.1, performance improvements were minimal. Although we could consider training a deeper network, the training time overhead (about 7 hours on 5 machines using the Levenberg-Marquardt algorithm with Bayesian regularization) is prohibitive. Even though we could simply try other training algorithms for the RNN, we found better results with more scalable methods.

MultRT We next consider RT with different forms of ensemble learning. The main idea is that RT, a nonparametric method, should be able to capture enough model complexity to fit to data, so we simply need to improve its generalization performance in problematic regions. Ensemble methods are well suited to reducing model variance without increasing model bias. We first consider a simple ensemble: we use k-means on the most important dimensions of the data (as determined via PCA) to separate our training data into separate regimes. We then train separate RT's on each subset, and compute a weighted average of each model's predictions during simulation according to the relevance of the model to the corresponding simulation regime. As shown in Table 3.1, this ensemble RT model (MultRT) does not make significant improvements. On the contrary, although it performs a bit better on turns than RT, it performs much worse on transitions between the turns and straights, the regions where we transition between separate regimes. Although we could simply create more training partitions, we found success with a related approach: bootstrap aggregation.

BagRT Bootstrap aggregation, also known as bagging, uniformly samples from training data to train multiple RT's on



(a) Regression SMSE vs. Trees (b) BagRT-40 error autocorrelation

Figure 5. Model performance for bootstrap aggregated regression trees (BagRT).

separate subsets of training data. Predictions are formed by averaging over the predictions of each separate tree. Bagging reduces the magnitude of our regression errors (Figure 5(a)). Although these errors indicate that performance improvements plateau with only a few trees, we saw continued improvements in error autocorrelation intensities with more trees. As seen in Figure 5(b)), the characteristic time scale for autocorrelation is only about 7 time steps for a bagged RT with 40 trees (BagRT40), which is half of that for RT (Figure 4). Although this improvement seems minor, error autocorrelations have a significant impact on performance. Even this minor improvement coupled with decreased error magnitudes leads to a 2 orders of magnitude decrease in simulation errors when compared to RT (Table 3.1). The obvious next step is to try random forests, which also use random subsets of *features* to train separate trees. This method can further reduce correlation between separate trees in the bagged model, thereby further reducing model variance.

4 Path/Policy Optimization via Reinforcement Learning

Developing a simulation model as described in Section 3 is useful only if it can aid in optimizing Shelley's path and associated policy beyond the human trials. To do so, we incorporate the simulation model into a reinforcement learning framework.

4.1 ADP with Cost Shaping

In conventional value iteration, the values $V(s)$ for *every* state must be updated in an iteration (either synchronously or asynchronously). To maintain simulation accuracy, we cannot afford to discretize our state space too coarsely, so value iteration quickly becomes intractable. Similarly, fitted value iteration was determined to be unnecessarily expensive for our problem, as our human trajectories were known to be already nearly optimal. Rather, we choose to generate an optimal trajectory by generating *dynamically feasible* perturbations to the original human trajectory.

In ADP, one steps through optimal states, performing a value update for a state only upon visiting it.[1] In our implementation, we start with a human trajectory and reward the simulation for following this trajectory as closely as possible (see $R(s)$ in Equation 4.1). However, to encourage some degree of exploration around this path, we add a shaping cost F to the reward in the potential form described by Ng et al. [2]. Under this ADP scheme, the original human trajectory remains optimal, but the simulation explores the state space around this trajectory as it iteratively traverses paths. Importantly, this exploration generates perturbations to the original path that are necessarily attainable by a sequence of actions, hence they are denoted *dynamically feasible* perturbations.

Since we treat our simulation model $M : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ as deterministic, each state-action pair (s, a) evolves to the state $M(s, a)$. Then our ADP framework is summarized as follows:

$$\begin{aligned} V(s_t) &:= R(s_t) + F + \gamma \max_{a \in \mathcal{A}_t} V(M(s_t, a)) \\ F &= \gamma c - c \\ R(s) &= \max_{s_{pro}} G(s, s_{pro}) \\ G(s, s_{pro}) &= \exp\left(-(s - s_{pro})^T \Sigma^{-1} (s - s_{pro})\right) \\ a_t &:= \arg\max_{a \in \mathcal{A}_t} V(M(s_t, a)) \\ s_{t+1} &:= M(s_t, a_t) \end{aligned} \tag{4.1}$$

where $0 \leq \gamma < 1$ is the discount factor, $\mathcal{A}_t$ is the set of actions at time t, s_{pro} is a state in the human trajectory, Σ is a diagonal matrix of characteristic length scales, and $c > 0$ is a constant (whereby $F < 0$).[2] We initialize $V(s) := R(s)$, and every time we reach a pre-determined "absorbing" state signifying the end of the simulation, we simply restart at s_0. Finally, to further restrict exploration to reasonable states, we consider $\mathcal{A}_t$ to be actions confined within a certain ball of the action considered by the human at the state $s^*_{pro} := \arg\max_{s_{pro}} G(s, s_{pro})$.

Iterative Optimization We greedily optimize our path perturbations to exploit our exploration of the human trajectory. First, we generate k perturbed trajectories from the human trajectory, and we choose the best trajectory among these $k+1$ trajectories. We then set this selected trajectory as the new "human" trajectory from which to generate perturbations using Equation 4.1; we continue the process as necessary or until a local optimum is attained.

