A Supervised Formulation of Reinforcement Learning: with super linear convergence properties

Amit Parag, Nicolas Mansard

A Supervised Formulation of Reinforcement Learning: with super linear convergence properties

Amit Parag
Université Fédérale Toulouse
LAAS-CNRS
aparag@laas.fr

Nicolas Mansard
LAAS-CNRS
nmansard@laas.fr

Abstract

Deep reinforcement learning uses simulators as abstract oracles to interact with the environment. In continuous domains of multi body robotic systems, differentiable simulators have recently been proposed but are yet under utilized, even though we have the knowledge to make them produce richer information. This problem when juxtaposed with the usually high computational cost of exploration-exploitation in high dimensional state space can quickly render reinforcement learning algorithms impractical. In this paper, we propose to combine learning and simulators such that the quality of both increases, while the need to exhaustively search the state space decreases. We propose to learn value function and state, control trajectories through the locally optimal runs of model based trajectory optimizer. The learned value function, along with an estimate of optimal state and control policies, is subsequently used in the trajectory optimizer: the value function estimate serves as a proxy for shortening the preview horizon, while the state and control approximations serve as a guide in policy search for our trajectory optimizer. The proposed approach demonstrates a better symbiotic relation, with super linear convergence, between learning and simulators, that we need for end-to-end learning of complex poly articulated systems.

1 Introduction

Reinforcement Learning (RL) [1] sets its goal as the search for an optimal policy to navigate its immediate environment. It does so by establishing its interactions with the environment as a Markov decision process where immediate action taken in the current state is driven to maximize an expected reward over a foreseeable future. In turn, it relies on an oracle to provide it with states, actions and expected rewards, with the implicit assumptions on the overall efficiency of the oracle. In high dimensional robotic systems, the oracle itself is a differentiable simulator, for instance [2], that can plan over long horizons. While an end to end RL framework can, in principle, be applied to robot learning, practical applications of RL on real world has been less successful. The algorithmic formulation of RL often requires a large number of samples [3] followed by considerable tuning. To mitigate these problems, in [4] an extension to soft actor critic approach [5] is developed to counter over sensitivity to hyper-parameters [6]. The quality of predictions of the actor itself, was examined in [7] with the conclusion that the actor learns better when learning to act optimally over a horizon rather than learning the next optimal state.

Noting that value functions are unique fixed points of *Bellman operators* of the corresponding Markov decision process and govern interactions of RL agent with its environment, an extensive analysis in [8] showed that the mathematical foundations of RL are fully not realised in the algorithmic implementation of policy gradient methods [9] : value function estimates never match the true value function and only marginally guide the search for policy. The mathematical foundation is the

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

minimization of some *stochastic objective* function based on the governing dynamics of the system and RL usually proceeds by estimating the 0^{th} order gradient of that objective [10, 11]. This is where differentiable simulators differ from RL and other derivative free methods [12, 13]. In robotic systems it is possible to compute either analytic [14] or approximate derivatives through automatic differentiation [15, 16].

Simulators, even though can handle non linearities, complex poly articulated behaviour and constraints [17], can become severely limited by the corresponding computation time, in particular shooting methods like Differential Dynamic Programming (DDP) [18, 19], can require a large number of iterations to converge. This is relatable to the application of RL solvers in continuous domain, where training can become prohibitively expensive and is strongly dependent on the sample efficiency of its exploration strategy.

However, planning over forecasts of process behaviour over long horizons by minimizing a cost function or learning optimal policies by maximizing a reward function are , in essence, complementary approaches to the underlying optimal control (Markov decision process) problem. This duality has spawned numerous approaches to combine them more rigorously, typically with the objective of making one benefit from the other. To overcome lack of consistency of RL, in [20] suitable guiding samples drawn from a trajectory optimizer are incorporated to assist direct policy search in high dimensional system. In [21], the sub optimal trajectories are refined using a trained policy which are then used as guiding samples. In [22] trajectory optimizers were used in exploration of the state space to minimize the risks associated by the RL agent acting *greedily*.

On the other hand, various approaches have tried to use learning to improve trajectory optimizers. This typically involves learning some quantity to improve the performance of trajectory optimizers. The learned quantity can be a dedicated dynamic model of the system [23] or a cost model for the task as in [24]. In [25] it was proposed to learn the value function at terminal state to offset the computation costs and serve as an anchor to drive trajectory optimizers toward goal. In [26], the authors build upon the idea of learning value function at the terminal state with high fidelity by coupling a trajectory optimizer with a sobolev supervised learning phase. Learning in Sobolev spaces, [27, 28], differs from classical regression in that it constrains learning to simultaneously match target derivatives with derivatives of the deep neural network while minimizing the error between predictions and target. This form of constrained learning has been shown to mitigate the additional cost of computing gradients of the neural network with respect to input, by being more data efficient and robust [29, 30, 31]. Supervised Sobolev training also opens the idea of learning control trajectories by forcing the learning agent to match the Riccati gains : these gradients (analytic or inexact) of control are provided by differentiable simulators.

