CS230

Tour Guide
Deep Learning for Trajectory Optimization

James Guthrie
jguthr@stanford.edu

Abstract

Real-time trajectory generation for autonomous vehicles requires solving nonlinear optimization problems that are parameterized by the current system state and objectives. Solver convergence is greatly affected by how close the problem is initialized to the true answer. We train a neural network on a library of optimal trajectories representing a reuseable launch vehicle that must reach a specified final position. The neural network is then used to initialize ("warm-start") the nonlinear optimizer on new, unsolved instances of the problem. We achieve a 2.2x speedup in solver runtime on average compared to using a default initialization.

1 Introduction

Autonomous systems such as self-driving cars and unmanned aerial vehicles perform motion planning in real-time. These plans must respect the governing dynamics of the system to be realizable. Further, to ensure acceptable performance, they should be optimal with respect to relevant criteria (e.g. fuel expenditure, final arrival time). This process is typically referred to as trajectory optimization. Due to the nonlinear dynamics of most autonomous systems, trajectory optimization often involves solving a nonlinear optimization problem. Unlike convex optimization, generic nonlinear optimization problems typically 1) converge to a local (vice global) minimum and 2) only have guaranteed convergence properties when the initial solution guess is sufficiently close to the final answer. Thus good initializations are essential to achieve fast and reliable convergence in most real-time trajectory optimization problems.

In this project we train a neural network to generate a good initial guess for a trajectory optimization problem that is parameterized by the current state (initial position) of the system and its desired final state. We generate a dataset of sample solutions offline using the optimal control software GPOPS-II [5] and then train a neural network to replicate these "seed" trajectories. Ideally the network will generalize well and provide good initial guesses for previously unsolved problems. This initial guess can then be quickly refined to the true answer by solving the nonlinear optimization problem online.

An alternative approach would be to attempt pure imitation learning in which the neural network exactly replicates the output of the nonlinear optimization solver. In theory, the online motion planning could then simply evaluate the neural network with the current system parameters. While desirable, this creates validation and verification issues in ensuring that the neural network always respects the various constraints imposed on the trajectory. Thus our focus will be on the less ambitious goal of simply being able to “warm-start” a nonlinear optimization problem via a trained neural network.

2 Related work

Trajectory optimization via nonlinear optimization is fundamentally limited by solver times. Recent research has explored leveraging neural networks as a means of accelerating solver runtimes. In [1] a neural network is used to approximate the cost-to-go function of a model predictive controller. This allows the trajectory optimization problem to optimize over a shorter time horizon yielding faster runtimes without sacrificing performance. Although theoretically appealing, the authors are not able to extend the approach beyond basic toy problems as the accuracy required in the value function approximation is "practically impossible."

More recently, [2] utilized a stored library of pre-computed trajectories to provide the initial guess to a nonlinear optimization solver. For a given problem instance the best initial guess from the trajectory library was selected using k-nearest neighbors. As expected, the resulting optimization problem showed improved convergence compared to a naïve initial guess. This came at the cost of 1) a potentially expensive search process for the best initial guess amongst the candidates within the trajectory library and 2) a large memory footprint which is not practical for many aerospace systems.

In [3], the authors first generate a family of relevant trajectories for an unmanned aerial vehicle which are dependent on the current state of the system. A neural network is then trained to approximate the trajectories. Finally, the neural network is used to warm-start a nonlinear optimization problem for real-time trajectory generation. The authors achieve quick convergence (2 to 5 iterations) for the system which consists of six states controlled over a short horizon (5s). This work will extend the approach of [3] to the more complex problem of a reusable launch vehicle landing problem taken from [4]. The problem has more challenging dynamics and involves much longer trajectories (approximately 150 seconds).

3 Dataset and Features

The dataset consists of 14,000 trajectories for a reusable space launch vehicle that is returning to earth. The vehicle is controlled by the angle-of-attack (aoa) and sideslip angle (bet) which are used to guide it to a terminal latitude and longitude at which point another guidance algorithm takes over. The vehicle must arrive at this terminal position with an altitude of 24kft, flight path angle (fpa) of $-5°$ and azimuth angle of $0°$. Due to uncertainty in the mission profile, the initial altitude (alt_i) and terminal latitude (lat_f) and longitude (lon_f) can vary as given in Table 1. Figure 1 shows two samples trajectories.