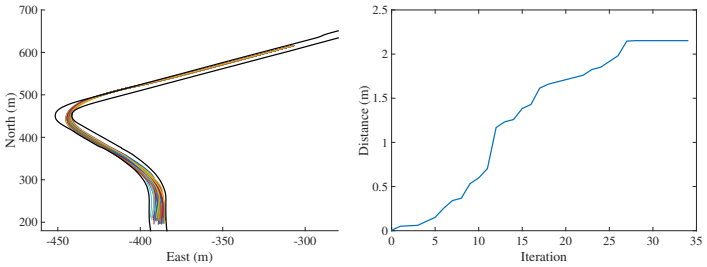
When simulating the whole track, the best of the $k+1$ trajectories is the one which completes the circuit in minimum time. When simulating only part of the track, however, we found the following to be more effective: we choose the trajectory that ends up the farthest along the track (as expressed in units along the centerline) within a specified time window.

4.2 Results

In our implementation of ADP with cost shaping, there is no need to discretize the state space $\mathcal{S}$; we only discretize the action space $\mathcal{A}_t$. We experiment over a $480m$ section along the track that has a straightaway followed by a sharp left turn and a soft right turn. Figure 6(a) illustrates some of the path perturbations during optimization, and Figure 6(b) indicates

[1] Q-learning is a special case of ADP without a model for the system.

[2] In general, we can have c be a function of state c(s) [2].



(a) (b)

Figure 6. (a) Illustrations of physical path perturbations in east/north coordinates. The start point is at the top right. Note that actual perturbations occur in $\mathcal{S}$, of which east/north position is simply a 2-dimensional subspace. (b) Distance traveled beyond the human trial in 13.4s of simulation vs. iteration of optimization scheme. A local optimum is reached at 26 iterations.

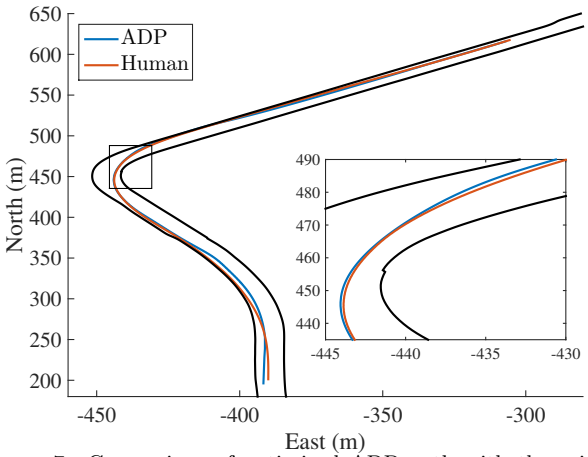


Figure 7. Comparison of optimized ADP path with the original human path. The inset displays a zoomed-in view of the left turn. The differences may appear subtle, but a $2m$ gain on every turn can yield a lead of roughly 1 second on a course with 15 turns. In Shelley's current best autonomous performances, it loses the most ground to this particular human driver in exactly these locations.

the degree of optimality with each iteration, as measured by the distance covered beyond the initial human trial. We see that the local optimum reached after 26 iterations yields a $2m$ gain over the human trial during 13.4 seconds of simulation. Figure 7 compares the optimized path with the original human trajectory. The optimized trajectory requires the car to brake later and deeper for the corner. Speed at corner entry is sacrificed for increased speed at corner exit, yielding an overall greater distance traveled.

Real-world Implementation When implementing this optimized trajectory in real life, the open-loop optimal policy will not always result in the car following the optimized path for a number of reasons. These include simulation accuracy as well as environmental variables such as wind variability, road conditions, tire wear, etc. To account for these factors, the open-loop policy can easily be augmented with feedback control techniques, from something as simple as a standard PID or LQR controller to a more sophisticated nonlinear model-predictive-control (MPC) scheme. This is currently how Shelley drives along a pre-determined trajectory. Importantly, our contributions are that we develop a dynamically feasible optimal trajectory as well as generate the feedforward component of the policy (the component of the policy before feedback augmentation). Each contribution has the potential to improve Shelley's performance compared to current methods.

5 Conclusions and Future Work

We have successfully developed a computationally efficient framework to optimize Shelley's racing performance beyond the capabilities of human drivers. Our approach includes two major components: a simulation model of the car's dynamics, and a path/policy optimization scheme designed using reinforcement learning techniques.

Our regression/simulation results illustrate an interesting phenomenon regarding our data. Because the data is biased towards regions away from those involving turning dynamics, naive regression models have a tendency to overfit in turning regimes, thereby corrupting simulation performance. Our best model, bootstrap-aggregated regression trees, reduces this model variance issue and also reduces error autocorrelations to provide reasonable simulation performance. The next step along this path towards a higher fidelity simulation model is to consider random forests and extremely randomized trees. Another way to improve model performance is to gather more data from turning regimes.

The main focus of the ADP approach for path optimization is computational efficiency. We have shown that it easily sidesteps the computational intractability of naive value iteration algorithms, and we have demonstrated its usefulness in determining optimized paths and optimized policies. We can further improve performance by sophisticating our shaping cost to more effectively balance exploration with exploitation.

We believe that our approach can easily be incorporated into current methods for autonomous vehicle racing at Stanford's Dynamic Design Lab. These machine learning techniques are a promising complement to traditional modeling and control methods employed by the group.

References

[1] Abbeel, P., Coates, A., and Ng, A. Y. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research* (2010).

[2] Ng, A. Y., Harada, D., and Russell, S. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML* (1999), vol. 99, pp. 278–287.

[3] Powell, W. B. *Approximate Dynamic Programming: Solving the curses of dimensionality*, vol. 703. John Wiley & Sons, 2007.