The overall goal is to find optimal solution to a Markov decision process and in the general landscape of robot learning, this is usually done in way where either *model based* help *model free* methods find that solution or the opposite. This point is important. In our work, we attempt to combine learning and trajectory optimization in a manner such that the overall combination finds the optimal solution.

1.1 Contributions

In this paper, we propose an actor-critic *esque* coupling as a solution to optimal control problem that combines a trajectory optimizer with a reinforced learning loop: the reinforced learning loop itself is the actor, while the role of critic is played by the trajectory optimizer. We achieve this by setting an iterative loop in the backdrop of the recursivity of Bellman's optimality principle, [32, 33]: the learning depends on the data provided by the trajectory optimizer, while the efficient computation of optimal trajectories over a preview horizon by the trajectory optimizer depends on accurate learning.

This synergistic coupling can alternately be viewed as a game between two players where the optimal outcome, i.e the optimal solution to the Markov Decision Problem, is contingent upon the strategy chosen by each player to be optimal, where the strategy of each player depends on the strategy of the other player. The purpose of utilizing Bellman' optimality principle is to ensure that the strategy of both players remain guided by optimality.

We explicitly focus on learning with high accuracy and with reduced rollouts. We use Differential Dynamic Programming (DDP) [18, 34], a particular class of (direct shooting) trajectory optimizer, to give us state-value pairs and state-control trajectories which we learn in the supervised phase using three neural networks. The estimates of value function is subsequently used inside DDP as an anchor

at the terminal position, while the approximation of optimal policies serve as a guide for DDP. We iterate over this process until *convergence*.

As DDP also requires the 1^{st} and 2^{nd} order derivatives of the value function, we explicitly use gradients during training using a Sobolev Loss. 1^{st} order supervised training also has the benefit of reducing the dependence of learning on hyper parameters.

2 Preliminary

In this section, we give a brief summary of the different foundations of our algorithm and establish notations. We choose to use the optimal control formulation of MDP.

2.1 Optimal Control

We consider a time discrete formulation of (finite state) Markov decision process for a system with autonomous dynamics.

In an n dimensional vector space, the change of the current state x to the next state x_+, under the application of a control u in an environment parameterized by Ω, where f represents the time independent state transition function, can be written as:

$$x_+ = f(x, u, \Omega) \tag{1}$$

where $x, x_+ \in \mathcal{X}$, can possibly also represent an element of Lie Group and $u \in \mathcal{U}$. The solution to the corresponding optimal control problem would then be to find a pair $x(t), u(t)$ which minimizes a cost functional $L(X, U)$. In a discretized transcription, $L(X, U)$ can be written as an infinite sum of running cost, $l(x, u)$:

$$L(X, U) = \sum_{k=0}^{+\infty} l(x_k, u_k) \tag{2}$$

where $l(x, u) : \mathcal{X} \times \mathcal{U} \to \mathbb{R}$.

For finite time horizon problems of length T, the $L(X, U)$ is split in two parts:

$$L(X, U) = \sum_{k=0}^{T-1} l(x_k, u_k) + l^f(x_T) \tag{3}$$

where $l^f(x_T)$ is the cost at the terminal position. We denote X^*, U^* as the optimal solution to this minimization problem over a finite time horizon T and from the initial starting state x_0.

Recursive Optimality If the solution pairs are optimal, then we can define a *value function*, $V : x \to V(x)$, as the optimal value of the cost functional when the system at the starting state x_0 moves along the optimal trajectory, X, while following an optimal policy $\pi : x \to \pi(x) \in \mathcal{U}$.

We can further define V_k as the cost-to-go over the horizon $T - k$ from a starting state x by rewriting Eq. 3 as:

$$V_{T-k}(x) = \min_U \sum_{j=k}^{T-1} l(x_j, u_j) + l^f(x_T) \tag{4}$$

If l^f can be replaced with value function, then the cost-to-go over the horizon $T - k$ can be reformulated to obtain recursive optimality. In that case, the minimization problem becomes:

$$V_{T-k}(x) = \min_u l(x, u) + V_{T-k-1}(f(x, u)) \tag{5}$$

On the assumption, that a fair estimate of the value function can be provided, the problem transforms into an infinite horizon problem while remaining solvable with finite resources. Furthermore, the cost-to-go becomes independent of timestep and is equal to value function: $V_T(x) = V(x) \quad \forall T > 0$

Differential Dynamic Programming DDP is a second order iterative algorithm, [18, 35] that takes advantage of the recursivity of Bellman' Optimality Principle by adding the boundary condition, $V_T(x) = l^f(x_T)$ to Eq. 5. In each iteration, it numerically solves the optimal control problem described above by performing a backward and a forward pass on the current estimate of the state-control trajectories (X,U) : a backward phase to estimate the value function as quadratic fit along the current candidate trajectory, a forward phase to refine the candidate trajectory based on the value function.

To construct a quadratic fit of the value function, DDP measures the deviations from the current candidate trajectory through Taylor's expansion [19], discards terms beyond second-order. It then returns an estimation of the cost-to-go and the hessian and gradient at every step along the preview horizon.

This implies that for the neural network that approximates the value function to be substituted as a proxy for $l_f(x_T)$, it has to be sufficiently accurate and twice differentiable, since, in the backward pass, DDP requires a estimate of value function along with its first and second order derivatives.