Table 1: Initial and Terminal Conditions

	Altitude	Longitude	Latitude	Velocity	Flight Path Angle	Azimuth
Initial	60-80kft	0	0	80m/s	$-1°$	$90°$
Terminal	24kft	$70° - 80°$	$25° - 35°$	Free	$-5°$	$0°$

The commercial optimal control software GPOPS-II [5] is used to determine the appropriate controls for arriving at the given terminal position in minimum time for 14,000 possible scenarios (alt_i, lat_f, lon_f). Due to memory limitations it is not feasible to store all possible trajectories on the flight computer and load the appropriate control sequence in real-time. Instead, the optimal control problem will be solved online for the system conditions using the nonlinear solver IPOPT [8]. This solver requires an initial guess of the solution which consists of both the system controls and states. The closeness of the guess to the true answer impacts the solver runtime. We aim to accelerate this by training a neural network to generate a good initial guess for a given (alt_i, lat_f, lon_f). Each control and state variable is represented by 100 equally-spaced points in time (i.e. $aoa, bet, alt, lon, lat, vel, fpa, azi \in \mathbb{R}^{100}$). Additionally we provide the flight duration $t_f \in \mathbb{R}$ which determines the time duration each point represents. The input is $x = [alt_i \quad lat_f \quad lon_f] \in \mathbb{R}^3$.

The output is $y = [aoa \quad bet \quad alt \quad lon \quad lat \quad vel \quad fpa \quad azi \quad t_f] \in \mathbb{R}^{801}$. Our problem is one of multivariate regression in which we want to minimize the average mean squared error:

$$J(y, \hat{y}) = \frac{1}{m}\frac{1}{n_y}\sum_{i=1}^{m}\sum_{j=1}^{n_y}(y_j - \hat{y}_j)^2 \tag{1}$$

Here $m = 14,000, n_y = 801, y \in \mathbb{R}^{801}$ is truth, and $\hat{y} \in \mathbb{R}^{801}$ is the output of the neural network which is given by $f(x)$. Using scikit learn [10], we shift (remove mean) and scale to unit variance each element of the input x and output y. The scaling of the input x is done to aid the convergence of the optimization methods. The scaling of the output y is done to ensure that our loss function weights each output equally. Without this scaling, the training would heavily focus on the velocity and altitude signals as these have larger mean relative to the other degree-based signals (which are numerically represented in radians). Scaling the output has the added benefit of making the loss function value easily understood. $J(y, \hat{y}) = 0.01$ means on average we have 1% error (in the 2-norm sense) in our representation of y with all elements weighted equally.

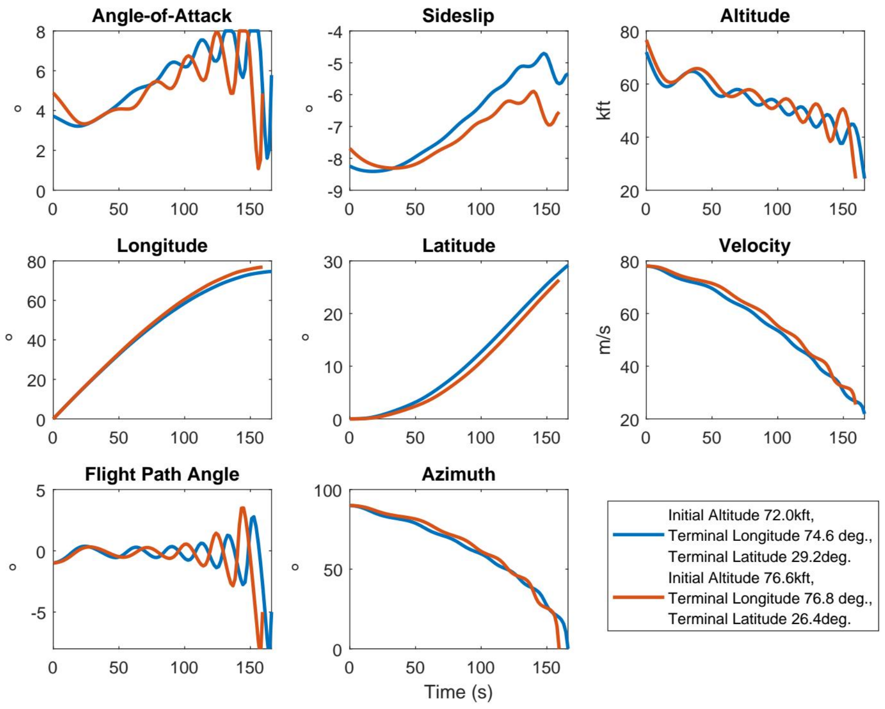

Figure 1: Example Trajectories

4 Methods

TensorFlow [7] in combination with Keras[6] was used to implement a fully-connected neural network with n_l hidden layers and n_h rectifier-linear units (ReLU) per hidden layer. The input was (alt_i, lat_f, lon_f). The output layer consisted of 801 linear weightings to generate $\hat{y}$. The 14,000 trajectories were split 90/10 into training and test set respectively. The ADAM [9] optimizer was utilized throughout.