3 Differential Policy Value Programming

Algorithmic Principles We propose an algorithm based on the data efficiency of Sobolev learning, recursivity of Bellman's optimality principle and robustness of DDP.

Eq 5 immediately shows that the global value function, V^*, can be approximated through the locally optimal rollouts of DDP if V is known. With this in view, we propose ∂PVP : Differential Policy Value Programming : to iteratively estimate the global time independent value function and state-control trajectories. We use three neural networks : a residual network, V_α^n to approximate the value function through Gauss decomposition, a state network, X_β^n, to learn the state trajectory, and a final control network, U_γ^n, to estimate the control trajectory[1]. α, β, γ are the parameters of the respective neural networks at the n^{th} iteration.

At the beginning of ∂PVP, we generate a batch of trajectories for a predefined horizon length T, but without any terminal cost model, i.e $l^f = 0$. We use this for offline training of $V_\alpha^{n=1}$, $X_\beta^{n=1}$, $U_\gamma^{n=1}$. We learn the value function through Sobolev regression, while we use classical regression to learn state and control trajectories. At the end of the first iteration, $V_\alpha^{n=1}$ approximates the cost-to-go for a horizon of length T, $X_\beta^{n=1}$ approximates the state trajectory from a given starting state x and $U_\gamma^{n=1}$ approximates the control to be applied at each step along the preview horizon.

Iterative Learning ∂PVP then proceeds to iteratively build upon its estimates of the value function, state and control trajectories. In subsequent iterations V_α^n acts as the proxy for terminal cost function. So at the end of the every iteration, Eq. 4 is changed to:

$$V^n(x) = \min_u \sum_{k=0}^{T-1} l(x_k, u_k) + V_\alpha^{n-1}(x_T) \tag{6}$$

where n is the iteration number V_α^{n-1} is the value function approximated in the previous iteration, while X_β^n, U_γ^n are used online in DDP to provide candidate trajectories in the forward pass.

The algorithm, as stated earlier, can be thought of as an *actor-critic* formulation, where V_α^n, X_β^n, U_γ^n play the role of actor, albeit in a supervised sobolev setting while DDP bases its decision on the quantities estimated by the actor.

With each iteration n, the learning should approximate the cost-to-go, state-control trajectories over a horizon $(n+1).T$. This in turn should provide a stable anchor for DDP at the terminal position, and drive it toward steady state solutions. Simultaneously, X_β^n, U_γ^n provide precise enough guesses of the steady state solutions.

By setting a reinforced loop between V_α^n, X_β^n, U_γ^n and DDP, the convergence of the algorithm should depend on $(n+1).T$. Therefore as n increases, V_α^n, X_β^n, U_γ^n should tend toward global V^*, X^*, U^*.

[1] see Sec 5 for use of Riccati Gains during training

and the algorithm should achieve super linear convergence in the number of attempts required to find optimal trajectories.

3.1 Architecture of the trainable models

The actors were initially implemented as the outcome of a three tailed feed forward network. This was specifically done to test the trade-off between the training time of a multi tailed network and the theoretical advantages of enabling the multiple tails to benefit from the rich information encoded in the common hidden layers. This resulted in bloated computation time of parameter updates during the training phase. The practical benefits of shorter training time far outweighed the theoretical advantages of common hidden layers. We do not include those results in this paper. For our experiments, we decided to learn the three different quantities - value function, state trajectory and control trajectory - on three different feed forward networks.

State-Control Approximators The state and control estimators are implemented with simple deep feed forward networks with 6 hidden layers with ReLU, ELU, Tanh alternately applied . The output and input shape depends on the dimensions of the optimal control problem.

Value Function Approximator In our experiments, we use the Gauss Newton approximation to write the value function as $V = r^T r$. Therefore, V_α learns the value function as a squared sum of residuals. We implement this with a feed forward network with three hidden layers and tanh activation. The final layer outputs a three-vector residual. The mathematical formulation of gradient and hessian of V_α is provided in the Appendix A.1. The update policy of this network is provided in Appendix A.2

4 Evaluation

We benchmark our algorithm on three classical control problems: unicycle [36], cartpole [10, 37] and inverted pendulum [37], along with a torque-controlled 7 dof manipulator arm[2]. Since the precision of our algorithm increases with $(n+1).T$, where n, T are the iterations and predefined horizon of the problem, the quality of the learned value function should depend on the n^th iteration or on T or both. We establish validation dataset for simple classical control systems by sampling for a large collection of locally long optimal trajectories of horizon length 1000. For the 7 dof manipulator, sampling for a similar validation dataset is not feasible.

In this section, we present certain results of interest for unicycle and 7 dof manipulator arm for the complete algorithm[3]. Additional plots are provided in the Appendix. The source code is available at `https://gitlab.laas.fr/aparag/kuka-arm-dpvp`

4.1 Manipulator Arm

The goal for the manipulator arm is to reach a static target with its End-Effector (EE): the OCP is formulated as a static end-effector pose reaching task, controlled in torque with additional regularization on both state and torque controls. The cost functional also penalizes deviation from state limits. The dynamics of this 7 dof manipulator arm were computed through [14]. The policy trails were tested on Bullet [38]. During the training phase, 150 locally optimal samples of horizon length 200 were drawn the 14 dimensional configuration space in each of the 20 iterations. This resulted in our TO computing 10347 rollouts overall. The training phase lasted 61 minutes.