5 Experiments/Results/Discussion

With the data properly scaled, it was straight-forward to find an appropriate learning rate α for the ADAM optimizer. We used $\alpha = 0.01$, which showed good convergence as seen in Figure 2. The

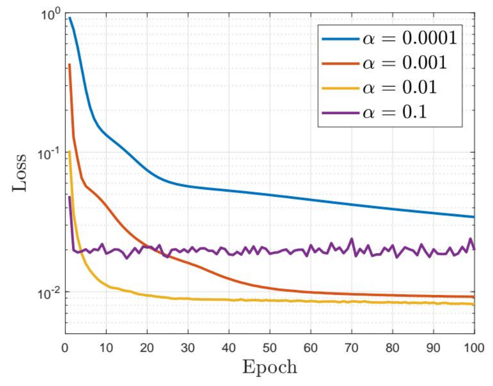

Figure 2: Impact of α on convergence

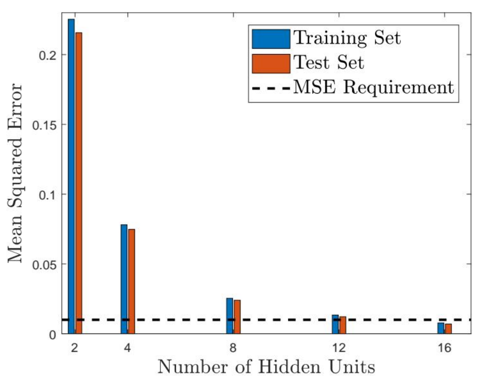

Figure 3: Impact of hidden units on MSE

remaining ADAM parameters were standard ($\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e-8$). A relatively small mini-batch size of 32 was used to ensure the model generalized well.

As the resulting neural network is intended to be deployed on a flight computer with limited computational resources, a strong emphasis was placed on minimizing the size of the network. Numerical experiments with the solver IPOPT [8] suggested that improving the initial guess gave diminishing returns once it was within 1% (as calculated by (1) of the true solution. Thus our hyperparameter tuning was done to find the smallest network that would achieve a 1% average mean squared error on the test and training sets. Instead of starting with a large network and enforcing sparsity through L_1 or L_2 penalty terms, we instead iterated on the number of hidden units until we achieved a satisfactory loss.

Figure 3 shows the reduction in cost function $J(y, \hat{y})$ as we increase the number of hidden units (n_h) in a fully-connected neural network with one hidden layer ($n_l = 1$). With the chosen batch size, we did not experience any over-fitting issues. With $n_l = 1, n_h = 16$ we achieve our 1% average mean squared error requirement. With this model, the cost function was 0.0077 and 0.0070 on the training and test set respectively (i.e. below the 1% requirement). Figure 4 shows an example of a reconstructed angle-of-attack profile with different numbers of hidden units. With $n_h = 16$ the resulting profile is nearly indistinguishable from truth.

Table 2 further summarizes the performance of our neural network on the test and training sets. While the average mean squared error satisfies the 1% requirement, it is not met on every individual test and and training example. Specifically, a small number of profiles that featured significant "clipping" of the angle-of-attack (it is limited to $8°$) had poorer performance. Figure 5 shows an example of one such profile. From time $130s - 170s$ the angle-of-attack is saturated. By increasing the number of hidden layers, we were able to improve the performance on these outlier cases as seen in Figure 5 where $n_l = 3$ closely matches truth. However, the resulting increase in neural network complexity was undesirable and computational testing (discussed later) confirmed the baseline performance was sufficient.