Fig 1 and Fig 2 shows the predicted state trajectories and control trajectories respectively, as compared with a corresponding locally long optimal trajectory. The advantage of learning state-control in a supervised setting can be seen in the smoothness of the predictions. The predicted state trajectories are nearly coincident with the corresponding long optimal trajectories. When these learned trajectories are used as a reference guide to direct the policy search in DDP, the number of rollouts/attempts required by DDP reduces, as opposed to the number of rollouts required by DDP when not provided any intelligent guesses.

[2]Kuka lwr iiwa R820 14

[3]The experiments were performed on core i9 processor with 32 Gb RAM

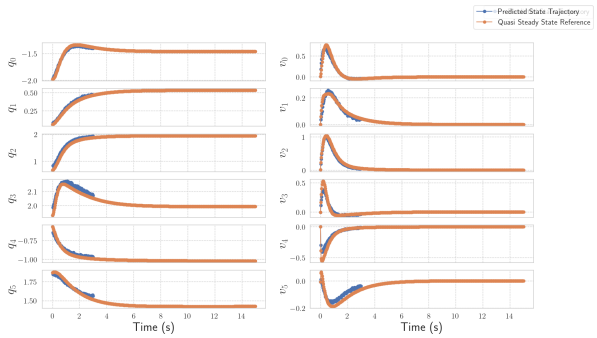


Figure 1: Comparison of predicted control trajectory and corresponding locally long optimal trajectory, for the manipulator task

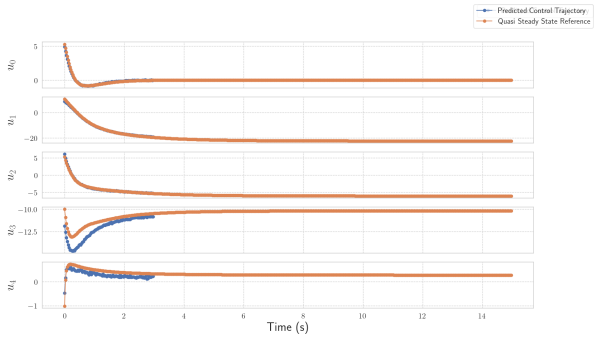


Figure 2: Comparison of predicted control trajectory and corresponding locally long optimal trajectory, for the manipulator task

In Fig 3, we show the number of rollouts required by our converged networks working in tandem with DDP to solve a problem. We use our trained networks inside our trajectory optimizer to solve for 600 uniformly sampled initial configurations from the configuration space. For those very same initial configurations, we also use trajectory optimizer but without any help from the trained networks. We then compare the number of rollouts needed in both cases. We see that when learning is used in DDP, the number of attempts made to find optimal policies decrease drastically. The average rollouts required by ∂PVP is 2.71, while the average rollouts required by DDP is 9.3.

Fig 4 shows the difference between the value functions predicted in the n^{th} and $(n+1)^{th}$ iteration. As we see, the relative bellman residuals stabilizes after the 10^{th} iteration. This effectively implies that 10 iterations are sufficient for convergence.

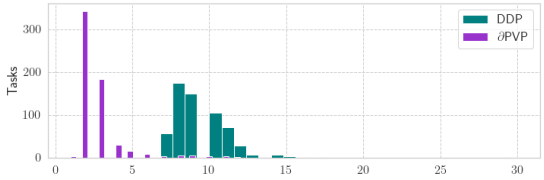


Figure 3: Histogram of attempts (rollouts) required by DDP to perform a task vs attempts required by DDP when guided by learning (∂PVP), shown here for the static pose reaching task with a manipulator arm.

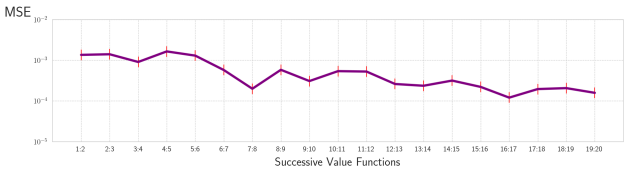


Figure 4: Bellman Residuals for the End Effector pose reaching task.

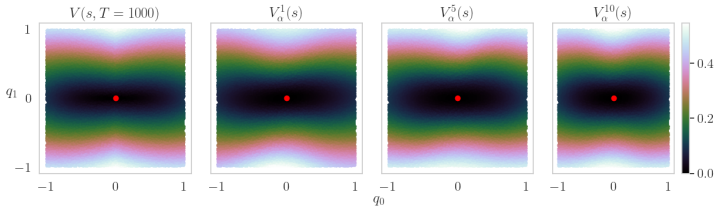


Figure 5: Comparison of the topology of value functions predicted by residual network at iterations 1, 5, 10 with value function at horizon length 1000: $V(s, T = 1000)$. The target position is shown in red. The unicycle can start anywhere in the configuration space and tries to reach for target

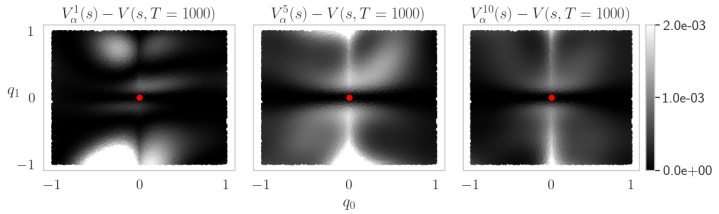


Figure 6: Errors between the predicted value function at iterations 1, 5, 10 and $V(s, T = 1000)$ for unicycle. The red dot is the goal position for the unicycle.

4.2 Unicycle

Fig 4.2 shows the evolution of the predicted value function for unicycle, as n increases. The unicycle is a 3 dof point mass in the 2D Cartesian horizontal plane, where it can either move forward or rotate on the spot with unconstrained longitudinal and angular velocity. The state space of the unicycle is a 3d cartesian vector space $s = [q_0, q_1, q_2] = [x, y, \theta]^T$. The control configuration is two dimensional vector: $u = [v, \omega]$. The task is to reach the state configuration $q = [0, 0, 0]^T$ while minimizing the weighted cost functional: $L(X, U) = ||w_1 q||^2 + ||w_2 u||^2$

To compare the differences in the predictions of the value function, we compute a *quasi-infinite* horizon value function validation dataset by sampling for locally long optimal horizons of length 1000. We use this dataset, $V(s, T = 1000)$, to establish distance to infinite horizon in value learning.

As we can see in Fig 4.2, the algorithm quickly learns the overall topology of value function across state space. As the number of iterations increase, it seems to be refine the inherent symmetry in the topology. Fig 6 shows the difference between quasi infinite horizon value function and predicted value functions at iterations 1, 5, 9. We observe that that the iterative aspect of ∂PVP allows it to learn value function over long horizons quite well despite initialization in short horizon. However, there seem to regions in configuration space difficult to handle (shown in white). We suspect that the non holonomic constraints in the unicycle environment lead to singularities which in turn destabilize learning. With more learning, the algorithm overcomes the presence of singularities. By the 10^{th} iteration, the algorithm had narrowed the location of singularities to be symmetrically distributed around the goal, coincident with the q_1 axis in Fig 6.

5 Discussion and Conclusion

In this work, we showed that reformulating reinforcement learning as a combination of iterative supervised learning phase, with emphasis on value functions, and simulators allows for reduction in trials needed to find on optimal solution. The iterative supervised learning phase enforces the stability of predictions through Sobolev regression while learning in the backdrop of recursive optimality further reduces the dependence on the hyper parameters.

There are a few points of interest in the way the information from policy trials are used in the supervised learning phase. First, in the learning phase the trajectory optimizer plays the role of an oracle to provide us, for each ocp, an estimation of time dependent value function at each node for every state trajectory, gradients and hessians of each value function along the state trajectory, control trajectory and first order derivatives of control. Of this information, we choose to keep the node with the highest preview horizon for learning value function. This can be seen as an under utilization of the state-value information. On the one hand, states with value function estimations of smaller preview horizon can augment the dataset but on the other hand, this approach can lead to a noisier dataset which can effectively slow down convergence.

To enforce a shorter training time, we also discarded the use of second order derivatives of value function. This also precluded the use of first order but high dimensional derivatives of the control trajectory.

Similarly we tested the feasibility of adding a kernel loss function based on Bellman' contraction operator on the learning of state and control trajectories. A kernel loss function that constrains trajectories to satisfy the Hamilton-Jacobi-Bellman criteria of optimal sub-structures: sub-solutions of an indefinite horizon optimal control problem should also be optimal solutions to the corresponding definite horizon sub problems, should in theory, elevate the quality of the predicted state and control trajectories. However implementing such a kernel loss function seemed to have little benefits at the expense of increased computation time. We leave it to a future work to explore these avenues.

The training time can be further reduced with the observation that the policies learned by the neural networks serve as a guide for DDP. In our experiments, we observe that the precision of learned state and control trajectories increases only slightly as ∂PVP iterates. This allows us flexibility in defining the number of training epochs in every iteration, which we need if the dimensionality of the problem under consideration increases. For experiments with manipulator arm, we choose to learn and improve state-control trajectories only in the initial and final iterations. We found this training strategy, where estimations of value functions are refined at every iteration, whereas estimation of state-control trajectories are refined only at start and end to be good enough for our purposes.

In this paper, we presented the foundational aspects of our algorithm. We believe that coupling approach presented in this paper is mature enough to be tested on legged robots such as quadrupeds and mini cheetah in real time. We plan to present those results in a future work.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540), 2015.

[2] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *IEEE International Conference on Intelligent Robots and Systems*. IROS, 2012.

[3] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. In *Proceedings of Robotics: Science and Systems*, Pittsburgh, Pennsylvania, June 2018.

[4] Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker, and Sergey Levine. Learning to walk via deep reinforcement learning. In *Robotics: Science and Systems XV*. RSS, 2019.

[5] Tuomas Haarnoja, Aurick Zhou, P. Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.