Table 2: Performance on test and training sets ($n_h = 16, n_l = 1$)

	Min. Loss	Max. Loss	Mean Loss
Train	0.0007	0.018	0.0077
Test	0.0007	0.016	0.0071

As previously discussed, the neural network developed herein is intended to provide a good "warm-start" to the optimization solver IPOPT. To demonstrate its utility, we solved 1000 instances of the re-entry problem with parameters (alt_i, lat_f, lon_f) drawn from ranges given in Table 1. We first solved the problems with IPOPT and a default initialization ("cold-start"). We then solved the same problems using warm-starts generated by the neural network. Table 3 lists the timing results. Although the solver obtains the same final solution in both instances, the warm-started solver

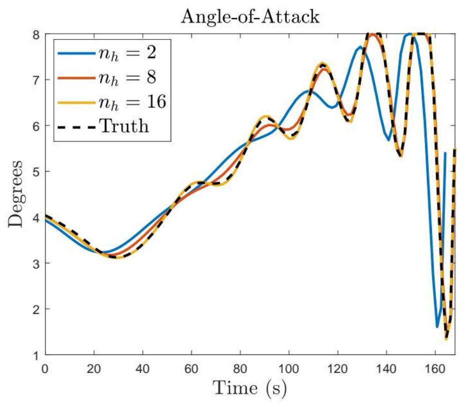

Figure 4: Approximation of angle-of-attack

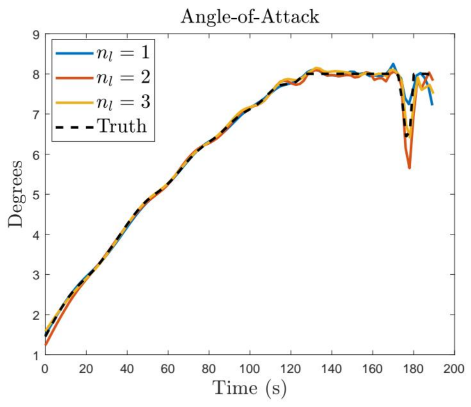

Figure 5: Saturating angle-of-attack profile

converges 1 second faster on average (a speedup factor of 2.2x). In one instance the warm-start reduces the solve time by 6.96s. For real-time trajectory planning these differences in timing can decide whether a given approach is feasible or not.

Table 3: Solve Times

	Minimum (s)	Maximum (s)	Mean (s)
Cold-Start	0.64	8.08	1.76
Warm-Start (w/Neural Network)	0.25	1.31	0.77
Δ = Cold Start - Warm Start	0.09	6.96	1.00

6 Conclusion/Future Work

Minimizing solver runtime is essential to successfully deploying optimization-based guidance and control algorithms on future autonomous vehicles. This project demonstrated that relatively simple neural networks can be used to help accelerate the nonlinear optimization solvers by providing good initial solutions guesses. Using a warm-start provided from a neural network, we were able to speed up the solver runtime by 2.2x on average. It was unexpected that such a simple neural network (one hidden layer with 16 ReLUs) could so closely replicate a database of trajectories. Unfortunately, time limitations prevented us from creating a new, more difficult set of trajectories. As future work we plan to increase the dimensionality of the trajectory data incrementally and pursue more complex neural network architectures as necessary.

7 Contributions

All work (data generation, data pre-processing, neural network development) was done by the author. Code for this project is available at: `https://github.com/guthriejd1/cs230_project`

References

[1] M. Zhong, M. Johnson, Y. Tassa, T. Erez, and E. Todorov, “Value Function Approximation and Model Predictive Control,” 2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning pp.100-107

[2] W. Merkt, W. Ivan, V. Vijayakumar, “Leveraging Precomputation with Problem Encoding for Warm-Starting Trajectory Optimization in Complex Environments,” 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 5877-5884

[3] N. Mansard, A. DelPrete, M. Geisert, S. Tonneau and O. Stasse, "Using a Memory of Motion to Efficiently Warm-Start a Nonlinear Predictive Controller," 2018 IEEE International Conference on Robotics and Automation

(ICRA), Brisbane, QLD, 2018, pp. 2986-2993.
[4] J. Betts Practical Methods for Optimal Control and Estimation Using Nonlinear Programming.SIAM Press, Philadelphia, PA. 2010
[5] M. Patterson, A. Rao "GPOPS-II A MATLAB Software for Solving Multiple-Phase Optimal Control Problems using hp-Adaptive Gaussian Quadrature Collocation Methods and Sparse Nonlinear Programming," ACM Transactions on Mathematical Software, Vol. 41, No. 1, October 2014.
[6] F. Chollet et. al, "Keras", 2015. `https://keras.io`
[7] M. Abadi et. al, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. `https://www.tensorflow.org`
[8] A. Wachter, L. Biegler, "On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming," Mathematical Programming 106(1), pp. 25-27, 2006
[9] D. Kingma, "Adam: A Method for Stochastic Optimization," Proceedings of the 3rd International Conference on Learning Representations, 2014.
[10] F. Pedregosa et. al, "Scikit-learn: Machine Learning in Python", Journal of Machine Learning Research, 2011