[6] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[7] David Hoeller, Farbod Farshidian, and Marco Hutter. Deep value model predictive control. In *Conference on Robot Learning*. CoRL, 2020.

[8] Andrew Ilyas, Logan Engstrom, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. A closer look at deep policy gradients. In *International Conference on Learning Representations*. ICLR, 2020.

[9] Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1291–1307, 2012.

[10] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[11] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[12] Luis Miguel Rios and Nikolaos V Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.

[13] Hong Qian and Yang Yu. Derivative-free reinforcement learning: a review. *Frontiers of Computer Science*, 15(6):1–19, 2021.

[14] Justin Carpentier, Guilhem Saurel, Gabriele Buondonno, Joseph Mirabel, Florent Lamiraux, Olivier Stasse, and Nicolas Mansard. The pinocchio c++ library: A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives. In *IEEE International Symposium on System Integration (SII)*. SICE, 2019.

[15] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. Difftaichi: Differentiable programming for physical simulation. *ICLR*, 2020.

[16] Russ Tedrake et al. Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems, 2014.

[17] M Huba, S Skogestad, M Fikar, M Hovd, TA Johansen, and B Rohal'-Ilkiv. Selected topics on constrained and nonlinear control. *Slovakia, ROSA. Dolnỳ Kubín*, 2011.

[18] David Q Mayne. Differential dynamic programming–a unified approach to the optimization of dynamic systems. In *Control and Dynamic Systems*, volume 10. Elsevier, 1973.

[19] Zhaoming Xie, C Karen Liu, and Kris Hauser. Differential dynamic programming with nonlinear constraints. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 695–702. IEEE, 2017.

[20] Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*. ICML, 2013.

[21] Nicolas Mansard, Andrea DelPrete, Mathieu Geisert, Steve Tonneau, and Olivier Stasse. Using a memory of motion to efficiently warm-start a nonlinear predictive controller. In *IEEE International Conference on Robotics and Automation*. ICRA, 2018.

[22] Kendall Lowrey, Aravind Rajeswaran, Sham Kakade, Emanuel Todorov, and Igor Mordatch. Plan online, learn offline: Efficient learning and exploration via model-based control. In *International Conference on Learning Representations*. ICLR, 2019.

[23] Ian Lenz, Ross A Knepper, and Ashutosh Saxena. Deepmpc: Learning deep latent features for model predictive control. In *Robotics: Science and Systems*. RSS, 2015.

[24] Aviv Tamar, Garrett Thomas, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. Learning from the hindsight plan—episodic mpc improvement. In *IEEE International Conference on Robotics and Automation*. ICRA, 2017.

[25] Mingyuan Zhong, Mikala Johnson, Yuval Tassa, Tom Erez, and Emanuel Todorov. Value function approximation and model predictive control. In *Symposium on Adaptive Dynamic Programming and Reinforcement Learning*. IEEE, 2013.

[26] Amit Parag, Sébastien Kleff, Léo Saci, Nicolas Mansard, and Olivier Stasse. Value learning from trajectory optimization and sobolev descent: A step toward reinforcement learning with superlinear convergence properties. In *International Conference on Robotics and Automation*, 2022.

[27] Wojciech Marian Czarnecki, Simon Osindero, Max Jaderberg, Grzegorz Świrszcz, and Razvan Pascanu. Sobolev training for neural networks. In *Advances in Neural Information Processing Systems*. Neural IPS, 2017.

[28] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2), 1991.

[29] Tom M Mitchell, Sebastian B Thrun, et al. Explanation-based neural network learning for robot control. In *Advances in Neural Information Processing Systems*. Neural IPS, 1993.

[30] Jeong-Woo Lee and Jun-Ho Oh. Hybrid learning of mapping and its jacobian in multilayer neural networks. *Neural computation*, 9(5), 1997.

[31] James B Witkoskie and Douglas J Doren. Neural network models of potential energy surfaces: Prototypical examples. *Journal of chemical theory and computation*, 1(1), 2005.

[32] Richard Bellman. Dynamic programming. *Science*, 153(3731), 1966.

[33] Shige Peng. A generalized dynamic programming principle and hamilton-jacobi-bellman equation. *Stochastics: An International Journal of Probability and Stochastic Processes*, 38(2):119–134, 1992.

[34] Carlos Mastalli, Rohan Budhiraja, Wolfgang Merkt, Guilhem Saurel, Bilal Hammoud, Maximilien Naveau, Justin Carpentier, Ludovic Righetti, Sethu Vijayakumar, and Nicolas Mansard. Crocoddyl: An efficient and versatile framework for multi-contact optimal control. In *International Conference on Robotics and Automation*. ICRA, 2020.

[35] Li-zhi Liao and Christine A Shoemaker. Advantages of differential dynamic programming over newton's method for discrete-time optimal control problems. Technical report, Cornell University, 1992.

[36] Sara Fleury, Philippe Soueres, J-P Laumond, and Raja Chatila. Primitives for smoothing mobile robot trajectories. *Transactions on Robotics and Automation*, 11(3), 1995.

[37] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[38] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2021.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 5
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] see Section 3
   (b) Did you include complete proofs of all theoretical results? [Yes] see Section 3
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] see Section 5
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Appendix

A.1 Architecture of Value Function Estimator

DDP as mentioned in Sec 2.1, is a second order algorithm. It computes the 2^{nd} and 1^{st} order derivatives of value function in the backward pass and therefore requires $V_\alpha^{'}$ and $V_\alpha^{''}$ in every iteration of our algorithm.

Since computing $V_\alpha^{''}$ is not feasible, especially in higher dimensional systems like manipulator arm, we use Gauss approximation to design V_α as squared sum of residuals:

$$V(x|\alpha) = R(x|\alpha)^2 \tag{7}$$

This immediately allows us to write the 1^{st} and 2^{nd} order derivatives as :

$$V'(x|\alpha) = 2R'(x|\alpha)^T R(x|\alpha) \tag{8}$$

$$V''(x|\alpha) \approx 2R'(x|\alpha)^T R'(x|\alpha) \tag{9}$$

A.2 Sobolev Regression

The trainable parameters of the learning model are consequently changed in response to cumulative errors accrued by two losses - 0^{th} and 1^{st} order. The 0^{th} order loss is identical to canonical losses used in functional regression that penalizes the difference in observations and target with some norm λ_d, while the 1^{st} order loss forces the neural network to match it derivatives with the corresponding derivatives of the target. Theoretically sobolev regression can involve higher order derivatives, however in our experiments we choose to stay at 1^{st} order with norm $d = 2$ to offset the increased workload on the automatic differentiation engine.

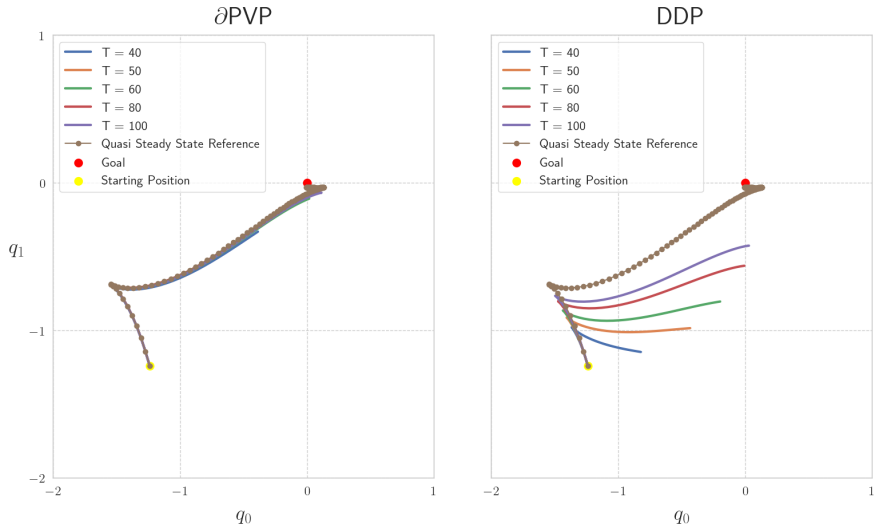


Figure 7: Trajectories followed by unicycle under ∂PVP and DDP.

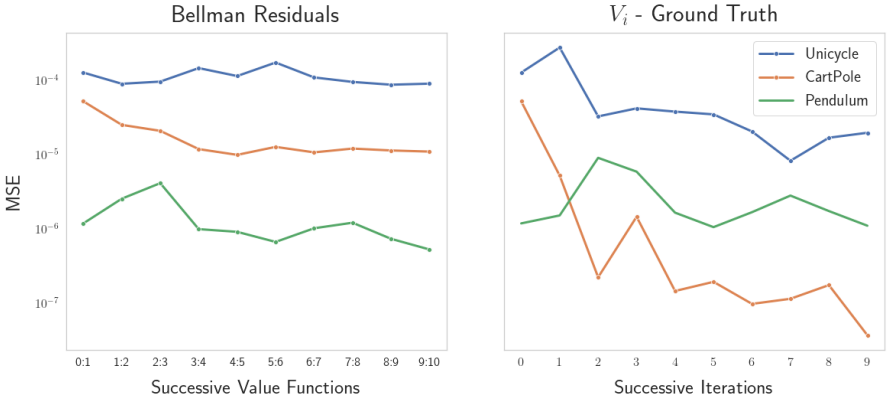
A.3 Stability of infinite horizon

V_α^n approximates value function at different horizons during its iterations. As noted before, the horizon approximated during an iteration is a function of n and T. In Fig A.3, we compute the solution to an identical task (for unicycle) over different horizon lengths. We see that the trajectories computed by a converged V_α^n are much closer to the quasi steady state solution even when the preview horizon is short. Comparatively, without a terminal proxy, the variance of trajectories computed by DDP is much higher.

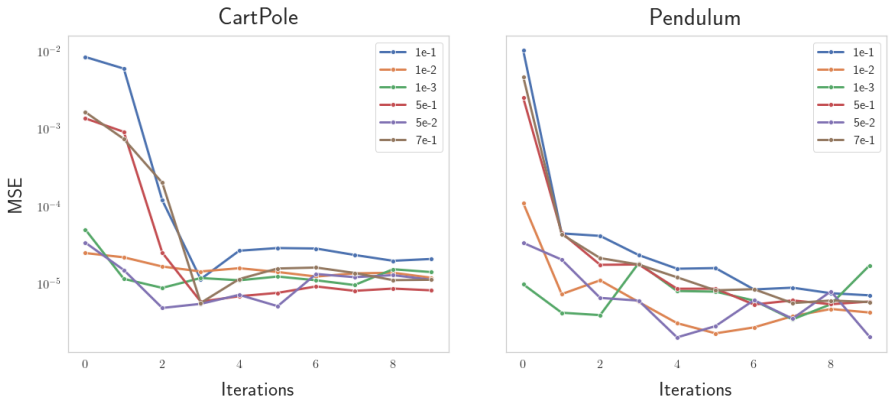
A.4 Related Work

The algorithm presented in this paper has two parts: learning value function and learning state-control policies to warmstart DDP. Previous investigations in solely learning value function [26] are shown here for self consistency.

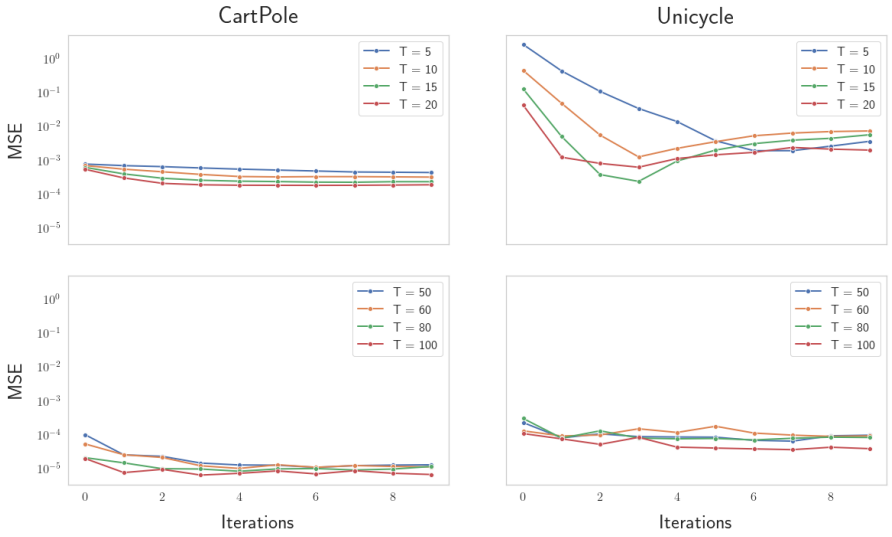
The following Figure shows bellman residuals and validations errors for systems where it is computationally cheap to generate huge validation datasets. The DDP solver computed locally optimal trajectories of horizon length 50 that served as datasets in every iteration.



The Figure below establishes the robustness under noise. Noise is injected into the system at the initial step. We observe the even under incorrect initialization, Sobolev regression and Bellman' optimality principle force the value estimation to converge

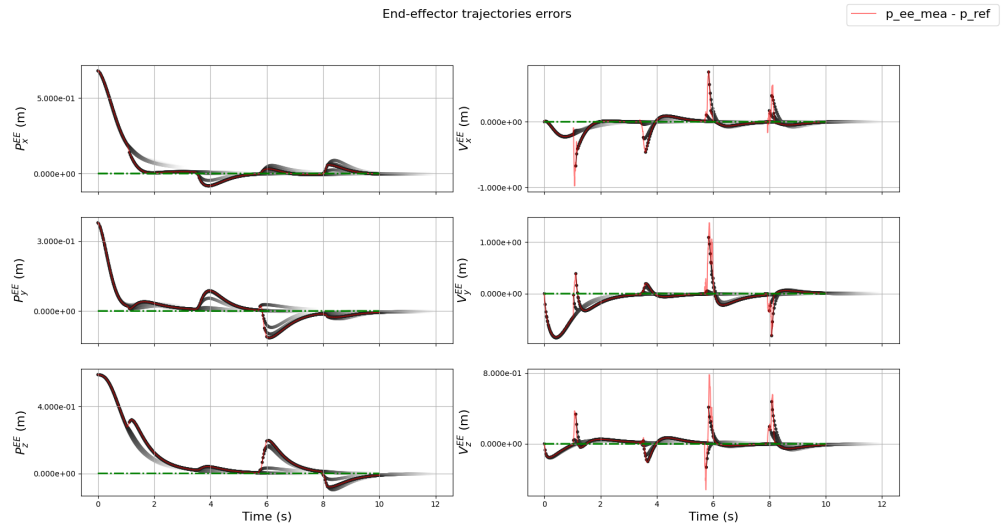


In Figure below, we see that learning converges faster when T increases. For $T > 40$, convergence is very quickly reached. For $T < 40$, convergence is slower.



The following experiment was run online in simulation in PyBullet. The terminal cost predicted by V_α serves as highly stable anchor allowing for quick replanning under disturbance. The figure below shows the evolution of meas squared errors be-

tween EE trajectories and ground truth, when disturbances are inject in the system



Finally in the Figure below, we show the optimal trajectories computed by solver with terminal proxy V_α, for the End Effector.